

API-Repo: API-centric Repository-level Code Completion

Zhihao Li
State Key Laboratory for Novel
Software and Technology
Nanjing University
Nanjing, China
zhihao.li@smail.nju.edu.cn

Chuanyi Li*
State Key Laboratory for Novel
Software and Technology
Nanjing University
Nanjing, China
lcy@nju.edu.cn

Changan Niu
State Key Laboratory for Novel
Software and Technology
Nanjing University
Nanjing, China
niu.ca@outlook.com

Ying Yan
State Key Laboratory for Novel
Software and Technology
Nanjing University
Nanjing, China
201250106@smail.nju.edu.cn

Jidong Ge
State Key Laboratory for Novel
Software and Technology
Nanjing University
Nanjing, China
gjd@nju.edu.cn

Bin Luo
State Key Laboratory for Novel
Software and Technology
Nanjing University
Nanjing, China
luobin@nju.edu.cn

Abstract

Code completion is one of the most common and important functions in software development. To complete code in a repository, code snippets from other files (i.e. cross-file context) is often needed. Text similarity-based retrieval is able to find some relevant code snippets and improve completion results, but it ignores the intention of retrieval. When programmers need retrieval, they may want to (1) find code snippets with similar process, or (2) find APIs that may be called by the unfinished code. To this end, we propose API-Repo, an API-centric repository-level code completion framework. In this framework, APIs have two uses: (1) focusing on the similarity of API calls when retrieving similar code snippets; (2) retrieving user-defined APIs that may be called by the unfinished code. API-Repo first collects all functions in the repository to build a function repository. It extracts API calls in each function and performs similar function retrieval to find code snippets with the most similar API calls, which indicates that they have similar processes. And it performs base class retrieval and related API retrieval to find APIs that may be called by the unfinished code. Three retrieval results are combined with the unfinished code for generation. We conduct experiments on five code LLMs with 7B parameters and two datasets. Experiments show that our method outperforms the baseline methods in code match and identifier match metrics. On RepoEval dataset, the two metrics increase by 2.94%/3.14% respectively, and on CrossCodeEval dataset, the two metrics increased by 4.58%/5.52% respectively.

CCS Concepts

• **Software and its engineering** → *Automatic programming*.

*Corresponding Author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Internetware 2025, Trondheim, Norway
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1926-4/25/06
<https://doi.org/10.1145/3755881.3755913>

Keywords

Code completion, Repository-level code completion, RAG, Code LLM

ACM Reference Format:

Zhihao Li, Chuanyi Li, Changan Niu, Ying Yan, Jidong Ge, and Bin Luo. 2025. API-Repo: API-centric Repository-level Code Completion. In *the 16th International Conference on Internetware (Internetware 2025)*, June 20–22, 2025, Trondheim, Norway. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3755881.3755913>

1 Introduction

Code completion is one of the most common and important functions in software development, which can help programmers improve coding efficiency. There are many types of code completion, including token-level completion, API completion, line-level completion, and block-level completion, etc. These different levels of completion have their own challenges. In development, writing code often requires referencing information from other files in the same repository. When performing code completion tasks, we also hope to add this information to improve the accuracy of completion. In this paper, we focus on repository-level code completion.

Repository-level code completion aims to improve code completion performance that is related to code repositories. For a code repository R and an unfinished code c , we need to find repository-level information (e.g. code snippets) $S = \{s_1, s_2, \dots, s_n\} (s_i \in R)$ that are related to c , and input these snippets and c into a generative model $\mathcal{G}(\cdot)$ to generate the code $\hat{Y} = \mathcal{G}(c, S)$. The generator needs to be trained to generate correct code. In recent years, generative code LLMs have developed rapidly. These models contain billions of parameters and are pre-trained on massive data. They have achieved good performance in software engineering tasks [14]. Code completion is also a generative task, and code LLMs can also solve the task of code completion. The data learned by the code model during pre-training are publicly available data. For repository-level code completion, however, there are a lot of user-defined content in the repository, such as variables, APIs, etc. These contents may not be learned during the pre-training stage of code LLMs, so the ability to complete these codes without repository-level information is

limited. We need to solve another challenge: How do we find more relevant repository-level information?

To solve the challenge, the most common retrieval approach is to treat the code as text and use text similarity methods to find the code snippet most similar to c , including Jaccard similarity [13], BM25 [25], and text embedding using pre-trained models (such as CodeBERT [8], UniXcoder [11]), etc. There are many methods based on this type of retrieval [17, 29, 31, 33]. These methods can retrieve some relevant code snippets and improve completion performance.

However, this approach ignores the programmer’s intention when triggering retrieval. When programmers want to find similar code in the repository, they may have two intentions: (1) there are code snippets in the repository that have a similar process with the unfinished code, which may access the same attributes or call the same APIs, so they can be used as a reference for completion. Figure 1a shows a completion task in RepoEval dataset [33]. The unfinished code contains a sequence of API calls, and we can find a similar sequence of API calls in another file in the repository, which can be retrieved as a snippet with similar process. (2) The current code needs to call the properties or methods defined in the project. Figure 1b shows a completion task in CrossCodeEval dataset [5]. The unfinished code needs to call a method defined in another file (i.e. a user-defined API). The user-defined API does not have a similar process to the unfinished code. Obviously, we need to formulate different retrieval strategies for these two intentions.

To this end, we proposed a novel repository-level code completion method API-Repo. For a completion request, we find relevant code snippets for the two purposes mentioned above. For intention (1), we propose *similar function retrieval*, which divides the code snippets by function and finds the snippets with the most identical API calls as the candidate snippets. For intention (2), we proposed *base class retrieval* and *related API retrieval* to retrieve the APIs that may be used by the unfinished code. We combined the above retrieval results and input them into the model together with the unfinished code for generation.

Our method is evaluated on the repository-level code completion datasets RepoEval [33] and CrossCodeEval [5]. The results show that our method outperforms the baseline code completion methods.

Our main contributions are summarized as follows:

- We propose a novel repository-level code completion framework API-Repo, which focuses on APIs to retrieve similar function snippets and possible API calls.
- We design a call-based similar function retrieval strategy to find the most similar code snippets and an API encoding strategy to find the most related APIs.
- We perform experiments to evaluate the performance of API-Repo on two publicly available datasets.

