Prompt-based Code Completion via Multi-Retrieval Augmented Generation

HANZHUO TAN[*†], Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
QI LUO[*], Department of Computer Science and Engineering, Southern University of Science and Technology, China
LING JIANG, Department of Computer Science and Engineering, Southern University of Science and Technology, China
ZIZHENG ZHAN, Kuaishou Technology, China
JING LI, Department of Computing and the Research Centre on Data Science and Artificial Intelligence (RC-DSAI), the Hong Kong Polytechnic University, China
HAOTIAN ZHANG, Kuaishou Technology, China
YUQUN ZHANG[†‡], Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China

Automated code completion, aiming at generating subsequent tokens from unfinished code, has significantly benefited from recent progress in pre-trained Large Language Models (LLMs). However, these models often suffer from coherence issues and hallucinations when dealing with complex code logic or extrapolating beyond their training data. Existing Retrieval Augmented Generation (RAG) techniques partially address these issues by retrieving relevant code with a separate encoding model where the retrieved snippet serves as contextual reference for code completion. However, their retrieval scope is subject to a singular perspective defined by the encoding model, which largely overlooks the complexity and diversity inherent in code semantics. To address this limitation, we propose ProCC, a code completion framework leveraging prompt engineering and the contextual multi-armed bandits algorithm to flexibly incorporate and adapt to multiple perspectives of code. ProCC first employs a *prompt-based multi-retriever system* which crafts prompt templates to elicit LLM knowledge to understand code semantics with multiple retrieval perspectives. Then, it adopts the *adaptive retrieval selection algorithm* to incorporate code similarity into the decision-making process to determine the

[*]Both authors contributed equally to this research.
[†]These authors are also affiliated with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. Hanzhuo Tan is also affiliated with the Hong Kong Polytechnic University, China.
[‡]Yuqun Zhang is the corresponding author.

Authors' Contact Information: Hanzhuo Tan, hanzhuo.tan@connect.polyu.hk, Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China; Qi Luo, 12232440@mail.sustech.edu.cn, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China; Ling Jiang, 11711906@mail.sustech.edu.cn, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China; Zizheng Zhan, zhanzizheng@kuaishou.com, Kuaishou Technology, Beijing, China; Jing Li, jing-amelia.li@polyu.edu.hk, Department of Computing and the Research Centre on Data Science and Artificial Intelligence (RC-DSAI), the Hong Kong Polytechnic University, Hong Kong, China; Haotian Zhang, zhanghaotian@kuaishou.com, Kuaishou Technology, Beijing, China; Yuqun Zhang, zhangyq@sustech.edu.cn, Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China.

ACM 1557-7392/2025/3-ARTXXX
https://doi.org/XXXXXXX.XXXXXXX

most suitable retrieval perspective for the LLM to complete the code. Experimental results demonstrate that ProCC outperforms a widely-studied code completion technique RepoCoder by 7.92% on the public benchmark CCEval, 3.19% in HumanEval-Infilling, 2.80% on our collected open-source benchmark suite, and 4.48% on the private-domain benchmark suite collected from Kuaishou Technology in terms of Exact Match. ProCC also allows augmenting fine-tuned techniques in a plug-and-play manner, yielding an averaged 6.5% improvement over the fine-tuned model.

CCS Concepts: • **Software and its engineering → Genetic programming**.

Additional Key Words and Phrases: Code Completion, Multi-Retriever, Prompting

ACM Reference Format:
Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2025. Prompt-based Code Completion via Multi-Retrieval Augmented Generation. *ACM Trans. Softw. Eng. Methodol.* XX, X, Article XXX (March 2025), 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Automated code completion aims at generating subsequent code tokens based on the ongoing incomplete code segments [19, 37, 45, 47, 56, 58, 59, 67, 69, 85]. Typically, it can significantly enhance the efficiency of software developers and reduce operating costs for corporations [36, 73]. Therefore, automated code completion has become widely incorporated into Integrated Development Environments (IDEs). For example, Microsoft's Pyright [49], a static type-checking tool, powers the auto-completion feature for Python in VScode. Conventional automated code completion techniques generally rely on program analysis to generate syntax-conforming completions [19, 47, 56]. However, they are often argued to be limited in capturing code semantics to generate line-level completions for real-world development [73]. To alleviate such issues, many existing techniques [37, 58, 59, 69] adopt statistical or corpus-based techniques like N-Gram, Deep Neural Network [33] (DNN), Recurrent Neural Network [64] (RNN), and Long Short-Term Memory [20] (LSTM) models to learn program semantics. However, they do not easily generalize across domains and require expensive data collection, annotation, and training efforts to adapt to new tasks, limiting their real-world applicability.

Recently, by leveraging pre-training, large language models (LLMs) have been largely adopted for code and shown capable of varied tasks including code completion [26, 39, 62, 66, 74, 77, 81]. These models, trained on large code corpora with trillions of tokens, can encode code knowledge and programming patterns into their large-scale parameters, remarkably outperforming the existing non-pre-trained techniques in both code understanding and generation tasks [83]. For instance, Qwen2.5-Coder-7B [22] is capable of completing 61.6% of the text-to-code programming problems upon the HumanEval [5] benchmark with no additional adaptation. However, when faced with complex code logic or required to extrapolate beyond their training data, LLMs can struggle with incoherent or repetitive generations [70]. They may also hallucinate plausible but incorrect outputs [41, 48]. One potential solution to these issues is fine-tuning [68, 82, 86], i.e., adapting a pre-trained language model to the code completion task by incrementally updating its parameters on the task-oriented dataset. However, this proves to be a costly endeavor both in terms of computational resources and time, making it impractical for many applications. For instance, fine-tuning the smallest LLaMA model typically requires a GPU workstation with eight A100 [9, 80]. On the other hand, the Retrieval Augmented Generation [35] (RAG) techniques offer a more feasible solution. Specifically, for an RAG-based technique, given an input, related information is retrieved from a pre-defined knowledge source and then used to assist the generation process. This retrieval process enables models to generate outputs that are coherent and relevant to the given context without the

need for expensive fine-tuning. Accordingly, multiple code completion techniques [18, 43, 45, 67, 85] have been proposed to leverage the power of RAG and have shown promising performance.

Despite the promising results shown in the RAG-based techniques, they are still somewhat limited. First, they depend on extrinsic encoding models. Existing techniques construct code representations for retrieval mainly by encoding the code with auxiliary models, which requires additional efforts for training the encoding model. Second, the representational scope of these techniques is subject to a singular perspective on lexical semantics defined by the corresponding encoding model, i.e., they fail to account for the complexity and diversity inherent in code semantics. To illustrate, for a complex missing line with rich context, only offering lexical semantics by the incomplete code is unlikely to fully represent the code intention. Under such a circumstance, we should retrieve code snippets that are most likely to produce analogous content, i.e., adopting more encoding perspectives other than lexical semantics only. However, existing systems based on pre-defined encoding models require substantially additional training to encode the code from more perspectives, making it impractical for the existing RAG-based techniques. Therefore, it is essential to adopt a flexible retrieval approach to code context with multiple perspectives for code completion.

In this paper, we propose ProCC, a code completion framework leveraging prompt engineering and the contextual multi-armed bandit algorithm [38] for the first time to flexibly incorporate and adapt to multiple perspectives of code. ProCC consists of two components—the *prompt-based multi-retriever system* and the *adaptive retrieval selection algorithm*. In particular, the *prompt-based multi-retriever system* provides diverse perspectives of code while enabling flexible implementation and seamless integration with existing retrieval systems. Instead of creating a series of new embedding models with significant extra cost, we adopt prompt engineering which advances LLMs to understand code semantics via following human preferences and instructions [24, 27, 76] such that we can access code semantics from different lenses for more comprehensive retrieval in a cost-effective manner. More specifically, our *prompt-based multi-retriever system* examines three perspectives that are prominently used in RAG-based techniques, namely lexical semantics [45], hypothetical line [85], and code summarization [78]. We craft the prompt "Embedding the following code snippets: [code]" to encode the lexical semantics, and "<PRE> [Prefix] <SUF> [Suffix] <MID>" to generate the hypothetical line serving as its representation, and "This code snippets of [code] means" to obtain the code summarization. To illustrate, "[code]" refers to the unfinished code, "[Prefix]" and "[Suffix]" represent the code snippets before and after the target insertion point. As our *prompt-based multi-retriever system* presents three distinct perspectives, directly concatenating all three retrievals with input may overwhelm the completion model with misaligned perspectives. To optimally select retrieved information with respect to the complex nature of incomplete code, we adopt the *adaptive retrieval selection algorithm* based on a contextual multi-armed bandit algorithm where the different retrieval perspectives are seen as the "arms" of the bandit and the goal is to identify which arm (i.e., perspective) yields the highest reward or performance for individual incomplete code, conditioned on the similarity between retrieved snippets and incomplete code. Accordingly, ProCC adapts to the dynamic nature of code completion, handles the uncertainty, and reliably determines the most suitable perspective for the LLMs to utilize in the code completion process.

To evaluate ProCC, we have formulated the following three research questions:
RQ1: How does ProCC perform compared with state-of-the-art code completion techniques?
RQ2: How do individual components of ProCC impact the performance?
RQ3: How does ProCC perform compared with fine-tuning? Can it further improve a fine-tuned model?

We evaluate the effectiveness of ProCC using the state-of-the-art (SOTA) models DeepSeek-Coder [17] and Qwen2.5-Coder [22] as the base models on two widely-adopted benchmarks, CrossCodeEval (CCEval) [7] and HumanEval-Infilling [12]. Meanwhile, we collected 20 open-source repositories to form our ProCC-Infilling benchmark, and 58 private-domain repositories from Kuaishou Technology, a billion-user e-commerce company, for extensive evaluation. We also evaluate each component of ProCC and investigate how ProCC impacts the performance of the fine-tuned models. Our evaluation results indicate that ProCC outperforms the widely-studied code completion technique RepoCoder [85] by 7.92% on the CCEval benchmark, 3.19% on the HumanEval-Infilling benchmark, 2.80% on our collected ProCC-Infilling open-source benchmark suite, and 4.48% on the Kuaishou private-domain benchmark suite in terms of Exact Match using DeepSeek-Coder-6.7B. Additionally, our evaluation results indicate that our single retrievers are robust across instructions and comparable to external encoders. Meanwhile, designing multiple retrievers to elicit distinct interpretations enables us to obtain a wider range of code semantics. Furthermore, incorporating the varied perspectives enriches the multifaceted representations, improving the code completion effectiveness. Finally, ProCC also allows augmenting the fine-tuned techniques in a plug-and-play manner, yielding an average 6.5% improvement over our studied fine-tuned model.

In summary, the contributions of this paper are listed as follows:

- **Novelty.** This paper opens up a new direction for multi-perspective code representation in retrieval-augmented code completion. We are the first to show incorporating prompt engineering and contextual multi-armed bandit can adapt to the most suitable code perspective without needing extra encoders. This provides a more comprehensive and adjustable encoding strategy compared to rigid representations in prior work [25, 45, 65].
- **Technique.** We propose and implement ProCC with two components, the *prompt-based multi-retriever system* and the *adaptive retrieval selection algorithm*. The *prompt-based multi-retriever system* examines three essential perspectives via crafted instructions for code semantics. The *adaptive retrieval selection algorithm* dynamically chooses the most relevant perspective based on incomplete code context to provide the optimal contextual support for code completion.
- **Evaluation.** We perform an extensive evaluation on ProCC against the SOTA techniques. ProCC outperforms the widely-studied technique RepoCoder [85] by 7.92% on the CCEval benchmark, 3.19% on the HumanEval-Infilling benchmark, 2.80% on the ProCC-Infilling benchmark, and 4.48% on the Kuaishou private-domain benchmark in terms of Exact Match. We also show that our single retrievers are robust across instructions and comparable to external encoders. Meanwhile, designing instructions to elicit distinct semantic interpretations could obtain a wider range of code semantics. Furthermore, incorporating the varied perspectives enriches the multifaceted representations for improving the code completion effectiveness. Moreover, applying ProCC to fine-tuned models results in a 6.5% gain.

