API Reranking for Automatic Code Completion: Leveraging Explicit Intent and Implicit Cues from Code Context

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have significantly advanced software development, particularly in automatic code completion, where selecting suitable API documents from vast third-party libraries plays a critical role. However, current solutions either focus on recommending APIs based on user queries or code context, without considering both aspects simultaneously. To bridge this gap, we propose a novel framework APIRANKER to rerank candidate API documents based on both the explicit developer intent and implicit cues embedded in the incomplete code context. To automatically construct ranking data, we introduce a self-supervised ranking framework that automatically constructs data by assessing the relevance of API documents to code context with a perplexity-driven approach via comments. To enhance API relevance detection, we propose a novel reranking model that predicts relevance scores by capturing a hidden reasoning state to detect relevance. The experimental results show the effectiveness of our approach, providing more accurate API recommendations and enhancing automatic code completion. The code is available[1] and the dataset will be released.

1 Introduction

The introduction of LLMs has led to advancements in automatic code completion (Husein et al., 2024), with recent models adopting the *retrieve-then-generate* paradigm (Nashid et al., 2024). This approach enables LLMs to dynamically retrieve up-to-date Application Programming Interface (API) information from documents, rather than relying solely on static training data.

A crucial aspect of this process is selecting suitable APIs from massive amounts of third-party libraries. The choice of API not only determines the functionality of the generated code but also affects its efficiency, maintainability, and overall integration with the existing software system (Wang et al., 2024b). Current research predominantly focuses on two approaches: **1)** retrieving APIs based on user queries (query-based API recommendation) (Wu et al., 2023), which neglects the code context, and **2)** completing user-written code based on the preceding code context (Peng et al., 2022), which fails to capture the developer's underlying intention behind the API usage. However, automating code completion with user preferable APIs requires a more comprehensive model that considers both the explicit intent conveyed by user queries and the implicit cues embedded in the code context.

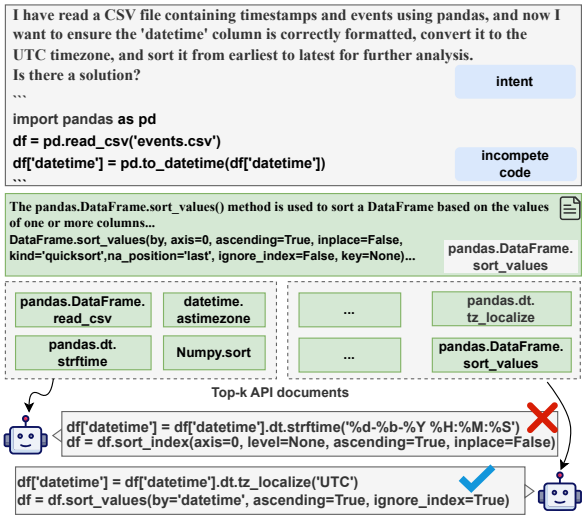

Figure 1: Example of retrieval-augmented code completion with different retrieved top-k API documents.

Consider the following practical scenario shown in Fig. 1: Alice is a developer, she encounters a problem during her daily work, i.e., "*convert the 'datatime' column to UTC timezone and sort it from earliest to latest for further analysis*". She asks the LLM to complete her code. The LLM completes the code using the *retrieve-then-generate* paradigm. However, without knowing Alice's intention or

[1] https://anonymous.4open.science/r/APIRanker-C442

the current code context, a large number of relevant but unsuitable APIs may be recommended (*e.g.*, `datetime.astimezone`, `numpy.sort`, etc.). If the target APIs are not ranked in the top-k retrieval results, they will not be used for code completion, causing the auto-completed code to misalign with her intended behavior. If Alice inputs both her intent and the current code context, and the target APIs (*e.g.*, `df.dt.tz_localize`, `pandas.DataFrame.sort_values`) are successfully recommended, the LLM is likely to complete her code by smoothly integrating with the recommended APIs, effectively solving her problem.

Based on our previous observations, there is a need for code completion using the correct APIs. However, recommending API based on both the developer's intent and the incomplete code is a challenging task: (i) **Lack of methods for large-scale data construction, hindering learning-based approaches**. Creating training datasets for API recommendation requires manually evaluating the relevance of API documentation to the developer's requirements. This process depends on domain expertise, which makes large-scale data collection impractical. Moreover, the lack of automated methods exacerbates this challenge, further limiting the development of learning-based approaches. (ii) **The relevance of API documents to the developer's requirements is hard to learn and capture**. API documentation exists in various formats and writing styles, often lacking consistency in structure and content, making it difficult to establish direct mappings to developer requirements. Moreover, the developer's requirements involve both intent and incomplete code, and the cues hidden within the code context are difficult to capture. This makes it challenging for API recommendations to satisfy both aspects simultaneously.

To tackle the above challenges, we propose a novel framework named APIRANKER, which is designed to rerank candidate API documents based on the developer's intent and their incomplete code. To address the challenges of lacking training data, we propose a self-supervised ranking framework to automatically mine ranking data. Specifically, we leverage a perplexity-driven relevance ranking approach, which uses LLM as an evaluator to automatically discover the relevance of API documents to developer requirements by measuring the perplexity of completed code. To bridge the gap between perplexity and true semantic relevance, we employ a perplexity alignment strategy via comments to reduce noise from perplexity shifts caused by code formatting and syntactic variations. To better learn and capture the relevance of API documentation to the developer's requirements, we design a novel reranking model architecture consisting of two main components that learn relevance by comparing the influence of different API documents on code completion. In particular, we leverage a hidden reasoning state extractor to capture the relevance of the API documentation to implicit cues from code context by extracting the reasoning state from LLMs during inference. The model then explicitly predicts a relevance score using a relevance detector, which learns and identifies relevance from the reasoning state.

In summary, our paper makes the following contributions: (1) Current research mainly focuses on a single requirement. To the best of our knowledge, no prior work has deeply explored how to recommend APIs based on both the developer's intent and their incomplete code. (2) We propose a self-supervised ranking framework to construct ranking data $\langle incomplete\ code, target\ code, API\ documents, relevance\ scores \rangle$ automatically. (3) We design a novel reranking model architecture that leverages LLM's semantic understanding to detect the relevance of API documents to a developer's requirements. The experimental results show the effectiveness of our model over a set of baselines, showing its potential to enhance automatic code completion by reranking candidate API documents. We hope our study can lay the foundations for this research topic.