2 Related Work

2.1 Code LLMs

In recent years, large language models (LLMs) have been widely applied across many domains. In software engineering, there are also a large number of code LLMs. These models have demonstrated strong capabilities in code-related tasks such as code generation and code completion.

```
# code snippets from train_dreambooth_lora.py
...
text_encoder_cls = import_model_class_from_model_name_or_path(...)
noise_scheduler = DDPMSScheduler.from_pretrained(...)
text_encoder = text_encoder_cls.from_pretrained(...)
vae = AutoencoderKL.from_pretrained(...)
unet = UNet2DConditionModel.from_pretrained(...)

# unfinished code from train_dreambooth.py
...
text_encoder_cls = import_model_class_from_model_name_or_path(...)
noise_scheduler = DDPMSScheduler.from_pretrained(...)
text_encoder = text_encoder_cls.from_pretrained(...)
vae = AutoencoderKL.from_pretrained(...)
[To be completed]
```

(a) Code snippets with similar API calls

```
# code snippet from command.py
class Command(object):
    ...
    def format_text(self, text):
        ...
        try:
            text = text.format_map(variables)
        except Exception as e:
            print('Text cannot be rendered with variables')
            if text.strip() in variables.keys():
                text = variables[text.strip()]
            return text

# code snippet from answer.py
class AnswerCommand(Command):
    def generate_response_to_human(self):
        if isinstance(self.content, list):
            content = self.content
        else:
            content = [self.content]
        items = []
        for entry in content:
            if isinstance(entry, str):
                entry = self.[To be completed]
```

(b) An API that may be called by the unfinished code

Figure 1: Two intentions of retrieval

Early pre-trained models [1, 22, 30] are pre-trained through specific pre-training objectives on code corpus, and are fine-tuned on downstream tasks. Their general capabilities are limited due to the relatively small scale of training data and parameter size [32]. Recent code LLMs [2, 9, 12, 16, 23, 24, 26] have significantly increased the scale of training data and parameter sizes, leading to a notable enhancement in model capabilities..

These models are evaluated by many public benchmarks such as HumanEval [4] and MBPP [3]. However, the tasks in these benchmarks are general programming questions that do not relate to specific repositories.

2.2 Retrieval-augmented generation

Retrieval-augmented generation (RAG) aims to improve generation results by retrieving relevant information from the external knowledge base [10, 15]. This approach comes from NLP, and is also widely used in software engineering tasks.

To retrieve relevant information, queries and candidates will be encoded by various encoding strategy. Sparse retrievers, such as Jaccard similarity [13] and BM25 [25], are simple but effective in encoding code snippets [5, 17, 33]. Dense retrievers use trainable

models, such as CodeBERT [8] and UniXCoder [11], to obtain the embeddings of the text and compute similarity [17, 29, 33].

Retrieval can be performed at different granularities [7]. kNLM [28] build a token-level code database and retrieve a token each time. Token-level retrieval is faster but also requires more space to save candidates. A more common granularity is chunk-level retrieval, which returns a code snippet each time [6, 17, 21, 29, 33].

To improve retrieval results, many pre-retrieval enhancements and post-retrieval enhancements are proposed [7, 10]. Pre-retrieval enhancements can be performed by polishing query statements [33]. Post-retrieval enhancements can be performed by processing the retrieval results [17, 19, 29].

2.3 Repository-level code completion

Repository-level code completion aims to complete the unfinished code based on the information from other files in the current repository. To solve the problem, the most straightforward approach is applying RAG. Considering that code contains unique semantic and structural information, many repository-level code completion methods are proposed to improve the RAG results.

Some work explores the impact of the number of retrieval iterations on the generation results. RepoCoder [33] proposes an iterative generation strategy. The generation results of the first iteration are used in the second iteration of retrieval and re-generate to improve the completions. Repoformer [31] proposes a self-supervised learning approach to train models to determine whether retrieval is necessary, and only perform retrieval if necessary.

Some work trains new retrievers to retrieve more relevant information and improve the completion results. ReACC [21] uses a semantic-preserving transformation strategy for data augmentation to pre-train a code retrieval model to retrieve related similar code snippets for generation. RLCoder [29] proposes a reinforcement learning framework to train retrievers on unlabeled data.

Some work trains new models to encode contexts and use the embeddings for generation. COCOMIC [6] builds a project context graph to find cross-file context and uses a model to encode both in-file context and cross-file context to predict the next token. RepoFusion [27] also extracts and encodes each context separately, and combines them to feed into a decoder for generation.

Some work proposes new retrieval approaches to find relevant snippets. GraphCoder [19] constructs code context graphs and find the most similar sub-graphs as candidate code snippets. RepoFuse [17] analyzes the import information to extract rationale context and analogy context, and proposes a rank truncated generation strategy to build prompts for generation.

There are some publicly available code completion datasets for repository-level completion, including RepoEval [33], CrossCodeEval [5] and RepoBench [18].

In this paper, instead of simply retrieving based on text similarity, we propose a new API-centric retrieval approach. Compared to text-similarity-based retrieval, our approach is able to better meet the users' intentions.

3 Methodology

3.1 Overview

The workflow of API-Repo is shown in Figure 2.

Given a repository, we first extract functions in the repository and build a function repository. We treat the extracted functions as user-defined APIs, use pre-trained models to obtain API embeddings, and build API candidates.

When users trigger a code completion request, they may (1) want to find code with similar functions or processes in the project; or (2) want to call user-defined functions (i.e., user-defined APIs) in the repository. For the first intention, we perform similar function retrieval to find similar functions based on API call sequences. For the second intention, we perform base class retrieval to find the superclasses of the class where the unfinished code is located. And we perform related API retrieval to find relevant APIs that may be called by the unfinished code.

We merge the results of similar function retrieval and API retrieval with the unfinished code into a prompt, and input the prompt into the model to generate code.

3.2 Building function repository

Many retrieved-based repository code completion methods build candidate repositories by splitting the code files into code snippets of equal length, which may split continuous information into different snippets.

Considering that functions and class methods are the basic units to implement a requirement, we build a code repository by functions. Given a code repository R , it first collects all code files $R = \{r_1, r_2, \dots, r_n\}$. For each $r_i \in R$, tree-sitter¹ is used to collect classes Cls^i , functions (or class methods) M^i and import lines I^i .

For languages like Python, global statements are allowed, and these statements may implement functions which are related to the unfinished code. So we also treat them as functions. Algorithm 1 shows the method for collecting global statements. We divide global statements into blocks according to location and add them to M^i .

