

A Deep Dive into Retrieval-Augmented Generation for Code Completion: Experience on WeChat

Zezhou Yang

Tencent

Guangzhou, China

zezhouyang@tencent.com

Ting Peng

Tencent

Guangzhou, China

sakurapeng@tencent.com

Cuiyun Gao*

The Chinese University of Hong Kong

Hong Kong, China

cuiyungao@outlook.com

Chaozheng Wang

The Chinese University of Hong Kong

Hong Kong, China

adf11178@gmail.com

Hailiang Huang

Tencent

Guangzhou, China

eraserhuang@tencent.com

Yuetang Deng

Tencent

Guangzhou, China

yuetangdeng@tencent.com

Abstract—Code completion, a crucial task in software engineering that enhances developer productivity, has seen substantial improvements with the rapid advancement of large language models (LLMs). In recent years, retrieval-augmented generation (RAG) has emerged as a promising method to enhance the code completion capabilities of LLMs, which leverages relevant context from codebases without requiring model retraining. While existing studies have demonstrated the effectiveness of RAG on public repositories and benchmarks, the potential distribution shift between open-source and closed-source codebases presents unique challenges that remain unexplored. To mitigate the gap, we conduct an empirical study to investigate the performance of widely-used RAG methods for code completion in the industrial-scale codebase of WeChat, one of the largest proprietary software systems. Specifically, we extensively explore two main types of RAG methods, namely identifier-based RAG and similarity-based RAG, across 26 open-source LLMs ranging from 0.5B to 671B parameters. For a more comprehensive analysis, we employ different retrieval techniques for similarity-based RAG, including lexical and semantic retrieval. Based on 1,669 internal repositories, we achieve several key findings: (1) both RAG methods demonstrate effectiveness in closed-source repositories, with similarity-based RAG showing superior performance, (2) the effectiveness of similarity-based RAG improves with more advanced retrieval techniques, where BM25 (lexical retrieval) and GTE-Qwen (semantic retrieval) achieve superior performance, and (3) the combination of lexical and semantic retrieval techniques yields optimal results, demonstrating complementary strengths. Furthermore, we conduct a developer survey to validate the practical utility of RAG methods in real-world development environments.

Index Terms—large language model, retrieval-augmented generation, code completion

I. INTRODUCTION

Code completion, which automatically predicts and suggests code fragments based on the surrounding programming context, has evolved from simple token-level suggestions to

generating entire code blocks [1], [2]. Studies have demonstrated that code completion tools substantially enhance developer productivity in real-world software development [3], [4]. Notably, 87% of professional developers report significant improvements in their coding efficiency when utilizing code completion tools in industrial settings [5]. Recent advances in large language models (LLMs) have further transformed various software engineering tasks [6]–[10], demonstrating unprecedented capabilities in code understanding and generation. These models have achieved particularly impressive performance in code completion tasks [11]–[13].

To enhance LLMs’ performance on domain-specific tasks, researchers have explored Retrieval-Augmented Generation (RAG), which augments model inference by retrieving and incorporating relevant context from the target codebase without requiring parameter updates [14], [15]. The emergence of RAG methods provides a promising approach to leverage the powerful abilities of LLMs for industrial software development [16]. It not only preserves the privacy of proprietary code but also enables models to adapt to specific coding styles. While RAG for code completion has shown promising results on public repositories and benchmarks [17], [18], the characteristics of closed-source codebases present unique challenges. Closed-source repositories often contain proprietary code patterns, custom frameworks, and domain-specific implementations that differ from open-source codebases [19]. The inherent differences raise practical concerns about the applicability of RAG methods in proprietary settings. However, a systematic study of RAG for code completion in closed-source repositories remains unexplored, leaving practitioners without clear guidance for leveraging RAG in proprietary development environments.

To mitigate the gap, we conduct a comprehensive investigation of RAG-based code completion in industrial settings, using the proprietary codebase from the WeChat Group in Tencent. As one of the largest social platforms with over 1 billion monthly active users [20], WeChat maintains a sophisticated codebase with internal well-defined development practices and complex business logic, providing a compelling real-world environment for evaluating RAG methods at an

*Cuiyun Gao is the corresponding author.

This research is supported by National Key R&D Program of China (No. 2022YFB3103900), National Natural Science Foundation of China under project (No. 62472126), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen-Hong Kong Jointly Funded Project (Category A, No. SGDX20230116091246007), and Shenzhen Basic Research (General Project No. JCYJ20220531095214031).

industrial scale. To enable systematic evaluation of RAG methods, we conduct two essential preparatory steps. First, we construct a carefully curated evaluation benchmark comprising 100 examples across seven domains, with manually annotated context and comments that reflect real-world code completion scenarios. Second, we collect 1,669 internal repositories from WeChat’s development ecosystem as the source of the retrieval corpus. Following this, we propose a data preprocessing algorithm that extracts multiple pieces of context information to construct a fine-grained retrieval corpus [21], [22].

Our experiments evaluate 26 open-source LLMs ranging from 0.5B to 671B parameters, thoroughly investigating the capabilities of RAG methods in closed-source scenarios. Specifically, we address three key research questions:

RQ1: How do different RAG methods perform in closed-source code completion? This research question compares two types of RAG paradigms: identifier-based RAG, which retrieves relevant definitions of identifiers to help LLMs understand their inner logic and usage; and similarity-based RAG, which retrieves similar code implementations using lexical (BM25) and semantic (CodeBERT, UniXcoder, CoCoSoDa, and GTE-Qwen) retrieval techniques. Our experimental results demonstrate the effectiveness of both types of methods, with similarity-based RAG exhibiting superior performance across different model scales.

RQ2: How do the retrieval techniques affect similarity-based RAG on code completion? We investigate the impact of using different query formulations (incomplete code context and complete code snippets) across various retrieval techniques within similarity-based RAG. Our experimental results reveal that lexical retrieval consistently shows strong performance across different code completion models, while the effectiveness of semantic retrieval scales positively with model capacity. Moreover, most retrieval techniques perform better with complete code snippets, whereas GTE-Qwen demonstrates superior performance with incomplete code context.

RQ3: What is the relationship between different types of retrieval techniques in similarity-based RAG? To answer this research question, we first conduct a comparative analysis between lexical (BM25) and semantic retrieval techniques by examining their retrieved results. Despite achieving similar performance levels individually, we discover minimal overlap in their retrieved candidates, which suggests they capture fundamentally different aspects of code similarities. Building upon this observation, we find that combining BM25 with GTE-Qwen yields optimal performance across most LLMs, demonstrating the value of hybrid approaches in RAG-based code completion.

The paper makes the following contributions:

- 1) We conduct a systematic study of retrieval-augmented code completion on closed-source codebases, providing comprehensive empirical insights into the effectiveness of different RAG methods using 26 open-source LLMs.
- 2) We propose a data preprocessing algorithm for constructing a fine-grained retrieval corpus from large-scale

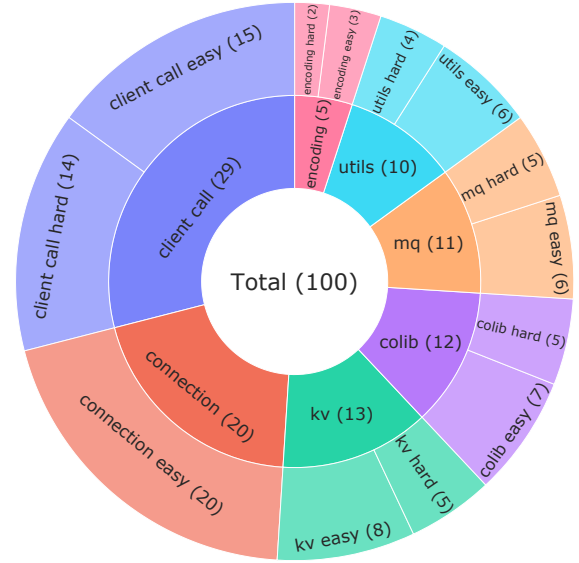


Fig. 1. Statistics of our benchmark.

codebases, addressing the challenge of context extraction in RAG for code completion.

- 3) Our experiment results reveal the complementary nature of lexical and semantic retrieval techniques, demonstrating that their combination can further enhance RAG-based code completion performance.
- 4) We validate our empirical findings through a developer survey, confirming that the observed performance improvements align with developers’ practical experiences in real-world scenarios.

II. RETRIEVAL-AUGMENTED CODE COMPLETION

A. Benchmark Construction

To comprehensively evaluate RAG’s performance in closed-source scenarios, we construct a function-level evaluation benchmark through manual annotation. The annotation process involves three senior developers from our group, each with over five years of industrial experience, following four rules:

- **Function Significance:** Selected functions must be integral to the daily development workflow within the codebase, representing real-world code completion challenges.
- **Context Selection:** Annotators manually identify relevant context and documentation (including line numbers and necessary explanations) based on their professional experience, simulating realistic code completion scenarios.
- **Difficulty Classification:** Each example is categorized as either ‘easy’ or ‘hard’ based on the complexity of the required completion, facilitating fine-grained analysis.
- **Quality Assurance:** All selected functions come from production systems that have undergone rigorous testing and are actively used in online environments. This ensures their correctness and practical value. After the initial annotation, all three developers perform cross-validation to ensure consistency and adherence to guidelines.