2 Background and Motivation

2.1 Code Completion

Recent natural language processing (NLP) breakthroughs have facilitated large pre-trained language models for the code completion task [52, 77, 84]. In general, LLMs are built on the Transformer architecture and pre-trained on large-scale text corpora using self-attention mechanisms [72]. LLMs efficiently model contextual relationships and facilitate the learning of general linguistic interpretations. In particular, LLMs exhibit substantial model size and volume of training data. For instance, the smallest version of the LLaMA2 model [71], launched in 2023, enables 7 billion parameters and

is trained on 2 trillion tokens. LLMs predominantly adopt a decoder-only architecture, in which they aim to auto-regressively generate tokens based on all previously generated ones.

The training loss for typical LLMs, depicted in Equation 1, minimizes the negative log probability for the ground truth token x_i:

$$\mathcal{L} = -\sum_{i} \log P_i(x_i|x_1, x_2, ..., x_{i-1}; \theta) \tag{1}$$

where the conditional probability P is modeled using a pre-trained language model $\mathcal{M}$ with parameters θ. These parameters are optimized by applying the gradient descent algorithms [63] with respect to the input sequence $x_1, x_2, ..., x_{i-1}$ preceding the given token x_i.

In particular, emerging code LLMs such as InCoder [12], StarCoder [39], Code Llama [62], DeepSeek-Coder [17], and Qwen2.5-Coder [22] are trained using the Fill-in-the-Middle (FIM) objective [4]. This technique involves randomly rearranging parts of a training sequence by moving them to the end and then generating predictions auto-regressively based on the reordered sequence. The pre-training loss for FIM remains consistent with Equation 1. Specifically for the code completion task, during inference, FIM leverages additional surrounding context by taking a prefix and suffix around the insertion point and generating the missing middle code tokens. Formally, the goal is to generate the token x_i that minimizes the negative log likelihood based on prefix tokens $[Prefix] = x_1, ..., x_{i-1}$ and suffix tokens $[Suffix] = x_{i+1}, ..., x_j$ before and after the insertion point:

$$\mathcal{L} = -\log P(x_i|[Prefix], [Suffix]; \theta) \tag{2}$$

For abbreviation, $\hat{X} = [Prefix], [Suffix]$ is denoted as the full unfinished code. Unlike conventional text generation models that are only conditional on preceding tokens, as formulated in Equation 1, access to suffix code is critical and practical for code completion. As a result, these infilling models largely outperform previous models in code completion. For instance, Qwen2.5-Coder-7B achieves an impressive pass@1 rate of 86.2% on the infilling benchmark [12], surpassing previous non-infilling SOTA techniques by a significant margin.

While LLMs have shown impressive capabilities on code completion tasks, they still face limitations when dealing with complex logic or are required to generalize beyond their training data [70]. They may produce incoherent text when the generation requires long-term reasoning or even generate hallucinated outputs that seem plausible but do not actually reflect valid behaviors [41, 48].

2.2 Retrieval-Augmented Generation

To address challenges on hallucination and enhance the production of coherent code, researchers have proposed that the models should be capable of accessing external memory or knowledge through information retrieval techniques, a.k.a. retrieval augmented generation (RAG) [35]. The RAG process can be formulated as follows. First, the code database is split into snippets $C = c_1, c_2, ...$, which are encoded by the retrieval model $\mathcal{R}$ to derive their representations $h^R_{c_1}, h^R_{c_2}, ...$ and form the corresponding database D. Second, for an incomplete snippet $\hat{X}$, the retrieval model $\mathcal{R}$ encodes it as $h^R_{\hat{X}}$ to retrieve relevant snippets C^R based on distance functions between $h^R_{\hat{X}}$ and the representations $h^R_{c_1}, h^R_{c_2}, ...$ stored in database D. The retrieval process can be described as:

$$p_R(C|x_1, x_2, ..., x_{i-1}, D) \tag{3}$$

The retrieved context, denoted as C^R, along with unfinished code $\hat{X}$ are then consumed by the model to conduct the code completion:

$$p_\theta(x_i|\hat{X}, C^R) \tag{4}$$

Consequently, the retrieved code snippets can be perceived as supplemental knowledge and interpreted by the LLMs to facilitate coherent generation.

There are two primary strategies for RAG—per-token and per-output [35]. In per-token RAG (RAG-token), distinct code snippets are assigned to individual tokens, whereby each new token generation requires the retrieval of a new code snippet. Therefore, this strategy escalates the retrieval and the encoding demands proportionally with the length of the generated sequence, leading to increased computational time. Moreover, the necessity for accessing each stage of token generation constrains its integration with closed-source systems, such as GPT [53], which operate as black boxes, i.e., exposing no intermediate generations. In contrast, per-output RAG (RAG-sequence) leverages the same code snippet to conduct the generation of the entire sequence, necessitating only one retrieval phase prior to generation. This strategy not only enhances efficiency of the retrieval process but also facilitates a more streamlined integration into the existing frameworks. Consequently, our subsequent discussions and analyses will be exclusively concentrated on the per-output RAG.

Following the standard RAG-sequence framework, the pioneering work ReACC [45] utilizes a dual-encoder model to function as a code-to-code search retriever and employs an auto-regressive language model to execute code completion. RepoCoder [85] suggests refining the retrieval process by iteratively utilizing the most recently generated content to retrieve information.

More complex RAG-based techniques have been recently proposed. GraphCoder [43] utilizes a code context graph (CCG), which includes control-flow, data-flow, and control-dependence relationships between code statements, to understand the context of the completion target. It further incorporates decay-with-distance subgraph edit distance to refine the CCG retrieval results. Similarly, FT2Ra [18] draws inspiration from the fine-tuning process and underscores the role of delta logits in boosting model predictions. It introduces a retrieval paradigm with a learning rate and multi-epoch retrievals that mimics fine-tuning. Despite the promising performance of these complex RAG-based techniques, they face significant challenges in integrating widely-employed retrieval acceleration frameworks like FAISS [60] and generation optimizations like vLLM [31]. As a result, they require substantial implementation effort for time-constraint and real-time code completion task. More detailed analysis will be provided in Section 4.2.1.

2.3 Motivation

As discussed in Section 1, the existing RAG-based techniques rely on external encoding models and are constrained to a singular perspective defined by the corresponding encoding model. They fail to account for the complexity and diversity inherent in code semantics.

Figure 1 presents three distinct code completion scenarios that illustarte the importance of different retrieval perspectives. In Scenario 1, Retriever 1 provides the relationship between “sslContext” and “SSLContext.getInstance()” which gives relevant context to assist the generator in completing the line. While Retriever 2 also hints “sslContext()”, it is insufficient to bridge the gap for completing the “SSLContext.getDefault()”. Conversely, in Scenario 2, Retriever 2 presents the list of parameters, i.e., “ResourceURL, UriPath, IndexFile, DefaultMediaType, DefaultCharset”, that should be included in the function “createServlet()”. This can be easily integrated by the generator to complete the return line “return new AssetServlet(resourcePath, uriPath, indexFile, defaultMediaType, StandardCharsets.UTF_8);”, whereas Retriever 1 presents noisy context and hinders the correct code completion. In Scenario 3, Retriever 2 retrieves a code segment that closely matches the source code in terms of lexicon. Note that this segment includes the line “sessionFactory = null;” which potentially misleads the generator towards an erroneous completion context. To address such an issue, Retriever 1 operates from a summary perspective, identifying segments with similar functionality to stopping and shutting down the “Factory” as in the source code which intends to

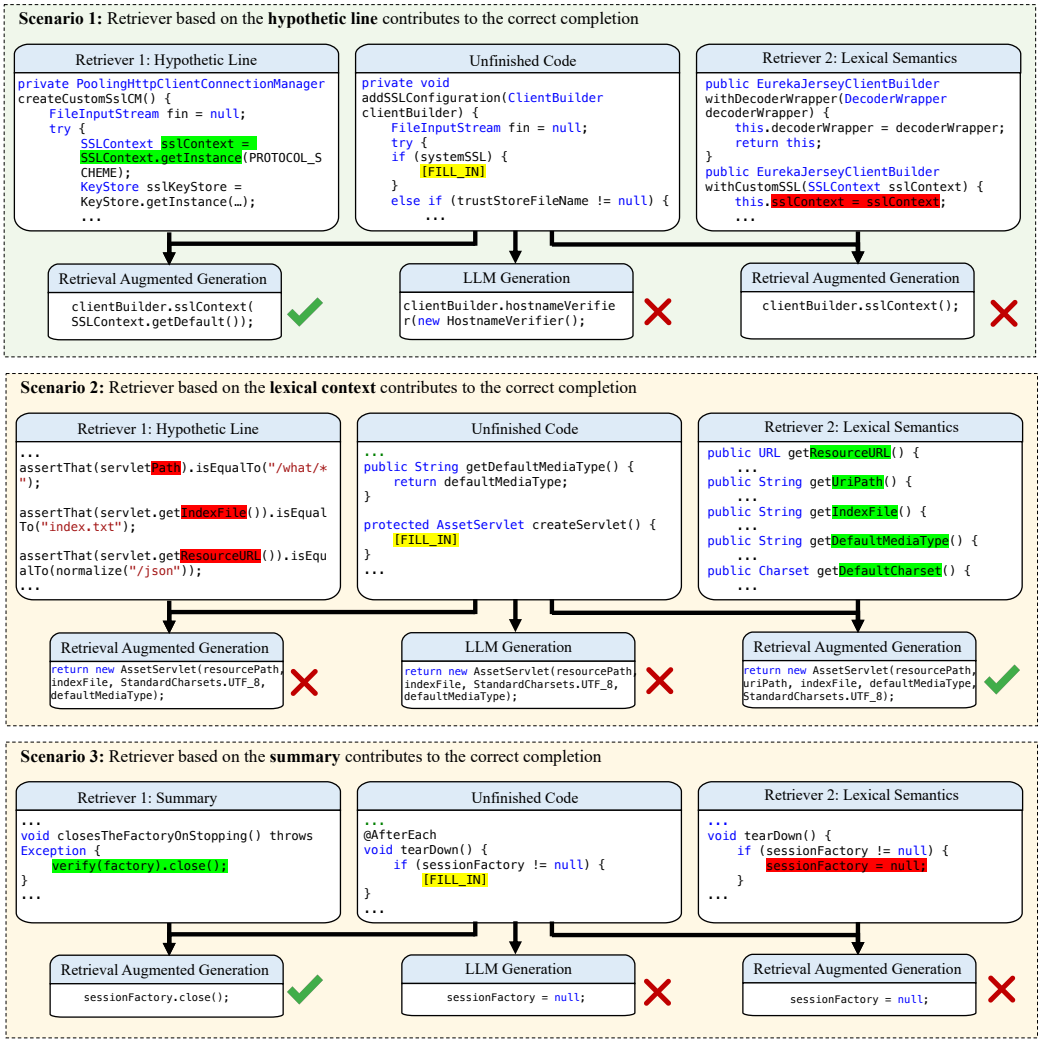

Fig. 1. Code completion scenarios demonstrating the contextual dependence for optimal retrievals. Red indicates the misleading information, Green represents the helpful hint.

"tearDown" the "sessionFactory". The hint "verify(factory).close();" can properly lead the generator to correctly manage the closure of the "sessionFactory".

