AlignCoder: Aligning Retrieval with Target Intent for Repository-Level Code Completion

Tianyue Jiang[1†], Yanli Wang[1†], Yanlin Wang[1*], Daya Guo[3], Ensheng Shi[2], Yuchi Ma[2], Jiachi Chen[1], Zibin Zheng[1]
[1] Sun Yat-sen University, Zhuhai, China
[2] Huawei Cloud Computing Technologies Co., Ltd., Shenzhen, China
[3] Independent Researcher, China

***Abstract*—Repository-level code completion remains a challenging task for existing code large language models (code LLMs) due to their limited understanding of repository-specific context and domain knowledge. While retrieval-augmented generation (RAG) approaches have shown promise by retrieving relevant code snippets as cross-file context, they suffer from two fundamental problems: misalignment between the query and the target code in the retrieval process, and the inability of existing retrieval methods to effectively utilize the inference information. To address these challenges, we propose AlignCoder, a repository-level code completion framework that introduces a query enhancement mechanism and a reinforcement learning based retriever training method. Our approach generates multiple candidate completions to construct an enhanced query that bridges the semantic gap between the initial query and the target code. Additionally, we employ reinforcement learning to train an AlignRetriever that learns to leverage inference information in the enhanced query for more accurate retrieval. We evaluate AlignCoder on two widely-used benchmarks (CrossCodeEval and RepoEval) across five backbone code LLMs, demonstrating an 18.1% improvement in EM score compared to baselines on the CrossCodeEval benchmark. The results show that our framework achieves superior performance and exhibits high generalizability across various code LLMs and programming languages.**

***Index Terms*—Repository-Level Code Completion, Query Enhancement, Reinforcement Learning, code LLMs**

I. Introduction

Recent developments in code large language models (code LLMs) [1]–[4] have demonstrated impressive capability in general code completion tasks [5]–[8], [8]–[11]. However, existing code LLMs demonstrate suboptimal performance on repository-level code completion tasks, primarily due to their insufficient understanding of repository-specific context and domain knowledge [12]. This limitation stems from the fact that target code repositories are often newly created, proprietary, or work-in-progress projects, making it hard for code LLMs to acquire repository-specific knowledge during pre-training and fine-tuning phases [13]. To address this challenge, one straightforward approach leverages the increasing context window length of modern models by concatenating all repository files into a single prompt. However, this naive concatenation introduces substantial irrelevant information that interferes with model generation [14], [15]. Consequently, recent methods have adopted the retrieval-augmented generation (RAG) paradigm [13], [16]–[21], which uses unfinished code in the current file as a query to retrieve relevant code snippets from the entire repository. These retrieved code snippets serve as cross-file context and are concatenated with the unfinished code to construct prompts for code LLMs. For instance, ReACC [16] integrates both sparse and dense retrieval methods. Sparse retrievers, such as BM25 [22], employ keyword matching algorithms that effectively capture lexical information. Conversely, dense retrievers encode both queries and code snippets into dense vectors, enabling the identification of semantically similar code snippets through vector similarity measurements. Despite these advances, most dense retrieval methods fail to leverage the reasoning and understanding capabilities of code LLMs to enhance the retrieval process, resulting in a semantic gap between query and target code in the retrieval process. To mitigate this misalignment, RepoCoder [17] proposes an iterative retrieval strategy where the generator produces intermediate completions based on retrieved code snippets, which are then incorporated into subsequent retrieval queries. While this approach represents a significant step toward addressing the query-target misalignment, the fundamental issue remains unresolved. Specifically, we have identified the following problems in existing retrieval processes:

- **P1** **The misalignment between query and target code remains unresolved.** To address the problem of misalignment between query and target code in RAG-based code completion, RepoCoder [17] introduces an iterative retrieval approach that concatenates the completion with the unfinished code to obtain a new query for the next retrieval round. However, this method has two problems: If LLMs produce an incorrect completion in an iteration, it will cause chain errors that affect the subsequent retrieval. Besides, multiple retrieval rounds significantly reduce efficiency.
- **P2** **Existing retrieval methods lack the ability to learn how to utilize inference information.** Although RepoCoder has demonstrated that the generated completion can significantly assist repository-level code completion, the method employs sparse (e.g., Jaccard index [23]) or dense (e.g., UniXcoder [24]) retrievers that are not

* Yanlin Wang is the corresponding author, wangylin36@mail.sysu.edu.cn.
† These authors contributed equally to this work.

specifically trained. These retrievers may fail to understand the relationship between the unfinished code and the candidate completion, and therefore cannot fully leverage it for effective retrieval.

In this paper, we propose a repository-level code completion framework AlignCoder to address the two aforementioned problems. First, we introduce a query enhancement mechanism that leverages sampled candidate completions to improve retrieval accuracy. Specifically, our approach employs the sampler to generate multiple candidate completions, which are then expanded to the unfinished code to construct an enhanced query representation. This enhanced query effectively bridges the gap between the initial query and the desired target completion (addressing **P1**). Secondly, we employ reinforcement learning to train the retriever, enabling it to learn how to utilize the multiple candidate completions contained in the enhanced query for more accurate retrieval (addressing **P2**). Specifically, given an enhanced query, we employ the retriever to retrieve multiple potentially relevant code snippets. We then utilize a reward model to evaluate the perplexity (PPL) of generating the target code using these retrieved code snippets, deriving rewards from this evaluation process to update the retriever's parameters.

We evaluate AlignCoder with extensive experiments using five backbone LLMs on two benchmarks: CrossCodeEval [25] and RepoEval [17]. These two benchmarks are widely used in repository-level code completion. Experimental results show that our framework achieves an 18.1% improvement in EM score compared with baselines on the CrossCodeEval Python. AlignCoder demonstrates high generalizability, showing effectiveness across various code LLMs and programming languages.

To summarize, our main contributions are:

- We introduce AlignCoder, a repository-level code completion framework. The proposed query enhancement mechanism allows the enhanced query to have a greater possibility of including key tokens relevant to the target code. This framework effectively addresses the misalignment problem between query and target code in the retrieval process.
- We train the retriever using reinforcement learning, resulting in Alignretriever. This retriever learns to leverage the inference information in enhanced queries to achieve more accurate retrieval.
- We perform extensive experimental evaluation on various benchmarks and code LLMs. The results show that AlignCoder achieves superior performance compared to previous approaches. We provide our code and data at https://anonymous.4open.science/r/AlignCoder.

II. Preliminaries

A. Retrieval-Augmented Code Completion Paradigms

Retrieval-augmented methods have been widely adopted for repository-level code completion. Such methods can be categorized into two primary paradigms:

1) Retrieve-then-Generate: This approach first retrieves relevant code snippets from the codebase using the unfinished code as a query, then uses the retrieved code snippets to assist the code completion. Methods like ReACC [16] and RLCoder [26] follow this paradigm. The retrieval process takes the unfinished code C_u as input to search through the repository R for the top-k most similar code snippets S, which are then provided to the language model P_θ as context for generating the target completion $\hat{C}_t$. However, this paradigm faces the challenge of semantic misalignment. These methods only use unfinished code as a query, which may not contain the key tokens related to the target code. Consequently, it becomes difficult to retrieve the relevant code snippets needed to generate the target code during the retrieval process [17].

