Context-Aware API Reranking for Custom Developer Requirements in Code Completion

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have made significant advancements in automatic code completion. Given the rapid pace of API updates and the restricted proprietary documentation of enterprise environments, selecting suitable APIs for automatic code completion from vast third-party and private libraries plays a critical role. However, existing solutions struggle to accurately recommend APIs when developers present customized requirements within an incomplete code context. To bridge this gap, we propose a context-aware approach API-RANKER to rerank candidate API documents based on both the explicit developer intent and implicit cues in the incomplete code context. To generate training data for this task, 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 estimate relevance. The experimental results show the effectiveness of our approach in recommending more accurate APIs. The code and the dataset is available[1].

1 Introduction

The introduction of LLMs has led to advancements in automatic code completion (Husein et al., 2024). Given the rapid pace of API updates and the restricted proprietary documentation of enterprise environments, relying on large-scale model training to acquire and keep up with such knowledge is not feasible. Therefore, a crucial aspect of code completion is selecting suitable API docs from massive amounts of third-party libraries and private libraries (Wang et al., 2024b). However, existing approaches struggle to accurately recommend APIs when developers present customized requirements within an incomplete code context: **1)** recommending API docs based on user queries (Wu et al., 2023), which neglects the code context, and **2)** recommending API docs based on the preceding user-written code context (Peng et al., 2022), which fails to capture the developer's underlying intention behind the API usage.

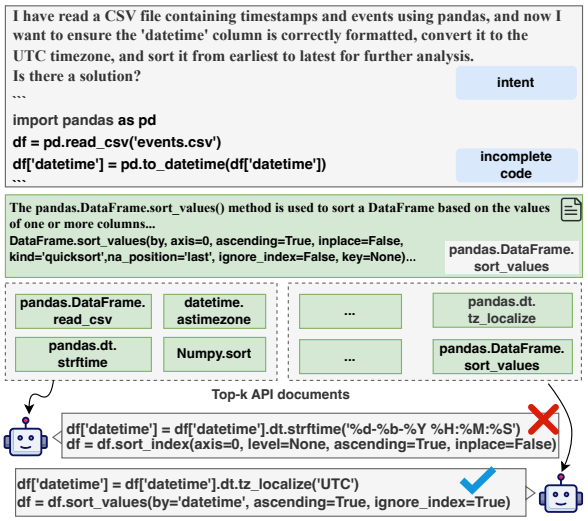

Figure 1: Example of retrieval-augmented code completion with different retrieved top-k API documents.

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 'datetime' column to UTC timezone and sort it from earliest to latest for further analysis*". However, without knowing Alice's intention or 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 recommended results, they will not be used for code completion, causing the auto-completed code to misalign with her intended behavior. Considering both the developer's intent and the incomplete code context, the target APIs (*e.g.*, `df.dt.tz_localize`, `pandas.DataFrame.sort_values`) could be suc-

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

cessfully recommended; the LLM is likely to complete her code by smoothly integrating with the recommended APIs, effectively solving her problem. Therefore, there is need a more comprehensive model that considers both the explicit intent conveyed by user queries and the implicit cues embedded in the code context.

However, recommending APIs for custom developer requirements is a significant challenge: (i) **Lack of code completion dataset annotated with relevant APIs hinders learning-based approaches**. Curating training datasets for API recommendation requires manually evaluating the relevance of API documentation to the developer's requirements. This process depends on domain expertise, additionally, developers rarely express intent in their code, which makes large-scale data collection impractical. (ii) **The relevance of APIs to the developer's requirements is hard to learn and capture**. API documentation varies significantly in format, writing style, and level of detail across different third party libraries, making it difficult to directly link to developers' requirements. Furthermore, developers' requirements are expressed through implicit intent and subtle semantics in the incomplete code, recommending APIs that align with both aspects effectively is challenging.

To tackle the above challenges, we propose a novel framework named APIRANKER, which is designed to rerank candidate APIs by jointly considering the developer's intent and the incomplete code context. To address the challenge of lacking training data, we propose a self-supervised ranking framework to automatically construct ranking data. Specifically, we leverage a perplexity-driven relevance ranking approach, which uses LLM as an evaluator to automatically estimate 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 using code comments as semantic anchors. To better learn and capture the relevance of API documentation to the developer's requirements, we design a novel reranking model consisting of two key components. First, a hidden reasoning state extractor is designed to capture the relevance of the API documentation to implicit cues within code context, by extracting reasoning states from LLMs during inference. Second, a relevance estimator uses the reasoning states to explicitly predict a relevance score, learning to differentiate the impact of various API docs on code completion effectiveness.

In summary, our paper makes the following contributions: (1) **A self-supervised ranking framework for ranking data construction**. We introduce a novel self-supervised approach that generates ranking data automatically, eliminating the need for manual annotation. (2) **An novel API reranking model for better code completion.** We design a novel reranking model that leverages the semantic understanding of LLMs to rerank APIs based on the relevance of API docs to a developer's requirements. The experimental results demonstrate the effectiveness of APIRANKER over a set of baselines. 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). RepoCoder (Zhang et al., 2023a) and APIRec-CST (Chen et al., 2021) could use code comments as the textual intent input, while our work focuses on explicit developers' requirements. 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 API docs. 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 performance in this task.

Retrieval-augmented Code Generation (RAG). 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). 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. For instance, CodeGen4Libs (Liu et al., 2023) recommends class-level API docs through a two-stage process of retrieval and fine-tuning. DocPrompting (Zhou et al., 2022) enables continuous updates to the documentation pool, en-

suring that the most current code libraries are used for generation. ToolCoder (Zhang et al., 2023b) integrates API search tools and uses automated data annotation to teach the model how to use tool usage information, thereby enhancing code generation. Since the core of RAG lies in the accuracy of the retrieved API documentation, our work notably focuses on optimizing the API recommendation process rather than building an end-to-end system, in order to avoid evaluation uncertainty.

