RLCoder: Reinforcement Learning for Repository-Level Code Completion

Yanlin Wang[1], Yanli Wang[1], Daya Guo[1], Jiachi Chen[1]*, Ruikai Zhang[2], Yuchi Ma[2], Zibin Zheng[1]
[1] Sun Yat-sen University, Zhuhai, China
{wangylin36, chenjch86, zhzibin}@mail.sysu.edu.cn, {wangyli58, guody5}@mail2.sysu.edu.cn,
[2] Huawei Cloud Computing Technologies Co., Ltd., Shenzhen, China
{zhangruikai1, mayuchi1}@huawei.com

***Abstract*—Repository-level code completion aims to generate code for unfinished code snippets within the context of a specified repository. Existing approaches mainly rely on retrieval-augmented generation strategies due to limitations in input sequence length. However, traditional lexical-based retrieval methods like BM25 struggle to capture code semantics, while model-based retrieval methods face challenges due to the lack of labeled data for training. Therefore, we propose RLCoder, a novel reinforcement learning framework, which can enable the retriever to learn to retrieve useful content for code completion without the need for labeled data. Specifically, we iteratively evaluate the usefulness of retrieved content based on the perplexity of the target code when provided with the retrieved content as additional context, and provide feedback to update the retriever parameters. This iterative process enables the retriever to learn from its successes and failures, gradually improving its ability to retrieve relevant and high-quality content. Considering that not all situations require information beyond code files and not all retrieved context is helpful for generation, we also introduce a stop signal mechanism, allowing the retriever to decide when to retrieve and which candidates to retain autonomously. Extensive experimental results demonstrate that RLCoder consistently outperforms state-of-the-art methods on CrossCodeEval and RepoEval, achieving 12.2% EM improvement over previous methods. Moreover, experiments show that our framework can generalize across different programming languages and further improve previous methods like RepoCoder. We provide the code and data at https://github.com/DeepSoftwareAnalytics/RLCoder.**

***Index Terms*—Repository-Level Code Completion, Reinforcement Learning, Perplexity, Stop Signal Mechanism**

I. Introduction

With the advancement of large language models for code (code LLMs) [1]–[5], code completion has emerged as one of the most important features in integrated development environments (IDEs) [6]–[11]. However, due to the vast size of code repositories and the limitations of context length in models, repository-level code completion, which involves generating code suggestions within the context of an entire repository, cannot practically leverage the entire repository directly as context [12]. Therefore, previous works [12]–[17] typically employ a retrieval-augmented-generation (RAG) strategy. In this approach, the unfinished code in the current file serves as a query to retrieve code candidates from the entire repository, providing cross-file context. These candidates are then concatenated with the unfinished code before being fed into code LLMs. To retrieve relevant code snippets from other files, various retrievers are adopted. RepoFuse [15] uses lexical-based method BM25 [18] as the retriever to retrieve code snippets that are textually similar with the unfinished code. RepoCoder [13] and RepoHyper [19] use the model-based approach that encodes code candidates and unfinished code into vectors and employs dense retrieval to find similar codes. Although these efforts have shown promising performance in repository-level code generation, we have identified the following problems in retrieval.

- **P1 Labeled Data Dependency.** Lexical-based methods such as BM25 [18] cannot capture code semantics, while model-based methods [12], [19], [20] are capable of understanding code semantics but are hampered by the lack of ground-truth candidate data for training. This labeled data is hard to obtain, as it requires significant effort in data parsing and expert labeling, limiting its generalizability.
- **P2 Candidate Construction Issue.** Previous methods of code candidate construction mainly employ the fixed window strategy [13] or dependency parsing [12], [21]. However, the fixed window strategy may disrupt the continuity of the code. Methods based on dependency parsing can only focus on limited context in the dependency graph and can not be applied to complex scenarios.
- **P3 Non-Selective Retrieval.** Previous studies typically directly retrieve several candidates to serve as the context for generation, neglecting when to retrieve and which candidates to retain. Unnecessary candidates can detract from the performance in completion scenarios that do not require repository context.

In this paper, we propose RLCoder, a reinforcement learning framework for repository-level code completion to address the aforementioned problems. Firstly, we propose a code-base construction pipeline with a simple yet effective Split-Aggregate strategy. This approach allows better code continuity of the candidates, which we refer to as natural candidates (addressing **P2**). Secondly, during the training stage, we diverge from supervised learning methods that depend on labeled data. Instead, we train a retriever named RLRetriever that learns what to retrieve based on feedback from a specifically

* Jiachi Chen is the corresponding author.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00014

designed evaluator, without needing labeled data (addressing **P1**). Specifically, we iteratively evaluate the usefulness of retrieved content based on the perplexity of the target code when provided with the retrieved content as additional context, and provide feedback to update the retriever parameters, which enables the retriever to learn from its successes and failures, gradually improving its ability to retrieve relevant and high-quality content. Moreover, to mitigate hallucinations often observed in repository-level code completions, typically due to incorrect identifier or API usage [16], we design a weighted perplexity (PPL) mechanism that allocates higher weights to certain important tokens in perplexity calculation. Furthermore, considering that not all candidates retrieved are useful for generation, we introduce a stop signal mechanism to evaluate the usefulness of candidates, allowing the retriever to decide when to retrieve and which candidates to retain autonomously (addressing **P3**). Finally, in the inference stage, given an unfinished code as input, RLCoder retrieves natural candidates from the codebase, retains the useful candidates, and then feeds them along with the unfinished code into the generator (a backbone LLM) for target code generation.

We evaluate RLCoder with extensive experiments with several LLMs on CrossCodeEval [22] and RepoEval [13]. Experimental results show that our framework achieves 12.2% improvement of Exact Match compared with previous methods. Furthermore, RLCoder demonstrates high generalizability, showing effectiveness across various LLMs and programming languages. Additionally, experiments show that RLCoder can be integrated into previous methods such as RepoCoder to enhance code completion performance further.

Our main contributions are:

- We propose RLCoder, a reinforcement learning framework for repository-level code completion. To our knowledge, we are the first to train the retriever without labeled data for repository-level code completion. Besides, we design a mechanism that uses the weighted perplexity of the target code as the reward to further enhance performance.
- We introduce a simple yet effective Split-Aggregate candidate construction strategy based on human programming habits. This method avoids the disruption of code continuity and outperforms fixed window candidates indicated by the experimental results.
- We propose a stop signal mechanism to evaluate the usefulness of candidates and discard useless candidates for more effective code completion.
- We perform an extensive evaluation of RLCoder. Experimental results show that RLCoder outperforms the state-of-the-art methods and demonstrates generalizability and applicability.

II. Background

A. Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) [23] is an approach that enhances the quality of generation by retrieving from external knowledge bases. This method includes three key components [24]: *retriever*, *generator*, and *augmentation techniques*. The *retriever* is used to find relevant information from a large-scale dataset or knowledge base, including pertinent documents, facts, or text snippets that are relevant to the input query or prompt. The retrieved information is fed into the *generator*, which integrates this external knowledge into the generation stage. *Augmentation techniques* focus on how retrieved information is integrated into the generation process. To formalize the RAG process, consider a scenario where we want to generate code based on a query q and a set of retrieved candidates $\{c_1, c_2, ..., c_n\}$. The process can be described by the following formula:

$$\text{Code} = \text{Generate}\left(q, \text{Retrieve}(q, \{c_1, c_2, ..., c_n\})\right) \quad (1)$$

where the Retrieve$(\cdot)$ function selects the most relevant candidates based on the query q from the candidate set $\{c_1, c_2, ..., c_n\}$, and the Generate$(\cdot)$ function then takes the query and the retrieved candidates to generate the target code.

In recent years, researchers have conducted a substantial amount of research related to RAG, highlighting its promising potential for future applications [23], [25]–[28]. Many studies have utilized RAG for code-related research [29]–[42]. In repository-level code completion, due to the massive amount of code in the repository and limited context of generator [12], it is impractical to use the entire repository as the context for generation. Therefore, most current methods employ the RAG method to retrieve suitable candidates from the repository for generation [12], [13], [16], [17].

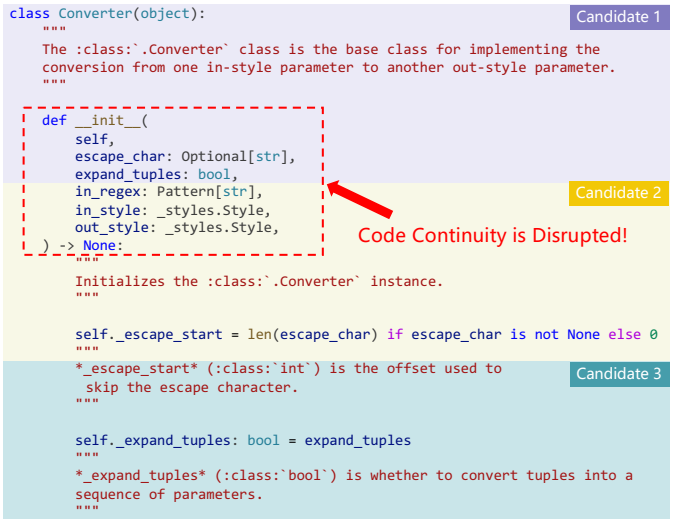

Fig. 1. Using fixed window candidates may disrupt the continuity of code semantics, resulting in the definition of functions being split across different code snippets.