After three weeks of effort, we obtain a benchmark dataset containing 100 examples across seven domains. These domains cover essential enterprise development scenarios, ranging from remote procedure calls (client call), coroutine (connection, colib, and encoding), data storage operations (kv), message queue (mq), and utility functions (utils), representing common patterns in the software development of WeChat. Figure 1 illustrates the distribution of examples across domains and difficulty levels.

B. Retrieval Corpus Construction

We collect 1,669 internal projects as our retrieval database source. These projects span multiple business units and development cycles, providing a comprehensive codebase for retrieval. To ensure data quality, we filter out duplicate code snippets and standardize the code format. However, constructing a retrieval corpus from closed-source C++ projects presents several unique challenges:

- **File Segmentation:** C++ heavily relies on header files for dependency management, which often contain extensive object declarations and definitions [23]. Using entire files as retrieval units would result in excessively long retrieved segments. Meanwhile, applying sliding window approaches to split these files would likely fragment object contents and break their semantic integrity [24].
- **Recursive Dependencies:** Header files frequently reference other header files, creating recursive dependency structures. This complexity prevents the direct application of dependency resolution approaches used in previous works [1], [11], [13].
- **Auto-generated Code:** Protocol Buffers (protobuf) is a language-agnostic data serialization format that allows developers to define structured data in proto files. While the protobuf compiler can automatically generate corresponding C++ files, these generated files contain numerous templates and predefined content that are irrelevant to the actual function logic [22].
- **Macro Specificity:** Unlike other object-oriented languages such as Python and Java, C++ extensively uses macro declarations, definitions, and implementations that play crucial roles in code functionality. These specific features require special handling during preprocessing.

To address these challenges, we develop a fine-grained preprocessing algorithm that extracts relevant objects from our internal codebase and uses these extracted objects as the basic units in our retrieval corpus. This granular organization ensures more precise and relevant retrievals while maintaining the semantic integrity of the code. Algorithm 1 presents our solution, which addresses each challenge as follows:

(1) To address the segmentation challenge, we extract class definitions and function definitions/declarations separately from cpp and header files, enabling fine-grained slicing and corpus construction. This is implemented as the main workflow in Algorithm 1.

(2) For recursive dependencies, we process all header files referenced in C++ source files recursively rather than only

Algorithm 1 Fine-grained Data Preprocessing

Input: Source file f , Processed headers set H

Output: Function definitions F_{def} , declarations F_{dec} , class definitions C , or message definitions M

```

1: Initialize  $F_{def}, F_{dec}, C, M, F_{macro} \leftarrow \emptyset$ 
2: if  $f.type = "proto"$  then
3:    $M \leftarrow \text{ExtractProtoMessages}(f)$ 
4:   return  $M$ 
5: else if  $f.type = "cpp"$  then
6:   // Extract elements from current file
7:    $C_f, F_{def}, F_{dec}, F_{macro} \leftarrow \text{Extract}(f, \emptyset)$ 
8:    $headers \leftarrow \text{GetRecursiveDependencies}(f)$ 
9:   for each  $h \in headers$  do
10:    if  $h \notin H$  then
11:      // Recursively process dependent headers
12:       $C_h, F_{def\_h}, F_{dec\_h}, F_{macro\_h} \leftarrow \text{Extract}(h, H)$ 
13:       $C \leftarrow C \cup C_h$ 
14:       $F_{def} \leftarrow F_{def} \cup F_{def\_h}$ 
15:       $F_{dec} \leftarrow F_{dec} \cup F_{dec\_h}$ 
16:       $F_{macro} \leftarrow F_{macro} \cup F_{macro\_h}$ 
17:       $H \leftarrow H \cup \{h\}$ 
18:    end if
19:  end for
20:  // Process macros
21:  for each  $m \in F_{macro}$  do
22:     $F_{def\_m}, F_{dec\_m} \leftarrow \text{TransformMacro}(m)$ 
23:     $F_{def} \leftarrow F_{def} \cup F_{def\_m}$ 
24:     $F_{dec} \leftarrow F_{dec} \cup F_{dec\_m}$ 
25:  end for
26:  // Remove redundant whitespace and comments
27:   $F_{def}, F_{dec}, C \leftarrow \text{Format}(F_{def}, F_{dec}, C)$ 
28:  return  $F_{def}, F_{dec}, C$ 
29: else
30:   Error: Unsupported file type!
31: end if

```

considering first-level dependencies. This process (lines 8-18 of Algorithm 1) ensures comprehensive dependency coverage while avoiding noise from unused dependencies.

(3) To handle auto-generated code issues, we first remove all the cpp and header files automatically generated from proto files. Considering that each protobuf message directly corresponds to a C++ class in the generated code, we design a specialized function `ExtractProtoMessages` (lines 2-4 of Algorithm 1) to extract message definitions directly from the more concise and structured protobuf files.

(4) For macro-specific features, we extract macro-related definitions and declarations from both cpp and header files, transforming them into function-like structures through pattern conversion. This transformation process is handled in lines 21-25 of Algorithm 1.

Specifically, a comprehensive retrieval corpus can be constructed from multiple C++ projects. The extraction process operates on both C++ source files (.cpp) and header files (.h), systematically identifying and extracting three types

of background knowledge. The set F_{def} encompasses all function definitions, and the set F_{dec} represents all function declarations, including both member functions and standalone functions. The set C contains all class definitions existing in the codebase, representing the object-oriented structure of the system. The extraction process also operates on protobuf files (.proto). The set M consists of protobuf message definitions, which are particularly important as they define the data structures used for communication and serialization.

C. Identifier-based RAG

Identifier-based retrieval-augmented generation aims to enhance code completion performance by incorporating the knowledge of relevant class, function, and protobuf message from the retrieval corpus. This section details the three main steps of this method: (1) index creation, (2) identifier extraction, and (3) prompt construction and code completion.

1) *Index Creation*: To facilitate efficient retrieval, we construct an indexed codebase that enables quick lookup of background knowledge based on the identifier and search type:

$$background_knowledge = Lookup(identifier, type) \quad (1)$$

This *Lookup* function provides a service to access the background knowledge of specific objects, where *identifier* refers to a string that uniquely identifies objects, such as the protobuf message name, class name, or function name. Type represents the search type, including protobuf message definition, function declaration, function definition, and class definition.

2) *Identifier Extraction*: The second step leverages a powerful LLM to analyze the current code snippet and identify relevant and important references that require definition lookup. Through carefully designed prompts, the LLM understands the code context and extracts three sets of identifiers:

$$M_{req}, F_{req}, C_{req} = Need_To_Lookup(current_code) \quad (2)$$

where M_{req} represents required protobuf message definitions, F_{req} represents required function definitions and declarations, and C_{req} represents required class definitions. Once the required references are identified, the corresponding background knowledge can be retrieved from the indexed codebases.

3) *Prompt Construction and Code Completion*: The final step involves constructing specialized prompts for different types of background knowledge and finishing code completions. The process is formalized as:

$$generated_code = LLM(prompt_template_{type}, knowledge_{type}, current_code) \quad (3)$$

where *type* has the same choices in Section II-B. In detail, we develop four distinct prompt templates to help LLMs understand different types of background knowledge.

D. Similarity-based RAG

Similarity-based retrieval augmented generation improves the performance of LLMs on code completion by providing code snippets similar to the current code. This process also includes three main steps: (1) index creation, (2) similar code retrieval, and (3) prompt construction and code completion.

1) *Index Creation*: Given that our primary focus is function completion, we only utilize function definitions as the retrieval source for the similarity-based RAG method. The construction of the retrieval corpus depends on the chosen similarity-based retrieval technique. The indexing process of lexical retrieval is shown as follows:

$$\{(term_i, tf_i, idf_i) | term_i \in F_{def}\} \rightarrow lexical_index \quad (4)$$

where tf_i represents the term frequency and idf_i represents the inverse document frequency of each term in the corpus. The indexing process of semantic retrieval is formalized as:

$$\{encode(f) | f \in F_{def}\} \rightarrow semantic_index \quad (5)$$

where each function is encoded into a fixed-dimensional embedding space and stored in a vector database for efficient semantic search.

2) *Similar Code Retrieval*: In this step, the current code snippet serves as a query to retrieve similar code from the previously constructed retrieval corpus. The specific retrieval process depends on the chosen retrieval technique.