These scenarios indicate the need to retrieve the code from multiple perspectives and select optimal retrieval. Thus, the complex nature of incomplete code poses the need for adjustable encoding perspectives to cover as much code semantics as possible. However, existing systems rely on pre-defined encoding models and require substantially additional training to encode the code from more perspectives. Thus they are limited in adjusting the retrieval perspective to cope with multifaceted code semantics. Accordingly, we can infer that it is essential for a flexible retrieval approach to adapt to code context with multiple perspectives for code completion.

In this paper, we attempt to address the limitations of prior code completion techniques by taking a more flexible approach based on *prompt-based multi-retriever system* and *adaptive retrieval selection algorithm* (as illustrated later). First, we leverage prompt engineering techniques to elicit a deeper, multi-faceted interpretation of the incomplete code based on LLM. By crafting prompts that guide the LLM to focus on multiple perspectives, we can expand the code representation beyond lexical features with no extra cost to train additional encoding models. This allows us to retrieve code snippets that can hint the completion even upon lexical dissimilarity. Second, we could attempt to employ adaptive selection to choose the most suitable retrieved results, i.e., dynamically making decisions from different retrieved information based on the specifics of the semantics of the target incomplete code snippet.

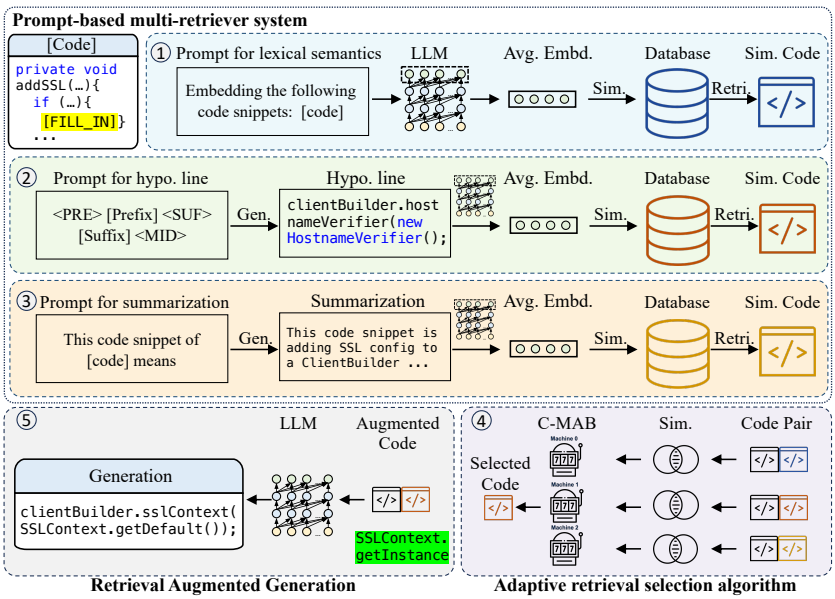

Fig. 2. The ProCC framework. The *prompt-based multi-retriever system* encodes the lexical semantics ①, hypothetical line ②, and code summarization ③ to derive multi-perspective representations. The *adaptive retrieval selection algorithm* ④ makes decisions based on code semantics similarities and selects the optimal context from retrievals. Finally, the selected code is concatenated with the unfinished code for augmented generation ⑤.

3 Approach

3.1 Overview

In this paper, we introduce ProCC, a novel framework leveraging prompt engineering and the contextual multi-armed bandit algorithm [38] to select suitable perspectives for code completion. As shown in Figure 2, ProCC adheres to the Retrieval-Augmented Generation (RAG) framework, as outlined in Equation 3 (the retrieval phase) and Equation 4 (the augmented generation phase). ProCC consists of two components—the *prompt-based multi-retriever system* and the *adaptive retrieval selection algorithm*. In particular, by adopting prompt engineering, the *prompt-based multi-retriever system* encodes the lexical semantics, hypothetical line, and code summarization to derive multi-perspective representations. More specifically, we employ three prompt retrieval models $\mathcal{R}_1$,

$\mathcal{R}_2, \mathcal{R}_3$ that encode the lexical semantics (①*Lexical Retri.*), generate a hypothetical line (②*Completion Retri.*), and produce the code summarization (③*Summarization Retri.*) respectively. This allows retrieving relevant snippets C^{R_1}, C^{R_2}, and C^{R_3} from the database based on representation similarities. Moreover, we use the contextual multi-armed bandit algorithm (④) to make decisions based on code semantics and select the optimal context from retrieved C^{R_1}, C^{R_2}, and C^{R_3}. Note that we are the first to introduce such an adaptive retrieval selection mechanism for code completion. The selected retrieval, for example, C^{R_1}, along with unfinished code $\hat{X}$ is then consumed by the model to perform the code completion, i.e. $P(x_1|\hat{X}, C_{R_1}; \theta)$ (⑤).

3.2 Prompt-based Multi-Retriever System

We formulate the multi-retriever in a unified paradigm by prompting the LLMs with designed prompt templates (or prompts for short for the rest of the paper). The construction of the retriever in the existing techniques is performed by encoding the code using auxiliary models, which requires additional resources for training the encoding model. By leveraging the LLM knowledge and crafting prompts, we can seamlessly represent code semantics from diverse perspectives with no need for extra models. This unified paradigm offers the advantage of flexibility and efficiency, thereby simplifying the process of constructing a multi-retriever system. Formally, given a language model $\mathcal{M}$, incomplete snippet $\hat{X}$, and a crafted prompt $Prompt$, we execute the model to process the input as Equation 5:

$$Out = \mathcal{M}(Prompt; \hat{X}) \tag{5}$$

We then extract the corresponding hidden states h of the output Out as the representation for the target perspective of the code $\hat{X}$. Finally, by crafting the prompts towards the lexical semantics, hypothetical line, and code summarization perspectives, we construct the following three retrievers.

Lexical Semantics. Incorporating lexical semantics into the retrieval process allows us to fetch relevant code snippets in terms of lexical semantics. Conventional techniques leverage contrastive pre-training to learn the code similarity [25, 45, 65]. For example, ContraCode [25] employs pre-training of an LLM to differentiate functionally similar program variants against non-equivalent distractors. However, in the domain of code completion, the code is typically unfinished and may not express coherent or consistent meaning, which is significantly different from the training samples of these contrastive models. Note that for code LLMs, as in Equation 2, they are pre-trained to auto-regressively generate the next token and complete the code. Hence, we deeply explore the knowledge embedded within LLMs by crafting prompts to encode incomplete snippets, as in Figure 2 ①. In particular, we craft the prompt, "Embedding the following code snippets: [code]" to encode the lexical semantics, where the "[code]" refers to the unfinished code. Then we extract the last hidden layer for the whole prompt and average them as the representation for lexical semantics. We name this retriever *Lexical Retri.*, which indicates that, despite its semantic retrieval intentions, its effective functionality is more aligned with lexical-level processing. This phenomenon largely stems from using the same model for both generation and embedding. Although this base model excels in generation, its direct application as an embedding model tends to prioritize textual information over semantic context. To address this, we attempt to design the prompts that could guide the model to focus on more in-depth contexts. This reclassification of lexical semantics helps understanding the capabilities and scope of different retrieval methodologies.

Hypothetical Line. Hypothetical line refers to the potential line that can complete the code, motivated by the concept of Hypothetical Document Embeddings (HyDE) [13]. In HyDE, a query question is passed into the model and guided to "write a document that answers the question", leading to the generation of a hypothetical document. This hypothetical document is then transformed

into an embedding in a vector space, enabling database search for retrieval. In our *prompt-based multi-retriever system*, as illustrated in Figure 2 ②, we use the prompt "<PRE> [Prefix] <SUF> [Suffix] <MID>" to generate the hypothetical line that acts as its representation where "[Prefix]" and "[Suffix]" represent the code snippets before and after the target insertion point respectively. Note that this structure is consistent with the pre-training format for Code Llama. For the code snippet C, we mask the line and input the surrounding code into the model $\mathcal{M}$ to generate the hypothetical line, storing the corresponding embedding to build the retrieval database D. For each incomplete code snippet $\hat{X}$, we follow the same process to generate its embedding for database search. Employing hypothetical line representations provides a critical benefit for retrieval—enriching the representation of incomplete code beyond the surrounding context. Generating embeddings solely from the available lexical semantics is prone to lack important semantics and patterns contained in the missing line itself. On the other hand, by prompting the model to generate a hypothetical line, the produced text exhibits relevant attributes like variable names, data types, and function signatures even when the lexical semantics alone does not offer such information. In other words, if two incomplete snippets generate hypothetical completions with similar or analogous variable names, function calls, etc., their overall functionality can be likely similar. This facilitates retrieving code snippet with conceptually relevant but lexically dissimilar information to the incomplete code. We name this retriever *Completion Retri.*.

Code Summarization. Code summarization allows capturing the overall functionality and the purpose of a code snippet in natural language. As humans tend to reason about code at a higher level of abstraction, summarization embeddings allow retrieval to focus on semantic similarity rather than superficial syntactic matches. Furthermore, natural language summaries provide a mechanism to inject human preferences into representation learning. This allows retrievals to better match human judgments of conceptual similarity. In our *prompt-based multi-retriever system*, as shown in Figure 2 ③, we use "This code snippet of [code] means" to produce code summarization and average the summary embeddings as the representation for the code. We name this retriever *Summarization Retri.*.

In fact, our prompt engineering can be easily extended beyond these perspectives, i.e., it can readily expand to encode any semantic dimensions of interest, which makes our *prompt-based multi-retriever system* potentially robust in the real world.

3.3 Adaptive Retrieval Selection Algorithm

After gathering retrieval information from multiple retrievers, it is essential to determine the optimal hints that aid code completion. Specifically, given the varied coverage and perspectives of different retrievers, directly concatenating all of them with the input could cause information overload and perspective confusion for code completion. Therefore, we aim to dynamically tailor the selection of the most suitable perspective for incomplete code.

We propose to tackle this perspective selection challenge as a learning problem [50]. The learning algorithm is presented with an action space that includes lexical semantics, hypothetical line, and code summarization perspectives. The algorithm's objective is to identify and select the most fitting prompt perspective while receiving rewards that reflect the quality of the selected perspective. To implement this learning approach, we adopt the multi-armed bandit algorithm, which is widely employed for recommendation systems [15, 28]. Specifically, we use the LinUCB algorithm [6], a variant of the multi-armed bandit algorithm that takes context into account. LinUCB is widely used in the multi-armed bandit problem [2, 21, 51]. It allows for effective handling of contextual information observed as the similarity between retrieved code and incomplete code. LinUCB enables a trade-off between exploring new options and using what is already known, adapting effectively

Algorithm 1: *adaptive retrieval selection algorithm*

Input: $\alpha \in \mathbb{R}_+$, num_arms, feature dimension $d \in \mathbb{N}$, Training set $\mathcal{D}$, Validation set $\mathcal{V}$, test set $\mathcal{T}$
Output: Trained parameters A, b and chosen retrieval for test set