B. Repository-Level Code Completion

Traditional code completion [1], [43] usually focused on generating code with in-file context. With the development of LLMs [3]–[5], [44], repository-level code completion is

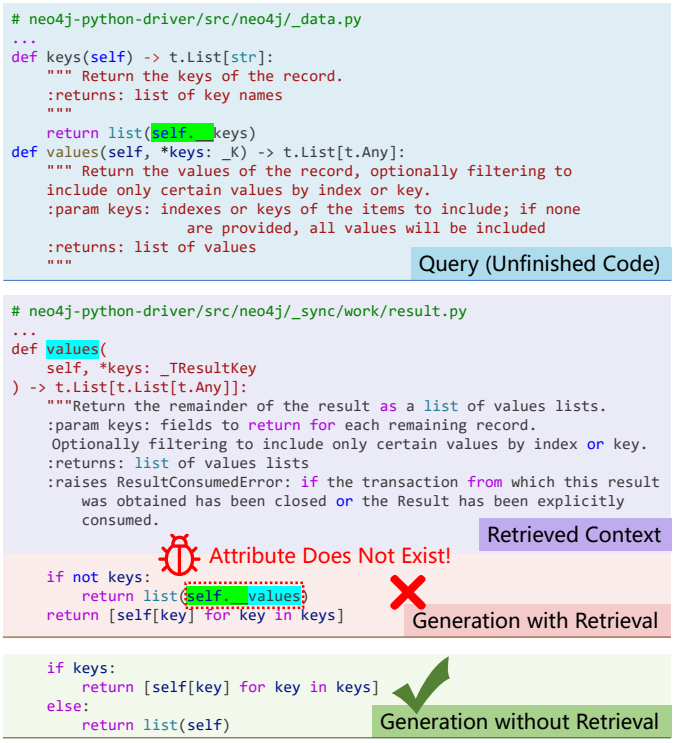

Fig. 2. Due to the limitations of LLMs, inappropriate retrieval can mislead generation, resulting in attempts to call an non-existent attribute.

gradually gaining attention as it better reflects real-world scenarios [12], [13], [22], [45]. To formalize repository-level code completion, we conceptualize the process as selecting the most relevant snippets (candidates) from a code repository and generating code based on the query. This can be encapsulated in a formula as follows:

$$\text{Code} = \text{Generate}(q, \text{Retrieve}(q, \text{codebase})) \tag{2}$$

where q represents the query or the prompt for code completion. Codebase symbolizes code snippets from the code repository. $\text{Retrieve}(q, \text{codebase})$ is the function that selects the most relevant code snippets (candidates) from the repository based on the query q. $\text{Generate}(q, \text{candidates})$ is the generation function that generates the target code based on the query q and the selected candidates.

Previous work highlights the importance of integrating both the in-file and cross-file context in repository-level code completion [12]. This implies that the model needs to understand not only the local context but also third-party libraries and global modules [21]. Fusing analogy context and rationale context can greatly ensure the integrity of the retrieval codebase [15]. Iterative retrieval and generation method [13], [16] involves concatenating the results generated from the previous iteration with the prior context to form a query. This query is then used for the subsequent round of retrieval and generation. Additionally, agents [17], [46]–[48] that assist in code completion through invoking tools or collaborating with each other is also a remarkable approach.

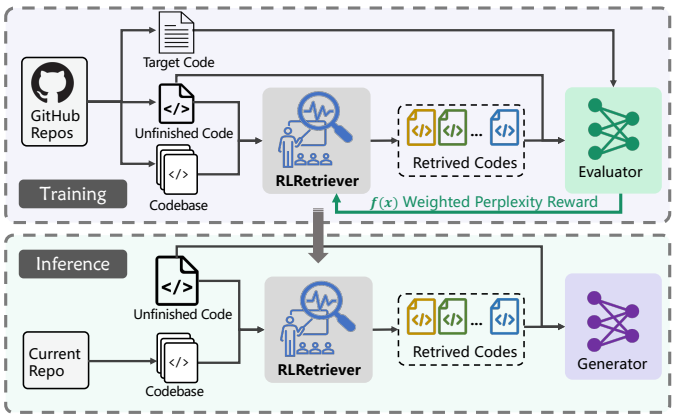

Fig. 3. Overview of RLCoder.

Limitations: There are still some issues that need to be addressed for current repository-level code completion methods. *First*, the lack of labeled data limits the generalizability of many learning-based approaches. For example, CoCoMIC [12] can only be used in trained repositories and struggles to expand to other languages and repositories. RepoHyper [19] uses a subset of the benchmark as training data and sets the gold candidate as the label. *Second*, previous works mostly adopted fixed window candidates [13] or candidates based on dependency parsing [12]. Methods based on dependency parsing only consider the nodes in the dependency parse graph, neglecting other code in the repository. This can lead to omitting many potentially useful code pieces during retrieval. Methods using fixed window candidates, as shown in Figure 1, may split the signature of the "`__init__`" function into two different candidates. This situation may lead to the retriever fetching the required code snippet without capturing the full parameter list. As a result, this partial information could confuse the generator leading to incorrect function calls. *Third*, current work lacks an evaluation of the necessity for candidates. As illustrated in Figure 2, the task can be correctly completed using only the in-file preceding context. However, if the context retrieved is blindly used, it may mislead the generation due to LLM's own capability limitation. In this case, there is a function definition for "`keys`" in the unfinished code, which calls the "`__keys`" attribute. The code snippet retrieved happens to have a function definition for "`values`", leading the model to mistakenly believe there is a corresponding "`__values`" attribute defined, thus calling a non-existent "`__values`" attribute during code generation.

III. Methodology

A. Overview

In this section, we introduce RLCoder, a reinforcement learning framework for repository-level code completion. The overview of RLCoder is shown in Figure 3, comprising two stages: training and inference. In the training stage, the major objective is to train the retriever RLRetriever, the key component of our framework. First, to train RLRetriever, we

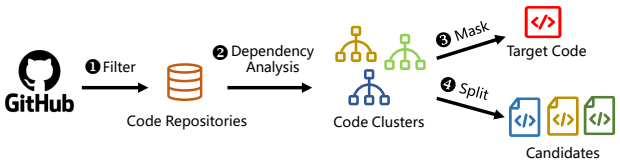

Fig. 4. Data construction pipeline.

construct data from repositories collected from GitHub and obtain unfinished code, target code, and candidate codebase. Then, RLRetriever will retrieve from the candidate codebase using unfinished code as the query. Finally, the retrieved code candidates will be evaluated by the evaluator and obtain the weight perplexity reward to update the parameters of RLRetriever. Through repeated iterations, RLRetriever enhances its retrieval capability via continuous feedback and learning. In the inference stage, given unfinished code and the current repository context, we first construct codebase from current repository. Then, we use the RLRetriever trained in the training stage to retrieve from the codebase using the unfinished code. Finally, we use the retrieved codes as context to concatenate with the unfinished code and feed them into the generator for target code generation.

B. Data Construction

Repository-level code completion tasks typically refer to generating partial code within the given repository code context, generally including a line, an API, or a part of a function body [13]. To ensure the retriever we train meets the requirements of repository-level code retrieval, we need to simulate such scenarios. Figure 4 shows the pipeline of our data construction process, which includes the following steps: repository filtering, dependency analysis, target code selection, and candidate construction.

1) Repository Selection: We randomly select 10,000 large-scale Python and Java repositories from GitHub that were created before March 2023 and meet the following requirements: (1) have cross-file dependencies for constructing our training dataset; and (2) not included in well-known benchmarks such as CrossCodeEval [22] and RepoEval [13], which are used in our evaluation. This filtering process aims at preventing potential data leakage, ensuring the reliability of the evaluation.

2) Dependency Analysis: To ensure our training data contains a substantial quantity of cross-file context dependencies, we implement a dependency analysis for each repository. The methodology is outlined in Algorithm 1. Specifically, to get the code files that are related to each other, we analyze the `import` statements within code files and construct a dependency graph that represents the relationships between these files. Based on whether dependencies exist between code files, we categorize them into clusters of interdependent code files. Through this process, we obtain 27,919 Python file clusters and 41,647 Java file clusters. We eliminate any cluster that contains only a single file. For clusters comprising multiple files, we employ a topological sorting based on the in-degree and out-degree of files. This means that, aside from the first file, each file in the cluster will contain code segments that depend on one or more of other files.

Algorithm 1 Dependency Analysis and Clustering.