Algorithm 1: Collecting Global Statements

Input: File information (r_i, Cls^i, M^i, I^i)
Output: Collected global statements
 $n \leftarrow \text{FileLineLen}(f_i)$;
 $Covered \leftarrow []$;
foreach $item \in Cls^i, M^i, I^i$ **do**
 $(start, end) \leftarrow \text{GetLocation}(item)$;
 Append $(start, end)$ to $Covered$;
end
 $Covered \leftarrow \text{MergeAndSort}(Covered)$;
 $Uncovered \leftarrow \{\}$, $CurrentStart \leftarrow 0$;
foreach $start, end \in Covered$ **do**
 if $CurrentStart < start$ **and**
 NonEmpty($CurrentStart, start - 1$) **then**
 $Uncovered \leftarrow$
 $Uncovered \cup \{(CurrentStart, start - 1)\}$;
 end
 $CurrentStart \leftarrow end + 1$;
end
return $Uncovered$;

¹<https://tree-sitter.github.io/tree-sitter/>

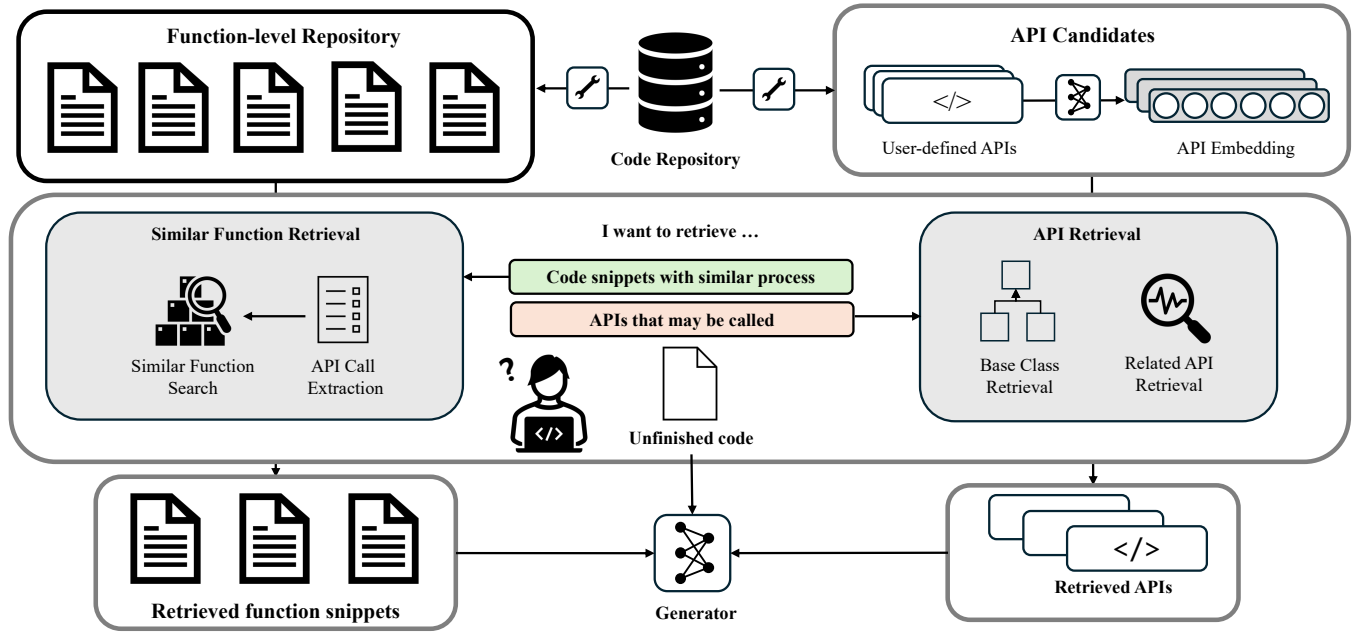


Figure 2: The workflow of API-Repo.

3.3 Similar function retrieval

When writing code, if the programmer wants to refer to the existing code in the code repository, it is very likely because unfinished has a similar process to the existing code. Considering that code often calls many APIs to implement specific functions, for codes with similar processes, they may call the same APIs. Therefore, if we want to find code snippets with similar processes in the code repository, we can look for code snippets that have the most identical API calls with the unfinished code.

In similar function retrieval stage, we refer to method calls and attribute calls as API calls. For each $m_j^i \in M^i$, we use tree-sitter to extract the API calls in m_j^i . We ignore the built-in functions (e.g. print in Python) as they are commonly used in code from various domains and contain less information.

When a completion request is triggered, we collect the call information $\tilde{C}$ of the function where the unfinished code is located. If the unfinished code lines are global statements, we also treat them as a function through Algorithm 1. We extract only from the n lines before the unfinished code to avoid including unrelated calls.

We use a sliding window strategy to traverse all collected functions in the repository. For each m_j^i , Algorithm 2 is used to find the most similar function snippet.

We compare the call similarity between $\tilde{C}$ and C_j^i through:

$$\text{CallSim}(\tilde{C}, C_j^i) = \begin{cases} \frac{|\tilde{C} \cap C_j^i|}{|\tilde{C}|} & \text{if } |\tilde{C}| > 0, \\ 0 & \text{else.} \end{cases} \quad (1)$$

The value of call similarity is between 0 and 1. We set a threshold σ and filter out candidate functions with similarity less than σ . We

Algorithm 2: Similar Function Retrieval

Input: Function m_j^i , query snippet calls $\tilde{C}$, window size n , threshold σ

Output: The most similar function snippet

$len \leftarrow \text{FileLineLen}(f_i)$;

$max_score \leftarrow 0$;

$max_snippet \leftarrow \phi$;

for $i \leftarrow 0$ **to** $len - n - 1$ **do**

$snippet \leftarrow m_j^i[i : i + n]$;

$C \leftarrow \text{GetCalls}(snippet)$;

$sim_score \leftarrow \text{CallSim}(\tilde{C}, C)$;

if $sim_score > max_score$ **then**

$max_snippet \leftarrow snippet$;

$max_score \leftarrow sim_score$;

end

end

if $max_score \geq \sigma$ **then**

return $max_snippet$;

else

return ϕ ;

sort the similarities to obtain the top k candidate functions with the highest similarity to the results of similar call retrieval.

Although the time complexity of the sliding window strategy is $O(n)$, the call information is extracted in advance, and it does not need to be extracted every time the sliding window is used, so the time spent is negligible.

Taking the task in Figure 1a as an example, there is a code snippet that is almost identical to the unfinished code. There are five

```
# super class: PromptSectionBase
# file path: Stevenic-promptrix-py-6ed7059/src/promptrix/PromptSectionBase.py
#
# attributes: ['self.separator', 'self.tokens', ...]
# methods:
#   def renderAsMessages(self, memory, functions, tokenizer, max_tokens)
#   def renderAsText(self, memory, functions, tokenizer, max_tokens)
#   ...
```

Figure 3: An example template of base class information.

API calls in the two snippets, four of which are shared by both. Therefore, their call similarity is 1.0. Due to space limitations, we only include partial snippets of the two functions for demonstration purposes. The actual functions contain more API calls.

3.4 API retrieval

Another scenario of referring to the existing code snippets is that users want to use a function they write before (i.e. a user-defined API). Existing work [17] finds related API information by analyzing import statements and improves performance. However, not all completion scenarios have the relevant APIs pre-imported. We aim to find related APIs in more general cases.

3.4.1 Base class retrieval. Firstly, if the unfinished code is in a class, the attributes and methods provided by the class are strongly related to the unfinished code, and there is a probability that the unfinished code may invoke these APIs. For languages such as Python, the APIs provided by a class are usually located in the same file (i.e., the in-file-context). However, if the class inherits from a base class, we also need to consider the APIs provided by the base class.