2 Related Work

API Recommendations. API recommendation methods typically rely on two main sources: natural language queries and contextual code information. Some studies focus on query intent, such as BIKER (Huang et al., 2018) and CLEAR (Wei et al., 2022), while others emphasize code context, like GAPI (Ling et al., 2021) and MEGA (Chen et al., 2023). Deep learning models like DeepAPI (Gu et al., 2016) and CodeBERT (Feng et al., 2020) enhance recommendations through embedding-based methods, using pretrained models to calculate similarities between queries and APIs. However, limited labeled data hampers model performance (Ma et al., 2024). In contrast, our approach leverages LLMs and automatically

generated data to reduce reliance on QA data, improving API recommendation performance.

Retrieval-augmented Code Generation. Retrieval-augmented generation (Gao et al., 2023) has proven valuable in code generation (Parvez et al., 2021), especially as code libraries are frequently updated (Lu et al., 2022). For instance, CodeGen4Libs (Liu et al., 2023) recommends class-level APIs through a two-stage process of retrieval and fine-tuning. DocPrompting (Zhou et al., 2022) enables continuous updates to the documentation pool, ensuring that the most current code libraries are used for generation. ToolCoder (Zhang et al., 2023) integrates API search tools and uses automated data annotation to teach the model how to use tool usage information, thereby enhancing code generation.

3 Methodology

In this section, we introduce a novel framework APIRANKER, aimed at reranking candidate documents based on their relevance to a given query (Section 3.1), thereby enhancing automatic code completion. To overcome the lack of training data, APIRANKER utilizes self-supervised ranking framework to generate data (Section 3.2). Moreover, a novel model architecture (Section 3.3), which includes a hidden reasoning state extractor and a relevance detector, is trained to predict the relevance score. After reranking training (Section 3.4), APIRANKER can rerank candidate documents to improve automatic code completion.

3.1 Task Definition

Given a query q, which contains natural language (NL) intent x and the corresponding incomplete code snippet c, the objective is to rerank the retrieved API documents D, prioritizing documents that are both relevant to NL intent x and beneficial for completing the code c. This process aims to improve automatic code completion by presenting the most pertinent API documents at the top of the ranking.

3.2 Self-supervised Ranking Framework

Given the lack of code completion data with API documents, training a reranking model becomes difficult. This is primarily due to the high cost of manual annotation and the challenges involved in verifying the correctness of generated code based on the API documents. To address this challenge,

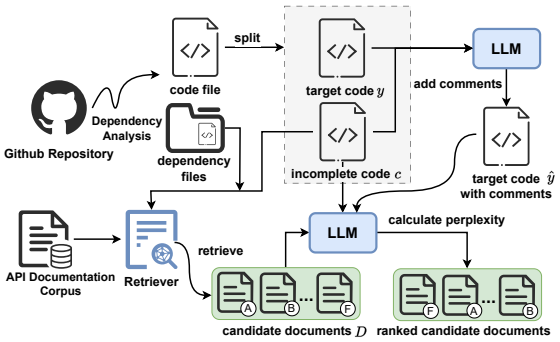

Figure 2: Overview of the Self-supervised Ranking Framework.

we propose a perplexity-driven relevance ranking approach, leveraging the perplexity of LLM-generated code to construct training data. Additionally, shifts in perplexity often arise from code formatting and syntactic variations, introducing noise that distorts the true semantic relevance of API documents to query. To further reduce the influence of code-specific perplexity, we incorporate an approach of perplexity alignment to enhance the information conveyed in the code by adding comments.

Perplexity-driven Relevance Ranking. Evaluating the relevance of documentation to query is a time-consuming manual process, and setting up execution environments can be complex (Wei et al., 2023). Therefore, these obstacles lead to a scarcity of training data, which restricts the development of learning-based ranking methods. To address this challenge, we propose a perplexity-driven relevance ranking method, which assesses the relevance by measuring the perplexity of LLM-generated code.

Specifically, as illustrated in Fig. 2, we construct data based on code repositories from GitHub[2], candidate documents, and an LLM as a perplexity evaluator. For a specific code file that has cross-file dependencies, we randomly select a middle position to split it into incomplete code c and the target code y, ensuring ample context for retrieval and completion. We directly use the incomplete code c as query q and retrieve the top n documents $D = \{d_1, d_2..., d_n\}$ using the retrieval model as candidate documents, which includes both API documentation and all the dependency (*i.e.*, direct and indirect dependency) files from the code file based on dependency tool[3]. The inclusion of dependencies files ensures that query q has relevant docu-