Require: Set of code files F
Ensure: Clusters of interdependent code files $Clusters$
  $G \leftarrow \text{ConstructDependencyGraph}(F)$
  $Clusters \leftarrow \text{IdentifyClusters}(G)$
  for all $cluster$ in $Clusters$ **do**
    if $\text{Size}(cluster) == 1$ **then**
      $Clusters \leftarrow Clusters - \{cluster\}$
    else
      $SortedCluster \leftarrow \text{TopologicalSort}(cluster)$
      Update $cluster$ in $Clusters$ with $SortedCluster$
    end if
  end for
  return $Clusters$

3) Target Code Selection: Within the clusters of interdependent code files, we designate files other than the first file as the ones to be completed. We select a random position within these files, excluding the beginning and end to ensure ample context for the code to be completed. This position serves as the starting point for the target code segment that needs completion. To formalize this process, we define the target code segment to be masked and completed as C_{target}, starting from position p_{start} with length l, where p_{start} is chosen randomly within the constraints mentioned above. The selection of p_{start} can be expressed as:

$$p_{start} = \text{Random}(p_{min}, p_{max}) \tag{3}$$

where p_{min} and p_{max} define the permissible range within the file, excluding the very beginning and ending segments to ensure sufficient context. The length l of the target code C_{target} is also determined randomly, with the constraint that the entire segment C_{target} must lie within the boundary of the code file:

$$C_{target} = C[p_{start} : p_{start} + l] \tag{4}$$

After identifying C_{target}, we *mask* this segment within the file to simulate an unfinished code scenario that needs completion. Upon masking C_{target}, the segment designated for completion, we intentionally exclude the file containing the masked code when assembling candidates. To formalize this concept, we define a binary selection function for candidate files as $S(f_i)$, where f_i represents a candidate file:

$$S(f_i) = \begin{cases} 0 & \text{if } C_{target} \in f_i, \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

4) Candidate Construction: Unlike previous works that utilized fixed window candidates [13], [16] or candidates parsed from dependencies [12], we propose a simple yet effective Split-Aggregate candidate construction strategy inspired by human programming habits. We term these candidates as natural candidates. Specifically, programmers often write code with continuous semantic information together, using blank lines as

separators to facilitate readability. As shown in the left part of Figure 5, code and its corresponding comments are usually not separated by blank lines. In fact, blank lines are usually used to separate code snippets with different semantics and usage. This practice naturally forms continuous code segments. The Split-Aggregate strategy is outlined in Algorithm 2. Specifically, we divide the code in a file into several mini-blocks based on blank lines and then aggregate these mini-blocks into candidates by a certain length. During aggregation, the mini-blocks are concatenated to form candidates in such a way that the length of any candidate does not exceed a preset threshold value T.

Algorithm 2 Split-Aggregate Strategy

Require: Code File F, Threshold T
Ensure: Candidate set C

```
Blocks ← SplitIntoBlocks(F)
C ← ∅
for all block in Blocks do
  if LineCount(block) < T then
    Aggregate ← block
    while LineCount(Aggregate) < T and block ≠ Last(Blocks) do
      block ← Next(block)
      Aggregate ← Aggregate + block
    end while
    C ← C ∪ {CreateCandidate(Aggregate)}
  else
    C ← C ∪ SplitBlock(block, T)
  end if
end for
return C
```

C. Reinforcement Learning-based RLRetriever Training

1) Design of Reward: For reinforcement learning, reward is feedback from the external environment that assists a model in learning specific capabilities based on the feedback. In the scenario of repository-level code completion, the most intuitive indicator of reward is whether the generated code can be executed to obtain the expected results. However, obtaining feedback through actual execution is difficult. On the one hand, it's challenging to set up the execution environment for repository code. Even if the execution environment is established, execution can be time-consuming, and there may be a lack of corresponding test cases to evaluate the accuracy of the execution results.

In the context of repository-level code completion, our primary aim is to identify the optimal candidate c from a set of possibilities that maximizes the likelihood of accurately generating the target code sequence y given the contextual information x. This objective can be formally articulated as:

$$\max_{c} P(y|x,c) \tag{6}$$

It is evident that this maximization is equivalent to minimizing the negative log-likelihood:

$$\min_{c} -\log P(y|x,c) \tag{7}$$

Perplexity (PPL), a standard measure for evaluating the predictive performance of probabilistic models, is defined as the exponential of the average negative log-likelihood (NLL) over a sequence. Minimizing NLL thereby directly corresponds to minimizing the perplexity of the target code sequence y:

$$\min_{c} PPL(y|x,c) = e^{-\frac{1}{N}\sum_{i=1}^{N}\log P(y_i|x,c,y_{<i})} \tag{8}$$

where y_i represents the i-th token in the target code sequence y.

In the domain of code completion, the first few tokens generated play a pivotal role in shaping the entire output. Considering this, we give more attention to the first few tokens. Besides, errors in repository-level code completion often occur due to hallucinations caused by a lack of understanding of the entire repository, such as generating incorrect or non-existent APIs. Therefore, we assign a higher focus on the identifier tokens. To further refine the model's focus, we introduce a weighted variant PPL_w:

$$PPL_w(y|x,c) = e^{-\frac{1}{\sum_{i=1}^{N} w_i}\sum_{i=1}^{N} w_i \cdot \log P(y_i|x,c,y_{<i})} \tag{9}$$

The weight w_i for each token of the target code is determined by a function that considers the token's position in the sequence and whether it is an identifier, which can be represented as:

$$w_i = \begin{cases} w_{first} & \text{if } i \le k, \\ w_{api} & \text{if } y_i \in \text{APIs}, \\ 1 & \text{otherwise} \end{cases} \tag{10}$$

where the first k tokens are assigned by a weight w_{first} to reflect their significant impact on the overall quality of the generated code. If the i-th token is part of an API or an identifier, it is assigned a weight w_{api} to acknowledge the importance of accurate and contextually appropriate identifiers in code completion.

We define the reward for choosing a particular candidate c_i, c_j from the set of all candidates C as follows:

$$r(c_i) = \begin{cases} 1 & \text{if } PPL_w(c_i) \le PPL_w(c_j), \forall c_j \in C, \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

where $PPL_w(c_i)$ denotes the weighted perplexity of the target code given candidate c_i, serving as an abbreviation for $PPL_w(y|x,c_i)$. The reward $r(c_i)$, equivalently referred to as $reward(c_i,x,C)$ in the formulations, is assigned a value of 1 if the candidate c_i exhibits a PPL that is equal to or lower than that of any other candidate in the set C. Conversely, a reward of 0 is allocated to c_i if it fails to meet this criterion.

Building on the concept of this reward mechanism, we further define our objective function, $\mathcal{L}$, as an aggregation of the logarithmic probabilities of choosing each candidate,

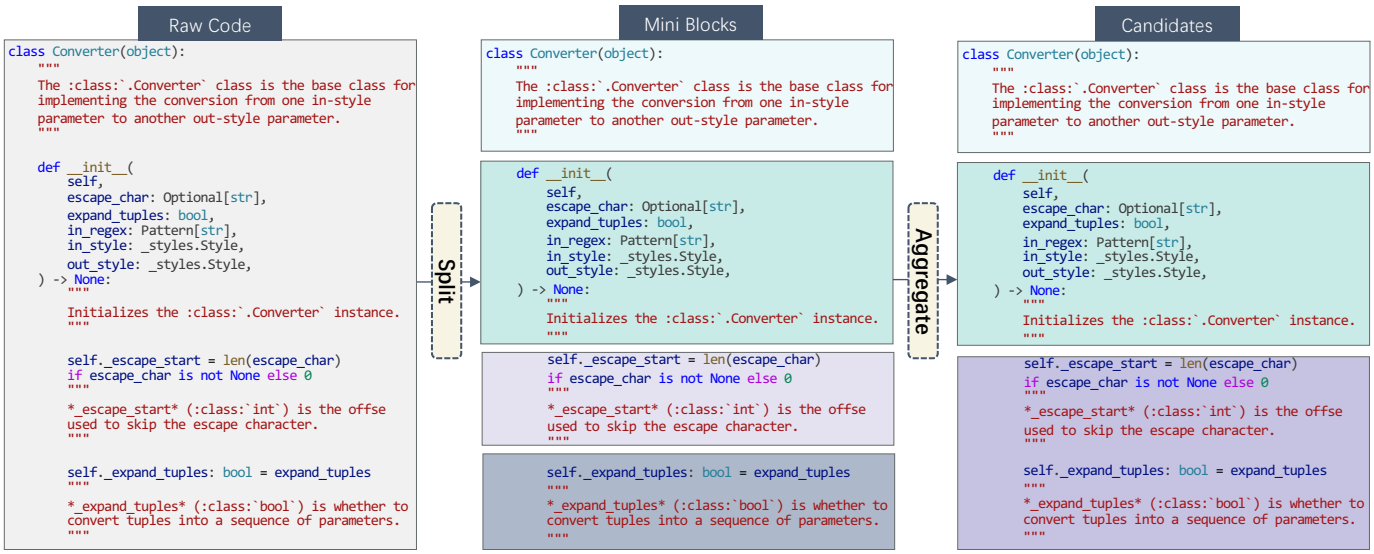

Fig. 5. Candidate construction strategy.

weighted by the corresponding reward. This is mathematically represented as:

$$\mathcal{L} = \sum_{i=1}^{n} (reward(c_i, x, C) \times \log p(c_i|x, C)) \qquad (12)$$

where n is the total number of candidates in set C, and $p(c_i|x, C)$ denotes the probability of selecting candidate c_i given the context x and the set of candidates C.