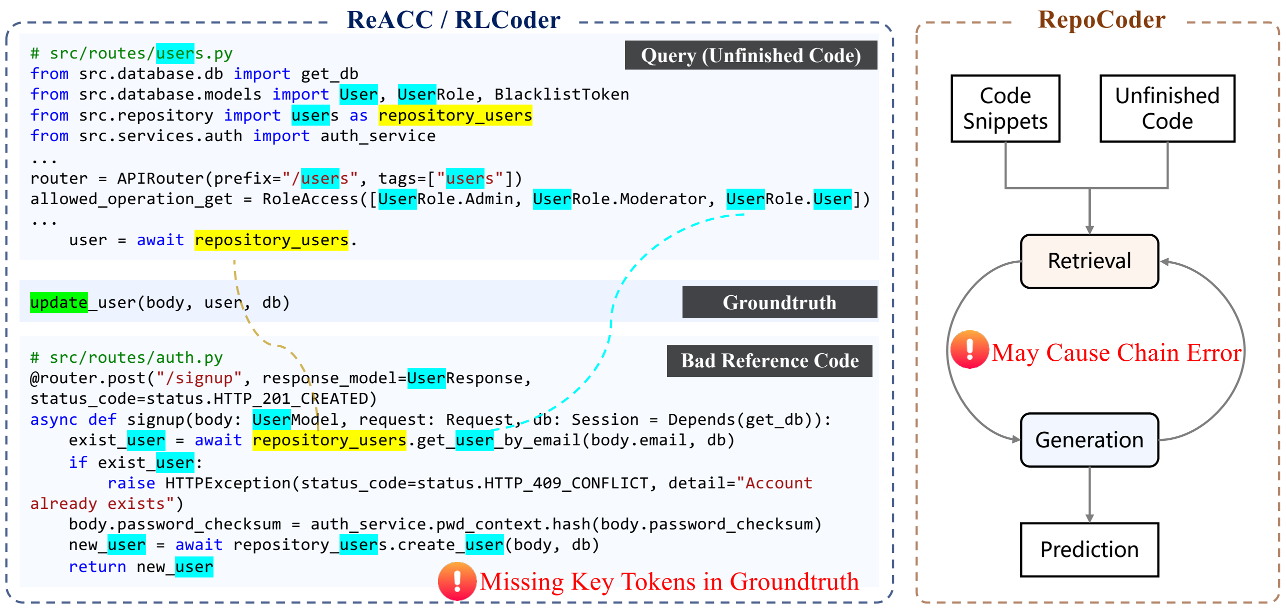
2) Iterative Generate-and-Retrieve: This approach alternates between generation and retrieval in multiple iterations. Starting with an initial retrieval using the unfinished code, the method generates a candidate completion and then uses the concatenation of the unfinished code and the generated candidate as a new query for the next retrieval iteration. Methods like RepoCoder [17] employ this iterative approach, where each iteration refines both the retrieved context and the generated completion. However, since each iteration relies on a single candidate completion, errors can propagate through the chain. If errors occur in the intermediate generation step or key tokens are still missing, the subsequent retrieval will be based on flawed information, leading to cascading errors throughout the remaining iterations.

B. Semantic Gap in Repository Code Retrieval

Repository-level code completion leverages contextual information across multiple files to generate accurate code completions. Formally, given an unfinished code C_u and a repository R, the objective is to generate target code C_t such that $C = C_u \oplus C_t$ is syntactically and semantically correct. A challenge in this task stems from the semantic gap between unfinished code and target code. Using C_u directly as a query is suboptimal because unfinished code and target code belong to different semantic spaces, making alignment difficult [27]. The semantic gap can be formalized as :

$$\mathcal{G}(C_u, C_r) = d(\Phi_{query}(C_u), \Phi_{target}(C_r)) \qquad (1)$$

where Φ_{query} and Φ_{target} be embedding functions mapping code to semantic spaces and d is a distance function in the semantic space. Previous approaches like RLCoder [26] attempt to bridge this gap through reinforcement learning, while RepoCoder [17] uses iterative retrieval-generation to update the query. However, the former requires semantic space alignment between unfinished code and target code, which is inherently challenging due to the differences in their representations. The latter relies on a single sample from each iteration, which can easily propagate errors in a chain reaction—if one iteration produces an incorrect completion, subsequent iterations are built upon this flawed foundation, leading to compounding errors throughout the retrieval-generation process.



```
# src/routes/users.py
from src.database.db import get_db
from src.database.models import User, UserRole, BlacklistToken
from src.repository import users as repository_users
from src.services.auth import auth_service
...
router = APIRouter(prefix="/users", tags=["users"])
allowed_operation_get = RoleAccess([UserRole.Admin, UserRole.Moderator, UserRole.User])
...
    user = await repository_users.
```

```
update_user(body, user, db)
```

```
# src/routes/auth.py
@router.post("/signup", response_model=UserResponse,
status_code=status.HTTP_201_CREATED)
async def signup(body: UserModel, request: Request, db: Session = Depends(get_db)):
    exist_user = await repository_users.get_user_by_email(body.email, db)
    if exist_user:
        raise HTTPException(status_code=status.HTTP_409_CONFLICT, detail="Account
already exists")
    body.password_checksum = auth_service.pwd_context.hash(body.password_checksum)
    new_user = await repository_users.create_user(body, db)
    return new_user
```

Fig. 1. Motivating example. Limitations of prior works: ReACC and RLCoder use the unfinished code as the query, which may miss key tokens in the ground truth. RepoCoder may cause chain errors and has potential efficiency issue.

C. *Multiple Sampling in LLMs*

LLMs exhibit inherent stochasticity in their generation process, making single-sample outputs unreliable. The probabilistic nature of token sampling introduces variability that can lead to inconsistent or incorrect responses, even when queried with identical prompts. However, through the multiple sampling strategy, we can significantly improve the likelihood of obtaining correct completions.

1) Single Sample Unreliability: For a given query q, an LLM generates a response by sampling from the learned probability distribution over the vocabulary V at each timestep. Let $p_\theta(y|x)$ denote the probability of generating sequence y given input x and model parameters θ. The sampling process can be formulated as:

$$y_t \sim p_\theta(\cdot|x, y_{<t}) \tag{2}$$

where y_t represents the token at position t, and $y_{<t}$ denotes all previously generated tokens.

Due to the stochastic nature of sampling, a single generation attempt has probability p_s of producing a correct answer, typically $p_s < 1$ and may be lower for complex queries. Threrefore, the probability of generating an incorrect response in a single attempt is:

$$P(\text{error}) = 1 - p_s \tag{3}$$

2) Multiple Sampling Benefits: When performing multiple sampling attempts, we acknowledge that individual samples are not truly independent events, as they originate from the same model with identical parameters and input prompt. However, due to the stochastic nature of the sampling process (temperature-based sampling, top-k, or nucleus sampling), we can approximate the samples as quasi-independent for analytical purposes.

Let ρ denote the correlation coefficient between samples, where $0 \leq \rho \leq 1$. For truly independent samples ($\rho = 0$), the probability of obtaining at least one correct answer from n attempts would be:

$$P_{\text{independent}}(\text{at least one correct}) = 1 - (1 - p_s)^n \tag{4}$$

However, accounting for inter-sample correlation, the actual probability can be approximated as:

$$P(\text{at least one correct}) \approx 1 - (1 - p_s)^{n \cdot (1-\rho)} \tag{5}$$

where the effective number of independent samples is reduced by the correlation factor $(1 - \rho)$.

In practice, modern sampling techniques (such as temperature scaling $T > 0$ or top-p sampling) introduce sufficient randomness that ρ remains relatively small, making the independence approximation reasonable:

$$P(\text{at least one correct}) \approx 1 - (1 - p_s)^n \quad \text{when } \rho \ll 1 \tag{6}$$

This relationship demonstrates that even with correlated samples, the success probability increases substantially with the number of sampling attempts. For practical applications, multiple sampling strategy consistently yields higher accuracy than single sampling, establishing it as an effective strategy for improving LLM reliability in critical applications.

The above provides an introduction to the theoretical foundations. In the following part, we demonstrate the effectiveness of multiple sampling strategy compared to single sampling through a preliminary experiment. This experiment utilizes the CrossCodeEval and RepoEval, where the prompt consists of BM25 retrieval code snippets concatenated with unfinished

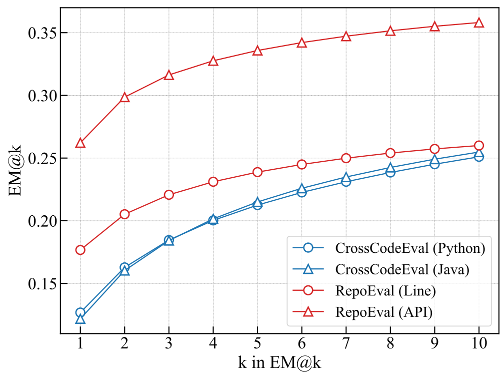


Fig. 2. EM@k performance trends on CrossCodeEval and RepoEval with varying k parameters. The trends indicate that multiple sampling strategy provides a higher probability of producing a correct completion compared tosingle sampling.

code. We present the *EM@k* results (k ranging from 1 to 10) for DeepSeekCoder-1B using the aforementioned prompt with 10 sampling attempts. Specifically, *EM@k* represents the probability of that at least one of the k samples exactly matches the target code (EM=1). As shown in figure 2, the *EM@k* values on both benchmarks increase with the value of k. This indicates that multiple sampling strategy provides a higher probability of generating a correct completion compared to single sampling, which also means it is easier to contain key tokens related to the correct completion.