3 Methodology

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 recommend correct API documents D for automatic code completion from a large corpus of candidates.

3.2 Self-supervised Ranking Framework

The absence of a code completion dataset annotated with relevant APIs makes training models a challenging task. This is primarily due to the high cost of manual annotation and the difficulties of verifying the correctness of generated code based on the APIs. To address this challenge, we propose a **perplexity-driven relevance ranking** approach, leveraging the perplexity of LLM-generated code to construct training data. However, perplexity can be influenced by factors such as code formatting and syntactic variations, introducing noise that distorts accurate relevance estimation. To mitigate this issue, we propose a **perplexity alignment** strategy that enriches the code context with comments, reducing the perplexity shifts and aligning code semantics.

Perplexity-driven Relevance Ranking. Evaluating the relevance of API documentation to preceding code context is a time-consuming process, requiring complex execution environments (Wei et al., 2023). These obstacles lead to a scarcity of training data, further limiting the learning-based approach's progress. 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 from GitHub repositories[2], API documents, and an LLM as a perplexity evaluator. For a specific

[2] https://github.com

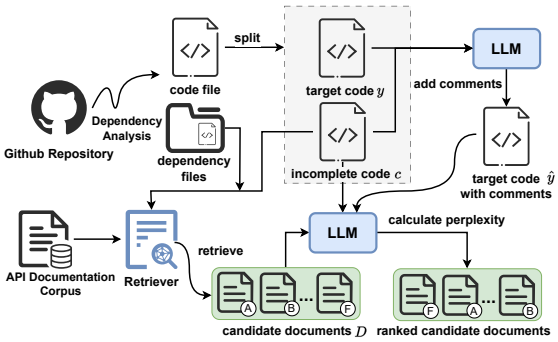

Figure 2: Overview of the Self-supervised Ranking Framework.

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. The detailed splitting strategy is presented in Appendix A.1. We use the incomplete code c as query q and retrieve the top n API documents $D = \{d_1, d_2..., d_n\}$ as candidates via the retrieval model. We use a dependency analysis tool[3] to identify direct and indirect dependency files of the code file, then each dependency file is regarded as a potential API document for user-defined functions, as it contains both the detailed implementations and definitions of those functions. Including these dependency files ensures that query q is paired with relevant API documents.

For each API 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. $y_{<i}$ denotes the first $i-1$ tokens. 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 API docs for the same query, since lower perplexity indicates that LLM has less difficulty in correctly completing the code with the API docs. A higher value of r indicates a higher relevance of the document to the query.

Perplexity Alignment via Anchor Comments. The perplexity score used for relevance estimation is intended to better reflect code semantics, while

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

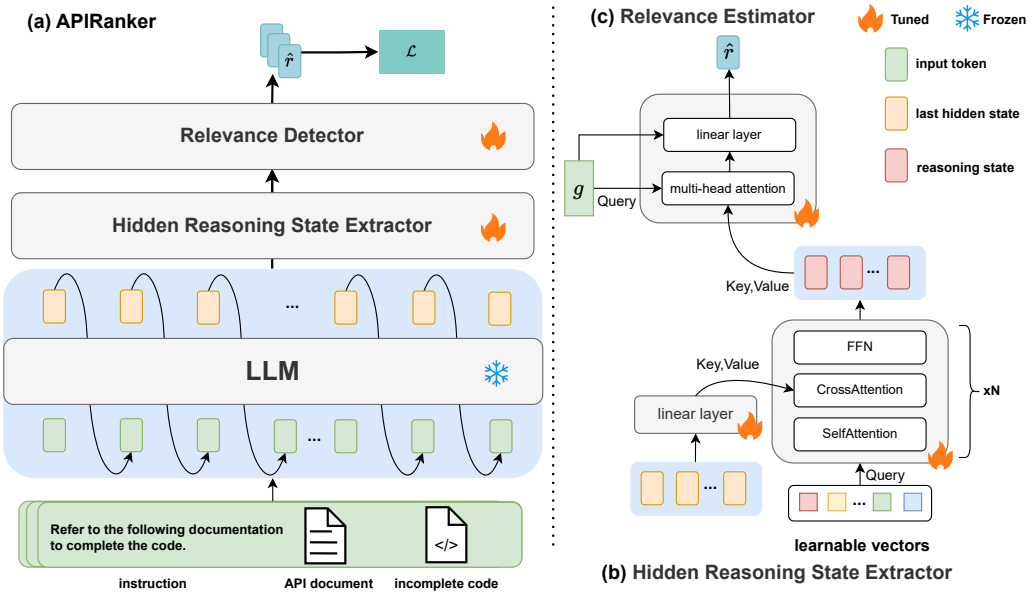

Figure 3: Overview of APIRANKER. (a) is the training process of APIRANKER based on the collection of different API documents D_r and the same incomplete code c as the query. (b) illustrates the structure of the hidden reasoning state extractor. (c) illustrates the structure of the relevance estimator.

minimizing the influence of non-semantic factors, such as formatting variations (*e.g.*, line breaks, indentation) and code syntactic variations (*e.g.*, bracket placement, variable declaration). These variations can cause significant shifts in perplexity, making it unreliable to reflect the true relevance of the API document and the code context. To address this issue, we incorporate a strategy of perplexity alignment via adding anchor comments into the target code. Anchor comments describe observable behavior (*e.g.*, operations and API usage), provide semantic information that enhances the logical representation during perplexity calculation, thereby reducing the sensitivity to perplexity shifts and improving the alignment between perplexity and logic relevance. Further theoretical analysis is provided in Appendix A.2.

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:

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

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

Based on the target code with comments $\hat{y}$, the relevance score r is calculated by Equation 2, offering a more accurate measure of the relevance between the API document and the query.