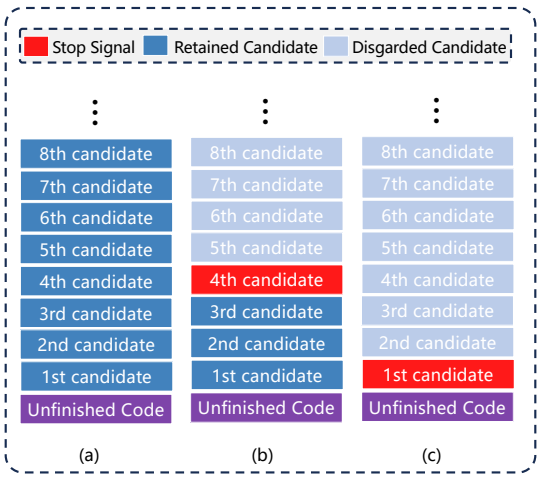

Fig. 6. Stop signal mechanism examples.

2) Stop Signal Mechanism for Candidates Selection: Previous works often overlook when to retrieve and which candidates to retain after retrieval. Specifically, after obtaining the top k candidates, traditional methods simply truncate this list to the top i candidates, determined by a predefined context length. However, this method overlooks the fact that not every retrieved candidate contributes positively to the generation process, and some may even have a negative impact. Therefore, discerning which candidates to retain is crucial for code completion performance.

The core of the stop signal mechanism is the empty candidate. In the retrieval phase, the empty candidate serves as a stop signal and is sent to the retriever together with other ordinary candidates. During training, the evaluator gives each candidate a weighted PPL score to measure the degree of helping generate target code. For candidates ranked below the stop signal, we consider them to be useless or harmful and should not be used; if the stop signal ranked 1st, it means that the generation doesn't require retrieval. During inference, we recall the candidates one by one until encountering the stop signal or reach the maximum context limit. As shown in Figure 6, the stop signal is outside the maximum context in example-(a). The stop signal is ranked 4th in example-(b), which means that only the first three candidates will be retained. The stop signal is ranked 1st in example-(c), which means that all candidates will be disgarded.

3) Learning from Reward: As depicted in Figure 3, RLRetriever is fine-tuned through a dynamic learning process where it receives rewards from the evaluator to update its parameters. This iterative learning process enables RLRetriever to progressively improve its retrieval results, leading to the selection of code candidates with progressively higher quality.

D. Code Completion with RLCoder

As shown in the lower half of Figure 3, during the inference stage, given unfinished code and the current repository context, we first construct the candidate codebase from the current repository using the Split-Aggregate method. Then, we retrieve code candidates using the unfinished code with the trained RLRetriever. Inherently, the stop signal strategy mentioned in Section III-C2 is embedded in the retrieved results to retain only the useful candidates. Finally, the unfinished code,

TABLE I
BENCHMARK STATISTICS.

Benchmark	Category	#Samples	Avg. #Lines	Avg. #Tokens
CrossCodeEval	Python	2665	1.00	14.45
	Java	2139	1.09	16.76
RepoEval	Python (Line)	1600	1.00	15.03
	Python (API)	1600	2.48	34.91

together with these selected candidates, is provided as input to the generator for code completion.

IV. EXPERIMENTAL SETUP

A. Baselines

1) For RLCoder: To evaluate the effectiveness of RLCoder, we compare it with the RawRAG method and RepoCoder framework. Besides, we use a popular dense retriever UniXcoder [49] in these experiments.

- **RawRAG** refers to the standard retrieval and generation approach in the repository-level code completion task. For the unfinished code to generate, RawRAG uses the left context of unfinished code as the query to find the relevant code in the repository to build prompts for generation.
- **RepoCoder** [13] is the state-of-the-art framework for repository-level code completion. It uses an iterative retrieval and generation approach to generate target code.

2) For RLRetriever: To evaluate the effectiveness of RLRetriever, we compare it with the following commonly used retrieval methods in RAG:

- **NoRetrieval** stands for direct generation with unfinished code, without retrieval.
- **BM25** [18] calculates scores for code candidates based on the frequency of query terms in each candidate. It adjusts for candidate length and the average candidate length across the entire database to prevent bias towards longer candidates.
- **UniXcoder** [49] is a dense retriever that encodes both the query and the code snippets into dense vector spaces. This encoding facilitates the identification and retrieval of semantically relevant code snippets from a large corpus based on the similarity of vector representations.
- **UniXcoder-SFT** is a retriever that we trained using supervised fine-tuning of UniXcoder. Due to the lack of labeled data, we use the candidate with the lowest perplexity of target code as the label to fine-tune the retriever.

B. Benchmarks

We evaluate RLCoder on widely used benchmarks for code completion: CrossCodeEval and RepoEval.

- **CrossCodeEval** [22] is a diverse and multilingual code completion benchmark, and we use the Python and Java parts of it.
- **RepoEval** [13] is a benchmark proposed simultaneously with RepoCoder [13]. The benchmark consists of the latest repositories that cover the line-level, API-level, and function-level completion tasks. We use the line-level and API-level tasks among it for evaluation.

Table I shows the statistics of the benchmarks. `#Samples` stands for the number of samples in a benchmark, `Avg. #Lines` and `Avg. #Tokens` stands for the average numbers of lines and tokens of the target code snippets in the benchmark, respectively. Tokens are tokenized by the tokenizer of DeepSeekCoder-1B.

C. Evaluation Metrics

We measure the performance of our approach using the widely used metrics ***Exact Match (EM)*** and ***Edit Similarity (ES)*** [50]. These metrics are widely used in previous code completion studies [12], [13], [16], [22]. EM assesses the precision of code completion by checking if the generated code matches the expected code exactly. It treats the entire code snippet as a single unit. ES measures the similarity between the generated code and the expected code by calculating the edit distance. It reflects the number of edits needed to transform the generated code into the expected code.

D. Experimental Details

All experiments are conducted on a machine with two Tesla A100 GPUs, each with 80 GB memory. In the training stage, we use the parameters of UniXcoder [49] to initialize RLRetriever and use DeepSeekCoder-1B as the evaluator. The batch size is 16 and the learning rate is $5e^{-5}$. We train the model for 20 epochs with 2000 samples per epoch and perform early stopping. In the inference and evaluation stage, we use five different backbone models as the generators.

V. EVALUATION RESULTS

In this section, we report and analyze the experimental results to answer the following research questions (RQs):

- **RQ1:** How effective is RLCoder in repository-level code completion?
- **RQ2:** How effective is RLRetriever compared to other retrieval methods?
- **RQ3:** Does each component of RLCoder contribute to its performance?
- **RQ4:** How is the generalizability of RLCoder?

A. RQ1: Effectiveness of RLCoder

To evaluate the effectiveness of RLCoder, we compare it with the RawRAG framework [35] and RepoCoder [13] with five backbone LLMs, i.e., CodeLlama-7B [3], StartCoder-7B [4], StarCoder2-7B, DeepSeekCoder-1B [5], and DeepSeekCoder-7B. We evalaute the performance on CrossCodeEval [22] and RepoEval [13] benchmarks.

From the experimental results shown in Table II, we can find that the proposed RLCoder demonstrates effectiveness on all backbone language models across the four evaluated datasets, except for RLCoder $_{\text{DeepSeekCoder-7B}}$ evaluated on RepoEval API, where its performance is on par with its corresponding best baseline RepoCoder $_{\text{DeepSeekCoder-7B}}$. We can also observe

TABLE II
PERFORMANCE OF DIFFERENT MODELS. THE SUPERSCRIPTS IN PERCENTAGE DENOTE THE IMPROVEMENT RATIOS OF RLCODER OVER THE CORRESPONDING BEST BASELINE.