[2] https://github.com

[3] https://github.com/IBM/import-tracker, https://maven.apache.org

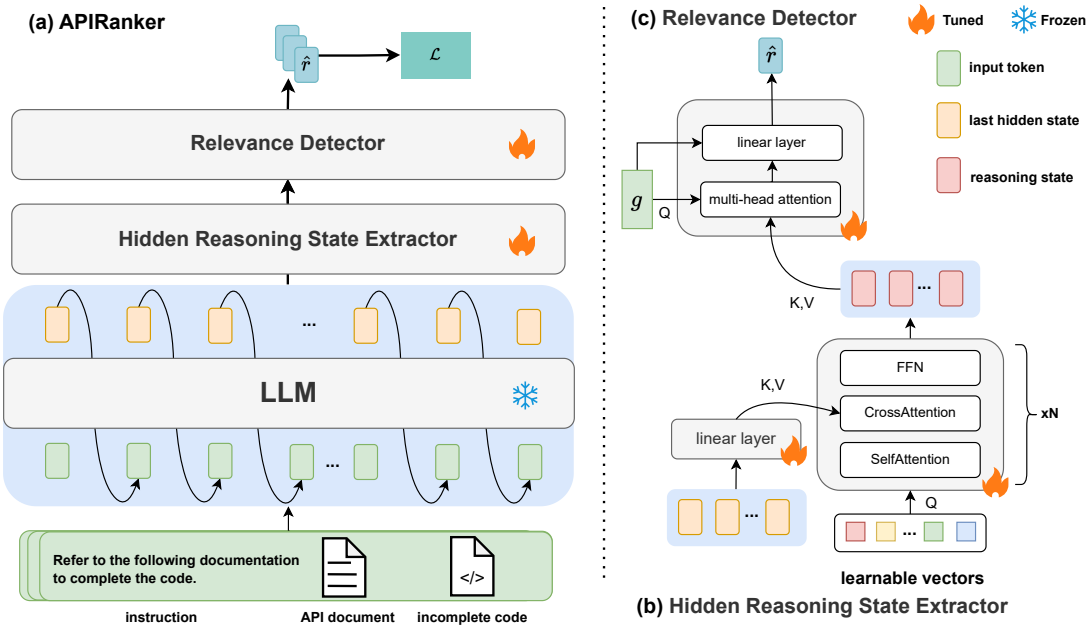

Figure 3: Overview of APIRANKER. (a) is the training process of APIRANKER based on the collection of different API documents D_r and the same incomplete code c as the query. (b) illustrates the structure of the hidden reasoning state extractor. (c) illustrates the structure of the relevance detector.

ments. For each document $d \in D$, the perplexity (PPL) of the target code y is defined as:

$$\text{PPL}(y|d,q) = e^{-\frac{1}{N}\sum_{i=1}^{N}\log P(y_i|d,q,y_{<i})}, \quad (1)$$

where P represents the probability distribution over the LLM's vocabulary, and N is the number of tokens in the target code y. The relevance score r between API document d and query q is then defined by the perplexity of the target code y as:

$$\text{r}(d,q) = \frac{1}{\text{PPL}(y|d,q)}. \quad (2)$$

Using the relevance score r, we can compare the relevance of different documents with the same query. A higher value of r indicates a greater relevance of the document to the query.

Perplexity Alignment via Comments. Code formatting variations (*e.g.*, line breaks, indentation) and code syntactic variations (*e.g.*, bracket placement, variable declaration) can introduce noise into the measurement of relevance. Since these surface-level code variations can introduce significant shifts in perplexity, the perplexity of the target code does not necessarily reflect the true relevance of the API document and the query. Therefore, we incorporate a strategy of perplexity alignment via comments into our method, aiming to bridge the gap between perplexity and semantic relevance.

Specifically, given an incomplete code c and the corresponding target code y, LLM is asked to add comments to each line of code y, as described by the following equations:

$$\hat{y} = \text{LLM}(c, y), \quad (3)$$

where $\hat{y}$ is the generated code with comments. The prompt of perplexity alignment via comments is then constructed as:

> • **Instruction**: Based on the following two consecutive parts of the same code, Part A (the first half) and Part B (the second half), both enclosed within <code> and </code> tags, you should add comments to each line of code in Part B as much as you can.
> • **Part A (the first half)**: c
> • **Part B (the second half)**: y

The score of relevance r based on the target code with comment $\hat{y}$ is defined as:

$$\text{r}(d,q) = \frac{1}{\text{PPL}(\hat{y}|d,q)}. \quad (4)$$

3.3 Reranking Model Architecture

Given that LLMs exhibit strong comprehension abilities (Naveed et al., 2023), we leverage them to reduce the need for learning and explicit NL intents. However, they still face challenges in capturing the implicit cues from code context and explicitly comparing the relevance of different documents to the same query. To tackle this issue, we propose a novel reranking model architecture, APIRANKER, which includes a hidden reasoning state extractor that leverages the reasoning state to capture the

relevance of the API document to implicit cues from the code context. Additionally, a relevance detector is employed to detect the reasoning state and explicitly predict a relevance score between the API document and the query.

Hidden Reasoning State Extractor. Most tokens in a language space are generated solely for fluency, contributing little to the actual reasoning process. Inspired by the previous studies on hidden reasoning state (Hao et al., 2024; Ouyang et al., 2022), we extract the representation of the reasoning state through the last hidden state of the LLM to capture the relevance, rather than relying on tokens from the language space.

Specifically, as illustrated in Fig. 3, given a query q (*i.e.*, the incomplete code c) and an API document d, we prompt the LLM to perform code completion based on d and extract the sequence of hidden states through the decoder layer of LLMs as:

$$h = \text{DecoderLayer}(d, q). \tag{5}$$

where $h = \{h_1, h_2, ..., h_m\}$ represents the sequence of hidden states, m is the number of hidden states. During this process, the LLM's parameters are kept frozen. To align the dimensions between the LLM and the state extractor, we introduce a linear layer as:

$$h' = W_s * h + b_s, \tag{6}$$

where h' is the hidden states after aligning, W_* and b_* denote the trainable parameters in this section. Each layer of the state extractor consists of self-attention, cross-attention, and a feed-forward network (FFN) followed by layer normalization as:

$$p' = \text{SelfAttention}(p, p, p), \tag{7}$$
$$p'' = \text{CrossAttention}(p', h', h'), \tag{8}$$
$$s = \text{LayerNorm}(\text{FFN}(p'') + p''), \tag{9}$$

where p denotes a set of learnable vectors used to capture the reasoning states and s represents the sequence of reasoning states, which serves as input for the next layer of the state extractor. We initialize the extractor with transformer weights pre-trained on code data, whereas the cross-attention layers are randomly initialized.

Relevance Detector. To detect relevance from the reasoning states, we propose a relevance detector that aggregates information of semantic understanding from the reasoning states and predicts relevance scores. Specifically, as illustrated in Fig. 3(c), we use a learnable vector as the query in multi-head attention, with s serving as both the key and value. The relevance score is then predicted by a neural network, as described by the following equations:

$$g' = \text{LayerNorm}(\text{MHA}(g, s, s)), \tag{10}$$
$$g'' = \text{LayerNorm}(g' + \text{FFN}(g')), \tag{11}$$
$$\hat{r} = W_r * g'' + b_r, \tag{12}$$

where g is a learnable vector, representing the relevance of states from the last reasoning states s of state extractor, MHA denotes multi-head attention, and $\hat{r}$ represents the predicted relevance score.