3) Diminishing Returns and Optimal Sampling Threshold: While multiple sampling generally improves the probability of obtaining correct completions, excessive sampling can introduce diminishing returns and potentially counterproductive effects. As the number of samples increases beyond an optimal threshold, several factors contribute to degraded performance:

Error Accumulation: With increased sampling numbers, the absolute number of incorrect responses grows, potentially overwhelming correct completions during aggregation or selection processes. If we define ϵ_n as the cumulative error rate after n samples, we have:

$$\epsilon_n = n \cdot (1 - p_s) \tag{7}$$

Selection Complexity: The probability of selecting the correct answer from a pool containing both correct and incorrect responses depends on the selection mechanism. For random selection from n samples, where k samples are correct, the probability of selecting a correct answer is:

$$P(\text{correct selection}) = \frac{k}{n} = \frac{n \cdot p_s}{n} = p_s \tag{8}$$

However, for more sophisticated selection mechanisms (e.g., majority voting, confidence-based selection), the relationship becomes more complex.

Optimal Sampling Threshold: To determine the optimal number of samples n^*, we must balance the benefit of increased correct sample probability against the cost of error accumulation. The expected utility can be formulated as:

$$U(n) = \alpha \cdot P(\text{at least one correct}) - \beta \cdot \epsilon_n - \gamma \cdot n \tag{9}$$

where α, β, and γ represent the weights for correctness benefit, error penalty, and computational cost, respectively.

Substituting our previous formulations:

$$U(n) = \alpha \cdot [1 - (1 - p_s)^n] - \beta \cdot n(1 - p_s) - \gamma \cdot n \tag{10}$$

Taking the derivative with respect to n and setting it to zero:

$$\frac{dU(n)}{dn} = \alpha \cdot (1 - p_s)^n \ln(1 - p_s) - \beta(1 - p_s) - \gamma = 0 \tag{11}$$

The optimal sampling threshold n^* can be approximated by solving:

$$n^* \approx \frac{\ln\left(\frac{\beta(1-p_s)+\gamma}{\alpha\ln(1-p_s)}\right)}{\ln(1 - p_s)} \tag{12}$$

This theoretical framework demonstrates that while multiple sampling is beneficial, there exists an optimal threshold beyond which additional samples provide marginal utility and may even degrade overall system performance due to increased complexity in answer selection and computational overhead.

D. Perplexity in LLMs for Code Assessment

Perplexity (PPL) quantifies how well a probability model predicts token sequences, serving as an intrinsic measure of model confidence in Large Language Models. For a token sequence $\mathbf{y} = (y_1, y_2, \ldots, y_T)$, perplexity is defined as:

$$\text{PPL}(\mathbf{y}) = \exp\left(-\frac{1}{T}\sum_{t=1}^{T} \log p_\theta(y_t | y_{<t})\right) \tag{13}$$

where $p_\theta(y_t|y_{<t})$ represents the conditional probability of token y_t given preceding context $y_{<t}$. Lower perplexity indicates higher model confidence.

While not directly measuring correctness, perplexity correlates with code quality through several mechanisms: (1) *Statistical regularity:* well-formed code follows learned patterns from training data; (2) *Syntactic consistency:* valid syntax exhibits predictable token transitions; (3) *Semantic coherence:* logical code patterns yield lower perplexity.

However, perplexity has limitations: uncommon but correct coding styles, domain-specific patterns, and model understanding gaps can cause discrepancies between perplexity and actual correctness. Empirical studies show moderate positive correlation between low perplexity and code correctness [28], [29], making it effective as a first-order heuristic for ranking code completion candidates when combined with other validation methods.

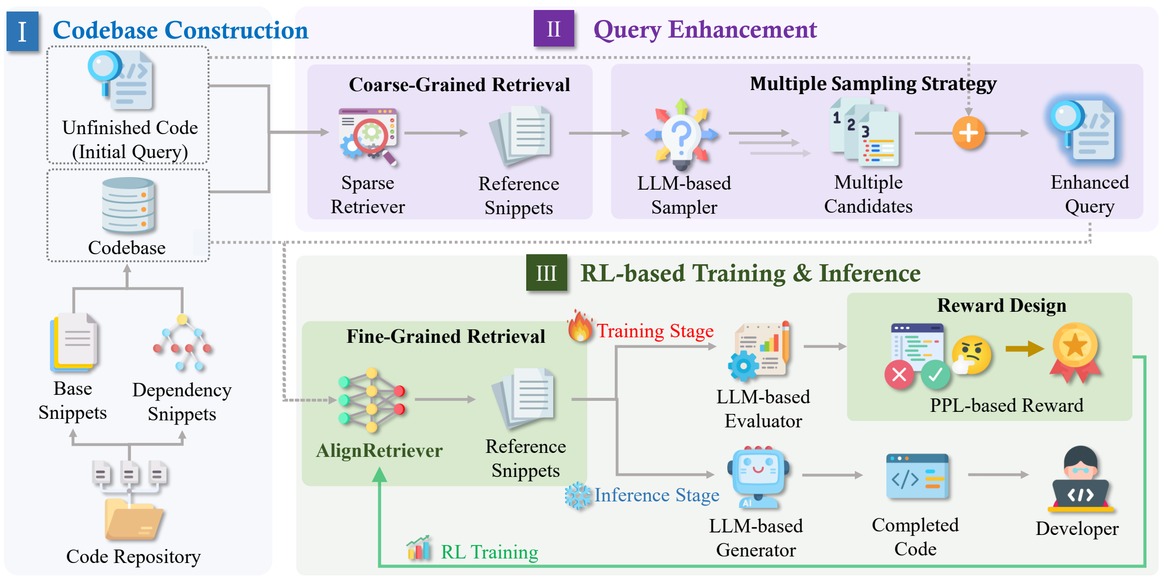


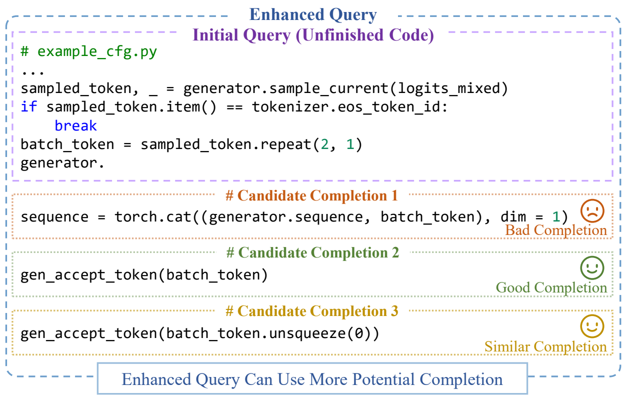
Fig. 3. Overview of AlignCoder. Our approach utilizes the inference capability of the LLM to sample multiple candidate completions and learn to understand the inference information for accurate retrieval for repository-level code completion.

III. Methodology

In this section, we introduce AlignCoder, a framework for repository-level code completion. The pipline of AlignCoder is shown in Figure 3, which contains three stages. ❶ Given a code repository, we extract two types of code snippets to construct the codebase for retrieval. ❷ For the unfinished code needs to be completed, we use it as the original query to perform the coarse-grained retrieval. Then we use the retrieved code snippets concatenated with the unfinished code as the prompt for sampler. The sampler is a lightweight LLM to sample multiple candidate completions. Then we concatenate the unfinished code with the sampled candidate completions to serve as the enhanced query. ❸ We use the enhanced query to perform fine-grained retrieval, which returns several reference code snippets. In the training phase, the reward model will give each reference code snippet a reward to evaluate the helpfulness. The parameter of the retriever will be updated by the reward, which is calculated through an LLM-based evaluator. In the inference phase, we concatenate the retrieved code snippets with the unfinished code to construct the prompt, based on which the generator performs the final code completion.