Model	CrossCodeEval (Python)		CrossCodeEval (Java)		RepoEval (Line)		RepoEval (API)	
	EM	ES	EM	ES	EM	ES	EM	ES
RawRAG $_{\text{CodeLlama-7B}}$	21.76	69.09	23.42	66.13	42.31	64.35	34.38	61.45
RepoCoder $_{\text{CodeLlama-7B}}$	23.34	70.84	24.17	66.56	43.94	65.81	37.00	63.51
RLCoder $_{\text{CodeLlama-7B}}$	**26.60** $^{\uparrow 14.0\%}$	**72.27** $^{\uparrow 2.0\%}$	**26.23** $^{\uparrow 8.5\%}$	**67.61** $^{\uparrow 1.6\%}$	**46.63** $^{\uparrow 6.1\%}$	**67.92** $^{\uparrow 3.2\%}$	**37.94** $^{\uparrow 2.5\%}$	**64.31** $^{\uparrow 1.3\%}$
RawRAG $_{\text{StarCoder-7B}}$	22.33	69.60	22.16	67.80	43.81	64.83	31.94	56.00
RepoCoder $_{\text{StarCoder-7B}}$	23.15	70.71	22.53	68.22	45.69	66.90	33.44	57.81
RLCoder $_{\text{StarCoder-7B}}$	**25.82** $^{\uparrow 11.5\%}$	**72.11** $^{\uparrow 2.0\%}$	**24.73** $^{\uparrow 9.8\%}$	**69.08** $^{\uparrow 1.3\%}$	**47.38** $^{\uparrow 3.7\%}$	**68.46** $^{\uparrow 2.3\%}$	**34.88** $^{\uparrow 4.3\%}$	**58.11** $^{\uparrow 0.5\%}$
RawRAG $_{\text{StarCoder2-7B}}$	22.89	70.66	23.42	69.13	44.44	65.95	34.50	58.78
RepoCoder $_{\text{StarCoder2-7B}}$	24.35	71.71	23.75	69.59	45.81	67.37	36.44	59.92
RLCoder $_{\text{StarCoder2-7B}}$	**27.17** $^{\uparrow 11.6\%}$	**73.24** $^{\uparrow 2.1\%}$	**26.23** $^{\uparrow 10.4\%}$	**70.51** $^{\uparrow 1.3\%}$	**48.25** $^{\uparrow 5.3\%}$	**68.61** $^{\uparrow 1.8\%}$	**38.00** $^{\uparrow 4.3\%}$	**61.21** $^{\uparrow 2.2\%}$
RawRAG $_{\text{DeepSeekCoder-1B}}$	19.74	67.68	18.89	62.47	39.31	62.04	33.00	60.41
RepoCoder $_{\text{DeepSeekCoder-1B}}$	20.23	68.78	19.59	62.35	40.88	63.56	35.13	61.92
RLCoder $_{\text{DeepSeekCoder-1B}}$	**23.98** $^{\uparrow 18.5\%}$	**70.44** $^{\uparrow 2.4\%}$	**20.80** $^{\uparrow 6.2\%}$	**63.39** $^{\uparrow 1.7\%}$	**44.19** $^{\uparrow 8.1\%}$	**66.48** $^{\uparrow 4.6\%}$	**36.06** $^{\uparrow 2.6\%}$	**62.72** $^{\uparrow 1.3\%}$
RawRAG $_{\text{DeepSeekCoder-7B}}$	23.30	70.84	22.49	66.78	45.69	66.67	38.00	65.66
RepoCoder $_{\text{DeepSeekCoder-7B}}$	26.98	72.96	24.96	66.52	46.38	67.51	39.31	**66.29**
RLCoder $_{\text{DeepSeekCoder-7B}}$	**30.28** $^{\uparrow 12.2\%}$	**74.42** $^{\uparrow 2.0\%}$	**26.09** $^{\uparrow 4.5\%}$	**67.31** $^{\uparrow 1.2\%}$	**48.75** $^{\uparrow 5.1\%}$	**69.43** $^{\uparrow 2.8\%}$	**39.88** $^{\uparrow 1.5\%}$	66.22 $^{\uparrow -0.1\%}$

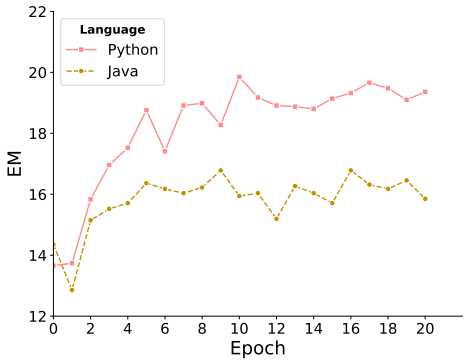

Fig. 7. Performance trajectory curve during the training process.

that among all models, RLCoder $_{\text{DeepSeekCoder-7B}}$achieves the best performance with EM score of 30.28, improving its corresponding best baseline RepoCoder $_{\text{DeepSeekCoder-7B}}$ by 12.2% on CrossCodeEval Python and 5.1% on RepoEval Line.

Furthermore, to investigate the efficacy of our training process, we plot the performance trajectory curve across the training epochs on CrossCodeEval, as illustrated in Figure 7. The result shows that the EM score gradually increases with each epoch until stabilizing, indicating the effectiveness of our training process.

RQ1 Summary: Our approach significantly outperforms current state-of-the-art methods for all backbone LLMs, improving the CrossCodeEval benchmark by 12.2% and RepoEval 5.1%. The performance trajectory further demonstrates the efficacy of our training process.

B. RQ2: Effectiveness of RLRetriever

To assess the effectiveness of RLRetriever, the key module of RLCoder, we conduct a comparative study. We evaluate RLCoder equipped with different retrieval methods described in Section IV-A, including NoRetrieval, BM25 [18], UniXcoder [49], and our enhanced model UniXcoder-SFT. Table III shows the experimental results on the CrossCodeEval and RepoEval benchmarks. The results yield the following findings:

- Our proposed RLRetriever consistently outperforms comparative baseline methods under all metrics in both benchmarks, underscoring its superior performance.
- All retrieval-based methods (i.e., BM25, UniXCoder, UniXcoder-SFT, and RLRetriever) perform better than NoRetrieval, showing the inherent value of the retrieval process itself.
- Both UniXcoder-SFT and RLRetriever show better performance than UniXcoder, indicating that retrieval training can enhance the performance. Notably, our reinforcement learning-based training method exhibits better performance over supervised fine-tuning.

We proposed the perplexity-based feedback, because it's both lightweight and can simulate the finetuning objectives of code completion. As shown in Figure 8, we have plotted the scores of EM, ES, and Perplexity (PPL) for DeepSeekCoder-7B on two datasets. It can be observed that lower PPL generally corresponds to higher EM and ES metrics.

RQ2 Summary: RLRetriever consistently outperforms other retrieval methods. Furthermore, the results affirms the significance of the retrieval and training processes, particularly highlighting the advantages of our reinforcement learning-based training approach.

C. RQ3: Contributions of Each Component

To understand the contributions of each component to RLCoder, we conduct an ablation study on RLCoder. Specifically, we remove each component of RLCoder each time

TABLE III

EXPERIMENTAL RESULTS OF RLCODER EQUIPPED WITH DIFFERENT RETRIEVAL METHODS. THE BACKBONE LLM USED IS DEEPSEEKCODER-7B. THE SUPERSCRIPTS IN PERCENTAGE DENOTE THE IMPROVEMENT RATIOS OF OUR RETRIEVAL MODEL RLRETRIEVER OVER THE CORRESPONDING BEST BASELINE RETRIEVER.

Retrieval Method	CrossCodeEval (Python)		CrossCodeEval (Java)		RepoEval (Line)		RepoEval (API)	
	EM	ES	EM	ES	EM	ES	EM	ES
NoRetrieval	9.46	62.79	11.41	63.81	39.63	61.95	30.44	59.40
BM25	18.31	68.38	17.48	65.23	45.94	66.67	38.25	65.04
UniXcoder	23.30	70.84	22.49	66.78	45.69	66.67	38.00	65.66
UniXcoder-SFT	27.28	72.90	25.11	66.39	46.75	67.28	37.69	65.00
RLRetreiver	**30.28** $^{\uparrow 11.0\%}$	**74.42** $^{\uparrow 2.1\%}$	**26.09** $^{\uparrow 3.9\%}$	**67.31** $^{\uparrow 1.4\%}$	**48.75** $^{\uparrow 4.3\%}$	**69.43** $^{\uparrow 3.2\%}$	**39.88** $^{\uparrow 5.8\%}$	**66.22** $^{\uparrow 1.9\%}$

TABLE IV

ABLATION STUDY RESULTS ON CROSSCODEEVAL AND REPOEVAL.

