

# PseudoBridge: Pseudo Code as the Bridge for Better Semantic and Logic Alignment in Code Retrieval

YIXUAN LI, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China

XINYI LIU, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China

WEIDONG YANG\*, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China

BEN FEI, Department of Information Engineering, The Chinese University of Hong Kong, China

SHUHAO LI, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China

MINGJIE ZHOU, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China

LIPENG MA\*, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China

Code search aims to precisely find relevant code snippets that match natural language queries within massive codebases, playing a vital role in software development. Recent advances leverage pre-trained language models (PLMs) to bridge the semantic gap between unstructured natural language (NL) and structured programming languages (PL), yielding significant improvements over traditional information retrieval and early deep learning approaches. However, existing PLM-based methods still encounter key challenges, including a fundamental semantic gap between human intent and machine execution logic, as well as limited robustness to diverse code styles. To address these issues, we propose **PseudoBridge**, a novel code retrieval framework that introduces pseudo-code as an intermediate, semi-structured modality to better align NL semantics with PL logic. Specifically, PseudoBridge consists of two stages: First, we employ an advanced large language model (LLM) to synthesize pseudo-code, enabling explicit alignment between NL queries and pseudo-code. Second, we introduce a logic-invariant code style augmentation strategy and employ the LLM to generate stylistically diverse yet logically equivalent code implementations with pseudo-code, then align the code snippets of different styles with pseudo-code, enhancing model robustness to code style variation. We build PseudoBridge across 10 different PLMs and evaluate it on 6 mainstream programming languages. Extensive experiments demonstrate that PseudoBridge consistently outperforms baselines, achieving significant gains in retrieval accuracy and generalization, particularly under zero-shot domain transfer scenarios such as Solidity and XLCOST datasets. These results demonstrate the effectiveness of explicit logical alignment via pseudo-code

---

\*Corresponding Authors

Authors' Contact Information: Yixuan Li, yxli24@m.fudan.edu.cn, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China; Xinyi Liu, liuxiny24@m.fudan.edu.cn, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China; Weidong Yang, wdyang@fudan.edu.cn, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China; Ben Fei, benfei@cuhk.edu.hk, Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong, China; Shuhao Li, shli23@m.fudan.edu.cn, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China; Mingjie Zhou, mjzhou19@m.fudan.edu.cn, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China; Lipeng Ma, lpma21@m.fudan.edu.cn, Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, Shanghai, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2018/9-ART

<https://doi.org/XXXXXXXX.XXXXXXXX>

and highlight PseudoBridge's potential as a robust, generalizable solution for code retrieval. Our source code and synthesized data are available at <https://github.com/yixuanli1230/PseudoBridge>.

**Additional Key Words and Phrases:** Code Retrieval, Large Language Model, Pretrained Language Model, Semantic Alignment

#### **ACM Reference Format:**

Xixuan Li, Xinyi Liu, Weidong Yang, Ben Fei, Shuhao Li, Mingjie Zhou, and Lipeng Ma. 2018. PseudoBridge: Pseudo Code as the Bridge for Better Semantic and Logic Alignment in Code Retrieval. 1, 1 (September 2018), 24 pages. <https://doi.org/XXXXXXX.XXXXXXX>

## 1 Introduction

Code retrieval refers to the task of searching relevant code snippets from a large codebase in response to a natural language query [40, 50, 51]. Natural language queries are typically unstructured textual descriptions, whereas code snippets are expressed in structured programming languages. Consequently, the core challenge in code retrieval is to effectively align natural language (NL) with programming language (PL). In recent years, researchers have proposed various methods to improve code retrieval performance, such as data augmentation [42] and high-quality training datasets construction [45]. With the rise of large language models (LLMs), code retrieval as a critical component of Retrieval-Augmented Generation (RAG) [52] and In-Context Learning (ICL) [26], has been widely adopted to enhance performance in tasks like code generation [9, 47, 54], underscoring its growing significance.

Early approaches to code retrieval are primarily based on information retrieval (IR) techniques [4, 34, 38], which treat code as plain text and rely on keyword matching. However, such token-based methods depend on lexical co-occurrence statistics and struggle to effectively capture the semantic correspondence between natural language (NL) queries and code snippets [15, 51]. The rapid development of deep learning (DL) has driven significant breakthroughs in code retrieval, thanks to its strong feature extraction and representation learning capabilities [10, 30, 51]. DL-based methods can be generally divided into two categories. The first category employs conventional neural architectures (e.g., RNN) [10, 30], where static word embedding models are used to achieve vectors for both NL and code snippets, then leverage neural networks to align their semantics with supervised training. The second category leverages pre-trained language models (PLMs) [14, 17, 18, 28, 42]. These approaches utilize transformer-based architectures pre-trained on large-scale code and natural language corpora to learn universal semantic representations across both modalities. Compared to earlier approaches, PLM-based methods achieve state-of-the-art performance by harnessing the powerful semantic representation capabilities of PLMs. Moreover, fine-tuning these models on task-specific datasets with cross-modal alignment further enhances their ability for specific code retrieval scenarios. As a result, PLM-based methods have become the dominant approach in modern code retrieval, where their effectiveness lies in their ability to align semantic consistency between NL and PL.