Function `TrainLinUCB`(α, *num_arms*, d, $\mathcal{V}$)**:**
  for $a = 1$ **to** *num_arms* **do**
    $A[a] \leftarrow I_d$ ▷ Identity matrix for arm a
    $b[a] \leftarrow 0_d$ ▷ Zero reward vector for arm a
  for t *in* $\mathcal{V}$ **do**
    $\theta_t \leftarrow A^{-1}b$
    for $a = 1$ **to** *num_arms* **do**
      Retrieve the code from $\mathcal{D}$ for arm a
      Observe features $x_{t,a}$ including cosine and jaccard similarities
      $p_{t,a} \leftarrow \theta_t^\top x_{t,a} + \alpha\sqrt{x_{t,a}^\top A^{-1} x_{t,a}}$
    $a_t \leftarrow \arg\max_a p_{t,a}$
    $r_t \leftarrow 0$ ▷ Initialize reward to 0
    if *action* a_t *is an exact match* **then**
      $r_t \leftarrow 1$ ▷ Set reward to 1 for exact match
    $A \leftarrow A + x_{t,a_t} x_{t,a_t}^\top$
    $b \leftarrow b + x_{t,a_t} r_t$
  return A, b
Function `SelectRetrieval`($A, b, \mathcal{T}$)**:**
  $\theta_t \leftarrow A^{-1}b$
  for $a = 1$ **to** *num_arms* **do**
    Retrieve the code from $\mathcal{D}$ for arm a
    Observe features $x_{t,a}$ including cosine and jaccard similarities
    $p_{t,a} \leftarrow \theta_t^\top x_{t,a}$
  $a_t \leftarrow \arg\max_a p_{t,a}$
  return a_t

to complex code completion situations. The LinUCB algorithm offers the following advantages: 1) it enhances decision-making by continuously learning from outcomes and refining the selection process, and 2) it adapts to different contexts, optimizing performance for each input. As illustrated in Figure 2 ④, the different retrieval perspectives are seen as the "arms" of LinUCB and the goal is to identify which arm (i.e., perspective) yields the highest reward for each individual incomplete code snippet conditioned on the context. We choose the similarity score from each retriever as one dimension of the context for our LinUCB algorithm. Additionally, we use the Jaccard similarity between the retrieved snippets and incomplete code as another dimension of the context for our LinUCB algorithm, following the Jaccard metrics [23] used in RepoCoder [85]. We also set the Exact Match score as the reward for the LinUCB algorithm where the reward is 1 upon an Exact Match and 0 otherwise. Note that the algorithm is flexible and can be readily extended to include additional dimensions, such as other possible retrieval perspectives, techniques or reward metrics. As a result, the LinUCB algorithm makes informed decisions about the optimal perspective for each incomplete code snippet.

$$
\begin{aligned}
A &\leftarrow A + x_{i,a} x_{i,a}^{\top} \\
b &\leftarrow b + x_{i,a} r_i \\
\text{retrieval} &= \arg\max_a \left(A^{-1} b x_a \right)
\end{aligned}
\tag{6}
$$

For training LinUCB, we first set up the retrieval training set $\mathcal{D}$, the validation set $\mathcal{V}$ and test set in the typical 80/10/10 split manner from the RAG retrieval database. Parameters A and b are initialized as the Identity matrix and zero vector for each arm, respectively (Lines 2-4). For each entry in the validation set $\mathcal{V}$, we extract similar code from the training set $\mathcal{D}$, assess the cosine and Jaccard similarity features, compute the probability for each arm, and select the arm with the highest probability (Lines 5-11). Rewards are assigned as 1 if the code retrieved by an arm facilitates correct code completion; otherwise, the reward is 0. Subsequently, we update the LinUCB parameters A and b (Lines 12-16). After training, the updated parameters A and b are documented (Line 17). During testing, we follow a similar procedure to retrieve and evaluate code, and select the best arm based on probability calculations (Lines 19-25).

Specifically, the matrix A, which accumulates the outer products of feature vectors for the samples i, is updated as $A + x_{i,a} x_{i,a}^{\top}$, while the vector b, aggregating the product of rewards and feature vectors, is updated as $b + x_{i,a} r_i$. Here, x denotes the feature vector of the current arm, incorporating the Jaccard and cosine similarities between the incomplete code and the retrieval results, r_i refers to the reward reflecting the Exact Match result in our situation. Finally, the selected retrieval from the LinUCB algorithm along with the unfinished code is then consumed by the model to conduct the code completion (⑤).

4 Evaluation

To evaluate ProCC, we have formulated the following three research questions:

- **RQ1**: How does ProCC perform compared with state-of-the-art code completion techniques?
- **RQ2**: How do individual components of ProCC impact the performance?
- **RQ3**: How does ProCC perform compared with fine-tuning? Can it further improve a fine-tuned model?

4.1 Experiment Setup

4.1.1 Models. We chose two state-of-the-art code LLMs, DeepSeek-Coder [17], and Qwen2.5-Coder[22] as the base models for our paper.

- **DeepSeek-Coder**, released in October 2023, is trained on 2 trillion tokens covering more than 80 programming languages. It features a window size of 16K, supporting project-level code completion and infilling, and achieves state-of-the-art performance among open code models. In particular, we use its versions with 1.3B and 6.7B parameters.
- **Qwen2.5-Coder**, released in September 2024, built on the Qwen2.5 architecture and has been further trained on a massive dataset of more than 5.5 trillion tokens. It's one of the current SOTA open-source code models.We use its version with 7B parameters.

4.1.2 Baselines. We adopt the following baseline techniques for comparison, focusing on code completion frameworks that can seamlessly integrate with mainstream systems. Techniques that require retrieving new context for every generated token [30, 67] are not included in our analysis, as they are not supported by current widely employed LLM systems [32] or inference libraries [31]. Incorporating such techniques would require substantial modifications to the LLM systems and inference pipelines, making them impractical for real-world deployment and integration with existing systems.

- **BM25 [55].** BM25 is proposed upon the BM25 ranking algorithm [61], which is one of the most widely employed retrieval algorithms in the Question and Answering (QA) domain. We leverage BM25 to search for code similar to the given incomplete code. The retrieved code is then concatenated with the incomplete code and input into the LLMs for completion.
- **ReACC [45].** ReACC, published in ACL2022, employs the vanilla RAG-based framework for code completion. Since it does not provide complete reproducible encoding models for retrieval, we implement the framework by using the widely-adopted retrieval model GTE-large [40], which was released in August 2023.
- **RepoCoder [85].** RepoCoder, published in ENMLP2023, is a widely-studied RAG-based code completion technique. RepoCoder refines the code retrieval process by iteratively utilizing the most recently generated content to retrieve information. To ensure consistency, we employ GTE-large as its embedding model.

4.1.3 Benchmarks. To comprehensively evaluate the performance of LLMs for code completion tasks, we first adopt two widely-adopted public benchmarks CrossCodeEval (CCEval) [7] and HumanEval-Infilling [12]. However, CCEval does not fully reflect the real-world deployment scenarios in industry settings. In particular, during our deployment of the code completion systems in Kuaishou Technology, we observed that over 70% of code completion requires involving suffix information, i.e., the code following the insertion point, as illustrated in Equation 2. However, CCEval does not provide such suffix information, causing a potential gap in practice. To bridge this gap, we decided to build new datasets that incorporate suffix information which are sourced from both open-source repositories and industry codebases.

Table 1. Statistics of test datasets

Domain	Type	Abb.	Count
ProCC-Infilling Open-Source	Function Body	FB.	1692
	Random Lines	RL.	1096
	ALL	-	2788
Kuaishou Private-Domain	Function Body	FB.	1972
	Random Lines	RL.	1102
	ALL	-	3074

We follow the protocol of previous work [85] to crawl 20 high-quality code repositories from GitHub, covering multiple levels of code completion—random line completion and function body completion. These scenarios, often encountered by developers, can largely reflect real-world development scenarios. We randomly split 10% of the files as the test set, 10% for validation used in training the *adaptive retrieval selection algorithm*, and the rest as retrieval data. Following [85], for the line completion, we randomly select three lines from each test file, and for the function body completion, we extract all functions in test files. Eventually, we obtained a test dataset with 2788 instances. We conduct the same process for the validation set with each test file as a test case. We name this benchmark as the ProCC-Infilling benchmark.

Given that LLMs are pre-trained on expanded GitHub datasets, it might inadvertently encompass elements from our test set and lead to the risk of test set contamination. To alleviate this issue, we construct another benchmark based on private-domain code from the Kuaishou Technology. We collect 58 repositories and construct the dataset with the same protocol as discussed above. In total, we construct a test dataset with 3074 instances. We name this benchmark as the Kuaishou private-domain benchmark. Table 1 shows the test set details.

4.1.4 Metrics. Following previous studies [45, 85], we select two widely recognized evaluation metrics for code generation—*Exact Match (EM)* and *Edit Similarity (ES)*. In particular, EM quantifies the percentage of generated code snippets that exactly match the ground truth. ES, adapted from the Levenshtein Edit Distance [34], measures the required edit operations from generated content to the ground truth. For the evaluations on the CCEval benchmark, we use the same metrics including the identifier match as in the original paper [7].

4.1.5 Implementation. We use the Python implementation of the DeepSeek-Coder models and Qwen2.5-Coder obtained on Hugging Face [79]. they are employed for encoding the incomplete code from various perspectives through prompt engineering, and concurrently, for executing the code completion task. We implement the dense vector retrieval using *Faiss* [60]. For the LinUCB algorithm, we set the coefficient of the upper confidence bound $\alpha = 0.1$. We conduct LLM generation and embedding using the *vLLM* framework [31]. For fine-tuning (Section 4.2.3), we set $batch\ size = 64$ and $learning\ rate = 2e{-}5$ and train the models with the AdamW optimizer [44] for 2 epochs. To ensure fairness in the analysis of time and space complexity, all experiments are performed on a cluster equipped with 8 NVIDIA A100-80GB GPUs.

4.2 Results and Analysis

Table 2. Results of the CCEval benchmark. Numbers are shown in percentage (%)

Model/Retireval	Code Match						Identifier Match					
	Java		TypeScript		C#		Java		TypeScript		C#	
	EM	ES	EM	ES	EM	ES	EM	F1	EM	F1	EM	F1
DeepSeek-Coder-1.3B	6.36	56.73	6.59	55.91	3.17	58.27	12.30	45.65	11.26	46.43	6.90	32.50
+ bm25	13.46	59.78	12.22	59.46	12.95	62.14	21.41	50.89	17.76	51.17	17.14	41.65
+ ReACC	14.12	59.55	11.68	58.67	13.57	63.58	21.46	50.61	16.90	50.19	17.59	43.62
+ RepoCoder	14.35	59.60	12.66	59.47	**14.38**	**64.28**	21.88	50.99	17.94	51.33	**18.51**	**44.69**
+ *Lexical Retri.*	11.78	58.55	11.17	58.85	10.75	61.24	19.17	49.47	16.84	50.43	14.42	39.64
+ *Completion Retri.*	13.62	59.65	11.89	59.35	10.97	61.77	20.82	50.15	17.46	51.55	15.10	40.48
+ *Summarization Retri.*	14.40	59.34	12.07	59.21	10.24	61.95	21.69	50.61	17.61	50.93	14.31	40.61
+ ProCC	**15.24**	**61.40**	**13.29**	**60.24**	12.16	62.66	**23.19**	**52.91**	**18.98**	**52.96**	16.01	42.11
DeepSeek-Coder-6.7B	11.22	61.91	9.95	60.35	5.88	60.66	18.09	52.59	15.11	51.65	8.82	35.80
+ bm25	20.15	63.78	16.06	63.81	16.40	62.54	28.38	56.36	22.20	56.65	20.70	44.29
+ ReACC	19.54	63.10	15.35	63.03	16.35	63.69	27.49	56.08	21.36	55.91	20.53	45.54
+ RepoCoder	20.37	63.82	16.43	63.81	17.67	64.39	28.41	56.80	22.81	56.81	21.47	46.71
+ *Lexical Retri.*	18.93	63.92	13.50	62.44	15.61	62.54	27.26	56.44	19.43	54.78	18.78	43.71
+ *Completion Retri.*	21.60	63.79	15.88	63.72	17.19	64.54	29.87	57.12	22.14	54.79	21.72	46.75
+ *Summarization Retri.*	21.37	63.87	16.54	64.23	16.46	63.67	29.41	57.29	22.88	57.68	21.04	45.72
+ ProCC	**23.05**	**65.34**	**17.67**	**65.15**	**18.21**	**65.53**	**31.70**	**59.03**	**23.84**	**58.07**	**22.68**	**47.69**
Qwen2.5-Coder-7B	11.59	64.29	8.31	60.28	4.47	61.03	19.26	52.78	13.89	49.94	7.24	32.23
+ bm25	20.24	67.82	13.92	63.03	13.46	66.43	28.75	58.43	19.93	53.69	16.69	42.33
+ ReACC	20.10	67.51	13.51	62.81	13.69	66.74	28.55	58.12	19.41	53.31	16.43	43.15
+ RepoCoder	20.57	67.88	14.02	63.20	14.12	67.05	29.02	58.65	20.12	53.89	16.89	43.91
+ *Lexical Retri.*	19.76	67.13	13.02	62.77	12.83	66.01	27.88	57.63	19.06	52.67	15.99	41.78
+ *Completion Retri.*	21.87	68.74	13.89	63.55	13.94	66.64	29.18	58.39	20.12	53.76	16.89	43.56
+ *Summarization Retri.*	21.72	68.59	14.02	63.49	13.76	66.21	28.92	58.21	20.01	53.45	16.67	43.34
+ ProCC	**23.23**	**69.77**	**14.80**	**64.23**	**14.95**	**67.88**	**32.12**	**59.71**	**21.02**	**54.61**	**18.12**	**44.95**