- **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

3.3 API Reranking Model Architecture

Inspired by LLMs' strong abilities (Naveed et al., 2023) in natural language comprehension, we incorporate LLM into our model architecture to capture user intents. This design allows us to bypass the need for modeling or learning from natural language intent. However, LLMs still struggle to capture implicit cues in code context and differentiate between different API documents for the same query. To tackle this issue, we propose a novel reranking model architecture, APIRANKER, for selecting the most suitable APIs from a set of candidates. APIRANKER 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 estimator is employed to detect the reasoning state and explicitly predict a relevance score between the API document and the query.

Hidden Reasoning State Extractor. A large number of tokens in natural language generation are produced solely for fluency, contributing little to the underlying 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 from the last hidden state of the LLM, allowing us to capture semantic relevance rather than relying on general tokens for linguistic coherence.

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:

$$h = \text{DecoderLayer}(d, q), \tag{4}$$

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:

$$h' = W_s * h + b_s, \tag{5}$$

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:

$$p' = \text{SelfAttention}(p, p, p), \tag{6}$$
$$p'' = \text{CrossAttention}(p', h', h'), \tag{7}$$
$$s = \text{LayerNorm}(\text{FFN}(p'') + p''), \tag{8}$$

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

$$g' = \text{LayerNorm}(\text{MHA}(g, s, s)), \tag{9}$$
$$g'' = \text{LayerNorm}(g' + \text{FFN}(g')), \tag{10}$$
$$\hat{r} = W_r * g'' + b_r, \tag{11}$$

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 consisting of pairwise comparisons between different API candidates for the same query. As illustrated in Fig. 3, we use a cross-entropy loss, where each training sample is labeled by performing comparisons between API document pairs. 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:

$$\mathcal{L} = -\frac{1}{\binom{K}{2}} \mathbb{E}_{(q,\hat{y},d_w,d_l)\sim\mathcal{D}_r} \left[\log \sigma(\hat{r}(q,\hat{y},d_w) - \hat{r}(q,\hat{y},d_l))\right], \tag{12}$$

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 docs 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 docs based on the relevance between document and the query. The first k docs ($k < n$) are reference docs for completion.

4 Experiments

4.1 Experimental Setup

Dataset. To study API recommendation task, we construct a dataset **APIRAC** (**API R**etrieval-**A**ugmented **C**ompletion) for this task. We collect 110,646 API docs 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

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.

in the RLCoder (Wang et al., 2024a), with an equal number of Python and Java repositories, and split them into training and validation sets with a 10:1 ratio. For a given code file, we add its associated dependency files to the retrieval sources as API candidates. Finally, we construct ⟨*incomplete code, target code, API docs, relevance scores*⟩ training data using our self-supervised ranking framework. For the test data, we select DS-1000 (Lai et al., 2023), which includes general open-domain code completion tasks. Each sample contains an NL query and incomplete code with executable environments. We use the human-annotated DS-1000 API documentation from CodeRAG-Bench for API recommendation, which specifies the ground-truth selected from a pool of 110,646 API documents. There is no overlap between the ground truth documents from test dataset and the APIs used in training. Overall statistics of the dataset are given in Table 1. Further details can be found in Appendix A.3.

Baselines. We consider the following mainstream API retrieval baselines from the Massive Text Embedding leaderboard (Muennighoff et al., 2022): Unixcoder (Guo et al., 2022), GIST-large (Solatorio, 2024), Arctic-Embed 2.0 (Yu et al., 2024), NV-Embed-v2 (Lee et al., 2024). We compare our method against four mainstream **LLM-based reranking methods**: Unsupervised Passage Re-ranker (UPR) (Sachan et al., 2022), Relevance Generation (RG) (Liang et al., 2022), Pairwise Ranking Prompting- Sorting (PRP-Sorting) (Qin et al., 2023) and Pairwise Ranking Prompting-Sliding (PRP-Sliding). Then we also consider the following mainstream **vector-based reranking model** in API recommendation: bge-reranker-large (Xiao et al., 2023) and jina-reranker-v2 (Wang et al., 2025). In addition, we adopt three widely adopted API recommendations approaches in software engineering, BIKER (Huang et al., 2018) and GAPI (Ling et al., 2021), which consider either user intent or the code context, respectively. RepoCoder (Zhang et al., 2023a) is a re-retrieval method based on the results of iterative completion. Further details on the above baselines can be found in Appendix A.4.

Implementation Details. For API recommendation, we rerank the top 50 documents retrieved by the retrieval model, then choose the top 10 documents after reranking. In our approach, we use Unixcoder as the initial weight for the hidden reasoning state extractor. Further details can be found in Appendix A.5.

Evaluation Metrics. To evaluate the performance of API recommendation, we report the common evaluation metrics (Zhang et al., 2017; Wei et al., 2023): Recall@k, NDCG@k, MRR@k, and MAP, with k 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 Improve@k metrics to measure the execution correctness of programs, with k set to 1. Further details can be found in Appendix A.6.

4.2 Experimental Results

API Recommendation Evaluation. We conducted API recommendation experiments using codellama-7b-instruct (Roziere et al., 2023) and deepseek-coder-6.7b-instruct (Guo et al., 2024). The results for deepseek-coder-6.7b-instruct are presented in Table 6 in Appendix A.7, while the experimental results for codellama-7b-instruct are shown in Table 2. Notably, NV-Embed-v2 (see Table 5 in the appendix) as a large-scale baseline with high latency and deployment cost but advanced performance, is used to assess the impact of reranking under the ideal retrieval upper bound. It is obvious that: (1) Regarding the Recall and overall ranking performance of recommending correct APIs, 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 35.99% on Arctic-Embed 2.0, surpassing the next best model (*i.e.*, jina-reranker-v2) by a significant margin of 8.35%. (2) APIRANKER demonstrates consistent and stable improvements across all evaluation metrics post-reranking, irrespective of the underlying retriever. Even in the case of ideal retrieval upper bound (*i.e.*, NV-Embed-v2), where other methods show degraded performance compared to the original retrieval results, APIRANKER still demonstrates stable improvements, outperforming the retriever