For lexical retrieval, the term frequencies of the current code are calculated, and the similarity is then computed based on the TF-IDF weights:

$$similar_code = \underset{term \in query}{argmax_{f \in F_{def}}} \sum TF-IDF(term, f) \quad (6)$$

For semantic retrieval, the current code is first encoded into an embedding with the same dimensionality as those in the retrieval corpus. The similarity is then measured using cosine similarity:

$$similar_code = \underset{encode(f)}{argmax_{f \in F_{def}}} \cos(encode(query), encode(f)) \quad (7)$$

3) *Prompt Construction and Code Completion*: In this step, similar code snippets are integrated into the input of LLMs by prompt construction. Specifically, the retrieved similar code snippets are directly concatenated with the current code to construct the prompt:

$$generated_code = LLM(prompt_template_{similar}, similar_code \oplus current_code) \quad (8)$$

where \oplus represents the concatenation operation. In the prompt template, we encourage LLMs to complete the current code according to the similar code snippets.

III. EXPERIMENT STUDY SETUP

A. Similarity-based Retrieval Techniques

In our experimental evaluation, we employ five distinct similarity-based retrieval techniques, including one lexical technique (i.e., BM25 [25]) and four semantic techniques (i.e., CodeBERT [26], UniXcoder [27], CoCoSoDa [28], and GTE-Qwen [29]):

BM25 [25] extends the basic TF-IDF approach by incorporating document length normalization and non-linear term frequency scaling. In our implementation, we use BM25 to retrieve similar code snippets from a function corpus F . For a given code query, BM25 first tokenizes it into terms $\{t_1, t_2, \dots, t_n\}$ and computes a relevance score for each function $f \in F$ as:

$$Score(query, f) = \sum_{i=1}^n (IDF_i \cdot TF_{mod}(t_i, f)) \quad (9)$$

The IDF component is calculated similarly to the standard TF-IDF approach, but with smoothing factors:

$$IDF_i = \log \frac{N - df_i + 0.5}{df_i + 0.5} \quad (10)$$

where N is the total number of functions in the corpus and df_i is the number of functions containing term t_i . The modified term frequency component TF_{mod} introduces saturation and length normalization:

$$TF_{mod}(t_i, f) = \frac{tf_i \cdot (k + 1)}{tf_i + k \cdot (1 - b + b \cdot \frac{len_f}{len_{avg}})} \quad (11)$$

where tf_i is the frequency of term t_i in function f , len_f is the length of function f , and len_{avg} is the average function length in the corpus. Parameters k and b control term frequency scaling and length normalization, respectively.

CodeBERT [26] is built upon the RoBERTa-base and pre-trained on both natural language and programming language corpus. It employs two types of objective modeling tasks, including Masked Language Modeling (MLM) and Replaced Token Detection (RTD), which enables the model to learn general representations.

UniXcoder [27] is pre-trained using three objectives: masked language modeling, unidirectional language modeling, and denoising tasks. Additionally, UniXcoder incorporates two pre-training strategies: (1) a multi-modal contrastive learning approach that leverages Abstract Syntax Tree (AST) to enhance code representations, and (2) a cross-modal generation task utilizing code comments to align embeddings across different programming languages.

CoCoSoDa [28], which has the same architecture as UniXcoder, enhances code representation through momentum contrastive learning. Specifically, the model employs dynamic data augmentation and negative sampling strategies to learn more robust and discriminative code embeddings.

GTE-Qwen [29], a member of the GTE (General Text Embedding) family, is a decode-only retrieval model based on Qwen2 [30]. Different from the general Qwen model,

GTE-Qwen incorporates bidirectional attention mechanisms to enrich contextual representations, making it particularly effective for multilingual understanding and retrieval tasks. We utilize GTE-Qwen2-1.5B-instruct in our experiment.

B. Large Language Models

To comprehensively evaluate code generation capabilities across different model scales, we conduct experiments with 26 open-source LLMs, ranging from 0.5B to 671B parameters.

Our evaluation encompasses both code-specialized LLMs and general-purpose LLMs. For code-specialized models, we select several prominent and emerging model series, including Qwen-Coder [31], Deepseek-Coder [32], [33], CodeLlama [34], Yi-Coder [35], OpenCoder [36], and Codestral [37]. For each series, we incorporate their latest versions across different parameter scales to ensure comprehensive coverage. To complement our evaluation and fill the gaps in model scale coverage, we also include several general-purpose LLMs, namely Llama-3.2-1B-Instruct, Llama-3.2-3B-Instruct [38], Llama-3.1-8B-Instruct [39], Llama-3.3-70B-Instruct [40], Qwen2.5-72B-Instruct [41], as well as DeepSeek-V2.5 [33] and DeepSeek-V3 [42].

C. Metrics

CodeBLEU (CB) [43] extends the BLEU metric [44] by incorporating additional factors to capture the structural and semantic aspects of code. Specifically, CodeBLEU is formally defined as a weighted combination of four factors:

$$CodeBLEU = \alpha \cdot N\text{-gram}_{match} + \beta \cdot N\text{-gram}_{weighted} + \gamma \cdot Similarity_{AST} + \delta \cdot Similarity_{DF} \quad (12)$$

where $N\text{-gram}_{match}$ measures the overlap of code tokens between the generated and reference code. $N\text{-gram}_{weighted}$ applies different weights to tokens based on their importance in context. $Similarity_{AST}$ evaluates the structural similarity by calculating the syntactic AST matching score, and $Similarity_{DF}$ assesses the semantic equivalence through data flow analysis. The coefficients α , β , γ , and δ control the relative contribution of each factor, allowing for flexible adaptation to different evaluation scenarios. In our experiments, all four coefficients are set to 0.25.

Edit Similarity (ES) measures the minimal number of token-level operations required to transform one string into another, normalized by the length of the longer string. Formally, given a generated code snippet c_g and its reference c_r , the edit similarity is calculated as:

$$ES(c_g, c_r) = 1 - \frac{EditDistance(c_g, c_r)}{\max(|c_g|, |c_r|)} \quad (13)$$

where $EditDistance(c_g, c_r)$ refers to the Levenshtein distance between two strings, counting the minimum number of single-token insertions, deletions, or substitutions needed to transform c_g into c_r . The resulting score ranges from 0 to 1, where 1 indicates identical sequences and 0 represents completely different strings.

For better readability, we present the two metrics as percentage scores (multiplied by 100) in our experimental results.

D. Implementation Details

1) *Data Preprocessing*: We implement the data preprocessing process following the algorithm outlined in Section II-B. For the `Extract` function, we employ the tree-sitter and s-expression pattern matching to efficiently extract the required patterns in constant time. Different from processing C++ source files and header files with tree-sitter, the protobuf files cannot be parsed into ASTs directly. To overcome this limitation, we carefully study the official protobuf documentation and design a regular expression-based approach to extract Message definitions.

For macro transformations, we develop a systematic approach based on three key components: macro parameters, internal logic, and return types. The macros can be converted into function-like structures by using macro names as function names and macro parameters as function parameters. Macros without internal implementation logic are transformed into function declarations, while those with both internal logic and return types are converted into function definitions.

2) *Retrieval Service*: For identifier-based RAG, we implement a retrieval service that provides specific background knowledge based on identifiers and retrieval types. Specifically, we adopt Tantivy¹, an efficient full-text search engine library, to construct separate retrieval databases for different retrieval types (e.g., protobuf message definition, function declaration, function definition, and class definitions). Each identifier serves as a unique index within its corresponding database. A unified `Lookup` function, mentioned in Section II-C1, provides the interface for retrieving specific background knowledge. As discussed in Section II-C2, the `Need_To_Lookup` function is needed to identify the specific type and identifier that require background knowledge. In our experiments, we employ Qwen2.5-72B-Instruct, due to its powerful ability to understand code, to extract protobuf messages, classes, and functions that lack sufficient background information for accurate completion.

For similarity-based RAG, the construction of retrieval databases varies depending on the specific similarity-based retrieval techniques. We utilize BM25S [45], a recently released efficient BM25 library, to implement the lexical retrieval service. For semantic retrieval, we employ Qdrant², a high-performance vector database, to construct and manage our retrieval database. The number of retrieved results is set to 4, ensuring that the length of the constructed prompt is less than 2k tokens, which aligns with the context length supported by most LLMs.

3) *Model Deployment and Inference Settings*: All LLMs and their corresponding tokenizers are obtained from their official Hugging Face repositories and deployed using the vLLM framework within Docker containers on the WeChat

testing platform. To ensure reproducibility and consistency of our experimental results, we maintain consistent inference parameters across all models, setting the temperature to 0 during generation. The hardware configuration for model inference varies according to model size: models with parameters under 200B are tested using 8 NVIDIA A100 GPUs (40GB each), while DeepSeek-Coder-V2-Instruct and DeepSeek-V2.5 are deployed on 8 NVIDIA H20 GPUs (96GB each). For the larger model, DeepSeek-V3, we utilize a cluster of 16 NVIDIA H20 GPUs (96GB each) to accommodate its extensive computational requirements. Except for DeepSeek-V3, which uses FP8, all other LLMs use FP16 precision for inference. To align with the structure format of our retrieval corpus and benchmark, we design our prompts in Chinese wrapped in C++ comment format for the closed-source code completion.