Model	CrossCodeEval (Python)		CrossCodeEval (Java)		RepoEval (Line)		RepoEval (API)	
	EM	ES	EM	ES	EM	ES	EM	ES
RLCoder	30.28	74.42	26.09	67.31	48.75	69.43	39.88	66.22
w/o RL	23.30 $^{\downarrow 23.1\%}$	70.84 $^{\downarrow 4.8\%}$	22.49 $^{\downarrow 13.8\%}$	66.78 $^{\downarrow 0.8\%}$	45.69 $^{\downarrow 6.3\%}$	66.67 $^{\downarrow 4.0\%}$	38.00 $^{\downarrow 4.7\%}$	65.66 $^{\downarrow 0.8\%}$
w/o WP	27.35 $^{\downarrow 9.7\%}$	72.82 $^{\downarrow 2.1\%}$	25.67 $^{\downarrow 1.6\%}$	67.43 $^{\uparrow 0.2\%}$	47.44 $^{\downarrow 2.7\%}$	67.83 $^{\downarrow 2.3\%}$	38.81 $^{\downarrow 2.7\%}$	65.25 $^{\downarrow 1.5\%}$
w/o NC	29.31 $^{\downarrow 3.2\%}$	73.91 $^{\downarrow 0.7\%}$	24.03 $^{\downarrow 7.9\%}$	66.49 $^{\downarrow 1.2\%}$	47.13 $^{\downarrow 3.3\%}$	68.11 $^{\downarrow 1.9\%}$	38.63 $^{\downarrow 3.1\%}$	65.56 $^{\downarrow 1.0\%}$
w/o SS	29.57 $^{\downarrow 2.34\%}$	74.49 $^{\uparrow 0.09\%}$	25.57 $^{\downarrow 1.99\%}$	67.42 $^{\uparrow 0.16\%}$	47.31 $^{\downarrow 2.95\%}$	68.23 $^{\downarrow 1.73\%}$	39.63 $^{\downarrow 0.63\%}$	65.87 $^{\downarrow 0.53\%}$

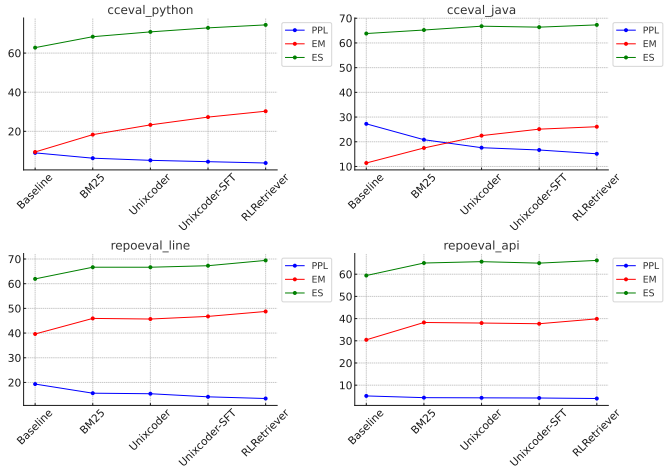

Fig. 8. The relationship between PPL and EM, ES: lower PPL corresponds to higher EM and ES.

TABLE V

ADDITIONAL ABLATION STUDY FOR THE STOP SIGNAL MECHANISM. NOTE THAT, CONTRARY TO EM AND ES, LOWER PPL SCORES CORRESPOND TO BETTER PERFORMANCE.

Model	EM	ES	PPL
RawRAG $_{\text{CodeLlama-7B}}$	10.3	65.2	2.1389
RLCoder $_{\text{CodeLlama-7B}}$	11.1	66.4	2.0835
w/o Stop Signal	9.9 $^{\downarrow 10.81\%}$	66.1 $^{\downarrow 0.45\%}$	2.1152 $^{\uparrow 1.52\%}$
RawRAG $_{\text{StarCoder-7B}}$	8.9	59.2	2.7400
RLCoder $_{\text{StarCoder-7B}}$	10.1	61.3	2.6695
w/o Stop Signal	9.1 $^{\downarrow 9.90\%}$	60.1 $^{\downarrow 1.96\%}$	2.7100 $^{\uparrow 1.52\%}$
RawRAG $_{\text{StarCoder2-7B}}$	9.1	60.5	2.6422
RLCoder $_{\text{StarCoder2-7B}}$	10.3	60.2	2.5716
w/o Stop Signal	9.3 $^{\downarrow 9.71\%}$	60.5 $^{\uparrow 0.50\%}$	2.6115 $^{\uparrow 1.55\%}$
RawRAG $_{\text{DeepSeekCoder-1B}}$	9.6	64.9	2.4370
RLCoder $_{\text{DeepSeekCoder-1B}}$	10.5	66.7	2.3764
w/o Stop Signal	10.0 $^{\downarrow 4.76\%}$	65.9 $^{\downarrow 1.20\%}$	2.4138 $^{\uparrow 1.57\%}$
RawRAG $_{\text{DeepSeekCoder-7B}}$	11.5	66.6	2.3334
RLCoder $_{\text{DeepSeekCoder-7B}}$	12.2	68.6	2.2824
w/o Stop Signal	11.8 $^{\downarrow 3.28\%}$	67.9 $^{\downarrow 1.02\%}$	2.3157 $^{\uparrow 1.46\%}$

and study the performance of the ablated model. The experimental results are shown in Table IV. "w/o RL" means using retriever without reinforcement learning. "w/o WP" means utilizing unweighted perplexity of the target code as the reward. "w/o NC" means using fixed window candidates instead of our natural candidates, "w/o SS" means using retriever without the stop signal mechanism. From Table IV, we can see that the performance of the model drops after removing any one component, indicating that each component contributes to the effectiveness of RLCoder. Especially, the performance drops the most significantly for "RLCoder w/o RL", indicating that the reinforcement learning mechanism is the most important component in RLCoder. We observe that unweighted perplexity and no stop mechanism can be beneficial to ES performance in some cases, but still harm EM performance. In fact, ES mainly considers the similarity between two pieces of code. The introduction of the stop signal and weighted perplexity can both affect code similarity. The stop signal reduces useless but similar candidates, while weighted perplexity emphasizes API tokens more rather than all tokens. This can potentially reduce the similarity between generated and target code. Although these strategies weaken similarity, they improve code correctness. So removing the stop signal and weighted perplexity decreases in EM across all benchmarks.

Since CrossCodeEval and RepoEval are specifically curated

to evaluate code completion ability in scenarios requiring cross-file context, the improvement brought by the stop signal is expected to be minor. To further evaluate the practical effectiveness of the stop signal mechanism, we construct a new dataset GitHubEval that construct code completion targets at random positions within repositories, thus incorporating instances that may not necessitate cross-file context. Following the same construction procedure as the training dataset, we obtain 1000 samples for evaluation. The results shown in Table V indicate that the stop signal is an important component in RLCoder, without which, the EM scores drop significantly by an average of 7.69%.

RQ3 Summary: Reinforcement learning mechanism is the most important component in RLCoder. Other components of RLCoder also contributes to its superior performance, with the stop signal mechanism showing further enhancements in scenarios involving target code completion that both require and do not require cross-file context.

D. RQ4: Generalizability of RLCoder

To explore the generalizability of RLCoder, we conduct an evaluation with a setting different from the training stage. Specifically, we train a new retriever by fusing RLCoder and RepoCoder. Then, we evaluate the performance of the fused model. As shown in Table VI, we find that RepoCoder trained using the framework of RLCoder significantly outperforms the original RepoCoder method. Specifically, the improvement rates in EM for Python and Java are 12.4% and 8.1% on CrossCodeEval, respectively. This result indicates that our training framework can be integrated into other models to further improve their performance. Note that when comparing "RepoCoder w/ RLCoder" to RLCoder, they have comparable performance (with "RepoCoder w/ RLCoder" slightly better). However, considering that RepoCoder requires multiple iterations of retrieval and generation, we opt for RLCoder, which accomplishes code completion in a single round, as the default setting in this work.

RQ4 Summary: The training pipeline of RLCoder shows generalizability on all datasets in applying to other frameworks.

E. Case Study

We illustrate the effectiveness of RLCoder through a case study presented in Figure 9. The left-hand side shows the incomplete code, groundtruth code, and the gold candidate we labelled for this case. We can see that RLCoder ranks the gold candidate as the first candidate and generates the correct code. A possible reason that other methods fail to retrieve the correct code is that these methods rely on the surface-level similarity between the query and the candidate. The bad candidate, despite sharing several tokens with the query (as highlighted in the figure), does not contribute meaningfully to the correct code completion. In contrast to RepoCoder, which iteratively uses generated code for retrieval and generation but still depends heavily on query-candidate similarity, RLCoder leverages the perplexity of generating target code from a given candidate. This approach enables RLRetriever to bypass candidates that are *seemingly useful but actually useless*, focusing instead on those more likely to aid in accurate code generation. This strategic prioritization explains RLCoder's success in both retrieving the gold candidate and generating the correct target code.

VI. Related Work

A. Code Completion

Code completion, recognized as one of the most crucial tasks in modern integrated development environments (IDEs), has garnered significant attention from researchers [51]–[55]. Traditional approaches to code completion typically rely on rule-based methods or leverage code examples to predict the next sequence of code [56]–[58]. While these methods have shown some effectiveness, they often struggle to adapt to the complexities and nuances of real-world programming scenarios. In recent years, deep learning-based methods [9], [10], [54], [59]–[71] have been explored to improve the performance of code completion. Recent studies found that code search can enhance code completion performance [72]. With the development of large language models [73]–[82], many researchers have introduced LLMs into code completion [83]–[87]. Equipped with LLMs, many studies have employed RAG for code completion/generation [32], [34]–[37], [40]. For example, RedCoder [35] enhances code generation and summarization by integrating relevant past work using dense retrieval techniques. To enhance private library code generation, APICoder [37] was proposed to employ API documentation to train models to better generate these libraries. DocPrompting [36] introduces a method to enhance code generation by using code documentation to address the challenge of generating code for unseen functions and libraries. AceCoder [34] improves code generation by integrating example retrieval and guided generation. ReCode [40] improves neural code generation by incorporating subtree retrieval from existing code examples. kNN-TRANX [32] improves code generation from natural language by using syntax-aware retrieval, reducing noise and computational time.