Retrieval	Reranking		Metric			
	Category	Method	Recall@10	NDCG@10	MRR@10	MAP
Unixcoder	-	-	2.44	1.35	0.98	1.15
BIKER	-	-	11.65	8.57	6.65	6.27
GAPI	-	-	7.18	4.67	3.55	2.99
GIST-large	-	-	15.25	6.88	3.87	4.79
	vector-based	bge-reranker-large	19.51	13.64	9.42	12.11
		jina-reranker-v2	23.96	12.82	11.49	10.72
	LLM-based	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
		RepoCoder	11.09	4.15	5.91	7.21
		APIRANKER	**26.62**	**14.52**	**12.33**	**11.83**
Arctic-Embed 2.0	-	-	18.86	10.83	7.72	8.71
	vector-based	bge-reranker-large	17.21	15.53	10.86	7.66
		jina-reranker-v2	27.64	14.51	11.20	13.09
	LLM-based	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
		RepoCoder	17.67	8.18	7.27	7.66
		APIRANKER	**35.99**	**28.25**	**23.56**	**23.81**

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.

Model	Complexity	Parameters	Time delay (ms)	Relevance	principle
RG	O(N)	6.7B	634	Pointwise	Perplexity
UPR	O(N)	6.7B	617	Pointwise	Perplexity
PRP-Sorting	O(logN*N)	6.7B	1,109	Pairwise	Perplexity
PRP-Sliding	O(K*N)	6.7B	31,306	Pairwise	Perplexity
APIRANKER	O(N)	6.7B+160M	652	Pointwise	Semantics

Table 3: Comparison of the reranking method. N is the number of retrieved docs, and K is the number of docs to be returned after reranking.

across all metrics, which highlights the effectiveness of our method in enhancing ranking quality and coverage in diverse retrieval scenarios. Further detailed analysis can be found in Appendix A.7. **Overall**, APIRANKER **shows substantial and stable improvements in reranking different API candidates compared to other models, validating the effectiveness for API recommendation.**

Methods Comparison and Analysis. As illustrated in Table 3, APIRANKER demonstrates several key advantages over other LLM-based reranking methods: (1) Its linear complexity $O(N)$ ensures scalability for large-scale applications. Under further engineering optimizations (e.g., KV cache, vLLM engine), the average time for APIRANKER to rerank 50 docs declines from 652 ms to 237 ms. Since we aim to serve function-level code generation, this time delay is acceptable, and APIRANKER can meet the requirements of developers and real-world deployment. (2) The pairwise training method improves its ability to learn relevance preference, while the pointwise inference ensures efficient inference. It achieves higher accuracy with just 160 M additional parameters. (3) By leveraging the semantic understanding of a relevance estimator, APIRANKER excels in tasks that require a deep comprehension of API documents, in comparison to models that rely solely on perplexity. Further details are provided in Appendix A.8.

Retrieval-augmented Completion Evaluation. Using a retrieval-augmented generation approach, we evaluated the performance of three reranking models in improving code completion performance on Arctic-Embed 2.0 with four mainstream Code LLMs (i.e., CodeLLama-7b-instruct, StarCode2-7b (Lozhkov et al., 2024), deepseek-coder-6.7b-instruct, Qwen2.5-Coder-7b-instruct (Hui et al., 2024)). The experimental results showed that: (1) As illustrated in Fig. 4, APIRANKER consistently outperforms other reranking baselines and leads to stable improvements in passing across all code LLMs. APIRANKER offers stable and reliable improvements, demonstrating practical usability in real-world applications. (2) As illustrated in Fig. 4(a) and (b), we further analyze the impact of API recommendations on the capabilities of Code LLMs. Code LLMs, which already possess strong coding capabilities, show even greater improvements when supplemented with external API knowledge. APIRANKER consistently outper-

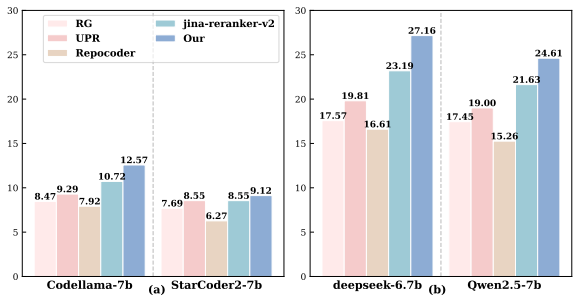

Figure 4: Effect of API Recommendations on Code Completion based on metric Improve@1 (%). The top 10 documents were used as context. The best result from 3 runs was reported.

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 Estimator	0.49	0.88

Table 4: Ablation study.

forms other reranking models, achieving significant improvements with both strong Code LLMs (e.g., deepseek-coder, Qwen2.5-Coder), highlighting its superior performance in scenarios where the Code LLMs' capabilities are strong but external API knowledge is required. The further detailed statements of this section is provided in Appendix A.9.

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 Estimator) 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 GIST-large and Arctic-Embed 2.0 when the Hidden Reasoning State Extractor and Relevance Estimator are removed separately. Notably, the removal of the Relevance Estimator 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 and Manual Evaluation. 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

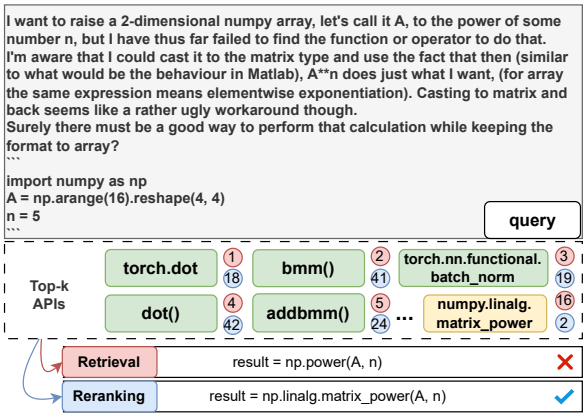

Figure 5: The example of code completion.

can't properly handle matrix exponentiation for `Numpy`, causing the completed code fails to pass test cases. APIRANKER reranks the retrieved APIs (*e.g.*, colored in blue), successfully moving the correct API (*e.g.*, colored in yellow) to the top, thus enhancing the LLM to produce the correct solution. This highlights that, APIRANKER can benefit the automatic code completion task by providing more accurate and effective API recommendations. To further validate the validity of the relevance score predicted by our framework, we conduct a manual evaluation by 3 users (each with over 4 years of programming experience) to assess the relevance between API docs and customed developer requirement. We calculate the agreement ratio between the manual evaluations and the automated scores. We calculate the Pearson correlation (Sedgwick, 2012) between the manual evaluations and the automated scores. The high consistency ratio of 94.8% indicates that our method aligns well with human evaluations, demonstrating its effectiveness in generating training data for API reranking. The details of the manual evaluation provied in Appendix A.11.