IV. EMPIRICAL RESULTS

A. RQ1: Effectiveness of RAG

As shown in Table I, the experimental results demonstrate that different RAG methods consistently outperform base models across different scales. For instance, the CB/ES metrics of Llama-3.1-8B-Instruct improve from 34.02/46.07 to 39.64/49.35 with function definition retrieval, when using GTE-Qwen-based RAG, Qwen2.5-Coder-14B-Instruct achieves a CB/ES metrics improvement from 29.79/48.56 to 51.12/61.96, representing a 71.60% and 27.59% relative increase, respectively. This enhancement pattern can be observed in larger models as well, where DeepSeek-V3 shows an enhancement from 35.23/54.85 to 60.28/73.11 using GTE-Qwen retrieval technique, corresponding to a 71.1% and 33.3% increase in performance metrics.

Among the identifier-based RAG methods, function definition retrieval consistently yields the highest performance gains. This is particularly evident in models like Qwen2.5-Coder-32B-Instruct, where function definition retrieval improves the base CB/ES metrics from 38.05/57.89 to 42.23/60.44, outperforming other background knowledge, such as message definition and class definition. In the similarity-based RAG, BM25 and GTE-Qwen-based retrieval techniques demonstrate superior performance, with BM25 achieving CB/ES metrics of 55.67/69.18 and GTE-Qwen reaching 55.29/68.21 for DeepSeek-V2.5, compared to other similarity-based retrieval techniques like CodeBERT and UniXcoder.

The comparative analysis between the two types of RAG methods reveals a clear advantage for similarity-based RAG. The superiority is consistently observed across different models and scales. For example, Qwen2.5-Coder-1.5B-Instruct achieves a maximum CB/ES metrics of 37.28/50.77 within identifier-based RAG, while similarity-based RAG pushes the performance to 46.69/56.04. DeepSeek-V3 reaches 42.24/61.75 performance with identifier-based RAG but achieves 60.28/73.11 with similarity-based RAG, representing a substantial improvement of 42.7% and 18.4% respectively.

¹<https://github.com/quickwit-oss/tantivy>

²<https://github.com/qdrant/qdrant>

TABLE I
PERFORMANCE COMPARISON OF LLMs WITH DIFFERENT RAG METHODS. THE METRICS SHOWING IMPROVEMENTS OVER THE BASE MODEL ARE HIGHLIGHTED BY GRAY . THE BEST PERFORMANCE METRICS WITHIN EACH RAG CATEGORY ARE MARKED IN **BOLD**.

Model	base CB/ES	msg-def. CB/ES	Identifier-Based RAG			Similarity-Based RAG				
			class-def. CB/ES	func-def. CB/ES	func-def. CB/ES	BM25 CB/ES	CodeBERT CB/ES	UniXcoder CB/ES	CoCoSoDa CB/ES	GTE-Qwen CB/ES
0.5B+ LLM										
Qwen2.5-Coder-0.5B-Instruct	27.57/41.54	24.39/ 38.31	24.25/37.36	21.38/36.09	26.72/37.71	31.43 /41.25	23.96/36.92	29.98 /39.43	30.39 /38.68	34.46 /41.67
1B+ LLMs										
OpenCoder-1B-Instruct	23.28/29.83	23.27/31.06	16.39/23.49	20.52/30.28	23.10/30.40	27.63 /32.45	23.60 /30.07	21.22/30.79	22.67/29.47	22.12/28.13
Llama-3.2-1B-Instruct	25.78/32.40	24.50/ 31.63	20.33/29.50	23.64/31.56	25.75/29.40	30.18 /32.59	24.30/29.86	25.39/30.02	25.12/30.06	30.93 /32.64
DS-Coder-1.3B-Instruct	21.85/35.22	22.08 / 36.08	24.85 /33.17	22.20 /35.52	23.62 /34.48	31.09 / 39.25	20.53/33.83	25.01 /35.54	27.76 /35.49	34.45 /37.06
Qwen2.5-Coder-1.5B-Instruct	24.42/42.83	35.05 / 51.13	31.82 /46.10	32.00 /46.99	37.28 /50.77	47.78 /57.62	17.33/34.26	35.82 /50.22	41.42 /50.85	46.69 /56.04
Yi-Coder-1.5B-Chat	19.56/31.69	21.66 /34.41	18.67/31.52	21.06 /34.47	20.97 / 34.37	30.17 /38.38	18.45/34.02	23.86 /34.07	24.23 /34.98	29.06 /36.40
3B+ LLMs										
Llama-3.2-3B-Instruct	32.79/45.74	32.23/44.82	32.97 /39.71	33.64 /45.13	36.02 /45.83	45.42 /47.37	34.15 /44.66	38.25 /44.80	41.19 /45.62	49.04 /51.07
Qwen2.5-Coder-3B-Instruct	15.41/38.15	17.07 /39.08	16.99 /38.72	14.45/34.97	19.86 /40.62	27.12 /44.48	18.25 /40.82	17.57 /38.52	24.89 /42.88	30.63 /48.48
7B+ LLMs										
DS-Coder-7B-Instruct-v1.5	33.24/49.54	32.59/48.35	27.75/40.52	34.19 /49.49	37.62 /50.11	43.21 / 55.20	34.05 /46.67	37.75 /50.37	35.12 /47.13	44.44 /52.36
Qwen2.5-Coder-7B-Instruct	33.00/50.27	34.60 /51.12	33.60 / 51.73	30.14/48.73	34.01 /51.10	45.16 /59.36	32.44/52.13	38.36 /54.75	44.23 /60.96	49.03 /62.66
Llama-3.1-8B-Instruct	34.02/46.07	36.42 /45.33	36.86 /46.34	35.55 /45.86	39.64 /49.35	49.80 /54.39	35.07 /45.55	40.01 /49.00	46.22 /51.65	53.47 /55.40
OpenCoder-8B-Instruct	29.69/31.42	30.28 /29.96	28.65/29.30	32.89 /29.42	37.35 /30.16	42.45 /32.17	29.70 /27.71	34.37 /30.08	37.15 /30.04	41.38 / 32.28
Yi-Coder-9B-Chat	33.78/47.11	33.69/45.25	32.00/43.63	34.03 /46.07	35.49 /46.98	51.66 /56.14	34.49 /46.57	40.28 /49.99	42.87 /50.49	49.59 /55.73
13B+ LLMs										
CodeLlama-13B-Instruct	26.81/35.52	24.09 /30.78	21.05/28.35	24.01/ 31.32	21.82/28.75	28.00 /30.51	23.23/29.54	23.84/28.66	24.63/29.24	26.58/ 30.70
Qwen2.5-Coder-14B-Instruct	29.79/48.56	33.43 /52.33	28.14/47.54	33.86 /53.56	35.02 /53.23	46.07 /59.97	35.06 /53.82	35.75 /52.25	43.01 /57.50	51.12 /61.96
DS-Coder-V2-Lite-Instruct-16B/24B	34.72/51.01	35.31 / 51.33	33.93/46.48	33.49/49.68	39.34 /48.15	51.45 /57.87	34.23/49.10	40.83 /50.75	45.60 /55.25	54.91 /57.98
20B+ LLMs										
CodeStral-22B-v0.1	34.12/55.25	36.13 /54.52	36.86 /54.64	34.51 / 55.03	36.11 /54.99	47.28 / 60.93	36.17 /54.65	36.16 /52.66	43.94 /57.05	49.37 /60.80
Qwen2.5-Coder-32B-Instruct	38.05/57.89	38.91 /58.84	40.54 /59.52	37.24/57.46	42.23 /60.44	55.76 /68.89	38.67 /59.28	44.16 /61.13	49.50 /65.54	60.79 /71.34
DS-Coder-33B-Instruct	28.48/45.20	31.45 / 48.34	24.47/39.66	30.36 /47.99	32.25 /46.74	38.91 / 50.19	29.37 /44.48	34.02 /47.96	34.50 /46.23	39.40 /47.43
CodeLlama-34B-Instruct	26.51/37.93	23.60/34.58	19.15/32.23	23.88/35.33	22.37/33.45	27.53 /36.48	21.36/32.30	18.01/31.58	22.12/33.93	28.35 /37.42
70B+ LLMs										
CodeLlama-70B-Instruct	22.50/33.10	17.88/28.95	12.06/17.85	12.59/21.99	11.60/17.39	19.17/26.95	13.65/20.57	15.23/21.34	16.05/20.30	16.26/20.70
Llama-3.3-70B-Instruct	34.14/53.04	35.15 /52.53	36.21 /53.30	36.92 /55.30	40.78 /58.39	50.21 /62.37	36.93 /54.84	39.33 /55.02	45.53 /58.86	52.64 /64.60
Qwen2.5-72B-Instruct	37.03/54.71	38.66 /55.97	38.07 /55.41	38.72 /56.86	41.90 /59.02	50.21 /62.45	37.89 /56.74	45.57 /60.38	48.16 /61.96	56.05 /65.35
200B+ LLMs										
DS-Coder-V2-Instruct-236B/21B	33.26/54.92	38.35 /57.95	37.63 /55.54	38.40 /58.50	43.52 /62.29	53.72 / 69.27	34.27 /55.73	44.56 /61.60	48.58 /63.33	55.92 /69.06
DeepSeek-V2.5-236B/21B	33.50/54.32	38.26 /56.55	36.43 /55.16	38.25 /58.74	41.81 /61.82	55.67 /69.18	32.67/53.75	44.70 /61.69	48.16 /63.85	55.29 /68.21
DeepSeek-V3-671B/37B	35.23/54.85	39.04 /58.54	37.74 /57.53	37.51 /58.63	42.24 /61.75	55.14 /68.55	38.13 /58.64	44.75 /62.27	50.43 /65.40	60.28 /73.11

Finding 1: Both types of RAG methods can consistently improve code completion performance across different models and scales in closed-source repositories. Moreover, compared to identifier-based RAG, similarity-based RAG substantially performs better in enhancing code completion quality.