3.4 Training and Inference

Training Objective. APIRANKER is trained on a dataset of comparisons between two documents on the same query. As illustrated in Fig. 3, We use a cross-entropy loss, where comparisons between document pairs act as labels. The difference in rewards represents the log odds of one document being preferred over the other, with this preference determined by the relevance function r (Section 3.2). To speed up comparison training, we construct pairs from a set of K documents selected evenly based on the difference in r values, chosen from the top n candidate documents, and train on all comparisons for each query as a single batch. Formally, the training objective of the reranking model is defined as:

$$\mathcal{L} = -\frac{1}{\binom{K}{2}} \mathbb{E}_{(q,\hat{y},d_w,d_l)\sim\mathcal{D}_r} [\log \sigma(\hat{r}(q, \hat{y}, d_w) - \hat{r}(q, \hat{y}, d_l))], \tag{13}$$

where σ denotes the logistic function, $\hat{r}(q, \hat{y}, d)$ is the scalar output of the reranking model for query q, target code with comments $\hat{y}$ and API document d. d_w is the preferred document out of the pair of d_w and d_l, and D_r is the training data based on score of relevance function r.

Inference. During the inference stage, given a set of candidate documents D retrieved by the retrieval model based on query q (*i.e.*, NL intent and incomplete code), each document $d \in D$ is evaluated by APIRANKER, which produces a new ordering of the candidate documents based on the relevance between document and the query.

4 Experiments

4.1 Experimental Setup

Dataset. To study retrieval-augmented code completion based on API documentation, we

construct a dataset **APIRAC** (**API R**etrieval-**A**ugmented **C**ompletion) for this task. We collect 110,646 API documentations from the dataset CodeRAG-bench (Wang et al., 2024c) as retrieval sources. Additionally, we gather 4,400 large-scale repositories from GitHub, based on the dataset presented in the RLCoder (Wang et al., 2024a), with an equal number of Python and Java repositories, and split them into training and validation sets at a 10:1 ratio. For a specific code file that has cross-file dependencies, we treat all the dependency files of the code file to be completed as the candidate documentation for the code file. Finally, we construct $\langle incomplete\ code, target\ code, API\ documents, relevance\ scores \rangle$ training data using our self-supervised ranking framework. For the test data, we select DS-1000 (Lai et al., 2023) as the automatic code completion dataset, which includes general open-domain coding completion tasks. We use the canonical API documentation in CodeRAG-bench, which provides human-annotated API documentation for queries in DS-1000. Overall statistics of the dataset are given in Table 1. Further details can be found in Appendix A.1.

Baselines. We consider the following retrieval baselines: (1) **Unixcoder**: Unixcoder (Guo et al., 2022) is a unified cross-modal pre-trained model for programming language. (2) **GIST-large**: GIST-large (Solatorio, 2024) is a method that improves text embedding fine-tuning by selectively choosing negative samples. (3) **Arctic-Embed 2.0**: Arctic-Embed 2.0 (Yu et al., 2024) is an open-source text embedding model built for accurate and efficient multilingual retrieval. (4) **NV-Embed-v2**: NV-Embed-v2 (Lee et al., 2024) is a generalist embedding model that ranks No.1 on the retrieval subcategory of the Massive Text Embedding leaderboard (Muennighoff et al., 2022).

Then we consider the following reranking baselines based on LLMs: (1) **Unsupervised Passage Re-ranker (UPR)**: UPR (Sachan et al., 2022) is a pointwise approach based on query generation. (2) **Relevance Generation (RG)**: RG (Liang et al., 2022) is a pointwise approach based on relevance generation. (3) **Pairwise Ranking Prompting-Sorting (PRP-Sorting)**: PRP-Sorting (Qin et al., 2023) is a pairwise method based on the log-likelihood of document generation, and it optimizes time complexity through heap sort. (4) **Pairwise Ranking Prompting-Sliding (PRP-Sliding)**: PRP-Sliding is a variant of PRP, which is based on the sliding window approach.

Sets	Avg. Number query	canonical	**Source**	Avg. Code Lines/Words intent	incomplete	target
train	4,000	-	Github	-	38.2	41.4
val	400	-	Github	-	37.9	40.5
test	513	1.4	Stackflow	84	10.7	5

Table 1: Dataset Statistics.

For automatic code completion, we consider the following code LLM: (1) Starcoder2-7B (Lozhkov et al., 2024), is trained on a vast programming dataset, achieving superior performance on code-related tasks. (2) CodeLlama-Instruct-7B (Roziere et al., 2023), is a fine-tuned version of Code Llama, optimized to follow natural language instructions for code generation. Further details on the above baselines can be found in Appendix A.2.

Implementation Details. For API recommendation, we rerank the top 50 documents retrieved by different retrieval models. In our method, we chose CodeLlama-Instruct-7B as the LLM and Unixcoder as the initial weight of the hidden reasoning state extractor. During decoding, code is generated using greedy decoding. Further details can be found in Appendix A.3.

Evaluation Metrics To evaluate the performance of API recommendation, we report the common evaluation metrics (Zhang et al., 2017; Wei et al., 2023): (1) Recall@k, measures the proportion of correct API documents in the the top-k recommendation results. (2) NDCG@k, evaluates the ranking of correct documents in the top-k recommendation results. (3) MRR@k, represents the reciprocal of the position where the first correct API appears in the top-k recommendation results. (4) MAP, evaluates the overall performance by taking into account the ranking of correct API documents. The value of k is set to 10. We use Recall@k as the primary metric since retrieval-augmented generation primarily relies on key information that appears in the context. To evaluate the performance of code completion based on API recommendation, we adopt Pass@k and Improve@k metrics to measure the execution correctness of programs, the value of k is set to 1, which means the number of generations. Further details can be found in Appendix A.4.