5 Conclusions

This research aims to recommend correct APIs to enhance automatic code completion, by jointly considering both natural language intent and incomplete code. To perform this task, we propose an approach APIRANKER that utilizes a self-learning ranking framework to automatically construct training data. Then we propose a novel reranking model 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 in both API recommendation and automatic code completion. We hope our study lays the foundations for this research and provides valuable insights.

6 Limitations

Several limitations are associated with our work. Firstly, due to the limited availability of code completion test sets that support code evaluation in multiple languages, and the difficulty in constructing queries that include both intent and incomplete code, our test is based on CodeRAGbench's DS-1000, which contains only a Python dataset. The available API recommendation benchmarks for other languages do not simultaneously consider both the query intent and the corresponding incomplete code context. Additionally, we were unable to find other suitable benchmarks that provide API documentation for code completion, apart from DS-1000, particularly given the scale of retrieval involved. APIRANKER does not rely on language-specific compilation, execution, or static analysis rules. The ranking signal is derived from the compatibility between code behavior and API documentation, which depends on API-level semantics (e.g., function names and input/output usage) rather than language-specific syntax. For this reason, we expect our approach to apply to other 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.

Thirdly, since the handling of fully code-only or NL-only scenarios is beyond the main focus of this work, we view further improving the robustness of different scenarios as a direction for future research.

References

Chi Chen, Xin Peng, Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao. 2021. Holistic combination of structural and textual code information for context based api recommendation. *IEEE Transactions on Software Engineering*, 48(8):2987–3009.

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*.

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

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.

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

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

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*.

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*.

Philip Sedgwick. 2012. Pearson's correlation coefficient. *Bmj*, 345.

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

Feng Wang, Yuqing Li, and Han Xiao. 2025. jina-reranker-v3: Last but not late interaction for listwise document reranking. *arXiv preprint arXiv:2509.25085*.

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.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding. *Preprint*, arXiv:2309.07597.

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

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

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. 2023b. 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 Splitting Details

During data construction, the split strategy ensures the target API usage is explicitly present in the target code y. The split position is randomly selected within some constraints. We sample from a bounded middle region (approximately 20%–80% of the file) and require sufficient left context and target length containing API usage; candidates that do not meet these conditions are discarded and resampled. Thus, the procedure is constrained random sampling rather than unrestricted randomness:

(1)The randomization is intended to cover diverse partial-context cases, since incomplete code in practice may end at different positions. Deterministic boundaries would bias the training data toward a narrow context pattern.

(2) Although some splits may not align perfectly with syntactic boundaries, the training signal is based on comparative perplexity and depends on behavioral compatibility rather than the exact split location. Therefore, our model is not sensitive to the split position.

(3) While certain split positions may individually form easier or harder examples, the randomness serves as data augmentation. It prevents the model from specializing to a particular code location and encourages learning completion behavior that is stable across different levels of context and code complexity.

A.2 Theoretical Analysis of Perplexity Alignment via Anchor Comments

Motivation. Raw code snippet of code completion is relatively out-of-distribution for LLM's prediction, while LLM-generated explanations are in-distribution. Adding such comments shifts the likelihood computation toward semantics and reduces sensitivity to stylistic or formatting artifacts. Due to traditional perplexity may be lower due to familiar patterns or tokens, we designed Perplexity Alignment via Anchor Comments that use LLM to generate fine-grained, fact-level comments for suffix code based on complete code, which describe specific operations (*e.g.*, API calls). This method shifts the focus from language fluency to measuring how well the API documentation aligns with observable code behavior. By doing so, we reduce potential interference from language-model fluency, ensuring that perplexity reflects the effectiveness of the candidate API documentation at explaining observable behavior rather than fluency. Therefore, our assumption is that under behavior-grounded anchor comments, the lower fact-level perplexity via anchor comments indicates stronger alignment between the document and code behavior.

Fact-level comments. Considering that perplexity is not a precise estimator of developer intent, our anchor comment design aims to mitigate biases in perplexity-based estimation. Our method removes existing code comments from the ground truth code completion and then generates fine-grained, line-level comments with the LLM to avoid sparse or poorly written comments. These comments describe observable behavior (e.g., operations and API usage) rather than intent; relevance is evaluated based on this observable behavior. During data construction, the target API usage is explicitly present in the code, and the generated comments describe observable behavior (e.g., function calls and input/output usage). The likelihood comparison focuses on whether an API document can

account for this behavior, rather than predicting arbitrary implementation details. Even for complex libraries, the decisive information primarily comes from API-level semantics (*e.g.*, function name and input/output types) rather than internal implementation complexity. Therefore, **the training signal is anchored to explicit API usage rather than unpredictable code generation**.

Avoiding impact from favoring certain APIs or styles. We use the same LLM as both the comment generator and the perplexity evaluator. Thus, model-specific style is consistent across candidates rather than changing the rank in relative comparisons. Moreover, our score are based on fact-level completion of code with comments; simply 'favoring' a particular API cannot lower perplexity unless the API document actually supports generating that factual description.