Although these PLMs methods for code retrieval have shown promising performance, they still face two fundamental challenges.

First, there remains a significant semantic gap between human intent and machine execution logic, which stems from the inherent differences between natural language and programming languages. NL tends to be ambiguous, context-dependent, and rich in semantics, while PL is designed to be precise, unambiguous, and strictly structured to instruct computers. For example, human concepts such as “efficient” do not have direct equivalents in PL, as they require an understanding of computational trade-offs like time or space complexity, and API optimizations. While PLMs have advanced cross-modal alignment [14], they focus on surface-level semantic representations, making

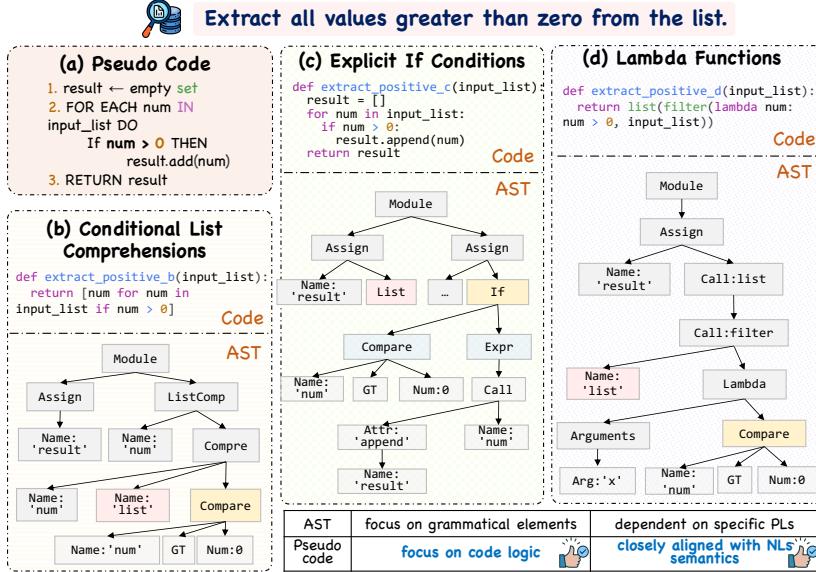


Fig. 1. Comparison between Pseudo-Code and AST, where AST focuses on the structure of the code, not only verbose but also hard to express the logic.

it difficult to bridge the inherent gap between semantics and logic. Many works [17, 36] attempt to address this issue by incorporating code structure information as additional supervision signals, particularly through Abstract Syntax Trees (ASTs), as illustrated in Figure 1. However, ASTs mainly capture the syntactic structure of code and introduce noise through their sheer volume and low-level granularity, lacking explicit modeling of the underlying program logic. Consequently, this gap remains unaddressed, and models struggle to fully align the intent behind user queries with the logical information encoded in programs.

Second, current PLMs often overlook the impact of code style diversity. In practice, the same functionality can be implemented in multiple code style implementations, as shown in Figure 1. For example, for the query requirement of “extracting all numbers greater than zero from the list,” there may exist several functionally equivalent code snippets: (b) list comprehensions that embed filtering logic, (c) explicit if conditions within loops, and (d) lambda functions combined with higher-order functions. This phenomenon means that a single natural language query can correspond to various code snippets that share the same underlying logic but differ significantly in their syntactic structure and implementation style. However, current PLMs struggle to align these logically equivalent yet stylistically diverse code snippets, leading to significant performance degradation and inconsistency in retrieval results. Moreover, as illustrated in Figure 1, variations of different styles manifest as substantial differences in their corresponding ASTs, causing models to focus excessively on superficial implementation details rather than core logic. This tendency not only increases the model’s sensitivity to code styles but also hinders its ability to capture deep code semantics. As a result, the models may misinterpret the semantic consistency between natural language queries and their possible code implementations, ultimately hindering the effectiveness of code retrieval models.