4.2.1 RQ1: the overall effectiveness of ProCC. Table 2 presents the evaluation results on the CCEval benchmark, where ProCC demonstrates significant improvements across various programming language test sets. Specifically, for the DeepSeek-Coder-6.7B, in code match mertics, ProCC achieves

a 13.16% EM increase in Java (from 20.37% to 23.05%), a 7.55% EM increase in TypeScript (from 16.43% to 17.67%), and a 3.06% EM increase in C# (from 17.67% to 18.21%), demonstrating superior performance across various programming languages compared to the widely-studied RepoCoder. In addition to the enhancements in code match metrics, ProCC also shows significant improvements in identifier match represented by EM and F1 scores. In the Java test set, ProCC achieves an EM increase of 11.58% (from 28.41% to 31.70%) and an F1 score improvement of 3.93% (from 56.80% to 59.03%). In TypeScript, ProCC improves the EM by 4.52% (from 22.81% to 23.84%) and the F1 score by 2.22% (from 56.81% to 58.07%). For C#, there is an increase in EM by 5.64% (from 21.47% to 22.68%) and an F1 score enhancement by 2.10% (from 46.71% to 47.69%). These results further underscore the effectiveness of ProCC in handling identifier matches, enhancing both the precision and recall of code retrieval tasks compared to previous techniques like RepoCoder. Similarly trends are also observed on Qwen2.5-Coder-7B, with an averaged 8.12% and 5.99% EM improvement over RepoCoder on Code Match and Identifier Match respectively.

From the experimental results, we can observe that the impact of retrieval-augmented generation is more substantial on the DeepSeek-Coder-1.3B model than on the 6.7B model, with the former achieving a 140% EM enhancement on CCEval-Java compared to the 105% improvement of the latter using ProCC. This is due to the inherent limitations of small models in encoding and understanding extensive data. Retrieval-augmented generation addresses these limitations by integrating external and relevant information during the generation process. Additionally, when comparing the DeepSeek-Coder-6.7B and Qwen2.5-Coder-7B, the latter shows superior performance in Java, while the former performs better in C# and TypeScript. Despite these differences, the consistent performance trends across four benchmarks for both models demonstrate the robustness of ProCC.

Table 3. Averaged results on the HumanEval-Infilling benchmark

Retrieval	DeepSeek-Coder-1.3B		DeepSeek-Coder-6.7B		Qwen2.5-Coder-7B	
	EM	ES	EM	ES	EM	ES
Base	70.86	85.57	71.52	87.13	80.35	93.11
+ bm25	72.13	86.72	73.54	88.62	80.94	93.67
+ ReACC	72.45	87.05	73.73	89.03	80.54	93.43
+ RepoCoder	72.67	87.38	73.94	89.23	81.02	93.80
+ ProCC	**74.25**	**88.25**	**76.30**	**90.40**	**83.45**	**95.78**

Table 3 summarizes the experimental results on the HumanEval-Infilling dataset. This dataset, intended for direct code completion from partial codes, is not designed for the usage of RAG-based techniques as it does not include a retrieval dataset. To address this issue, we adopt the CCEval benchmark dataset for the retrieval purposes. Notably, ProCC significantly enhances the code completion performance. On DeepSeek-Coder-1.3B, ProCC records a 4.78% and 2.17% EM gain over the vanilla model and RepoCoder respectively (increasing from 70.86%/72.67% to 74.25%). On DeepSeek-Coder-6.7B, ProCC achieves a 6.68% increase in EM compared to the vanilla model (improving from 71.52% to 76.30%). It also shows a 3.19% EM improvement over RepoCoder (from 73.94% to 76.30%). On the Qwen2.5-Coder-7B model, ProCC records a 3.86% and 3.00% EM gain over the vanilla model and RepoCoder respectively (increasing from 80.35%/81.02% to 83.45%). The performance gain of RAG-based techniques, as shown in Table 2 for CCEval, is reduced in the HumanEval-Infilling dataset. For instance, for the Qwen2.5-Coder-7B model, while ProCC achieved a notable 11.64 absolute EM improvement in CCEval (rising from 11.59% to 23.23%), the performance increase in HumanEval-Infilling was only 3.10 absolute EM (from 80.35% to 83.45%). This discrepancy is due to the mismatch between the retrieval and completion data sets used in the experiments (CCEval for retrieval and HumanEval for completion). However, it is encouraging to note that retrieval still benefits completion tasks, even when the datasets are not directly linked.

Table 4. Averaged results on the ProCC-Infilling benchmark

Model/Retrieval	EM			ES		
	FB	RL	Avg.	FB	RL	Avg.
DeepSeek-Coder-1.3B	42.43	54.47	47.17	68.91	81.23	73.75
+ bm25	44.86	57.12	49.68	70.41	81.13	74.63
+ ReACC	44.56	56.20	49.14	70.75	80.13	74.43
+ RepoCoder	44.93	56.03	49.69	70.81	80.67	74.69
+ *Lexical Retri.*	44.62	56.93	49.46	70.39	80.73	74.46
+ *Completion Retri.*	44.86	55.93	49.21	69.88	79.74	73.76
+ *Summarization Retri.*	45.04	57.48	49.93	70.57	80.56	74.50
+ ProCC	**46.17**	**58.03**	**50.83**	**71.34**	**81.78**	**75.44**
DeepSeek-Coder-6.7B	46.69	65.24	53.98	72.46	85.53	77.60
+ bm25	48.94	69.16	56.89	73.35	85.97	78.31
+ ReACC	48.94	68.34	56.56	73.73	85.81	78.48
+ RepoCoder	49.59	68.80	57.14	73.72	85.80	78.47
+ *Lexical Retri.*	47.93	66.33	55.16	72.72	84.40	77.31
+ *Completion Retri.*	49.53	68.52	56.99	73.59	**86.45**	78.52
+ *Summarization Retri.*	49.76	69.53	57.53	73.87	85.98	78.63
+ ProCC	**50.98**	**70.73**	**58.74**	**74.56**	86.42	**79.22**
Qwen2.5-Coder-7B	49.41	64.23	55.24	75.68	85.25	79.44
+ bm25	53.25	69.43	59.61	77.03	87.03	81.08
+ ReACC	51.48	67.43	57.57	76.17	86.69	80.31
+ RepoCoder	51.85	67.88	58.15	76.35	86.92	80.51
+ *Lexical Retri.*	50.12	66.71	56.64	75.82	86.02	79.83
+ *Completion Retri.*	51.35	67.52	57.71	76.21	86.74	80.35
+ *Summarization Retri.*	51.85	68.05	58.22	76.48	86.89	80.57
+ ProCC	**54.12**	**70.05**	**60.38**	**77.82**	**87.89**	**81.78**

Table 4 presents the performance comparison results between ProCC and the other RAG-based techniques on top of the ProCC-Infilling benchmark, where "FB" refers to the function body, "RL" refers to the random line, and "Avg" refers to averaged results. We can observe that ProCC can significantly outperform the baseline techniques. In particular, for the average results on DeepSeek-Coder-1.3B, ProCC significantly improves the code completion task with 7.76% EM improvement over the vanilla model (from 47.17% to 50.83%) and demonstrates a 2.29% EM improvement over RepoCoder (from 49.69% to 50.83%). For the average results on DeepSeek-Coder-6.7B, ProCC achieves 8.82% and 2.80% EM gain over the vanilla model and RepoCoder respectively (from 53.98%/57.14% to 58.74%). Similarly, on Qwen2.5-Coder-7B, ProCC achieves 9.30% and 3.83% EM improvement over the vanilla model and RepoCoder respectively (from 55.24%/58.15% to 60.38%). These results indicate the power of ProCC for effectively retrieving relevant contextual information to enhance code completion effectiveness.

Table 5 summarizes the experimental results for the Kuaishou private-domain benchmark suite. Within this benchmark suite, the DeepSeek-Coder-1.3B achieves an average EM score of 33.67%, in contrast to the 47.17% EM observed in the ProCC-Infilling benchmark. This difference indicates that LLMs generally face challenges when dealing with domain-specific tasks where the test data falls outside their training corpus. Notably, in this scenario, ProCC has demonstrated a significant enhancement, achieving a 30.89% EM increase over the vanilla model (from 33.67% to 44.07%) and surpassing the RepoCoder by 5.1% (from 41.93% to 44.07%) on DeepSeek-Coder-1.3B. On DeepSeek-Coder-6.7B, it achieves 24.87% and 4.48% EM improvement over the vanilla model and RepoCoder respectively (from 43.90%/52.47% to 54.82%). Meanwhile, on Qwen2.5-Coder-7B, it

Table 5. Averaged results on the Kuaishou private-domain benchmark

Retrieval	DeepSeek-Coder-1.3B		DeepSeek-Coder-6.7B		Qwen2.5-Coder-7B	
	EM	ES	EM	ES	EM	ES
Base	33.67	63.30	43.90	73.10	45.48	74.49
+ bm25	41.84	67.02	51.84	75.67	53.38	77.05
+ ReACC	41.70	66.95	52.30	76.60	53.97	78.12
+ RepoCoder	41.93	67.13	52.47	76.53	53.91	78.05
+ *Lexical Retri.*	41.42	66.96	51.32	76.61	52.98	78.04
+ *Completion Retri.*	42.10	67.24	52.05	76.85	53.78	78.42
+ *Summarization Retri.*	42.79	67.92	52.71	76.53	54.12	78.28
+ ProCC	**44.07**	**69.13**	**54.82**	**77.62**	**56.43**	**79.02**

achieves 24.08% and 4.67% EM improvement over the vanilla model and RepoCoder respectively (from 45.48%/53.91% to 56.43%).

Comparison with Complex RAG-based Techniques. We compare ProCC against the more complex RAG-based techniques FT2Ra [18] and GraphCoder [43]. We refer to them as "complex" because, unlike ReACC, RepoCoder, and ProCC, they are somewhat incompatible with standard acceleration pipelines and are more difficult to be integrated into existing code completion frameworks. Specifically, ProCC performs embedding similarity searches, which can be accelerated via Faiss [60], allowing highly efficient, millisecond-scale lookups over millions of vectors. In contrast, GraphCoder relies on subgraph edit distance calculations, i.e., an approach without widespread library support, resulting in higher computational costs for scaling. Meanwhile, ProCC uses a standard generation pathway that can work with closed-source LLMs such as GPT. Moreover, by leveraging mainstream inference frameworks (e.g., vLLM [31], TGI [10], and SGLang[87]), it achieves up to 10x speedups over the standard LLM generation. FT2Ra, however, must retrieve a new context for each token it generates and thus cannot be applied to closed-source LLMs. Its token-by-token retrieval and generation paradigm also lacks support from mainstream acceleration frameworks, requiring substantial effort to reduce generation time for a real-world deployment.