A. *Codebase Construction*

In constructing the retrieval codebase, we construct two distinct types of code snippets, namely base and dependency code snippets. Each category of code snippets is tailored to represent a corresponding type of cross-file information. The first type is the base context, which consists of initial code segments within the cross-files. The second type is dependency context, which provides deep semantic understanding of class hierarchies and API interactions within the codebase [30]. The following section provides a detailed introduction to the


Enhanced Query
Initial Query (Unfinished Code)

```
# example_cfg.py
...
sampled_token, _ = generator.sample_current(logits_mixed)
if sampled_token.item() == tokenizer.eos_token_id:
    break
batch_token = sampled_token.repeat(2, 1)
generator.
```

# Candidate Completion 1

```
sequence = torch.cat((generator.sequence, batch_token), dim = 1)
```

Bad Completion
# Candidate Completion 2

```
gen_accept_token(batch_token)
```

Good Completion
# Candidate Completion 3

```
gen_accept_token(batch_token.unsqueeze(0))
```

Similar Completion
Enhanced Query Can Use More Potential Completion

Fig. 4. Example: the initial query and the enhanced query obtained with the query enhancement mechanism. Task id: `project_cc_python/74`.

construction methodologies for base and dependency code snippets.

1) Base Code Snippets Construction: Previous works construct retrieval codebase using fixed window sizes [17] and dependency parsing approaches [13], [31], [32]. However, fixed window strategies may disrupt code continuity, while dependency parsing-based methods may only focus on limited context in the context graph and struggle to be applied to complex scenarios, such as when repository dependencies are highly intricate. Therefore, we adopt the Split-Aggregate strategy [26] to construct base code snippets. This approach is inspired by human programming habits, first dividing the cross-files into mini-blocks based on blank lines, then aggregating these mini-blocks into code snippets according to a predefined length. Formally, given a cross-file $F = \{l_1, l_2, \ldots, l_n\}$ where l_i represents the i-th line, we first split F into mini-blocks based on blank lines:

$$\text{Split}(F) = \{B_1, B_2, \ldots, B_k\} \text{ where } B_j = \{l_s, \ldots, l_e\} \quad (14)$$

where each mini-block B_j is a contiguous sequence of non-empty lines. Subsequently, we aggregate these mini-blocks into code snippets:

$$S_i = \text{Aggregate}(\{B_j, B_{j+1}, \ldots, B_{j+m}\}) \text{ s.t. } |S_i| \leq L \quad (15)$$

where S_i represents the i-th standard code snippet, L is the predefined maximum line counts, and $|S_i|$ denotes the line counts of snippet S_i. The aggregation process ensures that:

$$\sum_{k=j}^{j+m} |B_k| \leq L \text{ and } \sum_{k=j}^{j+m+1} |B_k| > L \quad (16)$$

2) Dependency Code Snippets Construction: We use tree-sitter [33] to extract import statements from in-file context and parse each statement to obtain module name, entity name, and alias. We filter out standard library and third-party imports, retaining only intra-repository module references. Formally, given import statements $I = \{i_1, i_2, \ldots, i_n\}$:

$$I_{intra} = \{i \in I \mid m(i) \notin (\text{StdLib} \cup \text{ThirdParty})\} \quad (17)$$

where $m(i)$ extracts the module name of import i. The intra-repository references are categorized into class imports I_c, method imports I_m, and function imports I_f.

We then parse cross-files using tree-sitter and extract matching code bodies based on these references. We refer to the corresponding code bodies as entities [13], [34], denoted by u. For function and method imports ($i \in I_f \cup I_m$):

$$\text{Dependency}(i) = \text{Signature}(u_i) \quad (18)$$

For class imports ($i \in I_c$):

$$\text{Dependency}(i) = \{\sigma_c, \Sigma_m, \Sigma_{nc}, \Sigma_{nm}\} \quad (19)$$

where σ_c is the main class signature, Σ_m contains all class method signatures, Σ_{nc} contains nested class signatures, and Σ_{nm} contains nested class method signatures. The complete dependency information is:

$$\mathcal{D} = \bigcup_{i \in I_{intra}} \text{Dependency}(i) \quad (20)$$

We construct dependency code snippets based on extracted information, treating each imported class's dependency information as an individual dependency code snippet. For imported methods or functions, we aggregate their signatures into a single snippet. Finally, we combine base code snippets and dependency code snippets to form the retrieval codebase.

B. Query Enhancemant Mechanism

If the query used for retrieval only contains the unfinished code, it is likely to retrieve code snippets that are similar to them. This raises a gap between the query and the target code, potentially leading to reduced retrieval accuracy. To address this issue, we propose a query enhancement mechanism. The details of the mechanism are illustrated below.

Our query enhancement mechanism consists of two phases: coarsed-grained retrieval and multiple sampling strategy. (1) In the coarse-grained retrieval, we use BM25 method to retrieve code snippets from the codebase using the unfinished code as the initial query. These code snippets may include both base and dependency code snippets, providing the original code information and deep semantic understanding of class hierarchies and API interactions. (2) These code snippets are then concatenated with the unfinished code to build a prompt. The sampler samples k candidate completions based on this prompt. Finally, these candidate completions are appended to the initial query to construct an enhanced query, which guides the retriever in the fine-grained retrieval phase. For example, Figure 4 presents a task `project_cc_python/74` with the initial query and the enhanced query after processing through the query enhancemant mechanism. In the task, we need to complete a specific line in example_cfg.py where a method is called on the `generator` object. The `generator` object is an instance of the `ExLlamaGenerator` class defined in `generator.py`, and the target code is a method `get_accept_token` defined in this class. Among the candidate completions sampled by the sampler, candidate completion **1** is an incorrect completion, while candidate completion **2** is the correct completion, and candidate completion **3** is a completion similar to the target code. We consider that candidate completions **2** and **3** contain key tokens related to the target code. The enhanced query may improve the ability of the retriever to retrieve more relevant code snippets in the following fine-grained retrieval phase.

C. AlignRetriever: RL-based Retriever Aglignment Training

1) Training data construction: We randomly selected 10,000 Python and Java repositories from GitHub. These repositories contain cross-file dependencies and were created before March 2023 and are not included in our evaluation benchmarks, CrossCodeEval [25] and RepoEval [17], ensuring no data leakage and maintaining the fairness of the evaluation. Constructing training data involves the following three steps: ① We divide each code repository into a series of clusters, where the code files within these clusters have interdependent relationships. We exclude clusters that contain only a single file. ② Then, we perform topological sorting on the remaining clusters. This sorting is based on the in-degree and out-degree relationships between the files. The final sorting result ensures that : The first code file is dependent upon other files in the same cluster, but does not contain dependencies on other files. All subsequent code files depend on one or more other files within the cluster. ③ For each cluster, we randomly select one non-first file to construct the target code for completion. To ensure the preceding context of the target code contains sufficient information, we avoid selecting starting positions of the target code from either the beginning or the end of the current file. The length of the target code is also randomly determined (set to be from 16 to 96 tokens in our experiment), and the entire code segment must lie within the boundaries of the current file.

2) The training objective function: The reward mechanism in reinforcement learning provides the model with environmental feedback, allowing it to progressively learn particular capabilities based on this feedback signal. We define the reward function as $\text{Reward}(\cdot)$, where the reward function needs to be maximized during AlignRetriever training period. The detailed mathematical expression for $\text{Reward}(\cdot)$ is:

$$\text{Reward} = \sum_{i=1}^{n} \mathbb{I}(c_i) \times \log \frac{\exp(s_{i,q})}{\sum_{j=1}^{n} \exp(s_{j,q})} \tag{21}$$

where q is the enhanced query, $C = \{c_1, c_2, \ldots, c_n\}$ is the set of retrieved code snippets, n is the number of code snippets, $\mathbb{I}(c_i)$ is the indicator function denoting correctness of snippet c_i, and $s_{i,q} = \cos(\text{emb}(c_i), \text{emb}(q))$ represents the cosine similarity between embeddings of code snippet c_i and query q. The definition of $\mathbb{I}(c_i)$ is:

$$\mathbb{I}(c_i) = \begin{cases} 1, & \text{if } c_i = c_{\text{mp}} \\ 0, & \text{otherwise} \end{cases} \tag{22}$$

where $c_{\text{mp}} \in C$ and satisfies:

$$\text{PPL}(t|q, c_{\text{mp}}) \leq \text{PPL}(t|q, c_i) \quad \forall i \in \{1, \ldots, n\} \tag{23}$$

where the formula $\text{PPL}(t|q, c_i)$ is defined as:

$$\text{PPL}(t|x, c_i) = e^{-\frac{1}{L}\sum_{j=1}^{L} \log P(t_j|c_i, q, t_{<j})} \tag{24}$$

where L denotes the total number of tokens in the target code.