B. RQ2: Impact of Retrieval Techniques

The experimental results in Table II reveal that among semantic retrieval techniques, CodeBERT consistently underperforms compared to UniXcoder, CoCoSoDa, and GTE-Qwen. This performance gap may be attributed to the differences in pre-training objectives: while CodeBERT relies solely on MLM and RTP, the other three models are specifically optimized for retrieval tasks through contrastive learning, which better captures code semantics and similarity relationships. An example is that Qwen2.5-Coder-14B-Instruct achieves CB/ES metrics of 51.12/61.96 with incomplete queries, still surpassing 23.23/29.54 gained by CodeBERT. Different from semantic retrieval techniques that require training, BM25, as a lexical retrieval technique, exhibits remarkable effectiveness through simple term-matching mechanisms. Our experimental results show that BM25-based RAG consistently achieves strong performance across various model scales and architectures.

In code completion tasks, only incomplete code context can be used as queries during retrieval, which creates a misalignment with the representation learning process of retrieval techniques. As shown in Table II, UniXcoder and CoCoSoDa demonstrate superior performance with the entire code snippets as queries, suggesting their potential ability to further improve RAG performance on code completion. BM25 also shows consistent improvements when using complete queries across all LLMs, with CB/ES metrics increasing from 31.43/41.25 to 36.40/43.48 in the 0.5B scale. In contrast, GTE-Qwen consistently demonstrates superior performance across different LLMs with incomplete code contexts as queries, particularly evident in larger LLMs. For example, when applied to DeepSeek-V3, GTE-Qwen achieves the highest CB/ES scores of 60.28/73.11, surpassing other retrieval techniques. The powerful retrieval ability may be attributed to its architecture, larger scale, and strong code-specific pre-training, making it particularly well-suited for code completion tasks.

Finding 2: Lexical retrieval technique consistently exhibits remarkable performance across different query formulations. While most retrieval techniques can be further improved by complete queries, GTE-Qwen demonstrates better performance with incomplete queries, making it particularly suitable for the code completion task.

TABLE II

COMPARATIVE ANALYSIS OF RETRIEVAL TECHNIQUES IN SIMILARITY-BASED RAG. THE BEST PERFORMANCE METRICS WITHIN EACH RETRIEVAL TECHNIQUE ARE HIGHLIGHTED BY GRAY. “INCOMPLETE” DENOTES USING PARTIAL CODE CONTEXT AS RETRIEVAL QUERY, WHILE “COMPLETE” REPRESENTS USING THE ENTIRE CODE SNIPPET FOR RETRIEVAL. THE BEST PERFORMANCE METRICS WITH INCOMPLETE CODE SNIPPETS AS QUERIES FOR EACH LLM ARE MARKED IN **BOLD**.

Model	BM25		CodeBERT		UniXcoder		CoCoSoDa		GTE-Qwen	
	Incomplete CB/ES	Complete CB/ES	Incomplete CB/ES	Complete CB/ES	Incomplete CB/ES	Complete CB/ES	Incomplete CB/ES	Complete CB/ES	Incomplete CB/ES	Complete CB/ES
0.5B+ LLM										
Qwen2.5-Coder-0.5B-Instruct	31.43/41.25	36.40 /43.48	23.96/36.92	24.00 /36.70	29.98/39.43	34.16 /41.65	30.39/38.68	34.27 /41.12	34.46 /41.67	33.77/39.62
1B+ LLMs										
OpenCoder-1B-Instruct	27.63/32.45	31.81 /32.31	23.60 /30.07	23.68 /29.82	21.22/30.79	26.89 /31.65	22.67/29.47	27.97 /28.32	22.12/28.13	23.87 /28.18
Llama-3.2-1B-Instruct	30.18/32.59	36.75 /33.68	24.30 /29.86	22.92/28.84	25.39/30.02	30.02 /31.76	25.12/30.06	29.66 /31.82	30.93 /32.64	30.58/32.71
DS-Coder-1.3B-Instruct	31.09/ 39.25	37.30 /40.53	20.53 /33.83	20.19 /34.15	25.01/35.54	31.54 /39.92	27.76/35.49	33.00 /37.49	34.45 /37.06	33.64/36.04
Qwen2.5-Coder-1.5B-Instruct	47.78/57.62	51.84 /59.39	17.33 /34.26	17.51 /33.72	35.82/50.22	42.89 /53.25	41.42/50.85	44.60 /51.68	46.69 /56.04	44.52/54.12
Yi-Coder-1.5B-Chat	30.17/38.38	34.57 /38.31	18.45 /34.02	17.78/32.87	23.86/34.07	32.23 /38.37	24.23/34.98	28.11 /37.00	29.06 /36.40	28.57/36.28
3B+ LLMs										
Llama-3.2-3B-Instruct	45.42/47.37	50.74 /49.82	34.15 /44.66	33.72/43.32	38.25/44.80	46.56 /50.82	41.19/45.62	44.50 /47.80	49.04 /51.07	47.20/49.27
Qwen2.5-Coder-3B-Instruct	27.12/44.48	31.59 /47.73	18.25 /40.82	17.98/40.27	17.57/38.52	27.05 /45.56	24.89/42.88	27.94 /45.05	30.63/48.48	30.90 /47.80
7B+ LLMs										
DS-Coder-7B-Instruct-v1.5	43.21/ 55.20	48.65 /58.02	34.05/46.67	34.12 /46.48	37.75/50.37	45.00 /55.39	35.12/47.13	37.86 /48.09	44.44/52.36	45.63 /53.78
Qwen2.5-Coder-7B-Instruct	45.16/59.36	52.92 /64.13	32.44/52.13	32.83 /52.68	38.36/54.75	46.75 /60.22	44.23/60.96	46.60 /60.95	49.03 /62.66	47.73/61.05
Llama-3.1-8B-Instruct	49.80/54.39	57.00 /60.19	35.07/45.55	35.53 /46.13	40.01/49.00	50.34 /54.69	46.22/51.65	48.96 /51.62	53.47 /55.40	52.89/54.27
OpenCoder-8B-Instruct	42.45/32.17	47.21 /32.98	29.70 /27.71	29.28/27.15	34.37/30.08	41.11 /31.36	37.15/30.04	40.96 /31.52	41.38 / 32.28	41.97 /31.84
Yi-Coder-9B-Chat	51.66/56.14	56.53 /58.91	34.49 /46.57	34.50 /45.80	40.28/49.99	48.88 /54.10	42.87/50.49	48.23 /54.00	49.59 /55.73	47.92/52.65
13B+ LLMs										
CodeLlama-13B-Instruct	28.00/30.51	34.15 /31.72	23.23 /29.54	22.88/29.69	23.84/28.66	29.51 /30.45	24.63/29.24	27.89 /29.69	26.58/ 30.70	28.10 /30.02
Qwen2.5-Coder-14B-Instruct	46.07/59.97	52.16 /63.72	35.06 /53.82	33.87/53.19	35.75/52.25	46.72 /60.36	43.01/57.50	46.39 /59.66	51.12 /61.96	49.28/60.16
DS-Coder-V2-Lite-Instruct-16B/2.4B	51.45/57.87	57.67 /62.19	34.23/49.10	34.65 /50.20	40.83/50.75	49.35 /55.13	45.60/55.25	49.64 /56.08	54.91 /57.98	53.50/56.00
20B+ LLMs										
CodeStral-22B-v0.1	47.28/ 60.93	52.46 /65.11	36.17 /54.65	36.02/52.65	36.16/52.66	48.87 /60.91	43.94/57.05	47.87 /58.87	49.37 /60.80	49.07/61.02
Qwen2.5-Coder-32B-Instruct	55.76/68.89	61.98 /72.38	38.67 /59.28	38.42/58.77	44.16/61.13	54.97 /69.16	49.50/65.54	52.63 /66.52	60.79 /71.34	59.75/70.25
DS-Coder-33B-Instruct	38.91/ 50.19	42.32 /53.85	29.37 /44.48	28.89/43.69	34.02/47.96	42.06 /52.79	34.50/46.23	37.40 /48.39	39.40/47.43	40.84 /48.89
CodeLlama-34B-Instruct	27.53/36.48	32.47 / 87.83	21.36/32.30	22.24 /33.61	18.01/31.58	23.71 /34.64	22.12/33.93	24.00 /35.15	28.35/37.42	28.44 /37.55
70B+ LLMs										
CodeLlama-70B-Instruct	19.17/26.95	25.13 /28.45	13.65 /20.57	13.97 /20.33	15.23/21.34	23.49 /26.13	16.05/20.30	19.53 /21.15	16.26/20.70	16.88 /20.88
Llama-3.3-70B	50.21/62.37	58.39 /66.70	36.93 /54.84	36.24/54.61	39.33/55.02	49.15 /62.24	45.53/58.86	50.09 /61.58	52.64 /64.60	52.03/62.48
Qwen2.5-72B	50.21/62.45	56.66 /66.74	37.89 /56.74	37.29/55.65	45.57/60.38	55.84 /67.22	48.16/61.96	52.24 /63.94	56.05/65.35	56.81 /65.75
200B+ LLMs										
DS-Coder-V2-Instruct-236B/21B	53.72/ 69.27	61.65 /72.39	34.27/55.73	34.54 /54.55	44.56/61.60	55.81 /70.07	48.58/63.33	51.99 /65.95	55.92/69.06	56.63 /68.51
DeepSeek-V2.5-236B/21B	55.67/69.18	60.92 /72.47	32.67/53.75	35.38 /54.09	44.70/61.69	55.00 /70.14	48.16/63.85	51.61 /66.04	55.29/68.21	57.80 /69.15
DeepSeek-V3-671B/37B	55.14/68.55	63.63 /73.70	38.13 /58.64	38.00 /58.91	44.75/62.27	57.81 /72.40	50.43/65.40	55.93 /69.14	60.28 /73.11	58.85/71.02