If the class where the unfinished code is located inherits from a base class, we can get the location of the base class by analyzing the import information. If the base class is defined in the repository, we can locate the file where the base class is located and collect the attributes and methods defined in the base class. This step is easy to achieve by the constructed function repository. The collected attributes and method signatures are made into a template as shown in Figure 3 as part of the API context.

3.4.2 Related API retrieval. In the stage of related API retrieval, we treat all collected functions and class methods M^i as repository-level APIs. For each $m_j^i \in M^i$, we extract the function signature s_j^i . If m_j^i is a class method, we also extract the class information $cls_j^i \in Cls_j^i$ and add it to s_j^i . Figure 4 shows an example of signature extraction. The code with orange background is extracted as the signature of the constructor `UNet2DConditionModel.__init__`.

After extracting all signatures, we use a small-size code pre-trained model $\mathcal{M}(\cdot)$ to encode the signatures. In this work, we use the encoder part of UniXcoder [11]. We input s_j^i into $\mathcal{M}(\cdot)$ and get the last hidden states $H^{(i,j)} = (h_1^{(i,j)}, h_2^{(i,j)}, \dots, h_t^{(i,j)})$, where t is the length of the signature. And we compute the embedding e_j^i of the signature as:

$$e_j^i = \frac{1}{t} \sum_{k=1}^t h_k^{(i,j)} \quad (2)$$

These embeddings are added to an API-level repository for retrieval. When a completion request is triggered, we extract the m

```
class UNet2DConditionModel(ModelMixin, ...):
    ...

    _supports_gradient_checkpointing = True

    @register_to_config
    def __init__(
        self,
        sample_size: Optional[int] = None,
        in_channels: int = 4,
        out_channels: int = 4,
        center_input_sample: bool = False,
        flip_sin_to_cos: bool = True,
        freq_shift: int = 0,
        down_block_types: Tuple[str] = (
            "CrossAttnDownBlock2D",
            "CrossAttnDownBlock2D",
            "CrossAttnDownBlock2D",
            "DownBlock2D",
        ),
        ...
    ):
        super().__init__()

        self.sample_size = sample_size
        time_embed_dim = block_out_channels[0] * 4
        ...
```

Figure 4: An example of extracting signatures of collected functions. Due to space limitations, some code details are omitted.

lines of code before the uncompleted line as context lines and set a mask token at the unfinished position. We use the same pre-trained model to embed the context and extract the hidden state at the mask position as embedding $\tilde{e}$. We compare the similarities between APIs and the context by cosine similarity:

$$\text{API_Sim}(\tilde{e}, e_j^i) = \frac{\tilde{e} \cdot e_j^i}{\|\tilde{e}\| \cdot \|e_j^i\|} \quad (3)$$

We select the top k most similar API signatures as the results of related API retrieval.

3.5 Merging retrieval results & Generation

After two retrievals, we merge the results to build prompts for generation.

In similar function retrieval, we set a threshold to filter out functions with too low similarity, and there may be too few candidates for some tasks. We use UniXcoder [11] to find 10 most similar snippets and re-rank them with retrieved function snippets.

Related APIs and similar function snippets are concatenated into the in-file context to construct the prompt. The constructed prompt is fed into the model to complete the code.

4 Experimental Settings

4.1 Baselines

To evaluate the performance of API-Repo, we compare API-Repo with the following methods:

(1) **Vanilla Completion**: We use the LLM to complete the code without retrieval. This method is used to test the completion capability of LLM itself.

(2) **RG-1**: RG-1 method uses several lines before the uncompleted code as the context and uses a retriever to retrieve the most similar code snippets. The retrieved snippets are added to the context to generate the target code.

(3) **RepoCoder** [33]: While RG-1 only performs one retrieval iteration, RepoCoder performs a second retrieval iteration to improve the completion results. After the first iteration of the retrieval, the intermediate results are used to retrieve again and generate the target code.

4.2 Generative models

API-Repo and all the baseline methods need code LLMs to complete the code. We use DeepSeek-Coder 7B [12], CodeGemma 7B [34], CodeLlama 7B [26], StarCoder2 7B [20] and StarCoder 7B [16]. All models are available from Hugging Face ².

4.3 Datasets

We use two datasets to conduct experiments:

(1) **RepoEval** [33]: RepoEval provides three levels of completion tasks: line-level, API-level, and function-level. The data set does not release the execution code that function-level completion tasks require, so we use only the line-level dataset and the API-level dataset. The two datasets share 8 publicly available Python repositories. For each repository, 200 line-level test cases and 200 API-level test cases are constructed. Both datasets contain 1,600 different test cases.

(2) **CrossCodeEval** [5]: CrossCodeEval provides repositories in four programming languages. We use the Python dataset, which contains 471 repositories and 2,665 test cases. Unlike RepoEval, the number of test cases contained in each repository in CrossCodeEval is different.

4.4 Metrics

To evaluate the performance of API-Repo and baseline methods, given a generation result $\hat{Y}$ and the ground truth Y , we use the following metrics to measure the accuracy of the result.

(1) **Exact Match (EM)**: If $\hat{Y}$ is identical to Y , the EM score is 1, otherwise it is 0.

(2) **Edit Similarity (ES)**: ES is based on Levenshtein distance Lev . It is calculated as

$$ES = 1 - \frac{Lev(\hat{Y}, Y)}{\max(|\hat{Y}|, |Y|)} \quad (4)$$

(3) **Identifier Exact Match (ID-EM)**: To compute identifier match, identifiers are extracted from $\hat{Y}$ and Y , resulting in two identifier lists $\hat{L}$ and L . If $\hat{L}$ is identical to L , the ID-EM score is 1, otherwise it is 0.

(4) **Identifier F1 (ID-F1)**: Given $\hat{L}$ and L , TP represents the number of identifiers that are present in both $\hat{L}$ and L , FP represents the number of identifiers found in $\hat{L}$ but not in L , and FN represents the number of identifiers that are present in L but not in $\hat{L}$. Based on

the statistical results, we can compute $\text{precision} = \frac{TP}{TP+FP}$, $\text{recall} = \frac{TP}{TP+FN}$, and F1 scores:

$$ID - F1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \quad (5)$$

4.5 Other Settings

We conduct all experiments on 4 NVIDIA Tesla V100 GPUs with 32GB RAM. All generative models and retriever models are downloaded from Hugging Face. Although some methods provide the source code for generation, we use the Hugging Face Transformers pipeline for all methods in generation stage for a fair comparison.

The input length of generation is 2048. For RG-1, RepoCoder and API-Repo, the ratio of in-file context length to cross-file length is 1:1. For API-Repo, the API retrieval results occupy 1/4 of the cross-file length, and the similar function retrieval results occupy 3/4 of the cross-file length. For RG-1, RepoCoder, all retrieval results are textually similar code snippets. Snippets that exceed the length limit will be truncated.

For Vanilla Completion, it only contains the in-file context.