4.2 Experimental Results

API recommendation Evaluation. Table 2 shows the experimental results of our approach and reranking baselines on the candidate documents re-

Retrieval Model	Size	dim	Reranking Method	Recall@10	NDCG@10	MRR@10	MAP
Unixcoder	126M	768	-	2.44	1.35	0.98	1.15
GIST-large	335M	1024	-	15.25	6.88	3.87	4.79
			RG	15.69	10.93	9.10	9.42
			UPR	16.65	9.56	7.01	7.66
			PRP-Sorting	7.00	2.35	0.87	2.42
			PRP-Sliding	12.65	10.15	9.15	10.04
			APIRANKER	**25.50**	**14.52**	**10.33**	**10.83**
Arctic-Embed 2.0	568M	1024	-	18.86	10.83	7.72	8.71
			RG	19.98	13.06	10.30	10.84
			UPR	20.64	12.00	8.38	9.22
			PRP-Sorting	8.45	2.80	1.12	3.11
			PRP-Sliding	12.51	11.52	11.20	12.73
			APIRANKER	**32.36**	**18.39**	**12.48**	**13.09**
NV-Embed-v2	7.9B	4096	-	27.12	13.65	8.76	9.80
			RG	22.53	14.37	**11.30**	**12.29**
			UPR	25.53	14.34	10.27	11.37
			PRP-Sorting	14.98	4.76	1.82	4.25
			PRP-Sliding	23.59	13.33	9.46	10.93
			APIRANKER	**30.17**	**15.49**	9.53	11.08

Table 2: Evaluation results on the APIRAC dataset. All results in the table are reported in percentage (%). The best method is in boldface, and the second best method is underlined for each metric.

trieved by different retrieval models. It is obvious that: (1) Regarding the API recommendation for Recall and overall ranking performance, our approach APIRANKER outperforms all other reranking baselines by a large margin across different retrieval models, demonstrating substantial improvements in both the coverage and ranking quality of relevant documents. For example, APIRANKER achieves a Recall rate of 32.36% on Arctic-Embed 2.0, surpassing the next best method (*i.e.*, UPR) by a significant margin of 11.72%. Similarly, in terms of NDCG@10, APIRANKER outperforms the next best method (*i.e.*, RG) with a value of 18.39%, surpassing it by a significant margin of 5.33%, which clearly indicates its superior ranking capability. (2) Our approach APIRANKER demonstrates consistent improvements across all evaluation metrics post-reranking, irrespective of the underlying retrieval model. Even in the case of the strong retrieval baseline (*e.g.*, NV-Embed-v2), where other methods all show degraded performance compared to the original retrieval results, APIRANKER still demonstrates stable improvements, outperforming retrieval baseline across all metrics, which highlights the effectiveness of our method in enhancing ranking quality and coverage in diverse retrieval scenarios. **Overall, our method shows substantial and stable improvements in reranking different candidate retrieved documents compared to other models, validating the effectiveness of our method for API recommendation.**

Model	Complexity	Parameters	Method		
			Train	Inference	principle
RG	O(N)	6.7B	-	Pointwise	Perplexity
UPR	O(N)	6.7B	-	Pointwise	Perplexity
PRP-Sorting	O(logN*N)	6.7B	-	Pairwise	Perplexity
PRP-Sliding	O(K*N)	6.7B	-	Pairwise	Perplexity
APIRANKER	O(N)	6.7B+160M	Pairwise	Pointwise	Semantics

Table 3: Comparison of the reranking method. N is the number of documents retrieved for reranking. K is the number of documents to be returned after reranking.

Method Analysis. As illustrated in Table 3, APIRanker demonstrates several key advantages over other reranking methods: (1) Its linear complexity $O(N)$ ensures scalability, making it suitable for large-scale applications. (2) It captures more complex dependencies and achieves higher accuracy with just 160M additional parameters. The pairwise training method improves its ability to compare documents in terms of relevance, while the pointwise inference ensures efficient processing. (3) By leveraging semantic understanding, APIRanker excels in tasks that require a deep comprehension of the documents in comparison to models that rely solely on perplexity.

Retrieval-augmented Completion Evaluation. Using a retrieval-augmented generation approach, we evaluated the performance of different reranking models in improving code completion performance on Arctic-Embed 2.0. The experimental results showed that: (1) As illustrated in Fig. 4(a), APIRANKER consistently outperforms the no-retrieval baseline and leads to stable improvements in passing across both code LLMs, whereas other

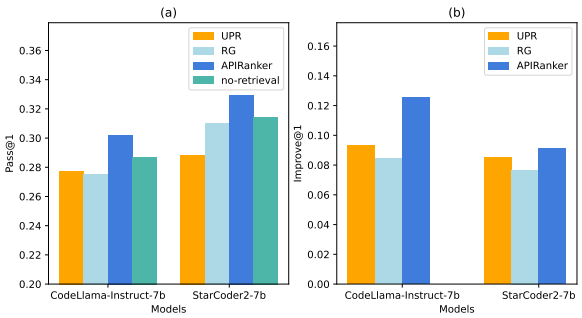

Figure 4: Effect of API Recommendations on Code Completion: Pass@1 and Improve@1 Evaluation for CodeLlama-Instruct-7B and StarCoder2-7B. The top 10 documents were used as context, with the number of documents incrementing from 1 to 10 in each trial. The best result from 10 runs was reported.

Model	Recall@10	
	GIST-large	Arctic-Embed
APIRanker	25.50	32.36
w/o Perplexity Alignment	19.90	31.57
w/o Reasoning State Extractor	24.66	27.91
w/o Relevance Detector	0.49	0.88

Table 4: Ablation study.

reranking models (*i.e.*, UPR, RG) cause performance degradation. This decline can be attributed to interference from some of the recommended documents, which disrupts the code LLM's ability to generate code that was previously correct. In contrast, APIRanker offers stable and reliable improvements in retrieval-augmented generation, demonstrating practical usability in real-world applications. (2) As illustrated in Fig. 4(b), we analyze the proportion of cases in which the code LLM generates the correct output with the recommended API documents, compared to when it initially failed without the recommended API documents. APIRanker consistently outperforms other reranking models, achieving higher improvements in both code LLMs, highlighting its superior performance in scenarios where the code LLM's capabilities fall short and external API knowledge is needed.