C. RQ3: Exploration on Retrieved Results

Due to the superior performance of similarity-based RAG, we conduct an exploratory analysis of retrieved results from lexical and semantic retrieval techniques. The results reveal minimal overlap between BM25 and semantic techniques: out of 100 test examples in our benchmark, there are 76, 74, and 64 completely distinct retrieved samples when comparing BM25 with UniXcoder, CoCoSoDa, and GTE-Qwen, respectively. This difference in retrieval distributions motivates us to explore the combination of these techniques.

As shown in Table III, the effectiveness of combining BM25 with different semantic retrieval techniques becomes more pronounced as model size increases. In the 200B+ scale, the BM25+GTE-Qwen combination achieves CB/ES metrics of 63.62/75.26 for DeepSeek-V3, substantially outperforming both individual techniques. Similarly, other semantic retrieval techniques also benefit from the combination of BM25. DeepSeek-V2.5 improves CB/ES metrics from 48.16/63.85 to 56.68/68.20 when combined with BM25. Notably, the advantage of BM25+GTE-Qwen combination is particularly striking for Qwen2.5-32B, which achieves impressive CB/ES scores of 63.73/72.25, rivaling or even surpassing models with significantly larger parameters such as DeepSeek-V2.5 and DeepSeek-V3. However, for smaller models (below 7B), the

combination shows limited or even negative impact, suggesting that the complementary benefits of hybrid retrieval methods are more effectively leveraged by larger models.

Finding 3: Lexical and semantic retrieval techniques exhibit distinct retrieval results distribution and demonstrate complementary characteristics in larger-scale models (7B+). With their combination, especially BM25+GTE-Qwen, similarity-based RAG achieves optimal performance in most LLMs.

V. DISCUSSION

A. Developer Survey

To further validate the superior outcomes achieved by integrating lexical and semantic retrieval results within similarity-based RAG, we conduct a developer survey involving three developers from our group (excluding the authors). This study aims to assess the quality of code completions generated using various retrieval techniques in similarity-based RAG, including BM25, GTE-Qwen, and a combination of both. The evaluation is performed on a random selection of 52 examples and three LLMs from the Qwen, Llama, and DeepSeek families, which demonstrated the best performance with the combined use of BM25 and GTE-Qwen.

TABLE III
PERFORMANCE COMPARISON OF THE COMBINATIONS BETWEEN LEXICAL-BASED AND SEMANTICS-BASED RETRIEVAL TECHNIQUES WITHIN SIMILARITY-BASED RAG FOR CODE COMPLETION.

Model	BM25 CB/ES	UniXcoder(U) CB/ES	U+BM25 CB/ES	CoCoSoDa(C) CB/ES	C+BM25 CB/ES	GTE-Qwen(Q) CB/ES	Q+BM25 CB/ES
0.5B+ LLM							
Qwen2.5-Coder-0.5B-Instruct	27.63/32.45	29.98/39.43	30.20/35.99	30.39/38.68	29.86/34.77	34.46/41.67	33.84/36.54
1B+ LLMs							
OpenCoder-1B-Instruct	27.63/32.45	21.22/30.79	20.36/24.25	22.67/29.47	15.82/20.09	22.12/28.13	18.53/20.04
Llama3.2-1B-Instruct	30.18/32.59	25.39/30.02	28.18/30.95	25.12/30.06	28.59/30.56	30.93/ 32.64	32.06 /31.50
DS-Coder-1.3B-Instruct	31.09/39.25	25.01/35.54	30.12/ 39.42	27.76/35.49	31.64/38.88	34.45/37.06	35.09 /38.04
Qwen2.5-Coder-1.5B-Instruct	47.78/ 57.62	35.82/50.22	45.47/53.89	41.42/50.85	47.04/53.66	46.69/56.04	48.78 /55.62
Yi-Coder-1.5B-Chat	30.17/38.38	23.86/34.07	23.40/33.24	24.23/34.98	23.81/31.88	29.06/36.40	29.54/34.11
3B+ LLMs							
Llama3.2-3B-Instruct	45.42/47.37	38.25/44.80	41.43/39.10	41.19/45.62	46.44/40.33	49.04/51.07	48.98/42.94
Qwen2.5-Coder-3B-Instruct	27.12/44.48	17.57/38.52	31.84/50.68	24.89/42.88	35.00/50.94	30.63/48.48	35.99/51.36
7B+ LLMs							
DS-Coder-7B-Instruct-v1.5	43.21/ 55.20	37.75/50.37	35.06/44.15	35.12/47.13	26.65/32.10	44.44 /52.36	29.63/37.12
Qwen2.5-Coder-7B-Instruct	45.16/59.36	38.36/54.75	50.72/63.73	44.23/60.96	53.58/65.55	49.03/62.66	55.57/66.07
Llama3.1-8B-Instruct	49.80/54.39	40.01/49.00	49.70/51.98	46.22/51.65	52.48/53.65	53.47/55.40	52.34/52.83
OpenCoder-8B-Instruct	42.45/32.17	34.37/30.08	39.80/33.51	37.15/30.04	39.44/33.09	41.38/32.28	42.56/34.21
Yi-Coder-9B-Chat	51.66/56.14	40.28/49.99	48.47/52.94	42.87/50.49	49.63/53.44	49.59/55.73	54.05/55.78
13B+ LLMs							
CodeLlama-13B-Instruct	28.00/30.51	23.84/28.66	30.24/30.65	24.63/29.24	30.84/30.25	26.58/30.70	32.62/32.15
Qwen2.5-Coder-14B-Instruct	47.28/60.93	35.75/52.25	52.92/65.00	43.01/57.50	55.20/66.38	49.37/60.80	57.80/66.89
DS-Coder-V2-Lite-Instruct-16B/2.4B	51.45/57.87	40.83/50.75	49.50/54.09	45.60/55.25	50.84/55.36	54.91/57.98	54.82/56.43
20B+ LLMs							
CodeStral-22B-v0.1	47.28/60.93	36.16/52.66	48.18/58.05	43.94/57.05	52.85 /60.24	49.37/60.80	52.25/ 61.49
Qwen2.5-Coder-32B-Instruct	55.76/68.89	44.16/61.13	56.66/69.34	49.50/65.54	56.90/67.66	60.79/71.34	63.73/72.25
DS-Coder-33B-Instruct	38.91/ 50.19	34.02/47.96	34.28/45.21	34.50/46.23	28.45/35.45	39.40 /47.43	33.84/41.83
CodeLlama-34B-Instruct	27.53/36.48	18.01/31.58	21.43/31.13	22.12/33.93	22.09/31.72	28.35/37.42	23.61/32.53
70B+ LLMs							
CodeLlama-70B-Instruct	19.17/26.95	15.23/21.34	14.27/17.76	16.05/20.30	15.59/19.87	16.26/20.70	13.72/18.10
Llama-3.3-70B	50.21/62.37	39.33/55.02	49.84/61.57	45.53/58.86	50.42/59.80	52.64/ 64.60	54.00 /64.37
Qwen2.5-72B	50.21/62.45	45.57/60.38	52.99/62.67	48.16/61.96	53.55/61.27	56.05/ 65.35	59.03 /64.89
200B+ LLMs							
DS-Coder-V2-Instruct-236B/21B	38.35/57.95	44.56/61.60	56.59/68.48	48.58/63.33	56.34/68.16	48.58/63.33	62.25/71.36
DS-V2.5-236B/21B	55.67/69.18	44.70/61.69	55.45/68.10	48.16/63.85	56.68/68.20	55.29/68.21	61.40/71.12
DS-V3-671B/37B	55.14/68.55	44.75/62.27	57.48/70.97	50.43/65.40	60.48/72.34	60.28/73.11	63.62/75.26