We also evaluate the **retrieval time cost** of our studied techniques. To ensure fair comparison, we conduct the experiments on using a single A100 GPU and Xeon Gold CPU on the ProCC-Infilling benchmark with DeepSeek-Coder-1.3B as the embedding model. The retrieval time cost for ProCC, FT2Ra, and GraphCoder is **0.080s**, **0.413s**, and **0.528s** respectively. We can observe that ProCC significantly outperforms both FT2Ra and GraphCoder (>3x faster). To further illustrate, it has been surveyed that 85% developers expect autocomplete suggestions within 200ms [73] which FT2Ra and GraphCoder both exceed.

Table 6. Comparison with complex RAG-based techniques

Retrieval	ProCC-Infilling bench.		Kuaishou private-domain bench.	
	EM	ES	EM	ES
Base	47.17	73.75	33.67	63.30
+ bm25	49.68	74.63	41.84	67.02
+ ReACC	49.14	74.43	41.70	66.95
+ RepoCoder	49.69	74.69	41.93	67.13
+ GraphCoder	50.67	74.99	43.58	68.76
+ FT2Ra	50.71	75.35	43.42	68.65
+ *Lexical Retri.*	49.46	74.46	41.42	66.96
+ *Completion Retri.*	49.21	73.76	42.10	67.24
+ *Summarization Retri.*	49.93	74.50	42.79	67.92
+ ProCC	**50.83**	**75.44**	**44.07**	**69.13**

Table 6 also highlights their performance differences on the ProCC-Infilling and Kuaishou private-domain Benchmarks. It can be observed that FT2Ra and GraphCoder outperform the baseline techniques (including our single retriever), achieving an absolute EM gain of 1.02/1.49 and 0.98/1.65 over RepoCoder on the ProCC-Infilling/Kuaishou private-domain benchmarks, respectively. This indicates the benefits of introducing more complex RAG-based techniques. Note that ProCC still performs better than these complex techniques. ProCC achieves an absolute EM gain of 0.16/0.49 and 0.12/0.65 over GraphCoder and FT2Ra on the ProCC-Infilling/Kuaishou private-domain benchmarks, respectively. By flexibly incorporating diverse semantic cues and employing a lightweight LinUCB-based decision engine, ProCC addresses code completion requests by autonomously choosing the most suitable perspective at runtime. Overall, ProCC offers a lightweight, scalable, and high-performing RAG-based solution for code completion.

4.2.2 RQ2: the effectiveness of the components of ProCC. In this section, we systematically evaluate the effects of the key components in ProCC. For simplification, all the experiments are conducted based on DeepSeek-Coder-1.3B.

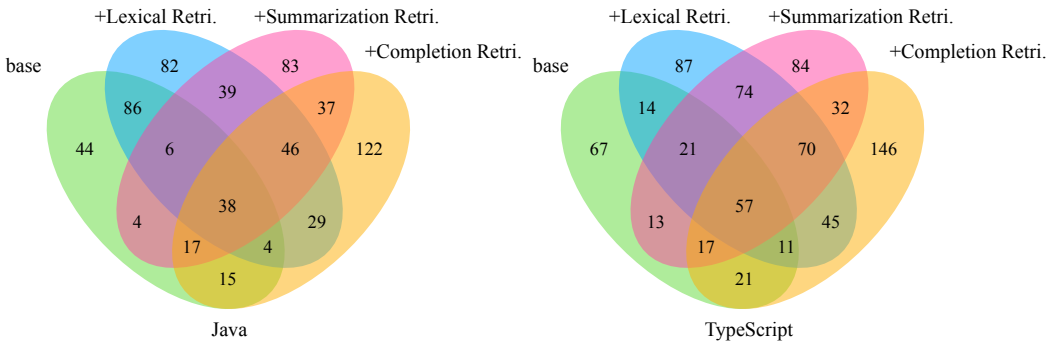

Fig. 3. Venn diagram of different retrievers. It shows the number of samples that are completed correctly in the CCEval Java and TypeScript benchmark.

Retrieval Perspectives. Our default individual retrievers are specialized in three perspectives—lexical semantics, hypothetical completion, and code summarization. Specifically, we plot a Venn diagram of the unique Exact Match achieved by each retriever for the CCEval benchmark in Figure 3. It is observed that each retriever brings to light different aspects of code semantics, thereby achieving exclusive successes in their targeted perspectives. For example, Figure 3 shows that the *Completion Retri.* uniquely retrieves correct contextual hints to complete 122 and 146 samples for Java and TypeScript respectively, demonstrating specialized strengths in its particular facet.

We further involved an evaluation of potential facets by designing various instructions to encode semantics from distinct perspectives. All the instruction perspectives utilized are presented in Table 7 and experiments are conducted on the ProCC-Infilling benchmark, where "[code]" symbolizes the incomplete code snippet, structured according to the FIM paradigm as described in Equation 2. The placeholder "[generation]" signifies the output generated by the LLM conditioned on the corresponding instruction template. We derive the instruction's average embedding from the last hidden layer of LLM, which forms the basis for the code representation in the retrieval process. Note that our default retrievers are No.1, No.3, and No.5 in Table 7. Our observations indicate that our single retriever with the directive instructions is effective and achieves comparative

Table 7. Retriever perspectives and instructions

No.	Perspect.	Instruction	AVG. EM	AVG. ES
0	Raw Code	[code]	49.32	73.49
1	Lexicon	Embedding the following code snippets: [code]	49.46	74.46
2	Lexicon	Representing the following code snippets: [code]	49.17	73.51
3	Hypo. Line	[code]->[generation]	49.21	73.76
4	Hypo. Line	Complete the code snippets [code]->[generation]	49.12	74.18
5	Summarization	This code snippets of [code] means -> [generation]	49.93	74.50
6	Summarization	Summarize the code snippets [code] -> [generation]	49.21	73.76

performance (between 49.12% to 49.93%) with the external encoding model (49.14%) as in Table 4. Note that the construction of the retriever perspectives only introduces minor variability in the code completion performance. For instance, a small variation like substituting “Embedding” with “Representing” in retriever No.1 leads to negligible 0.29 absolute EM differences. Nevertheless, the overall robustness and effectiveness of code completion tasks are consistently maintained. Similarly, using incomplete code directly (No.0) resulted in an EM score of 49.32%. In contrast, introducing the prompt "Embedding the following code" raised the EM score to 49.46%. These findings suggest that how prompts are structured within one perspective—whether by directly using the code or by incorporating a prefix—affects code completion performance only slightly. Additionally, the inference cost of five-word prefix is negligible, with no noticeable difference in timing whether the prefix is used or not. Therefore, we focus on designing three prompts from different perspectives which can elicit distinct semantic interpretations from the LLM to obtain a wider range of code semantics.

Table 8. Combination of different retrievers

Metrics	No.1+3	No.1+5	No.3+5	No.1+3+5	No.1-6
AVG. EM	50.27	50.33	50.46	50.83	50.98
AVG. ES	75.12	75.23	75.32	75.44	75.57

We also evaluate the combination of different retriever perspectives. Considering the prohibitive complexity of evaluating all combinations of retrievers, our investigation is confined to include the combinations between the most distinctive perspectives, i.e., retrievers No.1, No.3, and No.5 in Table 7. Table 8 presents the evaluation results on the ProCC-Infilling benchmark revealing that combining various perspectives outperforms individual ones, showcasing the value of the multi-retriever framework. For instance, combinations of retriever No.1+3, No.1+5, and No.3+5 yield EM scores of 50.27%, 50.33%, and 50.46% respectively, while the corresponding best single retriever presents an EM score of 49.93% (retriever No.5). Additionally, incorporating all six semantic perspectives from Table 7 further improves the code completion performance, though gains are marginal with only 0.15 absolute EM gain over 50.83% EM from the combination of three retrievers No.1, 3, and 5. This slight improvement indicates that while expanding a wide range of perspectives can offer benefits, potential overlapping information can limit its improvement. In conclusion, our multi-retriever framework approaches from three perspectives elicit distinct semantic interpretations from the LLM to obtain a wider range of code semantics. Incorporating these varied perspectives enriches the multifaceted representations and contributes to improved code completion performance.

Table 9. Decision-making using different algorithms

Method	EM	ES
Completion Retri.	49.21	73.76
Union	49.56	74.36
Max Similarity	49.57	74.48
Logistic Regression	49.71	74.68
LinUCB	50.83	75.44
LLM Decision	50.85	75.54

adaptive retrieval selection algorithm. Furthermore, we examine multiple decision-making algorithms. we first directly concatenate all three retrievals with input as one compared technique, namely "Union". Then we apply a naive decision approach that selects the retriever retrieved information based on the maximum dense vector similarity, called "Max Similarity". We also reframe this decision task as a classification problem, implementing logistic regression to choose the appropriate retriever, called "Logistic Regression". At last, we instruct the LLM to make decisions as in conventional multi-retriever systems, called "LLM". We include the optimal single retriever with the hypothetical completion as a reference. Table 9 shows the detailed results of the ProCC-Infilling benchmark. Our findings indicate that directly concatenating all three retrievers' retrievals with input achieves 49.56% EM. Direct concatenating risks present excessive information and only provide 0.35 slight absolute improvement over the best single retriever with the hypothetical completion, which achieves 49.21% EM. Employing the maximum similarity and logistic regression techniques generally enhances the performance of the best single retriever with 0.36 and 0.50 absolute EM improvement. The LinUCB algorithm, leveraging the similarity scores as contextual information, achieves performance with a 1.62 absolute EM improvement over the best single retriever and makes decisions about the optimal perspective for incomplete code. Notably, utilizing an LLM for decision-making achieves similar results to the LinUCB algorithm, suggesting that a lightweight decision-making approach is sufficient for the code completion task.

Model Ensemble. We attempt to assess the potential benefits of using a model ensemble strategy. We utilize DeepSeek-Coder-1.3B and Qwen2.5-Coder-1.3B as base models in our experiments, initially applying the same model for both embedding and code generation phases. To evaluate the model ensemble, we assign DeepSeek-Coder-1.3B for embedding to extract relevant context, and Qwen2.5-Coder-1.3B for the completion task, denoted as DeepSeek-Coder-1.3B + Qwen2.5-Coder-1.3B in Table 10, and vice versa. Our findings demonstrate that this model ensemble strategy has slight effect on the performance. While the ensemble approach achieves 46.95% EM points, using a single model (Qwen2.5-Coder-1.3B) for both procedures achieves 46.77%.

We left the exploration of model ensemble to the future work, as this study primarily explores the use of a single model for both retrieval and generation, highlighting that additional tuning of the retrieval model may not be essential.