To address these challenges, we propose **PseudoBridge**, a novel code retrieval framework that introduces pseudo-code as an intermediate bridge to better align NL semantics with PL logic. Pseudo-code, situated between informal natural language and formal programming language,

serves as a semi-structured modality that effectively models both semantics and logic. Specifically, our framework consists of two stages: in the first stage, we employ advanced LLM (e.g., GPT-4o) to synthesize pseudo-code for each  $\langle NL, PL \rangle$  pair, and leverage pseudo-code as an intermediate representation to narrow the gap between natural language query and pseudo-code. In the second stage, we introduce a logic-invariant code style augmentation strategy, where we leverage LLM to synthesize multiple style-different but logically equivalent code implementations from the constructed pseudo-code. This augmentation further aligns the pseudo-code representations with various code styles. This strategy not only bridges the inherent gap between different language structures but also improves robustness to code style variation and generalization across unseen domains.

To evaluate the effectiveness of PseudoBridge, we fine-tune PseudoBridge with 10 different PLMs as the backbone and conduct comprehensive experiments across 6 mainstream programming languages. Results demonstrate that PseudoBridge greatly improves retrieval performance, confirming the efficacy of explicitly constructing logical alignment with pseudo-code. Notably, under zero-shot settings, PseudoBridge exhibits strong performance on domain-specific datasets (e.g., Solidity [7] and XLCOST [56]), highlighting its exceptional generalization capability and practical potential.

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

- (1) We propose a novel code retrieval framework, called PseudoBridge, to our knowledge, PseudoBridge is the first to introduce pseudo-code as an intermediate bridge, effectively aligning NL semantics with PL logic. It captures code snippets intent in a structured yet readable form, effectively bridging the gap between ambiguous queries and precise code logic. This design significantly improves the model's ability to generalize in diverse retrieval scenarios.
- (2) We develop a two-stage training framework that progressively aligns NL with PL via pseudo-code, incorporating both semantic consistency and logic invariance. In the first stage, we synthesize pseudo-code with an advanced LLM and align the NL with the pseudo-code representation. In the second stage, we introduce a code style augmentation strategy and align the pseudo-code with diverse code implementations. This approach directs the focus of the model towards the underlying semantics rather than surface syntax, strengthening the robustness to stylistic variation and improving cross-domain generalization.
- (3) We conduct extensive experiments by fine-tuning PseudoBridge on 10 different PLMs and evaluate its effectiveness across 6 PLs. Experimental results show that PseudoBridge significantly improves code retrieval across all tested models and languages, demonstrating strong zero-shot generalization on specialized datasets like Solidity and XLCOST without training. These results confirm the efficacy of pseudo-code as an intermediate representation and underscore the framework's practical potential for real-world multilingual retrieval.

## 2 Related Works

### 2.1 Code Retrieval Techniques

Code retrieval is a fundamental technology in intelligent software engineering, as effective retrieval systems greatly improve development productivity. Existing approaches can be broadly classified into three categories: information retrieval (IR) methods, deep learning (DL) models, and pre-trained language models (PLMs). Traditional IR-based approaches make code plain text, relying on lexical matching and keyword indexing techniques [22, 34, 35]. While these techniques are efficient, they are inherently limited by their reliance on lexical similarity, which prevents them from capturing the deeper semantic relationships between NL and PL. This lexical dependency restricts their ability to bridge the semantic gap and model complex code semantics. With advances in deep

learning, researchers have begun to exploit large-scale datasets to model the relationships between NL and PL. DL-based approaches employ neural network architectures to map NL queries and code snippets into a shared vector space, thereby facilitating semantic alignment. For example, DeepCS [15] utilizes recurrent neural networks to embed both queries and code within a common representation space, while CodeMatcher [30] enhances retrieval accuracy through semantics-aware query expansion mechanisms.

More recently, pre-trained language models (PLMs) have become the mainstream approach to code retrieval [11]. Leveraging Transformer architectures, these models acquire comprehensive programming knowledge from large-scale code repositories [6, 14, 55]. PLMs are pre-trained on corpora containing both code and NL queries to learn cross-modal semantic representations [2, 5, 17], and are subsequently fine-tuned on task-specific datasets to further align NL and PL semantics, thus boosting retrieval performance [23, 46, 47]. Despite these advancements, most existing methods still emphasize lexical-level semantic similarity, often overlooking the logical consistency of code. As a result, models may fail to fully capture the intent behind natural language queries and struggle to establish deep logical alignment between queries and code snippets. This limitation undermines their effectiveness in accurately retrieving target code in real-world scenarios. To address this challenge, our work introduces **pseudo-code** as an intermediate representation, serving as a bridge to enhance both semantic and logical alignment between NL and PL.