IV. Experimental setup

In this section, we introduce the benchmarks, backbone models, baseline methods, and evaluation metrics used in our experiments.

A. Benchmarks and Backbone Models

In our evaluation, we use two benchmarks for repository-level code completion tasks: CrossCodeEval [25] and RepoEval [17]. These two benchmarks have been widely used in previous work [4], [26], [34]–[36].

- **CrossCodeEval**: CrossCodeEval is a diverse and multilingual code completion benchmark that requires a deep understanding of context across different files in the repository to ensure accurate code completion. We conduct the evaluation on Python and Java sets.
- **RepoEval**: RepoEval is allowed for assessments at three granularities, line, API invocation, and function body. In our evaluation, we focus on the line-level and API invocation tasks.

In our experiments, we select five code LLMs as generators. These code LLMs have been proven to perform well on repository-level code completion tasks in previous work [17], [18], [26], [30], [37]. These five code LLMs are: CodeLlama-7B [38], StartCoder-7B [39], StarCoder2-7B [1], DeepSeekCoder-1B [35], and DeepSeekCoder-7B [35].

B. Baseline Methods

In our experiments, we compare AlignCoder with previous RAG-based methods, encompassing ReACC, RepoCoder, and RLCoder.

- **ReACC**: ReACC [16] adopt the hybrid retriever [40], [41] framework by combining sparse and dense retriever.
- **RepoCoder**: RepoCoder [17] is an iterative retrieval and generation framework, where it searches for the relevant code snippets using the output generated by code LLMs from the previous iteration.
- **RLCoder**: RLCoder [26] is a reinforcement learning framework, which can enable the retriever to learn to retrieve useful snippets without the need for labeled data.

C. Evaluation Metrics

Following the established practice [13], [17], [26], [34], we use two metrics, Exact Match (EM) and Edit Similarity (ES), to evaluate code completion accuracy.

D. Experimental Details

We performed all experiments on a machine configured with 2 NVIDIA Tesla A100 GPUs, each with 80 GB of memory.

- **Training Stage**: In the training stage, we initialize the retriever using UniXcoder and use DeepSeekCoder-1B as the evaluator. We trained the retriever for a total of 20 epochs. Each epoch utilized 3,000 samples for training, and the learning rate was set to 5e-5.
- **Inference Stage**: In the inference stage, since our approach employs multiple sampling, we utilize the vLLM [42] framework to accelerate model inference.

V. Experimental Results

We aims to answer the following research questions (RQs):

- **RQ1**: How effective is AlignCoder in repository-level code completion?
- **RQ2**: The effectiveness of multiple sampling strategies, and what the optimal sampling numbers should be?
- **RQ3**: What is the contribution of each AlignCoder component to its performance?
- **RQ4**: What is the robustness of AlignCoder's parameter settings?

A. RQ1: Overall Performance of AlignCoder

In this section, we study the effectiveness of AlignCoder compared with three baseline methods. Table I presents the performance comparison of AlignCoder against three baseline methods across multiple benchmarks and backbone models. The results demonstrate that AlignCoder outperforms previous methods across all evaluation settings.

From the perspective of benchmarks, AlignCoder achieves greater improvements on CrossCodeEval compared to RepoEval. For the CrossCodeEval Python, AlignCoder demonstrates substantial performance gains, with EM score improvements ranging from 12.7% to 18.1% compared to RLCoder. For the CrossCodeEval Java, AlignCoder maintains its superiority

TABLE I
PERFORMANCE COMPARISON. SUPERSCRIPTED PERCENTAGES REPRESENT THE IMPROVEMENT OVER THE CORRESPONDING BEST BASELINE.

Method/Model	CrossCodeEval (Python)		CrossCodeEval (Java)		RepoEval (Line)		RepoEval (API)	
	EM	ES	EM	ES	EM	ES	EM	ES
ReACC $_{\text{CodeLlama-7B}}$	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 $_{\text{CodeLlama-7B}}$	26.64	72.26	26.27	67.60	46.63	67.86	37.94	64.32
AlignCoder $_{\text{CodeLlama-7B}}$	**30.13** $^{\uparrow 13.1\%}$	**74.29** $^{\uparrow 2.8\%}$	**28.80** $^{\uparrow 9.6\%}$	**68.38** $^{\uparrow 1.2\%}$	**46.96** $^{\uparrow 0.7\%}$	**67.92** $^{\uparrow 0.1\%}$	**39.56** $^{\uparrow 4.3\%}$	**66.01** $^{\uparrow 2.6\%}$
ReACC $_{\text{StarCoder-7B}}$	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 $_{\text{StarCoder-7B}}$	26.00	72.16	25.76	68.80	47.81	68.50	35.06	58.08
AlignCoder $_{\text{StarCoder-7B}}$	**30.43** $^{\uparrow 17.0\%}$	**74.33** $^{\uparrow 3.0\%}$	**28.00** $^{\uparrow 8.7\%}$	**70.04** $^{\uparrow 1.8\%}$	**48.38** $^{\uparrow 1.2\%}$	**68.56** $^{\uparrow 0.1\%}$	**36.25** $^{\uparrow 3.4\%}$	**59.49** $^{\uparrow 2.4\%}$
ReACC $_{\text{StarCoder2-7B}}$	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 $_{\text{StarCoder2-7B}}$	27.47	73.39	26.69	70.35	48.63	68.59	37.75	61.08
AlignCoder $_{\text{StarCoder2-7B}}$	**31.74** $^{\uparrow 15.5\%}$	**75.70** $^{\uparrow 3.1\%}$	**30.43** $^{\uparrow 14.0\%}$	**72.64** $^{\uparrow 3.3\%}$	**48.81** $^{\uparrow 0.4\%}$	**68.82** $^{\uparrow 0.3\%}$	**38.69** $^{\uparrow 2.5\%}$	**61.59** $^{\uparrow 0.8\%}$
ReACC $_{\text{DeepSeekCoder-1B}}$	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 $_{\text{DeepSeekCoder-1B}}$	24.02	70.45	20.66	63.17	44.06	66.05	36.00	62.50
AlignCoder $_{\text{DeepSeekCoder-1B}}$	**28.37** $^{\uparrow 18.1\%}$	**73.02** $^{\uparrow 3.6\%}$	**23.24** $^{\uparrow 12.5\%}$	**64.58** $^{\uparrow 2.2\%}$	**44.69** $^{\uparrow 1.4\%}$	**66.74** $^{\uparrow 1.0\%}$	**37.25** $^{\uparrow 3.5\%}$	**64.43** $^{\uparrow 3.1\%}$
ReACC $_{\text{DeepSeekCoder-7B}}$	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 $_{\text{DeepSeekCoder-7B}}$	30.09	74.43	26.37	67.28	48.81	69.48	39.75	66.01
AlignCoder $_{\text{DeepSeekCoder-7B}}$	**33.92** $^{\uparrow 12.7\%}$	**76.97** $^{\uparrow 3.4\%}$	**28.28** $^{\uparrow 7.2\%}$	**68.31** $^{\uparrow 1.5\%}$	**49.56** $^{\uparrow 1.5\%}$	**69.93** $^{\uparrow 0.6\%}$	**41.88** $^{\uparrow 5.4\%}$	**67.75** $^{\uparrow 2.6\%}$

with EM score improvements of 7.2% to 14.0% compared to RLCoder across backbone models.