Avoiding potential bias propagation in self-Supervised data construction. Considering bias propagation in self-supervised data construction, our method uses the same LLM for both evaluating perplexity and training. Since the LLM evaluates relevance based on complete code, misjudgments are less problematic than direct predictions of relevance. Even in cases where biases appear, these are inherent in the model from the start, not introduced during the training process. Therefore, the model doesn't degrade over time; at worst, it retains its initial biases or incorrect understandings, which remain manageable and accepted. Importantly, this does not prevent the model from learning from the correct cases, as the model has the opportunity to adjust based on the correct examples.

Generality of Internal Representations and Prompt. As shown in Fig. 9, we keep the same and simple prompt in our method. Furthermore, we designed the Reasoning State Extractor and the Relevance Estimator (Section 3.3) specifically to avoid relying on the prompt sensitivity or the internal representations of the LLM. These components extract reasoning states and estimate relevance based on predictive scores, rather than directly depending on the language outputs of the LLM. Additionally, we have validated the effectiveness of our approach on CodeLLama-7b-instruct (see Table 2) and deepseek-coder-6.7b-instruct (see Table 6), demonstrating that the method works consistently and robustly across different LLMs.

Comparing with other strategies. Regarding alternative strategies, we explored code normalization and style rewriting, but the model tended to copy or minimally paraphrase the reference code rather than normalize it. We also tried intent-level summaries, which introduced unstable variations and made the scores less reliable. We therefore adopt Fact-level comment-based alignment as the most stable approach.

A.3 Dataset Construction Details

The data used to construct our training corpus was restricted to repositories under permissive licenses (e.g., MIT and Apache). 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. An API documentation typically includes the API's purpose, function description, input parameters, output values, and example requests and responses. Howerever, we take into account the diversity of API descriptions in real-world scenarios and do not strictly restrict the structure or form of API documents, as long as they can be converted into text or code formats.

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`).

There is no overlap between the ground truth documents from the test dataset and the APIs used

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

in training. The relevance score label depends on whether the API description functionally explains the usage of the code, not on the source or type of the document. Although dependency-based API documentation and human-written documentation differ in format, both describe API functionality, so the document type itself is not an impact factor in our dataset.

A.4 Baselines Setup detail

Retrieval Baselines. 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.

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 in 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.

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. An API documentation typically includes the API's purpose, function description, input parameters, output values, and example requests and responses. Most API documentation includes essential information, and although some longer API docs may be truncated at the "example" section, the necessary details, including the description of the API's role and function, are typically present within the first 512 tokens.

Reranking Baselines. In this section, all methods are based on either pointwise or pairwise comparisons. For pairwise methods, we compare only two documents at a time, keeping the input length below 4k tokens. As a result, the context length does not exceed the LLM's maximum context limit. 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:

$$s_i = \begin{cases} 1 + p(\text{Yes}), & \text{if output Yes} \\ 1 - p(\text{No}), & \text{if output No} \end{cases} \tag{13}$$

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 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.

(5) **RepoCoder**: RepoCoder (Zhang et al., 2023a) is a reranking method through iterative retrieval of code snippets based on the result of code generation. The API documentation is provided as retrieval source for the RepoCoder. We conducted the experiment using a 2-iteration approach, following the method described in the RepoCoder paper.

In order to comparing the performence of different reranking methods, we uniformly use CodeLlama-Instrcut-7B as the base LLMs. The comparison between methods is made using the same retrieval source. The maximum token length of an API document is set to 512.

(6) **Vector-based reranking model**: We also consider the following mainstream **vector-based reranking models** in API recommendation: bge-reranker-large (Xiao et al., 2023) and jina-reranker-v2 (Wang et al., 2025). They have demonstrated competitiveness across a series of benchmarks, including function-calling-aware and document/code reranking tasks. bge-reranker-large is a strong off-the-shelf model pretrained on large-scale retrieval and reranking corpora. jina-reranker-v2 is trained on function-calling, text-to-SQL, and code retrieval tasks, providing substantial code–documentation reranking capability. In our experiment, we fine-tune these vector-based reranking models on the APIRAC dataset to ensure a fair comparison.

The baseline of API recommendations. We adopt two common approaches of API recommendations: (1) **BIKER** (Huang et al., 2018): BIKER is an API recommendation approach that bridges lexical and knowledge gaps by using word embeddings for similarity and similar questions retrieval for supplementary information. Here, we take the queries in the test set as the source of similar questions. (2) **GAPI** (Ling et al., 2021): GAPI uses the code context as the query for API usage recommendation and employs graph neural networks to capture high-order collaborative signals. However, due to differences in task setup and dataset, the project structure information is not available in our dataset. We only used text attributes to nodes as input for API prediction since lacking project structural information in our datasets.

Details of the Query Input. For all models except BIKER and GAPI, we use the concatenation of the natural-language intent and the incomplete code as the input to the model, as these methods natively support both natural-language and code as inputs.

For BIKER, we follow its original design and use the intent-only query, since it is built for pure textual input. Based on our experiment, concatenating code with the intent degraded its performance, likely because the added code tokens interfere with its text-oriented retrieval pipeline.

For GAPI, we also follow its native setting and use code-only input, as the model operates on a graph representation constructed from code rather than natural-language queries.

A.5 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.
- **API 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 API 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.6 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:

$$\text{Recall@k} = \frac{\text{R}}{\text{N}}, \tag{14}$$

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:

$$\text{NDCG@k} = \frac{DCG@k}{ideal\ DCG@k}, \tag{15}$$

$$\text{DCG@k} = \sum_{i=1}^{k} \frac{2^{rel(i)} - 1}{\log_2(i+1)}, \tag{16}$$

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:

$$MRR@k = \frac{1}{|Q|} \sum_{j=1}^{Q} \frac{1}{k_Rank_i}, \tag{17}$$

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:

$$\text{MAP} = \frac{1}{|Q|} \sum_{j=1}^{Q} \frac{\sum_{i=1}^{n}(P(i) \times \text{rel}(i))}{\#\text{correct answers}}, \tag{18}$$

$$P(i) = \frac{\#\text{correct answers in top i}}{i}, \tag{19}$$

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.