B. Repository-Level Code Completion

Repository-level code completion, which leverages the broader context of an entire code repository, has become a focal point for research in the field of code completion and many studies have attempted to improve repository-level code completion performance [12]–[17], [19], [21], [48]. CoCoMIC [12] and RepoHyper [19] enhance code completion capabilities

TABLE VI
EXPERIMENTAL RESULTS OF REPOCODER INTEGRATED WITH RLCODER.

Method	CrossCodeEval (Python)		CrossCodeEval (Java)		RepoEval (Line)		RepoEval (API)	
	EM	ES	EM	ES	EM	ES	EM	ES
RawRAG	23.30	70.84	22.49	66.78	45.69	66.67	38.00	65.66
RLCoder	30.28	74.42	26.09	67.31	48.75	69.43	39.88	66.22
RepoCoder	26.98	72.96	24.96	66.52	46.38	67.51	39.31	66.29
w/ RLCoder	**30.32** ↑12.4%	**74.79** ↑2.5%	**26.98** ↑8.1%	**67.81** ↑1.9%	**49.44** ↑6.6%	**69.76** ↑3.3%	**41.25** ↑4.9%	**67.08** ↑1.2%

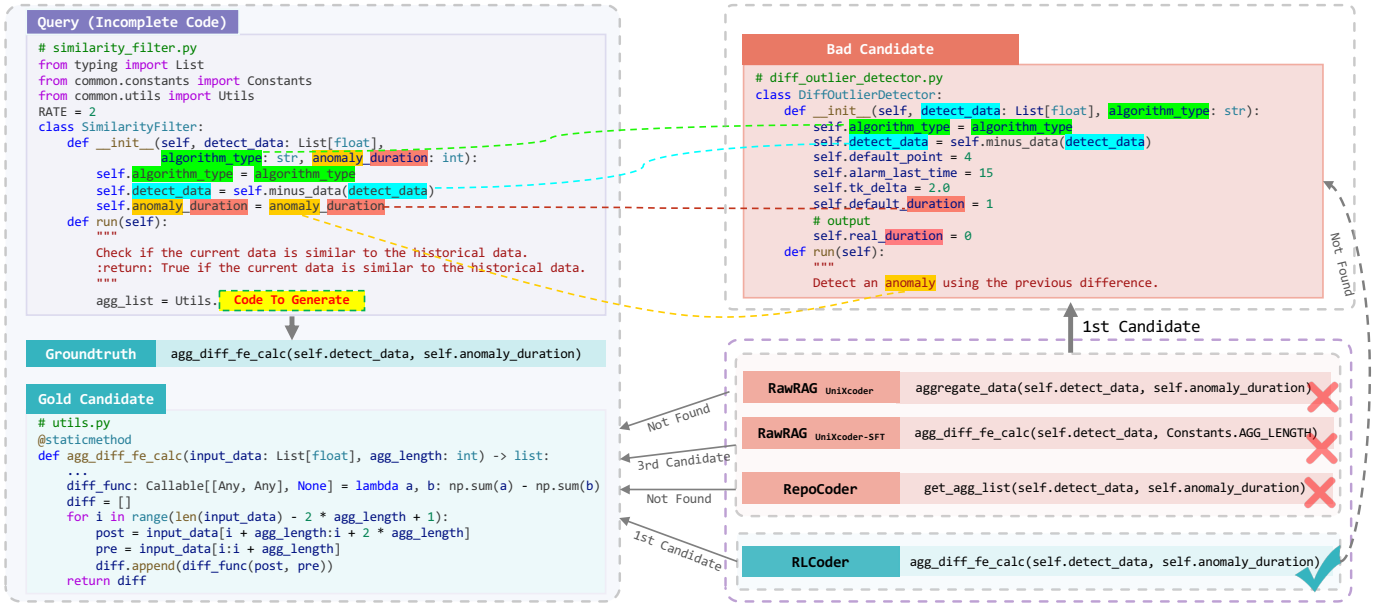

Fig. 9. Case study. An example sources from CrossCodeEval with the *task_id* being *project_cc_python/210*. The highlight token represents the identical tokens exists in both the query and the bad candidate.

through dependency analysis and learning-based methods but they encounter the problem of difficulty in obtaining training data and poor generalizability. CodePlan [14], RepoFuse [15] and A^3-CodeGen [21] employ static code analysis to obtain relevant candidates. RepoCoder [13] and De-Hallucinator [16] adopt an approach through iterative retrieval and generation. CodeAgent [17] and ToolGen [48] explore tool invocation to help code completion.

Although these efforts show promising performance, there are still many areas for improvement. Existing methods often overlook the importance of retriever, leading to the retrieved candidates not being needed for generation. Additionally, the lack of training data makes it difficult to fine-tune the retriever. RLCoder differs from them in that it does not require labeled data for training. Instead, it obtains feedback from the generator as a reward for training. Besides, to evaluate the usefulness of candidates, RLCoder uses a novel stop signal mechanism to determine when to retrieve and which candidates to retain. Furthermore, we propose a new candidate construction strategy for repository code.

VII. CONCLUSION

In this paper, we propose RLCoder, a novel reinforcement learning framework for repository-level code completion. We enable the retriever to learn iteratively by obtaining feedback from the evaluator. Besides, unlike using fixed window candidates or candidates parsing from dependency, we introduce a simple yet effective Split-Aggregate candidate construction method based on human programming habits. Moreover, we propose the stop signal to avoid using useless cross-file context. Experimental results indicate that RLCoder is capable of ignoring those *seemingly useful but actually useless* candidates, capturing ones that are beneficial for accurate code generation. This feature lets RLCoder achieve state-of-the-art performance on repository-level code completion. Besides, RLCoder demonstrates good generalizability and applicability to further enhancing existing methods.

ACKNOWLEDGEMENTS

The work described in this paper is supported by CCF-Huawei Populus Grove Fund CCF-HuaweiSE202301.

REFERENCES

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[3] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[4] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[5] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[6] F. Liu, Z. Fu, G. Li, Z. Jin, H. Liu, Y. Hao, and L. Zhang, "Non-autoregressive line-level code completion," *ACM Transactions on Software Engineering and Methodology*, 2024.

[7] C. Wang, J. Hu, C. Gao, Y. Jin, T. Xie, H. Huang, Z. Lei, and Y. Deng, "How practitioners expect code completion?" in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1294–1306.

[8] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: A case study on real-world completions," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 960–970.

[9] M. Izadi, R. Gismondi, and G. Gousios, "Codefill: Multi-token code completion by jointly learning from structure and naming sequences," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 401–412.

[10] W. Zhou, S. Kim, V. Murali, and G. A. Aye, "Improving code autocompletion with transfer learning," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 161–162.

[11] M. Izadi, J. Katzy, T. van Dam, M. Otten, R. M. Popescu, and A. van Deursen, "Language models for code completion: A practical evaluation," *arXiv preprint arXiv:2402.16197*, 2024.

[12] Y. Ding, Z. Wang, W. U. Ahmad, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang, "Cocomic: Code completion by jointly modeling in-file and cross-file context," 2023.

[13] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," 2023.

[14] R. Bairi, A. Sonwane, A. Kanade, A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, S. Shet *et al.*, "Codeplan: Repository-level coding using llms and planning," *arXiv preprint arXiv:2309.12499*, 2023.

[15] M. Liang, X. Xie, G. Zhang, X. Zheng, P. Di, H. Chen, C. Wang, G. Fan *et al.*, "Repofuse: Repository-level code completion with fused dual context," *arXiv preprint arXiv:2402.14323*, 2024.

[16] A. Eghbali and M. Pradel, "De-hallucinator: Iterative grounding for llm-based code completion," *arXiv preprint arXiv:2401.01701*, 2024.

[17] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, "Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," *arXiv preprint arXiv:2401.07339*, 2024.

[18] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[19] H. N. Phan, H. N. Phan, T. N. Nguyen, and N. D. Bui, "Repohyper: Better context retrieval is all you need for repository-level code completion," *arXiv preprint arXiv:2403.06095*, 2024.

[20] D. Wu, W. U. Ahmad, D. Zhang, M. K. Ramanathan, and X. Ma, "Repoformer: Selective retrieval for repository-level code completion," *arXiv preprint arXiv:2403.10059*, 2024.