From the perspective of models, results show that maximal improvement is achieved by DeepSeekCoder-1B on CrossCodeEval Python (EM: +18.1%, ES: +3.6%). For CrossCodeEval Java, StarCoder2-7B demonstrated the highest improvement (EM: +14.0%, ES: +3.3%). Regarding RepoEval performance, DeepSeekCoder-7B showed the most significant gains. It achieved EM score improvements of 1.5% on line-level tasks and 5.4% on API-level tasks.

RQ1 Summary: AlignCoder consistently outperforms all baseline methods across different benchmarks and backbone models. The best-performing setting improves 18.1% on the EM score, with particularly gains on CrossCodeEval Python.

B. RQ2: Mutiple Sampling Strategy

In this section, we investigate two questions: (1) whether multiple sampling is more effective than single sampling, and (2) what the optimal sampling numbers for AlignCoder. We conduct performance comparison experiments with sampling numbers set from 1 to 6. The generator used in this experiment is DeepSeekCoder-1B. The experimental results are presented in Table II.

(1) Effectiveness of Multiple Sampling. Multiple sampling (2, 3, and 4 samples) generally outperforms single sampling in EM and ES scores. The few exceptions show only minimal decreases: 0.2% lower EM on API-level tasks of RepoEval (sampling numbers set to 2), for instance. These results indicate that multiple sampling enhances the likelihood of generating key tokens related to target code, thus improving the retrieval performance.

(2) Performance degradation in certain datasets when sampling numbers exceed a threshold. A performance degradation is observed on certain datasets when sampling numbers surpass 4. When the sampling number is set to 5, the results on CrossCodeEval Java and line-level/API-level tasks of RepoEval are all inferior to single sampling. When the sampling number is set to 6, there is a decline compared to single sampling on CrossCodeEval Java and line-level tasks of RepoEval. This indicates that although multiple sampling is effective compared to single sampling, excessive sampling may cause the candidate completions to contain non-negligible noise, which could affect the accuracy of subsequent retrieval.

(3) Determining the optimal sampling numbers. We evaluate the performance of AlignCoder by computing average EM and ES scores under different sampling settings. As shown in Table II, the setting of sampling numbers at 4 demonstrates superior performance. This evidence establishes sampling numbers of 4 as the optimal balance for our approach, maximizing performance gains while maintaining computational efficiency.

RQ2 Summary: We demonstrate the effectiveness of the multiple sampling strategy compared to single sampling. When the sampling number exceeds a threshold, AlignCoder's performance declines. Experimental results show that the optimal sampling number is 4.

C. RQ3: Ablation Studies

To demonstrate the effectiveness of the three core components in AlignCoder, we conduct experiments focusing on: (1) incorporating dependency context as cross-file information by constructing and retrieving dependency code snippets; (2)

TABLE II
SUPERSCRIPTED PERCENTAGES INDICATE THE RELATIVE INCREASE OR DECREASE OF MULTI-SAMPLING COMPARED TO SINGLE-SAMPLING.

Method	**CrossCodeEval (Python)**		**CrossCodeEval (Java)**		**RepoEval (Line)**		**RepoEval (API)**		**Average**	
	EM	**ES**	**EM**	**ES**	**EM**	**ES**	**EM**	**ES**	**EM**	**ES**
Sampling Number=1	27.20	71.98	23.05	64.49	44.75	66.47	36.69	63.39	32.92	66.58
Sampling Number=2	27.65 $^{\uparrow 1.7\%}$	72.41 $^{\uparrow 0.6\%}$	**23.42** $^{\uparrow 1.6\%}$	**65.45** $^{\uparrow 1.5\%}$	**45.19** $^{\uparrow 1.0\%}$	**66.98** $^{\uparrow 0.8\%}$	36.63 $^{\downarrow 0.2\%}$	63.90 $^{\uparrow 0.8\%}$	33.22 $^{\uparrow 0.9\%}$	67.19 $^{\uparrow 0.9\%}$
Sampling Number=3	28.29 $^{\uparrow 4.0\%}$	72.71 $^{\uparrow 1.0\%}$	23.19 $^{\uparrow 0.6\%}$	63.93 $^{\downarrow 0.9\%}$	44.75 $^{\uparrow 0.0\%}$	66.35 $^{\downarrow 0.2\%}$	36.88 $^{\uparrow 0.5\%}$	63.93 $^{\uparrow 0.9\%}$	33.28 $^{\uparrow 1.1\%}$	66.73 $^{\uparrow 0.2\%}$
Sampling Number=4	28.37 $^{\uparrow 4.3\%}$	**73.02** $^{\uparrow 1.5\%}$	23.24 $^{\uparrow 0.8\%}$	64.58 $^{\uparrow 0.1\%}$	44.69 $^{\uparrow -0.1\%}$	66.74 $^{\uparrow 0.4\%}$	37.25 $^{\uparrow 1.5\%}$	**64.43** $^{\uparrow 1.6\%}$	**33.39** $^{\uparrow 1.4\%}$	**67.19** $^{\uparrow 0.9\%}$
Sampling Number=5	**28.44** $^{\uparrow 4.6\%}$	71.68 $^{\downarrow 0.4\%}$	22.81 $^{\downarrow 1.0\%}$	64.10 $^{\downarrow 0.6\%}$	44.37 $^{\downarrow 0.9\%}$	66.44 $^{\downarrow 0.1\%}$	36.63 $^{\downarrow 0.2\%}$	63.43 $^{\uparrow 0.1\%}$	33.06 $^{\uparrow 0.4\%}$	66.41 $^{\downarrow 0.3\%}$
Sampling Number=6	28.07 $^{\uparrow 3.2\%}$	72.54 $^{\uparrow 0.8\%}$	22.87 $^{\downarrow 0.8\%}$	64.18 $^{\downarrow 0.5\%}$	44.63 $^{\downarrow 0.3\%}$	66.38 $^{\downarrow 0.1\%}$	**37.81** $^{\uparrow 3.1\%}$	64.07 $^{\uparrow 1.1\%}$	33.35 $^{\uparrow 1.3\%}$	66.79 $^{\uparrow 0.3\%}$

TABLE III
ABLATION STUDY RESULTS.

Method	**CrossCodeEval (Python)**		**CrossCodeEval (Java)**	
	EM	**ES**	**EM**	**ES**
AlignCoder	33.92	76.97	28.28	68.31
w/o DC	31.44$^{\downarrow 7.3\%}$	75.82$^{\downarrow 1.5\%}$	27.44$^{\downarrow 3.0\%}$	67.54$^{\downarrow 1.1\%}$
w/o QH	31.33$^{\downarrow 7.6\%}$	75.21$^{\downarrow 2.3\%}$	27.12$^{\downarrow 4.1\%}$	67.63$^{\downarrow 1.0\%}$
w/o RL	28.14$^{\downarrow 17.4\%}$	73.39$^{\downarrow 4.7\%}$	25.62$^{\downarrow 9.4\%}$	66.85$^{\downarrow 2.1\%}$
Method	**RepoEval (Line)**		**RepoEval (API)**	
	EM	**ES**	**EM**	**ES**
AlignCoder	49.56	69.93	41.88	67.75
w/o DC	49.06$^{\downarrow 1.0\%}$	69.96$^{\uparrow 0.0\%}$	40.44$^{\downarrow 3.4\%}$	66.75$^{\downarrow 1.5\%}$
w/o QH	48.94$^{\downarrow 1.3\%}$	69.63$^{\downarrow 0.4\%}$	40.19$^{\downarrow 4.0\%}$	66.52$^{\downarrow 1.8\%}$
w/o RL	46.56$^{\downarrow 6.1\%}$	68.06$^{\downarrow 2.7\%}$	38.63$^{\downarrow 7.8\%}$	65.10$^{\downarrow 3.9\%}$

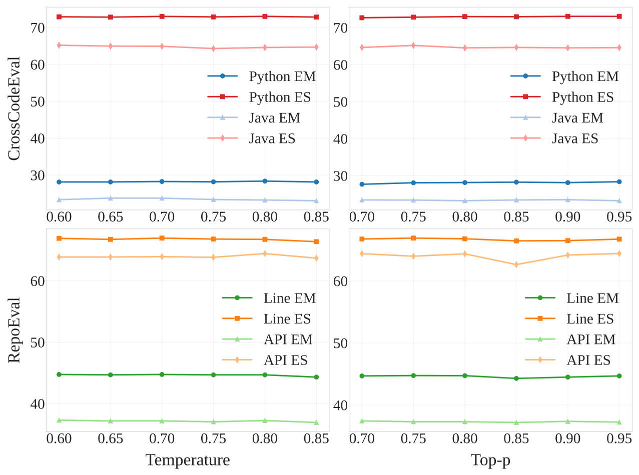


Fig. 5. Temperature and top-p sampling parameter effects on AlignCoder performance.

employing a query enhancement mechanism; and (3) using reinforcement learning to train the retriever to learn to utilize the inference information in the enhanced query. We conduct ablation studies on the CrossCodeEval and RepoEval. All experiments are implemented with DeepSeekCoder-7B as the generator, with detailed results shown in Table III.