Table 10. Evaluating model ensemble on the ProCC-Infilling benchmark

Model/Retrieval	EM			ES		
	FB	RL	Avg.	FB	RL	Avg.
DeepSeek-Coder-1.3B + DeepSeek-Coder-1.3B	46.17	58.03	50.83	71.34	81.78	75.44
Qwen2.5-Coder-1.5B + Qwen2.5-Coder-1.5B	46.77	58.03	51.20	72.03	82.26	76.05
Qwen2.5-Coder-1.5B + DeepSeek-Coder-1.3B	46.27	58.14	50.93	71.54	81.93	75.62
DeepSeek-Coder-1.3B + Qwen2.5-Coder-1.5B	**46.95**	**58.92**	**51.66**	**72.12**	**82.43**	**76.17**

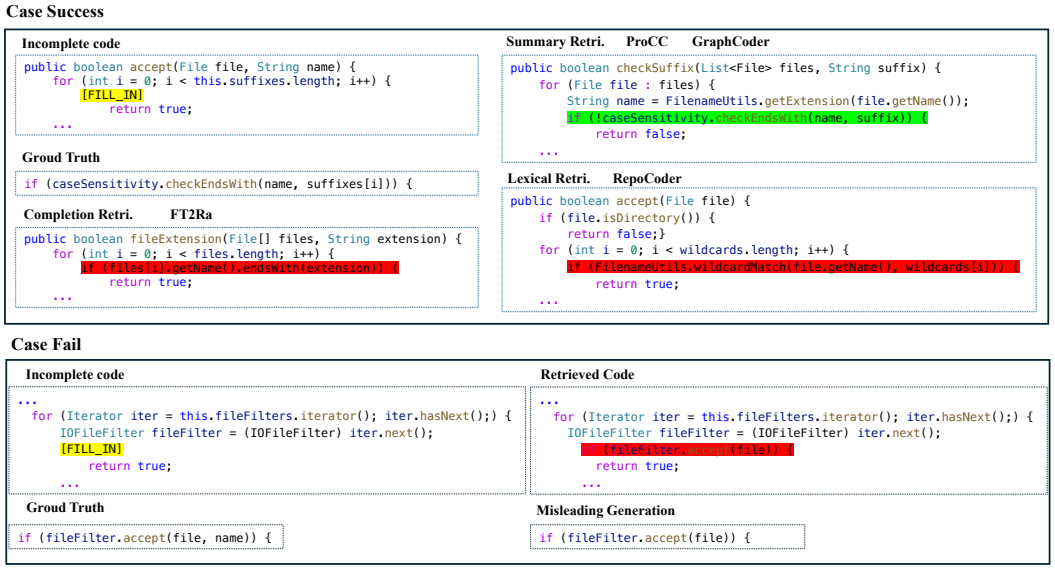

Fig. 4. Case study for the effectiveness of ProCC. Red indicates the misleading information, Green represents the helpful hint.

Case Study. We perform a case study for the baselines and each component of ProCC in Figure 4. In the Case Success, each component of ProCC retrieves a unique code segment. *Lexical Retri.* selects a segment sharing the same function name "accept" with the incomplete code, although the corresponding hint differs significantly from the ground truth. On the other hand, *Completion Retri.* identifies the line "if (files[i].getName().endsWith(extension))" to be the most appropriate for completing both the retrieved segment and the incomplete code. From *Summarization Retri.*, a summarization perspective, both the incomplete and retrieved codes primarily check the "suffix" of the variable "name", providing the correct context needed for code completion. ProCC leverages LinUCB to ensure that summarization provides the best guidance for completion. In the Case Fail, the correct completion "if (fileFilter.accept(file, name))" involves two parameters, "file" and "name". However, all retrievers favor a segment that is almost identical to the incomplete code but only takes "file" as an input, leading to misguided completion. This issue arises because LLMs tend to replicate the subsequent line from a closely matching retrieved snippet without accounting for slight variations in the context of completion. This is a fundamental challenge associated with the RAG-based techniques. While post-processing techniques such as self-reflection and agent systems [54] can help mitigate this problem, they are beyond the scope of our current study.

4.2.3 RQ3: performance compared with fine-tuning. This section presents a systematic comparison between ProCC and the conventional fine-tuning approach, evaluating their advantages and limitations. Moreover, we investigate the potential of ProCC to enhance the performance of models that have been already fine-tuned, thereby assessing its value as a supplementary optimization technique for fine-tuned models. For simplification, all the experiments are conducted based on DeepSeek-Coder-1.3B.

Training Dataset. For a fair comparison, the test dataset employed for fine-tuning is identical to that used for retrieval, i.e., a total of 2788 test samples from a corpus of 20 repositories (ProCC-Infilling). The remaining files within these repositories are utilized to construct the validation

and training sets, following the same protocol outlined in Section 4.1.3. This process yields 22,646 training samples used for the fine-tuning of hyper-parameters and 2790 validation samples. For the Kuaishou private-domain benchmark, we employed the same setup.

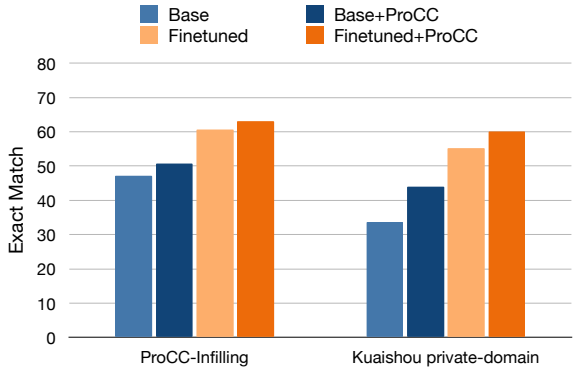

Fig. 5. Finetune v.s. ProCC

Results. In addition to the fine-tuning experiment, we apply ProCC to the fine-tuned models. Figure 5 presents the evaluation results which show that fine-tuning significantly enhances code completion performance, achieving an absolute 13.58 EM gain on the ProCC-Infilling benchmark suite and 21.51 EM gain on the Kuaishou private-domain benchmark suite compared to the baseline model respectively (from 47.17% to 60.75% and from 33.67% to 55.18%). Furthermore, the application of ProCC to this fine-tuned model yields an additional 2.46 EM improvement on the ProCC-Infilling benchmark suite and 4.94 EM improvement on the Kuaishou private-domain benchmark suite. These findings indicate that ProCC is an effective augmentation to an optimized fine-tuned system.

Table 11. Time costs (second)

Method	Processing	Completion	Total
base	-	0.261	0.261
ReACC	0.015	0.261	0.277
RepoCoder	0.285	0.261	0.547
ProCC	0.080	0.261	0.342

Training Cost. While fine-tuning demonstrates substantial efficacy in enhancing code completion tasks, it is essential to consider the associated computational cost. Fine-tuning the DeepSeek-Coder-1.3B model requires substantial hardware resources, typically involving a cluster with eight NVIDIA A100 GPUs. In contrast, deploying ProCC is considerably more resource-efficient and operable on a single A100 GPU. In terms of computation time, fine-tuning requires a training time of approximately 2.5 hours on the 8×A100 cluster. Conversely, ProCC eliminates the need for training time, with retrieval time aggregated to approximately 0.08 seconds on the same device.

Inference Cost. As shown in Table 11, using the 1.3b model for code completion takes an average of 0.261 seconds. Note that when using additional retrieval context, we allocate a fixed input length budget to the model to ensure that the prefill time remains consistent[7]. ReACC utilizes the vanilla RAG framework for code completion, which requires an average of 0.015 seconds for embedding and retrieval of code snippets from the memory. RepoCoder builds on ReACC by using

the results generated by the model to iteratively retrieve similar code snippets, thus requiring additional completion periods, i.e., 0.285 seconds in total for retrieval process. Our ProCC, which implements a lightweight multi-retriever framework that simplifies the iterative refinement, incurs only 0.08 seconds for the retrieval time for its retriever operation. To summarize, it could enhance the effectiveness with reasonable computational costs.

5 Threats to validity

Internal Validity. The threat to internal validity lies in potential implementation bugs. To mitigate this, for compared techniques, we obtained original source code from GitHub repositories and used identical hyperparameters from their papers. And we have conducted a thorough review of our code scripts to ensure their correctness.

External Validity. The threats to external validity mainly lie in the benchmarks and techniques studied. To reduce these threats, we not only used established benchmarks but also included industry data unknown to LLMs. Through an exhaustive literature review, we believe the compared RAG-sequence models are sufficiently representative. Another threat is randomness in results. To alleviate this threat, we averaged results over five runs, reducing variance.

Construct Validity. The threat to construct validity lies in our evaluation metrics. Following previous work [45, 85], we adopted two widely-used metrics—Exact Match and Edit Similarity to comprehensively assess performance. Using established metrics provides rigorous quantification of improvements.

6 Related Work

Language Model for Code Completion. To generate code completions of arbitrary lengths, researchers view code as a distinct variant of language and have subsequently used natural language processing techniques (NLP) to model code statistically. Earlier work leveraged N-gram models [59], recurrent neural networks such as LSTM [58], and attention mechanisms [37] to encode programming languages. With the emergence of transformer-based models, language models (LMs) are trained on large-scale code datasets, which has significantly advanced code completion. CodeBERT [11], one of the pioneering code LMs, performs the code completion task through masked language modeling. To facilitate the generation capability, later LMs mainly adopt either a decoder-only or an encoder-decoder model, which is trained to predict the subsequent token in an auto-regressive manner. For example, CodeGPT [46], which follows the architecture of decoder-only GPT [57], outperforms GPT2 in the code completion task. UniXCoder [16], a mixed encoder-decoder model, integrates multi-task learning strategies and leverages code structures to enhance pre-training and further advances code completion performance. Recent LLMs, such as Codex [5], CodeGen [52], InCoder [12], and StarCoder [39] employ billions of parameters and trained on trillions of code tokens, significantly excel in code generation tasks. Notably, more recent models like DeepSeek-Coder [17] and Qwen2.5-Coder [22] adopt the fill-in-the-middle pre-training objective [4], which resembles incomplete code contexts in code completion. This provides useful inductive bias, enabling DeepSeek-Coder and Qwen2.5-Coder to substantially outperform prior non-infilling models on completion benchmarks [3].

Retrieval Augmented Code Completion. Retrieval augmented generation [35] (RAG) has emerged as a technique to inject external knowledge into large language models (LLMs) to assist coherent text generation and mitigate hallucination for code completion. The RAG paradigm typically first retrieves the most relevant information using similarity measures such as BM25, dense embeddings such as SimCSE [14] or Dense Passage Retrieval [29] (DPR). The retrieved information is then

concatenated with the original input to guide the generation of LLM. Although initially explored for open-domain question answering, RAG has recently been adapted for code completion [45, 67, 85]. Early work in code completion [45] focused on code-to-code retrieval using dual encoder models with the retrieved results fed to autoregressive LMs. While RepoCoder [85] advances retrieval by iterating with incremental generations, KNM [67] incorporates in-domain code databases and utilizes Bayesian inference to finalize the code. Recently, GraphCoder [43] utilizes a code context graph (CCG) for retrieval and incorporates decay-with-distance subgraph edit distance to refine the CCG retrieval results. FT2Ra [18] introduces a retrieval paradigm with a learning rate and multi-epoch retrievals that mimics fine-tuning. Some other research works focus on cross-file retrieval or repository-level retrieval [75], i.e., drawing context from cross-file context dependencies like imported libraries (e.g., "$from\ transformers\ import\ GPTModelForCLM$") or header files ("$include\ bta\_hh\_co.h$"). CrossCodeEval [7] and RepoBench [42] construct benchmarks for such scenarios, while CocoMic [8] develops a cross-file context finder CCFINDER to identify and retrieve the most relevant cross-file context and integrates cross-file context to learn the in-file and cross-file context jointly by pre-trained code LLMs.

In this paper, we propose ProCC, a code completion framework leveraging prompt engineering and contextual multi-armed bandit for the first time to flexibly incorporate and adapt to multiple perspectives of code. Our extensive evaluation results indicate that ProCC can significantly enhance the code completion effectiveness over the existing RAG-based code completion techniques, indicating the strengths of our proposed RAG-based mechanisms.

7 Conclusion

In this paper, we propose ProCC, the first code completion technique to integrate prompt engineering and contextual multi-armed bandit to flexibly incorporate and adapt to multiple perspectives of code. ProCC first employs a *prompt-based multi-retriever system* which crafts prompt templates to elicit LLM knowledge to understand code semantics with multiple retrieval perspectives. Then, it adopts the *adaptive retrieval selection algorithm* to incorporate code similarity into the decision-making process to determine the most suitable retrieval perspective for the LLM to complete the code. Extensive evaluations across the CCEval, HumanEval-Infilling, ProCC-Infilling, and Kuaishou private-domain benchmarks demonstrate the superior performance and adaptability of ProCC, marking a significant advancement over the widely-studied RepoCoder by 7.92%, 3.19%, 2.80%, and 4.48% in terms of EM respectively. Additionally, ProCC offers the flexibility to augment fine-tuned techniques with an averaged 6.5% performance improvement over the fine-tuned model.