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:

$$\text{Improve@k} = \frac{\sum_{i=1}^{m} \text{correct}(i)}{\#\text{failures in } k \text{ samples}}, \tag{20}$$

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.

A.7 Experimental Comparison of API Recommendation.

The results show that APIRanker significantly outperforms both BIKER and GAPI, demonstrating its superior effectiveness over query-based methods and code context-based method, verifying the effectiveness of APIRanker for combining user intent and code context for API recommendation.

Based on the experimental results, our model APIRanker is bertter than RepoCoder. (1) APIRanker significantly outperforms RepoCoder in terms of different retrieval methods. This advantage is likely due to the larger scale of API document retrieval, which recalls a much larger number of similar documents compared to repository-level code retrieval. (2) APIRanker achieves consistent performance improvements, while Repocoder is more dependent on the quality of the retrieval model. For example, RepoCoder experiences a notable decline in terms of using GIST-large and Arctic-Embed, but shows an improvement when paired with the strong retrieval model NV-Embed-v2. Additionally, RepoCoder relies on a robust generative model to enhance API recommendations. However, APIRANKER still performs better overall than RepoCoder.

In our experiment with deepseek-coder-6.7b-instruct, as shown in Table 6, we observed that APIRANKER consistently outperformed all other baselines. Specifically, when using retrieval models like GIST-large and Arctic-Embed 2.0, APIRanker showed significant improvements in Recall@10, surpassing other reranking model by a notable margin. On the other hand, CodeLlama-7B-instruct, when combined with APIRANKER, consistently outperforms deepseek-coder-6.7b-instruct. The observed difference is due to the influence of pre-learned API knowledge. For Deepseek-coder-6.7b, its familiarity with API knowledge reduces the significance of document relevance when constructing training data, leading to slightly lower data quality in relevance judgment. This results in a slightly more challenging task for Deepseek in capturing the Hidden Reasoning State compared to CodeLlama.

Thus, the superior performance of CodeLlama-7b-instruct can be attributed to its ability to reason more effectively with API knowledge, with less influence from pre-learned API knowledge. However, we do not believe the performance gap is substantial, as the overall difference is minimal. For example, in Table 6, on GIST-large, CodeLlama achieves a Recall@10 of 26.62, slightly outperforming Deepseek at 25.78. Similarly, on Arctic-Embed 2.0, CodeLlama scores 35.99, while Deepseek scores 35.36. Moreover, APIRANKER demonstrates stronger performance than other methods on Deepseek, highlighting its effectiveness across different model families.

As a future direction, one potential improvement approach could be to anonymize the API names in the code and documentation to reduce model familiarity, which we consider a promising avenue for further work.

A.8 Experiment Analysis.

Experiment of Time delay. In our experiment, we use Milvus[6] in the retrieval stage, a high-performance vector retrieval database that enables millisecond-level queries on datasets with millions of entries, making the time overhead negligible **in the retrieval stage**. Retrieval over a corpus of about one million entries takes approximately 6 ms, and does not refer to the reranking stage. In the reranking stage, except for PRP-Sorting and PRP-Sliding, all models support data parallelism since the reranking complexity is $O(N)$. Since RepoCoder relies on the generative model, which cannot calculate time complexity and causes significant time delays, we exclude it from the discussion. In the architecture of APIRANKER, input data is processed in parallel rather than in an autoregressive manner, allowing for simultaneous computation of output hidden states for each position. Under further engineering optimizations (e.g., KV cache, vLLM engine), the average time for APIRANKER to rerank 50 docs declines from 652 ms to 237 ms. Our method is designed for function-level or multi-line completion after an explicit trigger. As illustrated in the introduction, the goal is to complete the remaining part of a function, which typically involves generating multiple lines. In this scenario, the interaction pattern is closer to code generation. Therefore, the time delay is

[6] https://github.com/milvus-io/milvus

Retrieval	Reranking		Metric			
	Category	Method	Recall@10	NDCG@10	MRR@10	MAP
NV-Embed-v2	-	-	27.12	13.65	8.76	9.80
	vector-based	bge-reranker-large	21.69	13.2	5.91	10.04
		jina-reranker	23.38	13.64	9.46	10.72
	LLM-based	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
		RepoCoder	28.41	13.72	8.00	9.09
		APIRANKER	**30.17**	**15.49**	9.53	11.08

Table 5: Evaluation results of NV-Embed-v2 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.

Retrieval	Reranking		Metric			
	LLM	Method	Recall@10	NDCG@10	MRR@10	MAP
GIST-large	-	-	15.25	6.88	3.87	4.79
	deepseek-6.7B-instruct	RG	7.90	4.07	3.12	4.16
		UPR	19.57	13.07	10.57	11.36
		PRP-Sorting	7.10	3.17	1.57	3.42
		PRP-Sliding	11.15	10.05	8.57	9.13
		RepoCoder	21.69	13.35	10.92	8.42
		APIRANKER	**25.78**	**15.77**	**11.29**	**12.31**
	CodeLlama-7B-instruct	APIRANKER	**26.62**	**14.52**	**12.33**	**11.83**
Arctic-Embed 2.0	-	-	18.86	10.83	7.72	8.71
	deepseek-6.7B-instruct	RG	8.90	4.96	3.52	4.89
		UPR	25.06	17.18	13.62	14.54
		PRP-Sorting	7.00	2.35	0.87	2.42
		PRP-Sliding	12.65	10.15	9.15	10.04
		RepoCoder	24.49	19.91	10.51	8.12
		APIRANKER	**35.36**	**16.05**	**10.98**	**13.51**
	CodeLlama-7B-instruct	APIRANKER	**35.99**	**28.25**	**23.56**	**23.81**

Table 6: Evaluation results of deepseek-6.7B-instruct.

acceptable considering the improvement of correctness and the context in which it is used for code completion.