Table III shows three ablation settings. w/o DC indicates that the dependency context is not considered. w/o QH denotes the configuration without the query enhancement mechanism applied. w/o RL indicates that reinforcement learning is not applied to the retriever, so the retriever has not learned how to utilize the key tokens in the enhanced query.

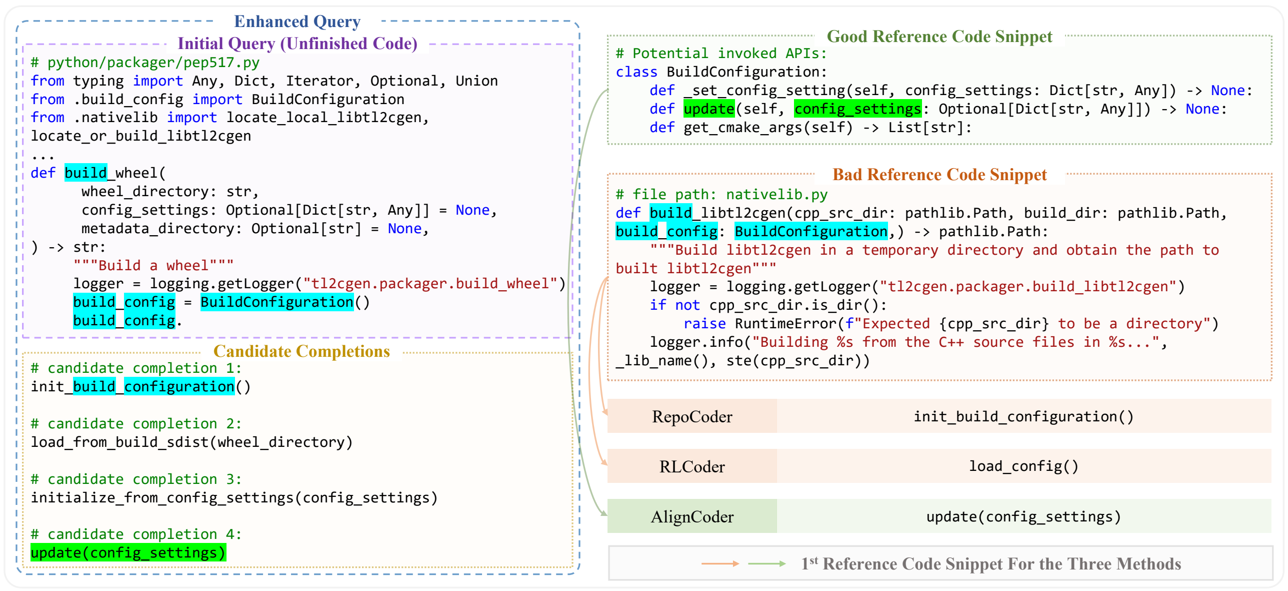
Our ablation study reveals performance patterns: (1) when w/o DC and w/o QH, both EM and ES scores consistently decrease across benchmarks. The only exception is a negligible 0.08% improvement in ES for RepoEval line-level tasks in the w/o DC setting. (2) Performance degradation becomes more pronounced under w/o RL. The experimental results show that w/o RL has a significant impact on AlignCoder's performance, which demonstrates that merely considering dependency context and query enhancement mechanism is insufficient, and it is also necessary to train the retriever to learn to utilize the additional information in the query.

RQ3 Summary: Our ablation studies demonstrate the critical roles of three components. Results show that w/o RL causes greater performance drops. This indicates that dependency context and query enhancement are incomplete without training the retriever to exploit the additional information in the enhanced query.

D. RQ4: Sampling Parameter Stability

In this section, we analyze the influence of two sampling parameters, temperature and top-p, on AlignCoder's performance when sampling candidate completions. Temperature and top-p are two crucial parameters that control the randomness and diversity of generation. Temperature regulates the degree of randomness in generated text, while top-p constrains the candidate vocabulary by sampling only from the highest-probability tokens whose cumulative probability reaches the specified p value. Since we employ the vLLM framework to accelerate model inference, the temperature and top-p settings in AlignCoder follow the default parameter configurations for the model sampling process as specified in the vLLM documentation, with temperature set to 0.8 and top-p set to 0.95. We conducted a series of experiments to examine the stability of AlignCoder's performance varying temperature (0.85, 0.75, 0.7, 0.65, 0.6) and top-p (0.9, 0.85, 0.8, 0.75, 0.7) values when using the optimal sampling number 4.

Our experiments used DeepSeekCoder-1B as the generator, Figure 5 demonstrates that parameter adjustments typically lead to modest performance changes. Most configurations exhibit EM and ES score variations of less than 1% across both benchmarks. However, five configurations show differ-


Enhanced Query
Initial Query (Unfinished Code)
Candidate Completions
Good Reference Code Snippet
Bad Reference Code Snippet
1st Reference Code Snippet For the Three Methods

```
# python/packager/pep517.py
from typing import Any, Dict, Iterator, Optional, Union
from .build_config import BuildConfiguration
from .nativelib import locate_local_libtl2cgen,
locate_or_build_libtl2cgen
...
def build_wheel(
      wheel_directory: str,
      config_settings: Optional[Dict[str, Any]] = None,
      metadata_directory: Optional[str] = None,
) -> str:
     """Build a wheel"""
     logger = logging.getLogger("tl2cgen.packager.build_wheel")
     build_config = BuildConfiguration()
     build_config.
```

```
# candidate completion 1:
init_build_configuration()

# candidate completion 2:
load_from_build_sdist(wheel_directory)

# candidate completion 3:
initialize_from_config_settings(config_settings)

# candidate completion 4:
update(config_settings)
```

```
# Potential invoked APIs:
class BuildConfiguration:
    def _set_config_setting(self, config_settings: Dict[str, Any]) -> None:
    def update(self, config_settings: Optional[Dict[str, Any]]) -> None:
    def get_cmake_args(self) -> List[str]:
```

```
# file path: nativelib.py
def build_libtl2cgen(cpp_src_dir: pathlib.Path, build_dir: pathlib.Path,
build_config: BuildConfiguration,) -> pathlib.Path:
    """Build libtl2cgen in a temporary directory and obtain the path to
built libtl2cgen"""
    logger = logging.getLogger("tl2cgen.packager.build_libtl2cgen")
    if not cpp_src_dir.is_dir():
        raise RuntimeError(f"Expected {cpp_src_dir} to be a directory")
    logger.info("Building %s from the C++ source files in %s...",
_lib_name(), ste(cpp_src_dir))
```

RepoCoder	init_build_configuration()
RLCoder	load_config()
AlignCoder	update(config_settings)

Fig. 6. Case study, task id: *project_cc_python/5407*. The highlighted words represent identical tokens that exist in both the query and the reference code.

ences: a 2.4% EM decrease on CrossCodeEval Python under temperature at 0.8 and top-p at 0.7, for instance.

RQ4 Summary: AlignCoder demonstrates robust performance under different temperature and top-p sampling configurations, consistently exhibiting stability in EM and ES metrics across most experimental configurations.

E. Case Study

We illustrate the effectiveness of AlignCoder through a case study. In the Figure 6, the model needs to complete the function `update` of the `BuildConfiguration` class. The highest similarity code snippet retrieved through AlignCoder is a dependency code snippet related to the `Validator` class. Based on this snippet, the generator is able to generate the correct completion. However, the most similar code snippet retrieved by RLCoder or RepoCoder is a base code snippet highly similar to the unfinished code, ultimately leading the model to produce an incorrect result.