5 Results And Discussions

5.1 Performance comparison

To evaluate the overall performance of API-Repo, we performed experiments on API-Repo and baseline methods using the settings in the previous section.

The results are shown in Table 1. It should be noted that the methods for EM and ES in the RepoEval dataset and the CrossCodeEval dataset are not identical. For a ground truth consisting of n lines, RepoEval computes EM/ES by truncating the first n lines of the generated result, whereas CrossCodeEval calculates EM/ES by truncating the first line of the generated result. The results of the two calculation methods may differ. Therefore, for each dataset, we employ the respective method to compute EM/ES.

From the results, we can see that our method outperforms existing approaches on all datasets and metrics. On RepoEval and CrossCodeEval dataset, our method improves the average EM metric by 2.94% and 4.58% respectively, and improves the average ID-EM metric by 3.14% and 5.52% respectively. Considering that dataset CrossCodeEval includes a larger number of repositories and covers a wider range of code domains, it demonstrates the generalizability of our method.

We can also see that retrieval is important for repository-level code completion. Code LLMs have been pre-trained on large-scale code corpus, and it can correctly complete some tasks. However, there is still a performance gap between raw generation and retrieval-based methods.

RG-1 retrieves the most similar code snippets from other files in the repository, and the performance improvement is substantial. RepoCoder performs two iterations of retrieval and generation to refine the output. From the results, it is evident that this method indeed shows improvement over RG-1. On the RepoEval dataset, it also comes quite close to the results achieved by our method. However, RepoCoder requires two iterations of retrieval and generation, implying that it requires double generation time for contexts of the same length.

²<https://huggingface.co>

Table 1: Performance comparison

Models & Methods	RepoEval (Line)				RepoEval (API)				CrossCodeEval			
	EM	ES	ID-EM	ID-F1	EM	ES	ID-EM	ID-F1	EM	ES	ID-EM	ID-F1
DeepseekCoder												
- Vanilla Completion	39.50	62.76	47.13	55.11	31.50	61.25	35.38	63.53	9.38	63.03	16.44	49.21
- RG-1	45.81	67.32	53.94	60.33	37.38	66.09	41.25	68.53	20.38	69.29	30.02	58.76
- RepoCoder	47.75	70.01	56.50	62.68	39.88	67.88	43.56	70.33	23.08	71.02	33.47	61.23
- Ours	50.19	71.45	58.81	63.92	42.75	69.20	46.50	72.12	28.63	74.06	39.85	66.62
CodeGemma												
- Vanilla Completion	41.75	64.84	49.38	57.45	32.75	61.60	35.75	64.53	10.69	63.69	17.60	49.90
- RG-1	46.69	67.97	54.69	60.85	40.06	67.58	43.56	69.80	21.73	70.05	30.81	59.53
- RepoCoder	48.31	69.31	56.44	62.47	41.38	69.04	45.44	71.26	24.84	72.31	34.75	62.71
- Ours	50.88	71.28	58.63	63.76	43.25	69.28	47.31	72.41	29.08	74.48	39.77	66.75
CodeLlama												
- Vanilla Completion	36.81	61.59	45.31	54.27	30.44	59.49	34.31	61.73	7.43	60.93	14.52	46.58
- RG-1	42.13	65.26	50.63	58.07	37.00	64.30	40.50	66.60	17.90	67.31	27.24	56.64
- RepoCoder	44.94	66.95	53.25	59.97	38.88	65.35	42.44	67.85	20.34	68.81	30.17	58.46
- Ours	48.31	70.03	56.63	62.43	40.31	66.70	44.25	69.67	24.69	72.00	35.46	63.93
StarCoder2												
- Vanilla Completion	37.00	61.42	45.44	54.08	31.75	60.31	35.25	62.72	7.13	61.65	14.15	47.14
- RG-1	45.25	67.09	53.06	59.45	35.69	62.03	39.13	62.17	18.42	68.27	27.77	56.93
- RepoCoder	46.88	68.68	54.94	61.37	37.25	63.31	40.81	63.68	21.39	69.96	30.84	59.85
- Ours	49.25	70.26	57.38	63.02	42.19	68.25	46.50	71.04	25.55	72.68	36.32	64.83
StarCoder												
- Vanilla Completion	34.31	58.79	42.56	51.54	27.88	56.97	31.38	59.67	6.49	59.71	13.17	45.19
- RG-1	42.81	65.20	51.31	57.89	31.88	57.81	34.94	58.55	17.56	66.19	26.27	55.60
- RepoCoder	44.75	66.44	52.94	59.24	34.63	59.46	37.88	60.39	19.47	67.57	29.12	57.45
- Ours	46.38	67.51	54.81	60.49	40.56	66.30	44.81	69.09	24.05	69.64	34.56	61.39

From the perspective of the model, the generation capabilities of different models are also different. From the results, we can see that the best CodeGemma and Deepseek Coder achieve the best performance. Although the performance of StarCoder is slightly lower, using our method, the performance of StarCoder can also be improved to the same level as other models. This also shows the generalizability of our method.

Answers and Implications. Our proposed method outperforms the baseline methods on two datasets, indicating the effectiveness of our method. On RepoEval and CrossCodeEval dataset, our method improves the average EM metric by 2.94% and 4.58% respectively, and improves the average ID-EM metric by 3.14% and 5.52% respectively.

5.2 Ablation study

We also perform an ablation study to find the contribution of each component in API-Repo.

In our methods, we proposed three types of information retrieval. We remove one type of information and keep the other settings unchanged. When removing similar function retrieval, we replace similar function snippets with code snippets with the highest textual similarity. When removing base class retrieval or related API retrieval, we only keep the other retrieval results. We use the DeepSeek Coder model for generation and observe the changes in each metric.

Table 2: Ablation results

Ablation Settings	EM	ES	ID-EM	ID-F1
RepoEval (Line)	50.19	71.45	58.81	63.92
- w/o SFR	46.19	67.87	53.94	60.48
- w/o BCR	49.81	71.10	58.13	63.65
- w/o RAR	48.63	68.75	46.44	71.64
RepoEval (API)	42.75	69.20	46.50	72.12
- w/o SFR	38.63	66.77	42.50	69.87
- w/o BCR	42.31	68.75	46.44	71.64
- w/o RAR	40.94	67.82	45.06	70.97
CrossCodeEval	28.63	74.06	39.85	66.62
- w/o SFR	23.68	71.43	34.18	62.83
- w/o BCR	27.65	73.46	38.39	65.37
- w/o RAR	26.19	72.48	36.51	63.99

The results are shown in Table 2, where *w/o SFR* represents the results of removing similar function retrieval, *w/o BCR* represents the results of removing base class retrieval, and *w/o RAR* represents the result of removing related API retrieval.