8 Data Availability

We provide the repository [1] for all the other available materials, including the source code of the artifact and the open-source dataset. Considering the deployment of the artifact within Kuaishou Technology and the privacy protection policy, the dataset containing the private-domain code of the company shall remain undisclosed.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (No. 62372220), the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. PolyU/25200821), the Innovation and Technology Fund (Project No. PRP/047/22FX), and PolyU Internal Fund from RC-DSAI (Project No. 1-CE1E). This work is also partially supported by Kwai Inc.

References

[1] 2024. Prompt-based Code Completion via Multi-Retrieval Augmented Generation. https://github.com/anonepo/procc GitHub repository.

[2] Parand A. Alamdari, Yanshuai Cao, and Kevin H. Wilson. 2024. Jump Starting Bandits with LLM-Generated Prior Knowledge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 19821–19833. https://doi.org/10.18653/v1/2024.emnlp-main.1107

[3] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alexander Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, J. Poirier, Hailey Schoelkopf, Sergey Mikhailovich Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Franz Lappert, Francesco De Toni, Bernardo Garc'ia del R'io, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luisa Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Christopher Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! *ArXiv* abs/2301.03988 (2023). https://api.semanticscholar.org/CorpusID:255570209

[4] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255* (2022).

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[6] Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. 2011. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 208–214.

[7] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. *ArXiv* abs/2310.11248 (2023). https://api.semanticscholar.org/CorpusID:264172238

[8] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. *ArXiv* abs/2212.10007 (2022). https://api.semanticscholar.org/CorpusID:254877371

[9] Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2023. How Abilities in Large Language Models are Affected by Supervised Fine-tuning Data Composition. *ArXiv* abs/2310.05492 (2023). https://api.semanticscholar.org/CorpusID:263830318

[10] Hugging Face. 2025. Large Language Model Text Generation Inference. https://github.com/huggingface/text-generation-inference GitHub repository.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *ArXiv* abs/2002.08155 (2020). https://api.semanticscholar.org/CorpusID:211171605

[12] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida I. Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *ArXiv* abs/2204.05999 (2022). https://api.semanticscholar.org/CorpusID:248157108

[13] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2022. Precise Zero-Shot Dense Retrieval without Relevance Labels. *ArXiv* abs/2212.10496 (2022). https://api.semanticscholar.org/CorpusID:254877046

[14] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. *ArXiv* abs/2104.08821 (2021). https://api.semanticscholar.org/CorpusID:233296292

[15] Kaan Gönç, Baturay Sağlam, Onat Dalmaz, Tolga Çukur, Serdar Kozat, and Hamdi Dibeklioğlu. 2023. User Feedback-based Online Learning for Intent Classification. *Proceedings of the 25th International Conference on Multimodal Interaction* (2023). https://api.semanticscholar.org/CorpusID:263742809

[16] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Annual Meeting of the Association for Computational Linguistics*. https://api.semanticscholar.org/CorpusID:247315559

[17] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[18] Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu. 2024. 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* (Vienna, Austria) *(ISSTA 2024)*. Association for

Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/3650212.3652130
[19] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 27–38.
[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[21] Mohanna Hoveyda, Arjen P. de Vries, Maarten de Rijke, Harrie Oosterhuis, and Faegheh Hasibi. 2024. AQA: Adaptive Question Answering in a Society of LLMs via Contextual Multi-Armed Bandit. arXiv:2409.13447 [cs.CL] https://arxiv.org/abs/2409.13447
[22] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] https://arxiv.org/abs/2409.12186
[23] Paul Jaccard. 1912. The distribution of the flora in the alpine zone. 1. *New phytologist* 11, 2 (1912), 37–50.
[24] Rolf Jagerman, Honglei Zhuang, Zhen Qin, Xuanhui Wang, and Michael Bendersky. 2023. Query Expansion by Prompting Large Language Models. arXiv:2305.03653 [cs.IR]
[25] Paras Jain, Ajay Jain, Tianjun Zhang, P. Abbeel, Joseph Gonzalez, and Ion Stoica. 2020. Contrastive Code Representation Learning. In *Conference on Empirical Methods in Natural Language Processing*. https://api.semanticscholar.org/CorpusID:220425360
[26] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 224, 13 pages. https://doi.org/10.1145/3597503.3639100
[27] Ting Jiang, Jian Jiao, Shaohan Huang, Zihan Zhang, Deqing Wang, Fuzhen Zhuang, Furu Wei, Haizhen Huang, Denvy Deng, and Qi Zhang. 2022. PromptBERT: Improving BERT Sentence Embeddings with Prompts. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 8826–8837. https://doi.org/10.18653/v1/2022.emnlp-main.603
[28] Jai Kannan, Scott Barnett, Anj Simmons, Taylan Selvi, and Luís Cruz. 2023. Green Runner: A tool for efficient model selection from model repositories. *ArXiv* abs/2305.16849 (2023). https://api.semanticscholar.org/CorpusID:258947508
[29] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Yu Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. *ArXiv* abs/2004.04906 (2020). https://api.semanticscholar.org/CorpusID:215737187
[30] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through Memorization: Nearest Neighbor Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HklBjCEKvH
[31] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.
[32] LangChain. 2023. How to Stream Results from Your RAG Application. https://python.langchain.com/v0.2/docs/how_to/qa_streaming/#streaming-final-outputs Accessed: 2024-05-29.
[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
[34] Vladimir I Levenshtein. 1965. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady* 10, 8 (1965), 707–710.
[35] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
[36] Jingxuan Li, Rui Huang, Wei Li, Kai Yao, and Weiguo Tan. 2021. Toward less hidden cost of code completion with acceptance and ranking models. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 195–205.
[37] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2017. Code Completion with Neural Attention and Pointer Networks. *ArXiv* abs/1711.09573 (2017). https://api.semanticscholar.org/CorpusID:11683607
[38] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. 2010. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 208–214.
[39] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
[40] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards General Text Embeddings with Multi-stage Contrastive Learning. *ArXiv* abs/2308.03281 (2023). https://api.semanticscholar.org/

CorpusID:260682258

[41] Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. TruthfulQA: Measuring How Models Mimic Human Falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 3214–3252. https://doi.org/10.18653/v1/2022.acl-long.229

[42] Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. *ArXiv* abs/2306.03091 (2023). https://api.semanticscholar.org/CorpusID:259075246

[43] Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. GraphCoder: Enhancing Repository-Level Code Completion via Coarse-to-fine Retrieval Based on Code Context Graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 570–581. https://doi.org/10.1145/3691620.3695054

[44] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101 [cs.LG]

[45] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 6227–6240.

[46] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[47] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. *ACM Sigplan Notices* 40, 6 (2005), 48–61.

[48] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. 2020. On Faithfulness and Factuality in Abstractive Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 1906–1919. https://doi.org/10.18653/v1/2020.acl-main.173

[49] Microsoft. 2023. Pyright: Static Type Checker for Python. https://github.com/microsoft/pyright

[50] Akshay Uttama Nambi, Vaibhav Balloli, Mercy Prasanna Ranjit, Tanuja Ganu, Kabir Ahuja, Sunayana Sitaram, and Kalika Bali. 2023. Breaking Language Barriers with a LEAP: Learning Strategies for Polyglot LLMs. *ArXiv* abs/2305.17740 (2023). https://api.semanticscholar.org/CorpusID:258959055

[51] Duy Nguyen, Archiki Prasad, Elias Stengel-Eskin, and Mohit Bansal. 2024. LASeR: Learning to Adaptively Select Reward Models with Multi-Armed Bandits. arXiv:2410.01735 [cs.CL] https://arxiv.org/abs/2410.01735

[52] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG]

[53] OpenAI. 2023. *GPT-4 Technical Report.* Technical Report. OpenAI. https://arxiv.org/pdf/2303.08774.pdf

[54] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 2, 22 pages. https://doi.org/10.1145/3586183.3606763

[55] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. arXiv:2108.11601 [cs.SE]

[56] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 275–286.

[57] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[58] Md. Mostafizer Rahman, Yutaka Watanobe, and Keita Nakamura. 2020. A Neural Network Based Intelligent Support Model for Program Code Completion. *Sci. Program.* 2020 (2020), 7426461:1–7426461:18. https://api.semanticscholar.org/CorpusID:225584181

[59] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014). https://api.semanticscholar.org/CorpusID:13040187

[60] Facebook Research. 2023. FAISS: A library for efficient similarity search and clustering of dense vectors. https://github.com/facebookresearch/faiss GitHub repository.

[61] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[62] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*

(2023).
[63] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).
[64] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.
[65] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2022. CoCoSoDa: Effective Contrastive Learning for Code Search. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2022), 2198–2210. https://api.semanticscholar.org/CorpusID:256827724
[66] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 3473–3487. https://doi.org/10.18653/v1/2024.emnlp-main.203
[67] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain Adaptive Code Completion via Language Models and Decoupled Domain Databases. *arXiv preprint arXiv:2308.09313* (2023).
[68] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.
[69] Kenta Terada and Yutaka Watanobe. 2019. Code Completion for Programming Education based on Recurrent Neural Network. *2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA)* (2019), 109–114. https://api.semanticscholar.org/CorpusID:210694727
[70] Zhao Tian and Junjie Chen. 2023. Test-Case-Driven Programming Understanding in Large Language Models for Better Code Generation. *ArXiv* abs/2309.16120 (2023). https://api.semanticscholar.org/CorpusID:263136277
[71] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288
[72] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Neural Information Processing Systems*. https://api.semanticscholar.org/CorpusID:13756489
[73] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. How Practitioners Expect Code Completion?. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1294–1306. https://doi.org/10.1145/3611643.3616280
[74] Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. 2024. How and Why LLMs Use Deprecated APIs in Code Completion? An Empirical Study. *CoRR* abs/2406.09834 (2024). https://doi.org/10.48550/ARXIV.2406.09834 arXiv:2406.09834
[75] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. 2025. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). https://doi.org/10.1145/3714462 Just Accepted.
[76] Liang Wang, Nan Yang, and Furu Wei. 2023. Query2doc: Query Expansion with Large Language Models. *ArXiv* abs/2303.07678 (2023). https://api.semanticscholar.org/CorpusID:257505063
[77] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).
[78] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. *Code Generation as a Dual Task of Code Summarization.* Curran Associates Inc., Red Hook, NY, USA.
[79] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[80] Chaoyi Wu, Xiaoman Zhang, Ya Zhang, Yanfeng Wang, and Weidi Xie. 2023. PMC-LLaMA: Towards Building Open-source Language Models for Medicine. https://api.semanticscholar.org/CorpusID:258417843

[81] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhang, Haotian Zhang, and Yuqun Zhang. 2024. How Far Can We Go with Practical Function-Level Program Repair? arXiv:2404.12833 [cs.SE] https://arxiv.org/abs/2404.12833

[82] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. WizardLM: Empowering Large Language Models to Follow Complex Instructions. *ArXiv* abs/2304.12244 (2023). https://api.semanticscholar.org/CorpusID:258298159

[83] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. arXiv:2212.09420 [cs.SE]

[84] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/3533767.3534390

[85] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 2471–2484. https://aclanthology.org/2023.emnlp-main.151

[86] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. arXiv:2306.05685 [cs.CL]

[87] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: Efficient Execution of Structured Language Model Programs. arXiv:2312.07104 [cs.AI] https://arxiv.org/abs/2312.07104

Received Oct 28 2024; revised Feb 03 2024; accepted Mar 17 2024