Specifically, we design an evaluation website for the study. The current code context and the ground truth code are provided as reference. The participating developers could assess the quality of the generated code by comparing the generated completion with the code context and ground truth. The scoring system uses a 1-5 scale, where 1 indicates significant differences from the original code with serious errors, and 5 represents near-perfect alignment with the original code and excellent quality. Additionally, based on our preliminary analysis of code completion failures, we identified three common issue categories that we provide as predefined options in the evaluation interface: **code logic errors**, **external code dependencies missing**, and **reference to non-existent functions**. These additional evaluation criteria help us better understand the root causes of low-quality code completion.

The developer survey reveals several important insights about different retrieval techniques and their effectiveness. The combination of BM25 and GTE-Qwen consistently achieves higher scores than using either technique alone across all three LLMs. This advantage is particularly evident in the win-rate analysis, where the combined technique outperforms single technique in about half of all test cases. While DeepSeek-V3 shows no strong preference between the retrieved results

of BM25 and GTE-Qwen individually, both Qwen2.5-32B-Instruct and Llama3.3-70B-Instruct perform better with GTE-Qwen-based RAG. The error analysis reveals a consistent pattern across all LLMs. Missing or Incorrect Logic dominates the error types at around 52%. This suggests that improving logical reasoning capabilities should be a priority for future development. Llama3.3-70B-Instruct exhibits slightly more errors overall compared to other LLMs, but the general distribution of error types remains similar across all three LLMs.

B. Implication of Findings

Our findings have several important implications for the development and application of RAG for code completion in closed-source scenarios:

Leveraging open-source LLMs with RAG for proprietary code development: Our results demonstrate that RAG can effectively leverage knowledge from closed-source codebases to improve the code completion performance of open-source LLMs. It is particularly valuable for proprietary development environments where access to extensive training data may be limited. The transparent and accessible nature of open-source models addresses privacy concerns in the devel-

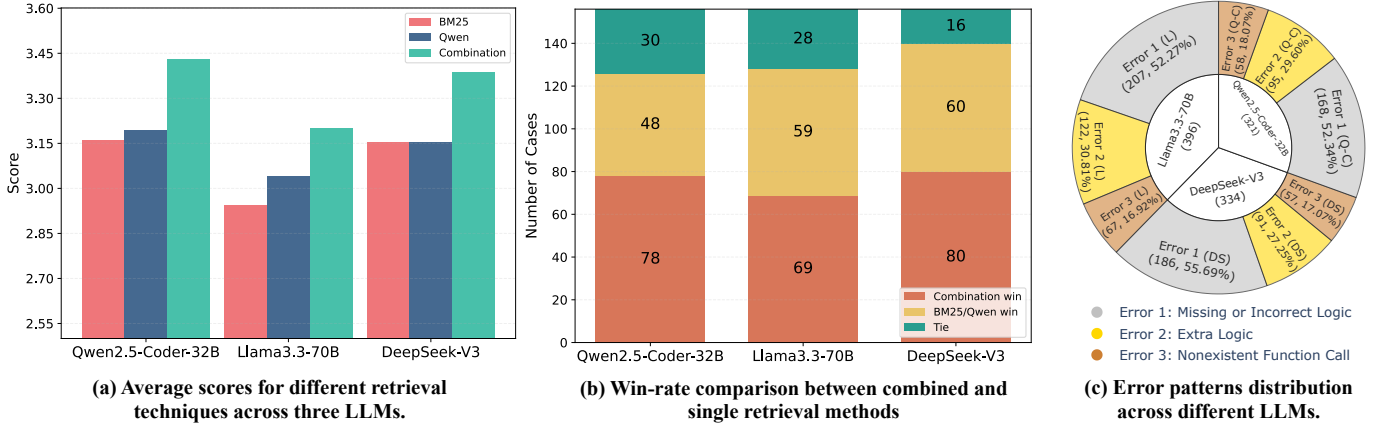


Fig. 2. The results analysis of developer survey.

opment process, making them particularly suitable for real-world industrial applications. Moreover, the similarity-based RAG proves to be a more effective solution for improving the performance of various LLMs compared to the identifier-based RAG.

Exploring task-specific retrieval techniques: We identify a gap between the training and application scenarios of semantic retrieval techniques. These models are typically trained on complete code snippets. However, code completion tasks require retrieving complete code snippets based on incomplete code fragments. This misalignment suggests two potential directions for improvement: (1) developing specialized retrieval techniques optimized for incomplete queries by reducing the semantic distance between partial and complete code snippets, and (2) utilizing more powerful retrieval models like GTE-Qwen that naturally accommodate incomplete query scenarios without additional fine-tuning.

Combining different types of retrieval techniques within RAG: Previous works often employ one type of retrieval technique for initial candidate selection, followed by another type of retrieval technique for re-ranking [46]–[48]. Our findings suggest a more nuanced relationship between the two types of retrieval techniques. The distinct yet complementary nature of lexical and semantic retrieval techniques indicates potential for further performance improvements in similarity-based RAG systems through their strategic combination, beyond the traditional retrieve-then-rerank pipeline.

C. Threats to Validity

Internal Validity. In our exploration of RAG using various LLMs and retrieval techniques, the performance of these deep learning-based models can be influenced by multiple factors, including parameter settings and hardware devices. To address this potential threat, we maintain consistency by using default parameter configurations across all models and set the temperature parameter to 0 during LLMs inference to ensure reproducible results.

External Validity. Our experiments are conducted on the specific enterprise codebase in WeChat group, which might

exhibit distinct characteristics from other organizations’ codebases. To mitigate this limitation, we select a diverse set of projects (total 1,669) covering different domains and development periods within our codebase. This dataset encompasses various development practices, coding standards, and business scenarios, providing a comprehensive representation of software development patterns in a closed-source environment.

Construct Validity. We use automated metrics (CodeBLEU and Edit Similarity) to measure code quality, but since code completion tools are ultimately used by developers, these metrics might not fully capture the semantic correctness and functionality of generated code in real development scenarios. To address this limitation, we supplement human evaluation with a developer survey, categorizing potential error types and identifying future research directions for optimization.

VI. RELATED WORK

Code completion, which aims to predict subsequent code elements based on the existing context, is a crucial task for improving developer productivity in software engineering. Early approaches primarily rely on statistical methods to implement code completion functionality [49]. Recently, with the advancement of deep learning, particularly LLMs, code completion has achieved superior performance in production environments and provide developers meaningful suggestions [50], [51]. Several recent works have focused on improving code completion through various context selection techniques. Liang et al. [11] extract dependency definitions from the current context and retrieve similar code snippets from a code repository, aggregating both to construct prompts that help LLMs better understand the context for code completion. Cheng et al. [13] introduce code dependencies through data flow graphs in a directed manner. Liu et al. [1] locate context segments relevant to code completion using structured patterns and implement a reranking algorithm based on decay-with-distance sub-graph edit distance. Additionally, Liu et al. [52] incorporate multiple static analysis methods across different stages of code completion to enhance the reliability of completed code.

VII. CONCLUSION

In this paper, we conduct a systematic investigation of retrieval-augmented generation (RAG) for code completion in closed-source repositories. Through comprehensive experiments on 26 open-source LLMs ranging from 0.5B to 671B parameters, we demonstrate the consistent effectiveness of both identifier-based and similarity-based RAG methods. Our in-depth analysis of similarity-based RAG reveals that BM25 and GTE-Qwen achieve superior performance in code completion. Furthermore, we explore the relationship between lexical and semantic retrieval techniques, identifying the BM25+GTE-Qwen combination as the optimal improvement strategy. We summarize our findings and provide valuable insights for researchers and practitioners to apply RAG methods for code completion systems in their proprietary environments.