VI. Related Work

A. Code Completion

Code completion has been a fundamental task in software engineering, aiming to predict subsequent code tokens or statements to assist programmers during development [8], [43]. Traditional approaches relied on rule-based methods or statistical methods [44]–[47], but recent advances have been driven by deep learning techniques. Modern neural approaches [48]–[50] have significantly improved completion accuracy by learning from large codebases. The emergence of large language models has further revolutionized this field, with studies exploring LLM-based completion systems [28], [51]. Retrieval-augmented generation (RAG) has become particularly influential, with methods like RedCoder [52] enhancing code generation relevant code snippets retrieval, DocPrompting [53] leveraging documentation for unseen functions, and AceCoder [54] integrating examples retrieval with guided generation to improve completion quality.

B. Repository-Level Code Completion

Repository-level code completion extends traditional completion by leveraging broader repository context to improve accuracy and relevance [17], [18], [30], [31], [55], [56]. Existing approaches can be categorized into several paradigms. Learning-based methods such as CoCoMIC [31] and RepoHyper [18] employ dependency analysis and adaptive learning mechanisms, though they face challenges in training data acquisition and generalizability. Static analysis approaches, including CodePlan [55], RepoFuse [30], and A^3-CodeGen [56], utilize program analysis techniques to identify relevant code candidates. GraphCoder [13] captures the context by leveraging the structural information in the source code via a constructed code context graph. Iterative retrieval strategies have been explored by RepoCoder [17], which performs multi-round retrieval and generation, and De-Hallucinator [57], which refines completion through iterative processes. Tool-augmented methods such as CodeAgent [58] and ToolGen [59] investigate external tool invocation, while RLCoder [26] employs reinforcement learning for repository-specific retrieval optimization.

Despite these advances, AlignCoder differs by leveraging LLMs' powerful inference capabilities to obtain multiple reference code snippets, combining this with a reinforcement learning-based approach for training the retriever to achieve more accurate retrieval and superior repository-level code completion performance.

VII. Conclusion

In this paper, we present AlignCoder, a repository-level code completion framework that addresses the fundamental challenges of query-target misalignment and ineffective inference information utilization in existing retrieval-augmented generation approaches. Our key contributions include a query enhancement mechanism that generates multiple candidate completions to bridge the semantic gap between queries and target codes, and a reinforcement learning-based Align-retriever that learns to leverage inference information for more precise retrieval. Extensive experiments on CrossCodeEval and RepoEval benchmarks across five backbone models demonstrate the effectiveness of our approach, achieving an 18.1% improvement in EM score on the Python subset of CrossCodeEval and showing strong generalizability across various code LLMs and programming languages. This work contributes to the development of repository-level code generation tools and may help improve developer productivity in programming tasks.

Acknowledgements

This work is supported by CCF-Huawei Populus Grove Fund CCF-HuaweiSE202403.

References

[1] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

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

[3] 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.

[4] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[5] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, Y. Wang, and J.-G. Lou, "Large language models meet nl2code: A survey," *arXiv preprint arXiv:2212.09420*, 2022.

[6] Z. Zhang, C. Chen, B. Liu, C. Liao, Z. Gong, H. Yu, J. Li, and R. Wang, "Unifying the perspectives of nlp and software engineering: A survey on language models for code," *arXiv preprint arXiv:2311.07989*, 2023.

[7] 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.

[8] M. Izadi, J. Katzy, T. Van Dam, M. Otten, R. M. Popescu, and A. Van Deursen, "Language models for code completion: A practical evaluation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[9] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley, "Longcoder: A long-range pre-trained language model for code completion," in *International Conference on Machine Learning*. PMLR, 2023, pp. 12 098–12 107.

[10] Q. Ren, C. Gao, J. Shao, J. Yan, X. Tan, W. Lam, and L. Ma, "Codeattack: Revealing safety generalization challenges of large language models via code completion," *arXiv preprint arXiv:2403.07865*, 2024.

[11] D. Shrivastava, D. Kocetkov, H. de Vries, D. Bahdanau, and T. Scholak, "Repofusion: Training code models to understand your repository," *arXiv preprint arXiv:2306.10998*, 2023.

[12] Z. Tang, J. Ge, S. Liu, T. Zhu, T. Xu, L. Huang, and B. Luo, "Domain adaptive code completion via language models and decoupled domain databases," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 421–433.

[13] W. Liu, A. Yu, D. Zan, B. Shen, W. Zhang, H. Zhao, Z. Jin, and Q. Wang, "Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model," *arXiv preprint arXiv:2406.07003*, 2024.

[14] F. Shi, X. Chen, K. Misra, N. Scales, D. Dohan, E. H. Chi, N. Schärli, and D. Zhou, "Large language models can be easily distracted by irrelevant context," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 210–31 227.

[15] O. Yoran, T. Wolfson, O. Ram, and J. Berant, "Making retrieval-augmented language models robust to irrelevant context," *arXiv preprint arXiv:2310.01558*, 2023.

[16] 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.

[17] 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," *arXiv preprint arXiv:2303.12570*, 2023.

[18] 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," *CoRR*, 2024.

[19] J. Wang, Y. He, and H. Chen, "Repogenreflex: Enhancing repository-level code completion with verbal reinforcement and retrieval-augmented generation," *arXiv preprint arXiv:2409.13122*, 2024.

[20] T.-D. Bui, D.-T. Luu-Van, T.-P. Nguyen, T.-T. Nguyen, S. Nguyen, and H. D. Vo, "Rambo: Enhancing rag-based repository-level method body completion," *arXiv preprint arXiv:2409.15204*, 2024.

[21] M. Liang, X. Xie, G. Zhang, X. Zheng, P. Di, W. Jiang, H. Chen, C. Wang, and G. Fan, "Repogenix: Dual context-aided repository-level code completion with language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2466–2467.

[22] 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.

[23] P. Jaccard, "The distribution of the flora in the alpine zone. 1," *New phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[24] 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.

[25] 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.

[26] Y. Wang, Y. Wang, D. Guo, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, "Rlcoder: Reinforcement learning for repository-level code completion," *arXiv preprint arXiv:2407.19487*, 2024.

[27] L. Wang, N. Yang, and F. Wei, "Query2doc: Query expansion with large language models," 2023. [Online]. Available: https://arxiv.org/abs/2303.07678

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] J. Liu, Y. Chen, M. Liu, X. Peng, and Y. Lou, "Stall+: Boosting llm-based repository-level code completion with static analysis," *arXiv preprint arXiv:2406.10018*, 2024.

[33] M.Brunsfeld, P.Thomson, A.Hlynskyi, J.Vera, P.Turnbull, T.Clem, D.Creager, A.Helwer, R.Rix, H. Antwerpen, M.Davis, Ika, T.-A.Nguyen, S.Brunk, N.Hasabnis, bfredl, M.Dong, V.Panteleev, ikrima, S.Kalt, K.Lampe, A.Pinkus, M.Schmitz, M.Krupcale, narpfel, S.Gallegos,

V.Martí, Edgar, and G.Fraser, "Tree-sitter," site:https://github.com/tree-sitter/tree-sitter, 2020, accessed: 2023-11-03.

[34] W. Cheng, Y. Wu, and W. Hu, "Dataflow-guided retrieval augmentation for repository-level code completion," *arXiv preprint arXiv:2405.19782*, 2024.

[35] 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.

[36] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," *arXiv preprint arXiv:2312.02120*, 2023.

[37] 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.

[38] 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.

[39] 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.

[40] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 6769–6781. [Online]. Available: https://aclanthology.org/2020.emnlp-main.550/

[41] X. Ma, K. Sun, R. Pradeep, and J. Lin, "A replication study of dense passage retriever," 2021. [Online]. Available: https://arxiv.org/abs/2104.05740

[42] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[43] 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.

[44] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[45] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014, pp. 419–428.

[46] 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.

[47] 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.

[48] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel, "Learning python code suggestion with a sparse pointer network," *arXiv preprint arXiv:1611.08307*, 2016.

[49] 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.

[50] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with bayesian networks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–31, 2015.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.