Memory Usage. We agree that LLM-based methods are more expensive and require greater memory usage than lightweight rerankers such as BERT. The difference is that our score is not a similarity function but a conditional likelihood: it evaluates whether the API document can account for the observed code behavior. This quantity may not be directly computed by standard cross-encoders, which are trained to model relevance similarity. Therefore, the LLM forward pass is required to obtain this signal. Since this signal is not explicitly modeled by lightweight rerankers, the additional computational cost is a necessary trade-off for measuring the signal rather than similarity.

Analysis on Failure Cases due to the Initial Retriever. Failure cases where the correct API is not included in the top 50 are inherent to the initial retriever, which relies on embedding similarity for large-scale retrieval. This is a common issue in the retrieval stage of reranking systems more broadly, since such methods often fail to fully capture the intent behind code task requirements, which may demand deeper semantic understanding and reasoning. Handling these cases falls outside the main focus of this work. Instead of adopting an idealized setting, we retain this realistic pipeline and assess the practical improvements achievable by reranking models under imperfect retrieval results.

A.9 Discussing Completion Evaluation.

In the code completion experiment, we focused on Improve@k is to reduce the potential negative impact that generation models, which are highly sensitive to context, can have on the results. When evaluating Pass@k, models sometimes fail on tasks that they could otherwise solve due to context sensitivity. By focusing on the Improve@k metric, we isolate and evaluate the benefits brought by document retrieval, without the additional noise from model generation errors.

Method	CodeLlama-7B	StarCoder2-7B	DeepSeek-Coder-6.7B	Qwen2.5-Coder-7B
base	28.65	31.58	38.99	37.43
RG	27.29	30.99	42.88	39.57
UPR	27.68	28.85	44.64	40.16
Repocoder	26.51	29.24	42.88	38.01
jina-reranker-v2	29.04	31.77	46.78	41.52
APIRANK	**30.41**	**32.94**	**49.71**	**44.44**

Table 7: Performance comparison of different methods across four code LLMs on Pass@3. Best results in each column are highlighted in bold.

To further provide a view of how reranking impacts the final execution-based metrics, we assessed Pass@3 to provide a clearer view of how reranking impacts the final execution-based metrics, and the results are provided in Table 7.

As shown in the Table 7, our method consistently improves performance across different base models. The improvements are more pronounced in models like deepseek-Coder-6.7b and Qwen2.5-Coder-7b, which are less sensitive to irrelevant document interference or noise, making their performance better in Retrieval-augmented generation.

Notably, our goal is not to compete with commercial code completion systems, but to study API retrieval and recommendation as a complementary knowledge mechanism for code models. This completion evaluation is a supplementary experiment. We do not need to further compare with larger-scale and commercial models as follows:

(1) APIRANKER operates at the retrieval and reranking level rather than the code completion level. Comparisons with larger completion models would not isolate our contribution. Improvements from larger models mainly reflect stronger generation capacity, whereas our evaluation measures whether API ranking provides useful external knowledge to the same base model.

(2) In addition, scaling model size does not guarantee coverage of third-party or private APIs, which are absent from pretraining data. Our method is designed to supply such external API knowledge instead of relying on memorization by larger models.

A.10 Discussing PPL and Log-Probability Based Uncertainty.

PPL is the exponentiated average of the log-probability based uncertainty, which is more suitable for assessing the overall model performance, as it aggregates token-level uncertainties into a comprehensive score. While the Log-Probability based uncertainty is used to calculate the generation probability of each token, which offers more granular insight on individual token-level. In terms of our research of code completion, we care more about the model's ability to generate complete code sequence, thus PPL is more appropriate. Regarding tasks for more detailed token-level analysis (e.g., keyword analysis, code style analysis), examining log-probability based uncertainty could be more informative.

A.11 Manual Evaluation.

To further validate the effectiveness of our reference, we conduct a user study. In particular, we randomly select 400 training API-query pairs that are scored based on our framework and ask 3 users (each with over 4 years of programming experience) to assess them. Users are asked to answer the question: "Which of the two API documents is more helpful for the query?", and every user is provided with three options (i.e., A is Better, B is Better, Cannot Determine/Both Equally). We calculate the agreement ratio between the manual evaluations and the automated scores. The results of the user study are as follows:

Consistency	Inconsistency	Indeterminate
85%	11%	4%

Table 8: The result of the user study.

We calculate the Pearson correlation between the manual evaluations and the automated scores. The high consistency ratio of 94.8% indicates that our method aligns well with human evaluations, demonstrating its effectiveness in generating training data for API reranking.

A.12 Analysis of Model's Cross-architecture Generalization.

We conducted experiments by using CodeLlama-7b-Instruct's PPL results and training deepseek-

Retrieval	PPL Evaluator	Reranker	Recall@10	NDCG@10	MRR@10	MAP
GIST-large	CodeLlama-7b	CodeLlama-7b	26.62	14.52	12.33	11.83
	deepseek-6.7b	deepseek-6.7b	25.78	15.77	11.29	12.31
	CodeLlama-7b	deepseek-6.7b	26.27	15.50	11.47	11.29
Arctic-Embed 2.0	CodeLlama-7b	CodeLlama-7b	35.99	28.25	23.56	23.81
	deepseek-6.7b	deepseek-6.7b	35.36	16.05	10.98	13.51
	CodeLlama-7b	deepseek-6.7b	36.53	27.16	23.81	21.36

Table 9: Performance comparison across different PPL evaluators and Rerankers.

coder-6.7b as a reranker. The experimental results are shown in Table 9. The experimental results demonstrate that CodeLlama's PPL further enhanced Deepseek's performance at the primary metric Recall@10, still outperforming other methods that use Deepseek as the reranker. This validates the effectiveness of our method even when paired with disparate LLMs. The improvement may be attributed to CodeLlama's relatively fewer pre-learned API knowledge, which allows it to respond more accurately to API relevance early on in our training set. Additionally, deepseek-coder-6.7b has a relatively stronger code reasoning capability, making it an effective reranker.