[21] D. Liao, S. Pan, Q. Huang, X. Ren, Z. Xing, H. Jin, and Q. Li, "Context-aware code generation framework for code repositories: Local, global, and third-party library awareness," *arXiv preprint arXiv:2312.05772*, 2023.

[22] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[23] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023.

[24] P. Zhao, H. Zhang, Q. Yu, Z. Wang, Y. Geng, F. Fu, L. Yang, W. Zhang, and B. Cui, "Retrieval-augmented generation for ai-generated content: A survey," *arXiv preprint arXiv:2402.19473*, 2024.

[25] B. Cao, D. Cai, L. Cui, X. Cheng, W. Bi, Y. Zou, and S. Shi, "Retrieval is accurate generation," *arXiv preprint arXiv:2402.17532*, 2024.

[26] Z. He, Z. Zhong, T. Cai, J. D. Lee, and D. He, "Rest: Retrieval-based speculative decoding," *arXiv preprint arXiv:2311.08252*, 2023.

[27] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462.

[28] Y. Liu, S. Yavuz, R. Meng, D. Radev, C. Xiong, and Y. Zhou, "Uniparser: Unified semantic parser for question answering on knowledge base and database," *arXiv preprint arXiv:2211.05165*, 2022.

[29] S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," *arXiv preprint arXiv:2203.07722*, 2022.

[30] C. Yu, G. Yang, X. Chen, K. Liu, and Y. Zhou, "Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 82–93.

[31] J. A. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 155–166.

[32] X. Zhang, Y. Zhou, G. Yang, and T. Chen, "Syntax-aware retrieval augmented code generation," in *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[33] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1385–1397.

[34] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, "Acecoder: Utilizing existing code to enhance code generation," *arXiv preprint arXiv:2303.17780*, 2023.

[35] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," *arXiv preprint arXiv:2108.11601*, 2021.

[36] S. Zhou, U. Alon, F. F. Xu, Z. Wang, Z. Jiang, and G. Neubig, "Docprompting: Generating code by retrieving the docs," *arXiv preprint arXiv:2207.05987*, 2022.

[37] D. Zan, B. Chen, Z. Lin, B. Guan, Y. Wang, and J.-G. Lou, "When language model meets private library," *arXiv preprint arXiv:2210.17236*, 2022.

[38] A. Madaan, S. Zhou, U. Alon, Y. Yang, and G. Neubig, "Language models of code are few-shot commonsense learners," *arXiv preprint arXiv:2210.07128*, 2022.

[39] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[40] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," *arXiv preprint arXiv:1808.10025*, 2018.

[41] N. Beau and B. Crabbé, "The impact of lexical and grammatical processing on generating code from natural language," *arXiv preprint arXiv:2202.13972*, 2022.

[42] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 1004–1004.

[43] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[44] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[45] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the*

46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–12.

[46] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023.

[47] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.

[48] C. Wang, J. Zhang, Y. Feng, T. Li, W. Sun, Y. Liu, and X. Peng, "Teaching code llms to use autocompletion tools in repository-level code generation," *arXiv preprint arXiv:2401.06391*, 2024.

[49] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[50] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[51] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[52] J. Chen, C. Chen, J. Hu, J. Grundy, Y. Wang, T. Chen, and Z. Zheng, "Identifying smart contract security issues in code snippets from stack overflow," *arXiv preprint arXiv:2407.13271*, 2024.

[53] Y. Wang, T. Jiang, M. Liu, J. Chen, and Z. Zheng, "Beyond functional correctness: Investigating coding style inconsistencies in large language models," *arXiv preprint arXiv:2407.00456*, 2024.

[54] Y. Wang and H. Li, "Code completion by modeling flattened abstract syntax trees as graphs," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 16, 2021, pp. 14 015–14 023.

[55] W. Tao, Y. Zhou, Y. Wang, H. Zhang, H. Wang, and W. Zhang, "Kadel: Knowledge-aware denoising learning for commit message generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[56] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2009, pp. 213–222.

[57] D. Hou and D. M. Pletcher, "Towards a better code completion system by api grouping, filtering, and popularity-based ranking," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 26–30.

[58] R. Robbes and M. Lanza, "How program history can improve code completion," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 317–326.

[59] Y. Chen, C. Gao, X. Ren, Y. Peng, X. Xia, and M. R. Lyu, "Api usage recommendation via multi-view heterogeneous graph representation learning," *IEEE Transactions on Software Engineering*, 2023.

[60] C. Wang, X. Peng, M. Liu, Z. Xing, X. Bai, B. Xie, and T. Wang, "A learning-based approach for automatic construction of domain glossary from source code and documentation," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 97–108.

[61] M. Liu, X. Peng, A. Marcus, Z. Xing, W. Xie, S. Xing, and Y. Liu, "Generating query-specific class api summaries," in *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 120–130.

[62] X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu, "Compilable neural code generation with compiler feedback," *arXiv preprint arXiv:2203.05132*, 2022.

[63] G. A. Aye and G. E. Kaiser, "Sequence model design for code completion in the modern ide," *arXiv preprint arXiv:2004.05249*, 2020.

[64] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint meeting on foundations of software engineering*, 2017, pp. 763–773.

[65] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1073–1085.

[66] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[67] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[68] S. Nguyen, T. Nguyen, Y. Li, and S. Wang, "Combining program analysis and statistical language model for code statement completion," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 710–721.

[69] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1433–1443.

[70] F. Wen, E. Aghajani, C. Nagy, M. Lanza, and G. Bavota, "Siri, write the next method," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 138–149.

[71] Y. Yang, Y. Jiang, M. Gu, J. Sun, J. Gao, and H. Liu, "A language model for statements of software code," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 682–687.

[72] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code search is all you need? improving code suggestions with code search," *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024.

[73] J. Lu, W. Zhong, Y. Wang, Z. Guo, Q. Zhu, W. Huang, Y. Wang, F. Mi, B. Wang, Y. Wang *et al.*, "Yoda: Teacher-student progressive learning for language models," *arXiv preprint arXiv:2401.15670*, 2024.

[74] F. Hu, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and X. Li, "Split, encode and aggregate for long code search," *arXiv preprint arXiv:2208.11271*, 2022.

[75] Y. Liu, J. Chen, T. Bi, J. Grundy, Y. Wang, T. Chen, Y. Tang, and Z. Zheng, "An empirical study on low code programming using traditional vs large language model support," *arXiv preprint arXiv:2402.01156*, 2024.

[76] Y. Wang, Y. Huang, D. Guo, H. Zhang, and Z. Zheng, "Sparsecoder: Identifier-aware sparse transformer for file-level code summarization," *arXiv preprint arXiv:2401.14727*, 2024.

[77] J. Zhou, W. Zhong, Y. Wang, and J. Wang, "Adaptive-solver framework for dynamic strategy selection in large language model reasoning," *arXiv preprint arXiv:2310.01446*, 2023.

[78] E. Shi, F. Zhang, Y. Wang, B. Chen, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Sotana: The open-source software development assistant," *arXiv preprint arXiv:2308.13416*, 2023.

[79] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv preprint arXiv:2311.10372*, 2023.

[80] Z. Zheng, K. Ning, J. Chen, Y. Wang, W. Chen, L. Guo, and W. Wang, "Towards an understanding of large language models in software engineering tasks," *arXiv preprint arXiv:2308.11396*, 2023.

[81] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *arXiv preprint arXiv:2309.05520*, 2023.

[82] W. Zhong, R. Cui, Y. Guo, Y. Liang, S. Lu, Y. Wang, A. Saied, W. Chen, and N. Duan, "Agieval: A human-centric benchmark for evaluating foundation models," *arXiv preprint arXiv:2304.06364*, 2023.

[83] M. Liu, Y. Yang, Y. Lou, X. Peng, Z. Zhou, X. Du, and T. Yang, "Recommending analogical apis via knowledge graph embedding," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1496–1508.

[84] X. Jiang, Y. Dong, Z. Jin, and G. Li, "Seed: Customize large language models with sample-efficient adaptation for code generation," *arXiv preprint arXiv:2403.00046*, 2024.

[85] B. Li, Z. Sun, T. Huang, H. Zhang, Y. Wan, G. Li, Z. Jin, and C. Lyu, "Ircoco: Immediate rewards-guided deep reinforcement learning for code completion," *arXiv preprint arXiv:2401.16637*, 2024.

[86] Y. Zhu, J. A. Li, G. Li, Y. Zhao, J. Li, Z. Jin, and H. Mei, "Improving code generation by dynamic temperature sampling," *arXiv preprint arXiv:2309.02772*, 2023.

[87] L. Guo, Y. Wang, E. Shi, W. Zhong, h. Zhang, j. Chen, R. Zhang, y. Ma, and Z. Zheng, "When to stop? towards efficient code generation in llms with excess token prevention," in *Proceedings of the 33st ACM SIGSOFT international symposium on software testing and analysis*, 2024.