From the results, we can see that after removing each type of information, the metrics of the generated results will decrease. Among them, the decrease is the largest after removing SFR, which shows that SFR can effectively retrieve code with processes similar to the unfinished code and can better help complete code. After removing

BCR, it decreases slightly on the RepoEval dataset and decreases more on the CrossCodeEval dataset. Since BCR only contains superclass information, the total amount of information provided is relatively small, and not all tasks in the dataset require superclass information. However, if the code really needs the information in the superclass, then such retrieval is useful, so metrics is still decreasing. After removing RAR, the metrics also decreases, indicating that providing user-defined APIs within the project is also necessary for generation.

Answers and Implications. Each component of API-Repo contributes to the performance of generation, among which SFR contributes the most, and BCR and RAR also have an impact on the performance.

5.3 Impact of context length

To further investigate the characteristics of API-Repo, we perform two experiments to study the impact of context length.

5.3.1 The ratio of in-file context to cross-file context. From Table 1, we see that cross-file context is important for repository-level code completion. However, it is impossible to complete code without in-file context. Therefore, it is crucial to set a reasonable ratio between in-file context and cross-file context.

For a total context length of 2048 tokens, we allocate n tokens to in-file context and $(2048 - n)$ tokens to cross-file context. For cross-file context, we still keep 1/3 as extended API context and 2/3 as similar function snippets. We set the value of n to 256, 512, 768, 1024, 1280, 1536, and 1792 respectively, to investigate the variations in each metric of the generated results.

The results are shown in Figure 5. From the figures, we can see that different ratios of in-file context to cross-file context lead to variations in the generation performance. Different datasets have different trends in terms of context ratio changes.

For CrossCodeEval dataset, although we set the in-file context and cross-file context to 1024 tokens each in the experimental setting, reducing the proportion of in-file context can achieve better generation results. This shows that the completion tasks in CrossCodeEval dataset requires fewer in-file context; on the other hand, reducing the length of the cross-file context leads to a decrease in performance, which means that there is still room for improvement in retrieval, and it is necessary to find more similar candidates and rank them to the front.

For RepoEval dataset, we find that shorter in-file context does not achieve the best results. Setting in-file context length between 1024 and 1280 produces better results. This shows that for completion tasks in RepoEval dataset, the in-file context contains more information that is useful for completion, and a longer context length is required. The cross-file context is also important. When the cross-file context length continues to decrease, the performance also decreases significantly.

5.3.2 The total length of input context. We also want to explore the impact of total context length on the generated results. We keep the ratio of in-file context length to cross-file context length unchanged at 1:1, adjust the length of the total context, and explore the changes of each metric. We also use DeepSeek Coder as the

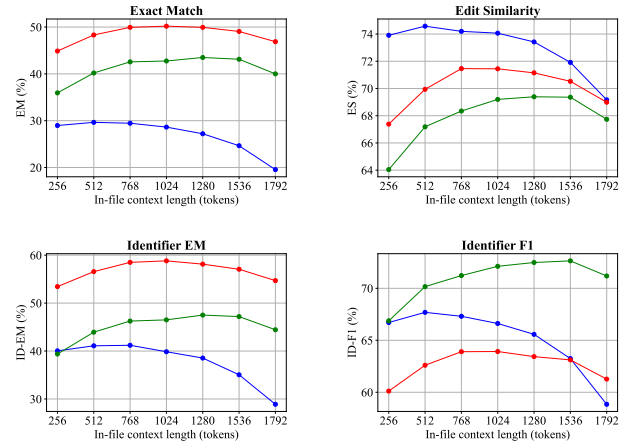


Figure 5: Impact of the ratio of in-file context to cross-file context. —●— : CrossCodeEval, —●— : RepoEval Line, —●— : RepoEval API

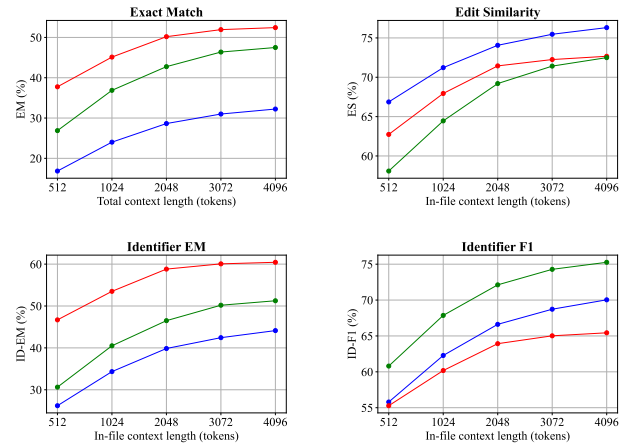


Figure 6: Impact of total context length. —●— : CrossCodeEval, —●— : RepoEval Line, —●— : RepoEval API

backbone model. We adjust the context length to 512, 1024, 2048, 3072, and 4096 respectively, and the metrics of generated results are shown in Figure 6.

As seen in the figure, the changing trends of different datasets are similar. As the length of the context increases, the overall metrics of the generated results are improved. From 512 to 2048 tokens, the metrics have increased significantly; from 2048 to 4096 tokens, the improvement has become more gradual. This shows that a shorter context length cannot fully contain all the context information that is useful for generation, and increasing the context length can introduce more useful context information. However, it is worth noting that a longer context means a longer generation time, resulting in a high overhead.

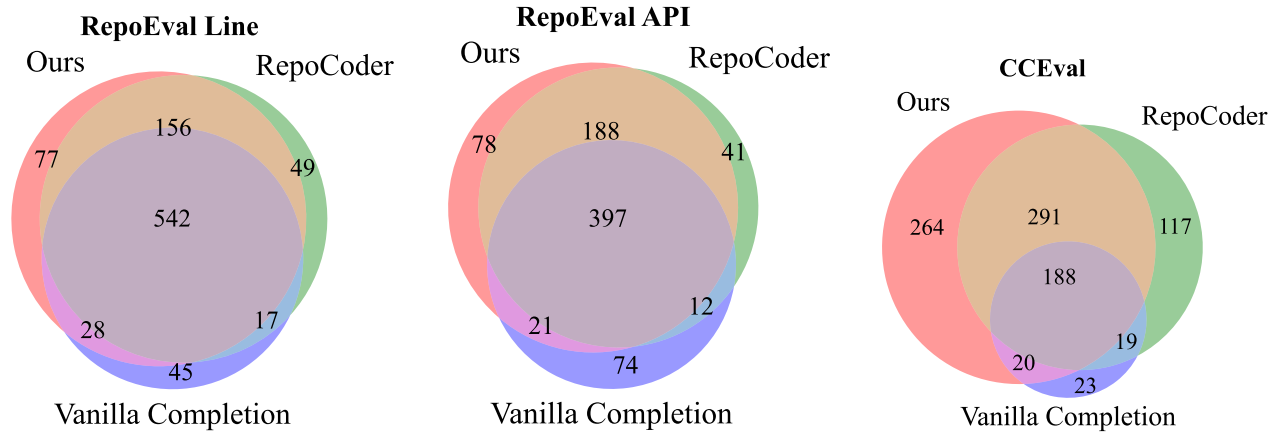


Figure 7: EM task IDs of different generation methods

Answers and Implications. The ratio of in-file context length to cross-file context length affects the performance of generation. The ratio for achieving the best performance varies depending on the data. The longer the total length of the context, the better the performance, but it also results in more overhead.