REFERENCES

- [1] 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.
- [2] 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.
- [3] 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 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 79:1–79:13.
- [4] T. Liu, C. Xu, and J. J. McAuley, "RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.
- [5] 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, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1294–1306.
- [6] S. Gao, X. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What Makes Good In-Context Demonstrations for Code Intelligence Tasks with LLMs?" in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 761–773.
- [7] Y. Peng, C. Wang, W. Wang, C. Gao, and M. R. Lyu, "Generative Type Inference for Python," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 988–999.
- [8] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating Large Language Models in Class-Level Code Generation," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 81:1–81:13.
- [9] X. Du, M. Wen, J. Zhu, Z. Xie, B. Ji, H. Liu, X. Shi, and H. Jin, "Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 10 507–10 521.
- [10] S. S. Daneshvar, Y. Nong, X. Yang, S. Wang, and H. Cai, "Exploring RAG-based Vulnerability Augmentation with LLMs," *arXiv preprint arXiv:2408.04125*, 2024.
- [11] M. Liang, X. Xie, G. Zhang, X. Zheng, P. Di, W. Jiang, H. Chen, C. Wang, and G. Fan, "REPOFUSE: Repository-Level Code Completion with Fused Dual Context," *CoRR*, vol. abs/2402.14323, 2024.
- [12] D. Wu, W. U. Ahmad, D. Zhang, M. K. Ramanathan, and X. Ma, "Repoformer: Selective Retrieval for Repository-Level Code Completion," in *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.
- [13] W. Cheng, Y. Wu, and W. Hu, "Dataflow-Guided Retrieval Augmentation for Repository-Level Code Completion," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 7957–7977.
- [14] 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.
- [15] L. Zhang, H. Zhang, C. Wang, and P. Liang, "RAG-Enhanced Commit Message Generation," *arXiv preprint arXiv:2406.05514*, 2024.
- [16] T. Ahmed, C. Bird, P. Devanbu, and S. Chakraborty, "Studying LLM Performance on Closed-and Open-source Data," *arXiv preprint arXiv:2402.15100*, 2024.
- [17] S. Lu, N. Duan, H. Han, D. Guo, S. Hwang, and A. Svyatkovskiy, "ReACC: A Retrieval-Augmented Code Completion Framework," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 6227–6240.
- [18] Q. Guo, X. Li, X. Xie, S. Liu, Z. Tang, R. Feng, J. Wang, J. Ge, and L. Bu, "FT2Ra: A Fine-Tuning-Inspired Approach to Retrieval-Augmented Code Completion," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 313–324.
- [19] M. Andersen-Gott, G. Ghinea, and B. Bygstad, "Why do commercial companies contribute to open source software?" *International journal of information management*, vol. 32, no. 2, pp. 106–117, 2012.
- [20] Tencent, "Tencent Announces 2024 Third Quarter Results," <https://www.tencent.com/en-us/investors/financial-news.html>, 2024.11.13.
- [21] A. Arusoae, S. Ciobăca, V. Craciun, D. Gavrilit, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in c/c++ code," in *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNAS)*. IEEE, 2017, pp. 161–168.
- [22] L. I. Hatledal, A. Styve, G. Hovland, and H. Zhang, "A language and platform independent co-simulation framework based on the functional mock-up interface," *IEEE Access*, vol. 7, pp. 109 328–109 339, 2019.
- [23] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Identifying and understanding header file hotspots in c/c++ build processes," *Automated Software Engineering*, vol. 23, pp. 619–647, 2016.
- [24] P. Xu, W. Ping, X. Wu, L. McAfee, C. Zhu, Z. Liu, S. Subramanian, E. Bakhturina, M. Shoenybi, and B. Catanzaro, "Retrieval meets long context large language models," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: <https://openreview.net/forum?id=xw5nxFWMlo>
- [25] A. Trotman, A. Puurula, and B. Burgess, "Improvements to BM25 and Language Models Examined," in *Proceedings of the 2014 Australasian Document Computing Symposium, ADCS 2014, Melbourne, VIC, Australia, November 27-28, 2014*, J. S. Culpepper, L. A. F. Park, and G. Zuccon, Eds. ACM, 2014, pp. 58–65.
- [26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, 2020, pp. 1536–1547.
- [27] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified Cross-Modal Pre-training for Code Representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 7212–7225.
- [28] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "CoCoSoDa: Effective Contrastive Learning for Code Search," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2198–2210.
- [29] Z. Li, X. Zhang, Y. Zhang, D. Long, P. Xie, and M. Zhang, "Towards General Text Embeddings with Multi-stage Contrastive Learning," *CoRR*, vol. abs/2308.03281, 2023.

- [30] Q. Team, “Qwen2 Technical Report,” *CoRR*, vol. abs/2407.10671, 2024.
- [31] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, A. Yang, R. Men, F. Huang, X. Ren, X. Ren, J. Zhou, and J. Lin, “Qwen2.5-Coder Technical Report,” *CoRR*, vol. abs/2409.12186, 2024.
- [32] DeepSeek-AI, “DeepSeek LLM: Scaling Open-Source Language Models with Longtermism,” *CoRR*, vol. abs/2401.02954, 2024.
- [33] DeepSeek-AI, “DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence,” *CoRR*, vol. abs/2406.11931, 2024.
- [34] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, “Code Llama: Open Foundation Models for Code,” *CoRR*, vol. abs/2308.12950, 2023.
- [35] A. Young, B. Chen, C. Li, C. Huang, G. Zhang, G. Zhang, H. Li, J. Zhu, J. Chen, J. Chang, K. Yu, P. Liu, Q. Liu, S. Yue, S. Yang, S. Yang, T. Yu, W. Xie, W. Huang, X. Hu, X. Ren, X. Niu, P. Nie, Y. Xu, Y. Liu, Y. Wang, Y. Cai, Z. Gu, Z. Liu, and Z. Dai, “Yi: Open Foundation Models by 01.AI,” *CoRR*, vol. abs/2403.04652, 2024.
- [36] S. Huang, T. Cheng, J. K. Liu, J. Hao, L. Song, Y. Xu, J. Yang, J. H. Liu, C. Zhang, L. Chai, R. Yuan, Z. Zhang, J. Fu, Q. Liu, G. Zhang, Z. Wang, Y. Qi, Y. Xu, and W. Chu, “OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models,” *CoRR*, vol. abs/2411.04905, 2024.
- [37] M. AI, “Codestral,” <https://mistral.ai/news/codestral/>, 2024.05.29.
- [38] Meta, “Llama 3.2: Revolutionizing edge AI and vision with open, customizable models,” <https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/>, 2024.09.25.
- [39] Meta, “Introducing Llama 3.1: Our most capable models to date,” <https://ai.meta.com/blog/meta-llama-3-1/>, 2024.07.23.
- [40] Meta, “Llama-3.3-70B-Instruct,” <https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct>, 2024.12.06.
- [41] Q. Team, “Qwen2.5: A Party of Foundation Models,” <https://qwenlm.github.io/blog/qwen2.5/>, 2024.09.
- [42] DeepSeek-AI, “DeepSeek-V3 Technical Report,” *CoRR*, vol. abs/2412.19437, 2024.
- [43] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “CodeBLEU: a Method for Automatic Evaluation of Code Synthesis,” *CoRR*, vol. abs/2009.10297, 2020.
- [44] K. Papineni, S. Roukos, T. Ward, and W. Zhu, “Bleu: a Method for Automatic Evaluation of Machine Translation,” in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318.
- [45] X. H. Lù, “BM25S: Orders of magnitude faster lexical search via eager sparse scoring,” *CoRR*, vol. abs/2407.03618, 2024.
- [46] M. R. Glass, G. Rossiello, M. F. M. Chowdhury, A. Naik, P. Cai, and A. Gliozzo, “Re2G: Retrieve, Rerank, Generate,” in *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*, M. Carpuat, M. de Marneffe, and I. V. M. Ruiz, Eds. Association for Computational Linguistics, 2022, pp. 2701–2715.
- [47] J. Dong, B. Fatemi, B. Perozzi, L. F. Yang, and A. Tsitsulin, “Don’t Forget to Connect! Improving RAG with Graph-based Reranking,” *arXiv preprint arXiv:2405.18414*, 2024.
- [48] P. Finardi, L. Avila, R. Castaldoni, P. Gengo, C. Larcher, M. Piau, P. Costa, and V. Caridà, “The Chronicles of RAG: The Retriever, the Chunk and the Generator,” *arXiv preprint arXiv:2401.07883*, 2024.
- [49] V. Raychev, M. T. Vechev, and E. Yahav, “Code completion with statistical language models,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O’Boyle and K. Pingali, Eds. ACM, 2014, pp. 419–428.
- [50] Z. Sun, X. Du, F. Song, S. Wang, and L. Li, “When Neural Code Completion Models Size up the Situation: Attaining Cheaper and Faster Completion through Dynamic Model Inference,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.
- [51] T. Zhu, Z. Liu, T. Xu, Z. Tang, T. Zhang, M. Pan, and X. Xia, “Exploring and Improving Code Completion for Test Code,” in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 137–148.
- [52] 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.