Ablation Study. As illustrated in Table 4, we conduct an ablation study to assess the contribution of different techniques by removing key components (*i.e.*, Perplexity Alignment via Comments, Hidden Reasoning State Extractor and Relevance Detector) of our approach separately. The experimental results show that: (1) No matter which component we drop, it hurts the overall performance of our model, which signals the importance and effectiveness of all three components. (2) The recall rate shows a significant drop in reranking performance on the candidate documents retrieved by

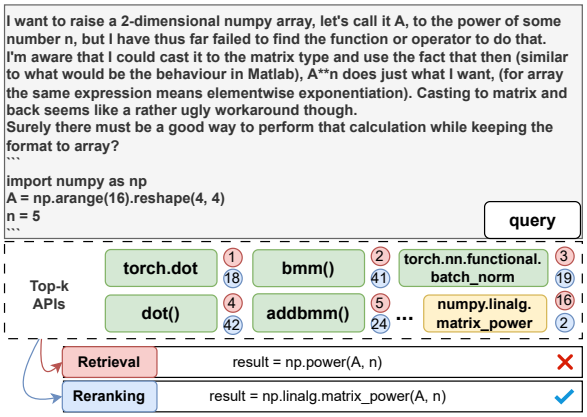

Figure 5: The example of code completion based on API recommendation.

GIST-large and Arctic-Embed 2.0 when the Hidden Reasoning State Extractor and Relevance Detector are removed separately. Notably, the removal of the Relevance Detector causes an enormous decrease, which makes the model fail to work properly. This justifies the importance and necessity of these two components in our reranking model architecture.

Case Study. As illustrated in Fig. 5, we present an example of generation using CodeLlama, based on API documents retrieved (*e.g.*, colored in red) by Arctic-Embed 2.0 and reranked by our model. The retrieved APIs can't properly handle matrix exponentiation for Numpy, causing the LLM to fail in performing the required matrix operations. APIRanker reranks the retrieved APIs (*e.g.*, colored in blue), successfully moving the correct API (*e.g.*, colored in yellow) to the top, thus providing the correct solution. This highlights that, in code completion tasks where higher precision and specific requirements are crucial, APIRanker offers more accurate and effective API recommendations by optimizing the ranking.

5 Conclusions.

This research aims to rerank retrieved API documents to enhance automatic code completion, considering both NL intent and incomplete code. To perform this task, we propose an approach APIRanker that utilizes a self-learning ranking framework to automatically construct data for ranking. Then we propose a novel reranking model architecture to predict the relevance score between the API documents and the query, based on the LLM's reasoning capabilities. The experimental results show the effectiveness of our approach for this task. We hope our study lays the foundations for this research and provides valuable insights.

6 Limitations.

Several limitations are concerned with our work. Firstly, due to the limited availability of code completion test sets that support code evaluation in other languages, and the difficulty in constructing queries that simultaneously include both intent and incomplete code, our test is based on Python, one of the most popular programming languages used by developers. However, during the training of our method, we used data from two programming languages Java and Python, and we believe that our approach can easily adapt to other programming languages. Secondly, our approach does not explicitly create intent but rather leverages the language comprehension ability of LLMs to reduce the need for learning natural language intent. Exploring how to automatically generate high-quality intent from code is an interesting research topic for our future work.

References

Yujia Chen, Cuiyun Gao, Xiaoxue Ren, Yun Peng, Xin Xia, and Michael R Lyu. 2023. Api usage recommendation via multi-view heterogeneous graph representation learning. *IEEE Transactions on Software Engineering*, 49(5):3289–3304.

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

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 631–642.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, et al. 2023. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents. *arXiv preprint arXiv:2310.19923*.

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

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. 2024. Training large language models to reason in a continuous latent space. *arXiv preprint arXiv:2412.06769*.

Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. 2018. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 293–304.

Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. 2024. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces*, page 103917.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. 2024. Nv-embed: Improved techniques for training llms as generalist embedding models. *arXiv preprint arXiv:2405.17428*.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*.

Chunyang Ling, Yanzhen Zou, and Bing Xie. 2021. Graph neural network based collaborative filtering for api usage recommendation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 36–47. IEEE.

Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. Codegen4libs: A two-stage approach for library-oriented code generation. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 434–445. IEEE.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seungwon Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722*.

Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024. Compositional api recommendation for library-oriented code generation. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 87–98.

Marcellino Marcellino, Davin William Pratama, Steven Santoso Suntiarko, and Kristien Margi. 2021. Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. In *2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI)*, volume 1, pages 154–160. IEEE.

Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2022. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*.

Noor Nashid, Taha Shabani, Parsa Alian, and Ali Mesbah. 2024. Contextual api completion for unseen repositories using llms. *arXiv preprint arXiv:2405.04600*.

Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2023. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*.

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

Arnold Overwijk, Chenyan Xiong, Xiao Liu, Cameron VandenBerg, and Jamie Callan. 2022. Clueweb22: 10 billion web documents with visual and semantic information. *arXiv preprint arXiv:2211.15848*.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Yun Peng, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2022. Revisiting, benchmarking and exploring api recommendation: How far are we? *IEEE Transactions on Software Engineering*, 49(4):1876–1897.

Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Le Yan, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, et al. 2023. Large language models are effective text rankers with pairwise ranking prompting. *arXiv preprint arXiv:2306.17563*.

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*.

Devendra Singh Sachan, Mike Lewis, Mandar Joshi, Armen Aghajanyan, Wen-tau Yih, Joelle Pineau, and Luke Zettlemoyer. 2022. Improving passage retrieval with zero-shot question generation. *arXiv preprint arXiv:2204.07496*.

Aivin V Solatorio. 2024. Gistembed: Guided in-sample selection of training negatives for text embedding fine-tuning. *arXiv preprint arXiv:2402.16829*.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024a. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487*.

Yong Wang, Yingtao Fang, Cuiyun Gao, and Linjun Chen. 2024b. Api recommendation for novice programmers: Build a bridge of query-task knowledge gap. *IEEE Transactions on Reliability*.

Zora Z. Wang, Akari Asai, Xinyan V. Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024c. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*.

Moshi Wei, Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Lin Shi, Song Wang, and Zhen Ming Jiang. 2023. A survey on query-based api recommendation. *arXiv preprint arXiv:2312.10623*.

Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. Clear: contrastive learning for api recommendation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 376–387.

Di Wu, Xiao-Yuan Jing, Hongyu Zhang, Yang Feng, Haowen Chen, Yuming Zhou, and Baowen Xu. 2023. Retrieving api knowledge from tutorials and stack overflow based on natural language queries. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–36.

Puxuan Yu, Luke Merrick, Gaurav Nuti, and Daniel Campos. 2024. Arctic-embed 2.0: Multilingual retrieval without compromise. *arXiv preprint arXiv:2412.04506*.

Jingxuan Zhang, He Jiang, Zhilei Ren, and Xin Chen. 2017. Recommending apis for api related questions in stack overflow. *IEEE Access*, 6:6205–6219.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023. Toolcoder: Teach code generation models to use api search tools. *arXiv preprint arXiv:2305.04032*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. *arXiv preprint arXiv:2207.05987*.

A Appendix

A.1 Dataset Construction Details

We collect 110,646 API documentations from the dataset CodeRAG-bench (Wang et al., 2024c) as retrieval sources. These documents come from two main sources: official Python library documentation provided by devdocs.io[4] and content obtained from ClueWeb22 (Overwijk et al., 2022), a large-scale web corpus, covering a wide range of topics, from basic programming techniques to advanced library usage. Each page in ClueWeb22 includes code snippets and textual explanations. To support efficient vector queries based on cosine similarity, we create vector search libraries using Milvus[5], a high-performance vector database designed for scalability and providing fast, scalable similarity search and retrieval.

In the training and evaluation data, for a specific code file that has cross-file dependencies, we treat all the dependency files (i.e., both direct and indirect dependencies) of the code file as the candidate documentation for the code file. For the test data, we select DS-1000 (Lai et al., 2023) as the query (*i.e.*, NL intent and incomplete code) and code completion dataset, which includes general open-domain coding completion tasks (*e.g.*, `Matplotlib`, `Numpy`, `Pandas`, `Sklearn`, `Tensorflow`).

A.2 Baselines Setup detail

Retrieval Baselines. We consider the following retrieval baselines, which are dense retrievers that encode both the query and code documentation into vector spaces for retrieving semantically relevant documentation based on vector similarity: (1) **Unixcoder**: Unixcoder (Guo et al., 2022) is a unified cross-modal pre-trained model for programming language. (2) **GIST-large**: GIST-large (Solatorio, 2024) is a method that improves text embedding fine-tuning by selectively choosing negative samples. (3) **Arctic-Embed 2.0**: Arctic-Embed 2.0 (Yu et al., 2024) is an open-source text embedding model built for accurate and efficient multilingual retrieval. (4) **NV-Embed-v2**: NV-Embed-v2 (Lee et al., 2024) is a generalist embedding model that ranks No.1 on the retrieval sub-category of the Massive Text Embedding (MTEB) leaderboard (Muennighoff et al., 2022). GIST-large and Arctic-Embed 2.0 are also ranked highly on the MTEB leaderboard.

Since the performance of code retrieval models (*e.g.*, Unixcoder, CodeBert (Feng et al., 2020), jina-base-v2-code (Günther et al., 2023)) is not ideal (with poor retrieval performance), we do not conduct reranking experiments on it. Additionally, CodeBert and jina-base-v2-code are unable to recall any relevant API documents in the top 50, we do not report retrieval results for these models. Considering the context limitations of retrieval and code generation, as well as the excessive length of some API documentation, the retrieval model's maximum token encoding length is uniformly set to 512.

Reranking Baselines. We consider the following reranking baselines, which are based on LLMs:

(1) **Unsupervised Passage Re-ranker (UPR)**: UPR (Sachan et al., 2022) is a pointwise approach based on query generation. The prompt template for UPR is shown in Fig. 6. In this approach, the relevance score of an API document d to the query q is measured by the probability of generating the query.

- **Instruction**: Please write a question based on this passage.
- **Passage**: d
- **Question**: q

Figure 6: The prompt template for UPR. d is the API document, q is the query.

(2) **Relevance Generation (RG)**: RG (Liang et al., 2022) is a pointwise approach based on relevance generation. The prompt template for RG is shown in Fig. 7. In this approach, the relevance of an API document d to the query q is defined as:

$$s_i = \begin{cases} 1 + p(\text{Yes}), & \text{if output Yes} \\ 1 - p(\text{No}), & \text{if output No} \end{cases} \quad (14)$$

where $p(\text{Yes})$ and $p(\text{No})$ denote the probabilities of LLMs generating the tokens of "Yes" or "No" respectively.

- **Instruction**: Does the passage answer the query?
- **Passage**: d
- **Query**: q

Figure 7: The prompt template for UPR. d is the API document, q is the query.

(3) **Pairwise Ranking Prompting- Sorting (PRP-Sorting)**: PRP-Sorting (Qin et al., 2023) is

[4] https://devdocs.io
[5] https://github.com/milvus-io/milvus

a pairwise method based on the log-likelihood of document generation, and it optimizes time complexity through heap sort algorithm (Marcellino et al., 2021). The prompt template for PRP-Sorting is shown in Fig. 8. In this approach, to compare two API documents d_A and d_B, the one that is more relevant to the query q is determined based on which has a higher probability of generating "Passage A" or "Passage B".

• **Instruction**: Given a query "q", which of the following two passages is more relevant to the query?
• **Passage A**: d_A
• **Passage B**: d_B

Figure 8: The prompt template for PRP-Sorting and PRP-Sliding. d is the API document, q is the query.

(4) **Pairwise Ranking Prompting-Sliding (PRP-Sliding)**: PRP-Sliding is a variant of PRP, which is based on the sliding window approach. The prompt template and comparison function for PRP-Sliding are the same as those for PRP-Sorting.

In order to comparing the performence of different reranking models, we uniformly use CodeLlama-Instrcut-7B as the base LLMs. The maximum token length of an API document is set to 512.