5.4 Qualitative analysis

Lastly, we perform qualitative analysis to show the effectiveness of the proposed API-Repo.

We first count the EM task IDs of generation methods in the overall performance comparison (i.e., Section 5.1) and obtain the Venn diagrams shown in Figure 7. As can be seen from the figure, each method has data instances that it can correctly complete, while other methods cannot. Our method can correctly complete most data instances, especially in the CrossCodeEval dataset. However, it is worth noting that there are also some data instances that can be correctly completed by Vanilla Completion while the RAG method cannot correctly complete them, which shows the limitations of the RAG method. It is necessary to determine whether retrieval is needed in real generation.

To further explore why our method works well, we manually analyze the retrieval results and the generation results in performance comparison experiments. We show three examples in Figure 8. These examples demonstrate completion tasks where our method successfully retrieves relevant code snippets and correctly completes them, in contrast to the baseline methods which fail to achieve correct completion.

More accurately locate related code. As shown in Figure 8(a), the left side is a completion task, and [to be complete] is the hole that needs to be completed. The right side shows the similar code snippets retrieved by API-Repo and RepoCoder. It can be seen that both methods find the most similar file. However, RepoCoder splits the code snippets into fixed lengths, so the functionally continuous code snippets are split into two snippets, and finally an inappropriate code snippet is selected. API-Repo uses sliding windows and API call similarity to accurately find the most similar code snippet, and thus generates the correct result.

More valid information. As shown in Figure 8(b) and (c), the API calls that need to be completed in both examples come from other files in the project. RepoCoder is unable to find the corresponding API information. For Figure 8(b), it can be seen that the target API is a method in the base class. Our method can find the corresponding API through *base class retrieval*, and the model uses this information to correctly complete the code. For Figure 8(c), the target API comes from other files. Our method adds the target API to the context through *related API retrieval*, and the model uses this information to correctly complete the code.

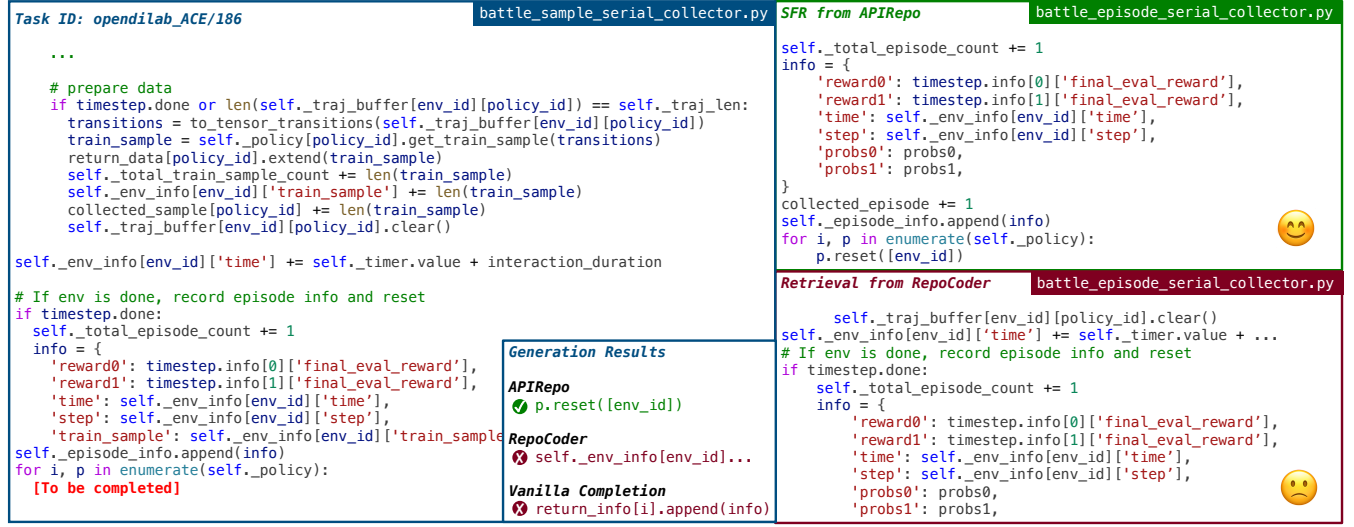
Answers and Implications. Our method can locate similar code snippets more accurately and provide more effective API information in the repository, allowing the model to correctly complete more completion tasks. However, retrieval has a negative impact on some completion tasks. It is necessary to determine whether retrieval is needed in real generation.

6 Threats and Validity

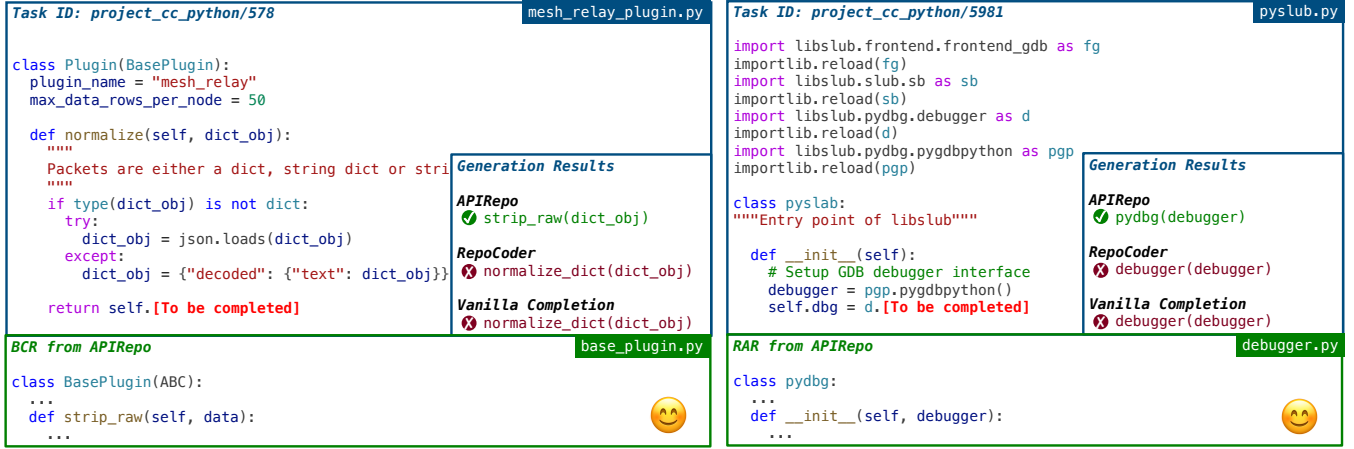
Internal Validity. Threats to internal validity arise from API call analysis and data leakage. The proposed method relies on API call analysis. However, the structures of code repository in the datasets vary, and some calls cannot be analyzed using existing tools, especially some test code snippets. We investigated some repositories with abnormal analysis results and implemented the analysis process based on these cases.

Since code LLMs are trained on large code corpora, there is a risk of data leakage. In order to reduce the impact of data leakage, we use Vanilla Completion as a baseline to test the model's own completion ability.

External Validity. Threats to external validity arise from the generalization of the proposed method. We selected RepoEval dataset and CrossCodeEval dataset. Since RepoEval dataset only contains 8 repositories, which is not enough to prove the effectiveness of the method, we also selected CrossCodeEval dataset with 471 Python repositories. We conducted experiments on the Python dataset and proved its effectiveness on Python, but did not verify it on other programming languages.



(a)



(b)

(c)

Figure 8: Completion examples that API-Repo can complete correctly.

Construct Validity. Threats to construct validity arise from the metrics of generation. We used code match metrics (EM and ES) and identifier match metrics (ID-EM and ID-F1), which are commonly used by repository-level code completion methods. However, ES, ID-EM and ID-F1 only test the similarity of the code and cannot guarantee the generated code can run correctly. EM guarantees the correctness of the code, but in actual programming, the same function can be implemented in multiple ways, so the correctness cannot be fully evaluated. RepoEval constructs a function-level completion dataset and uses an execution-based metric pass rate. However, it does not release the required execution code.

7 Conclusion

In this paper, we propose API-Repo, an API-based repository-level code completion framework that adopts the RAG paradigm. API-Repo takes into account the different intentions for retrieval. Similar function retrieval is used to retrieve code snippets that have a similar process to the unfinished code. Base class retrieval and related API retrieval are used to retrieve APIs that may be used by the unfinished code. We conducted experiments on five backbone code LLMs and two datasets to evaluate the performance of API-Repo. The experimental results show that API-Repo outperforms the baseline methods, which validate its effectiveness.

References

- [1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings*

- of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6–11, 2021, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tür, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, 2655–2668. doi:10.18653/V1/2021.NAACL-MAIN.211
- [2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! CoRR abs/2301.03988 (2023). doi:10.48550/ARXIV.2301.03988 arXiv:2301.03988
 - [3] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. CoRR abs/2108.07732 (2021). arXiv:2108.07732 <https://arxiv.org/abs/2108.07732>
 - [4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. CoRR abs/2107.03374 (2021). arXiv:2107.03374 <https://arxiv.org/abs/2107.03374>
 - [5] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/920f2dced7d32ab2ba2f1970bc306af6-Abstract-Datasets_and_Benchmarks.html
 - [6] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. CoCoMIC: Code Completion by Jointly Modeling In-file and Cross-file Context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20–25 May, 2024, Torino, Italy*, Nicoletta Calzolari, Min-Yen Kan, Véronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue (Eds.). ELRA and ICCL, 3433–3445. <https://aclanthology.org/2024.lrec-main.305>
 - [7] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyuan Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25–29, 2024*, Ricardo Baeza-Yates and Francesco Bonchi (Eds.). ACM, 6491–6501. doi:10.1145/3637528.3671470
 - [8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16–20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. doi:10.18653/V1/2020.FINDINGS-EMNLP.139
 - [9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1–5, 2023*. OpenReview.net. <https://openreview.net/forum?id=hQwb-lbM6EL>
 - [10] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, and Haofen Wang. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. CoRR abs/2312.10997 (2023). doi:10.48550/ARXIV.2312.10997 arXiv:2312.10997
 - [11] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22–27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. doi:10.18653/V1/2022.ACL-LONG.499
 - [12] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. CoRR abs/2401.14196 (2024). doi:10.48550/ARXIV.2401.14196 arXiv:2401.14196
 - [13] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.
 - [14] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. CoRR abs/2406.00515 (2024). doi:10.48550/ARXIV.2406.00515 arXiv:2406.00515
 - [15] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6–12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). <https://proceedings.neurips.cc/paper/2020/hash/6b492320205f780e1bc26945df7481e5-Abstract.html>
 - [16] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *Trans. Mach. Learn. Res.* 2023 (2023). <https://openreview.net/forum?id=KoFoG41haE>
 - [17] Ming Xiang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Wei Jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. 2024. REPOFUSE: Repository-Level Code Completion with Fused Dual Context. CoRR abs/2402.14323 (2024). doi:10.48550/ARXIV.2402.14323 arXiv:2402.14323
 - [18] Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024*. OpenReview.net. <https://openreview.net/forum?id=pJZIOuQuf>
 - [19] Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. GraphCoder: Enhancing Repository-Level Code Completion via Coarse-to-fine Retrieval Based on Code Context Graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 570–581. doi:10.1145/3691620.3695054
 - [20] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osa Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian J. McAuley, Han Hu, Torsten Scholach, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, and et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. CoRR abs/2402.19173 (2024). doi:10.48550/ARXIV.2402.19173 arXiv:2402.19173
 - [21] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22–27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 6227–6240. doi:10.18653/V1/2022.ACL-LONG.431
 - [22] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and*

- Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html>
- [23] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *CoRR abs/2305.02309* (2023). doi:10.48550/ARXIV.2305.02309 arXiv:2305.02309
 - [24] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=iaYcJKpY2B_
 - [25] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (2009), 333–389. doi:10.1561/15000000019
 - [26] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoming Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR abs/2308.12950* (2023). doi:10.48550/ARXIV.2308.12950 arXiv:2308.12950
 - [27] Disha Shrivastava, Denis Kocetkov, Harm de Vries, Dzmitry Bahdanau, and Torsten Scholak. 2023. RepoFusion: Training Code Models to Understand Your Repository. *CoRR abs/2306.10998* (2023). doi:10.48550/ARXIV.2306.10998 arXiv:2306.10998
 - [28] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguang Huang, and Bin Luo. 2023. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 421–433. doi:10.1109/ASE56229.2023.00076
 - [29] Yanlin Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. RLCoder: Reinforcement Learning for Repository-Level Code Completion. *CoRR abs/2407.19487* (2024). doi:10.48550/ARXIV.2407.19487 arXiv:2407.19487
 - [30] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. doi:10.18653/V1/2021.EMNLP-MAIN.685
 - [31] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoforformer: Selective Retrieval for Repository-Level Code Completion. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. <https://openreview.net/forum?id=moyG54Okjr>
 - [32] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 7443–7464. doi:10.18653/V1/2023.ACL-LONG.411
 - [33] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 2471–2484. doi:10.18653/V1/2023.EMNLP-MAIN.151
 - [34] Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. 2024. CodeGemma: Open Code Models Based on Gemma. *CoRR abs/2406.11409* (2024). doi:10.48550/ARXIV.2406.11409 arXiv:2406.11409