Code Completion Baselines. For automatic code completion, we consider the following code LLMs: (1) Starcoder2-7B (Lozhkov et al., 2024), which is trained on a vast programming dataset and achieves superior performance on code-related tasks. (2) CodeLlama-Instruct-7B (Roziere et al., 2023), which is a fine-tuned version of Code Llama, optimized to follow natural language instructions for code generation.

A.3 Implementation Details

In our approach, we chose CodeLlama-Instruct-7B as the perplexity evaluator in the self-supervised learning ranking framework and as the base LLM of the reranking model. Additionally, UnixCoder is chosen as the retriever in the self-supervised learning ranking framework and as the initial weight of the hidden reasoning state extractor. All experiments were conducted on two A800 GPUs.

In our self-supervised learning ranking framework, we set the total length of the incomplete code and the target code to be no more than 1024 tokens, ensuring that the ratio of 0.4 to 0.5 of the total length is considered as the incomplete code. The prompt template for the perplexity evaluator is shown in 9.

In the design of the reranking model, we set the number of learnable vectors in the hidden reasoning state extractor to 32. We employed the AdamW optimizer with a learning rate of 1e-4. The learning rate schedule was managed using the WarmupCosineLR scheduler, where the learning rate linearly warms up for the first 75 steps and then follows a cosine decay towards a minimum ratio of 0.0001 over a total of 750 steps. The batch size was set to 384, and the number of gradient accumulation steps was 4. The input length was capped at a maximum of 1152 tokens. We constructed pairs from a set of 4 documents, selected evenly based on the difference in values from the perplexity evaluator, chosen from the top 20 candidate documents retrieved by the retriever. The prompt template for training is shown in Fig. 9. During the inference stage, we reranked the top 50 documents retrieved by different retrieval models. The input length was capped at a maximum of 1600 tokens. The prompt template for inference was the same as for training.

• **Instruction**: Refer to the following documentation (between "— Documentation —" and "— End Documentation —") to complete the code.
• **APIs document**: d
• **query**: q

Figure 9: The prompt template for APIRANKER. d is the API document, q is the query.

For retrieval-augmented code completion, we use top-k API documents as a context for automatic code completion, keeping only the first 512 tokens in each document. The prompt template of retrieval-augmented code completion is shown in Fig. 10. During decoding, code is generated using greedy decoding. The length of the output to a maximum of 2048 tokens.

• **Instruction**: Refer to the following documentation (between "— Documentation —" and "— End Documentation —") to complete the code. Based on the following problem description and existing code, please write the code to achieve the desired output. Place the executable code between <code> and </code> tags, without any other non-executable things.
• **the top-k APIs documents**: $d1, ..., d_k$
• **query**: q

Figure 10: The prompt template for PRP-Sorting and PRP-Sliding. d_i is the i-th API document, q is the query.

A.4 Evaluation Metrics

To evaluate the performance of API recommendation, we report the common evaluation metrics (Zhang et al., 2017; Wei et al., 2023): (1) Recall@k, measures the proportion of correct API documents in the the top-k recommendation results. It is defined as follows:

$$\text{Recall@k} = \frac{\text{R}}{\text{N}}, \tag{15}$$

where N is the total number of relevant documents, and R is the number of relevant documents in top-k recommended results. (2) NDCG@k, evaluates the ranking of correct documents in the top-k recommendation results. As a normalized Discounted Cumulative Gaine, NDCG is calculated by dividing by a special ideal DCG, where all relevant documents are ranked higher than irrelevant ones. It is defined as:

$$\text{NDCG@k} = \frac{DCG@k}{ideal\ DCG@k}, \tag{16}$$

$$\text{DCG@k} = \sum_{i=1}^{k} \frac{2^{rel(i)} - 1}{\log_2(i+1)}, \tag{17}$$

where i represents the rank. $rel(i)$ is a binary function to check whether the API in rank i is correct or not. If the API at rank i is a correct API, then the value $rel(i)$ is 1; otherwise, the value is 0. (3) MRR@k, represents the reciprocal of the position where the first correct API appears in the top-k recommendation results. It is defined as:

$$MRR@k = \frac{1}{|Q|} \sum_{j=1}^{Q} \frac{1}{k\_Rank_i}, \tag{18}$$

where $|Q|$ is the number of queries Q, and k_Rank_i means the rank position of the first correct answer in the top k recommended list for the i-th query. (4) Mean Average Precision (MAP), evaluates the overall performance by taking into account the ranking of correct API documents. It is defined as:

$$\text{MAP} = \frac{1}{|Q|} \sum_{j=1}^{Q} \frac{\sum_{i=1}^{n} (P(i) \times \text{rel}(i))}{\#\text{correct answers}}, \tag{19}$$

$$P(i) = \frac{\#\text{correct answers in top i}}{i}, \tag{20}$$

where $p(i)$ is the precision at a given cut-off rank i. The value of k is set to 10, and n is set to 50. We use Recall@k as the primary metric since retrieval-augmented generation primarily relies on key information that appears in the context.

To evaluate the performance of code completion based on API recommendation, we adopt Pass@k and Improve@k metrics to measure the execution correctness of programs: (1) Pass@k, is an evaluation metric that has been widely used in previous work (Jiang et al., 2024), computing the fraction of problems having at least one correct prediction within k samples. It is defined as:

$$\text{pass@k} := \mathbb{E}_{\text{task}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right], \tag{21}$$

where n is the total number of sampled candidate code solutions, k is the number of randomly selected code solutions from these candidates for each programming problem, with $n \geq k$, and c is the count of correct samples within the k selected. (2) Improve@k, is the proportion of cases in which the code LLM generates the correct output with the recommended API documentation, compared to when it initially failed without the recommended API documentation. It is defined as:

$$\text{Improve@k} = \frac{\sum_{i=1}^{m} \text{correct}(i)}{\#\text{failures in } k \text{ samples}}, \tag{22}$$

where m is the number of problems that initially failed to generate the code in the k samples, and correct(i) is 1 if the i-th problem passes in the k samples, and 0 if it fails. The value of k is set to 1 in our experiment. Given the differences in the capabilities of code LLMs, there are instances where a model, initially capable of generating correct outputs, may fail when code completion is based on API documents. Therefore, we use Improve@k to explore the potential for improvement.