## 2.2 Data Synthesis for Code Intelligence

The synthesis of high-quality data has played a crucial role in advancing code intelligence tasks, particularly in improving the performance of LLMs for code generation and understanding. Early research explored the use of simulated coding environments to create large-scale datasets by generating and validating programming problems and solutions [21, 43]. For instance, techniques such as self-play [21] and Chain-of-Thought prompting [43] have enabled the automated creation of diverse and correct code samples. The development of instruction-following code LLMs has also greatly benefited from synthetic data. Models like WizardCoder [33], InstructCoder [27], UniCode [44], and Code Llama [39] have significantly expanded the scale and diversity of instruction-tuning datasets by generating synthetic programming instructions, problems, and corresponding solutions. Recent efforts have focused on increasing the diversity and realism of programming problems and solutions. For example, MagicCoder [48], InverseCoder [49], and Phi series [1, 16] have explored new ways for generating realistic and controllable coding instruction data, further broadening data coverage and diversity.

Despite these advancements, most existing data synthesis efforts focus on improving code generation and completion, rather than supporting code retrieval. In contrast, this paper targets data synthesis specifically for code retrieval scenarios, aiming to bridge the gap between natural language queries and code snippets.

## 3 Methods

### 3.1 Overview

The conceptual framework of PseudoBridge is depicted in Figure 2(c). In contrast to retrieval methods that rely on IR direct keyword matching (Figure 2(a)) or align NL and PL directly through PLMs (Figure 2(b)), PseudoBridge introduces a novel two-stage approach to more effectively align NL semantics with PL logic. Specifically, PseudoBridge first employs pseudo-code as an intermediate representation to effectively narrow the semantic gap between human intent, expressed through natural language queries, and machine executable logic, represented by code snippets. This design facilitates the alignment and learning of core logic and semantics between the two modalities.

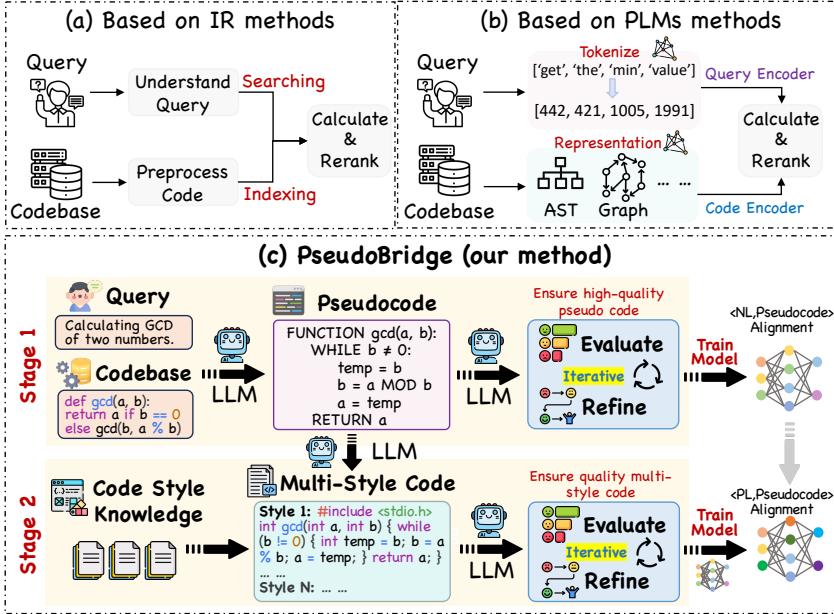


Fig. 2. Comparison of Existing Code Retrieval Frameworks. IR-based and PLM-based methods achieve superficial alignment between code and queries. Our proposed framework utilizes pseudo-code and logic-invariant code style enhancement to better align NL semantics with PL logic.

Subsequently, PseudoBridge constructs diverse code variants to simulate realistic scenarios in which the same query corresponds to different code implementations. This strategy enhances the model’s generalization capability and ultimately improves retrieval performance. The framework operates in two distinct stages:

- **Stage 1:** In this stage, PseudoBridge utilizes advanced LLMs to synthesize pseudo-code from each aligned  $\langle \text{NL}, \text{PL} \rangle$  pair. The generated pseudo-code captures the core algorithmic logic and control flow described in the code, while maintaining a human-readable abstraction consistent with the original NL query. This pseudo-code acts as a semantic intermediary, allowing the model to better disentangle high-level intent from low-level implementation details.

- **Stage 2:** Based on the synthesized pseudo-code, PseudoBridge generates multiple code implementations that are functionally equivalent but syntactically diverse. These variants are produced using LLMs with controlled prompts to preserve logic while introducing stylistic diversity. By aligning the pseudo-code representation with both the original and the augmented variants, the model learns to focus on invariant logical structures rather than incidental syntactic features. This strategy enhances the model’s robustness to code style variability and improves retrieval performance across heterogeneous codebases.

The following sections provide a detailed explanation of each component in PseudoBridge: pseudo-code generation (Section 3.2), multi-style code generation (Section 3.3), and model training (Section 3.4).

### 3.2 Pseudo-Code Generation

Due to the fundamental differences between NL and PL, bridging their semantic gap requires an effective intermediate representation. Although ASTs accurately capture code structure, they

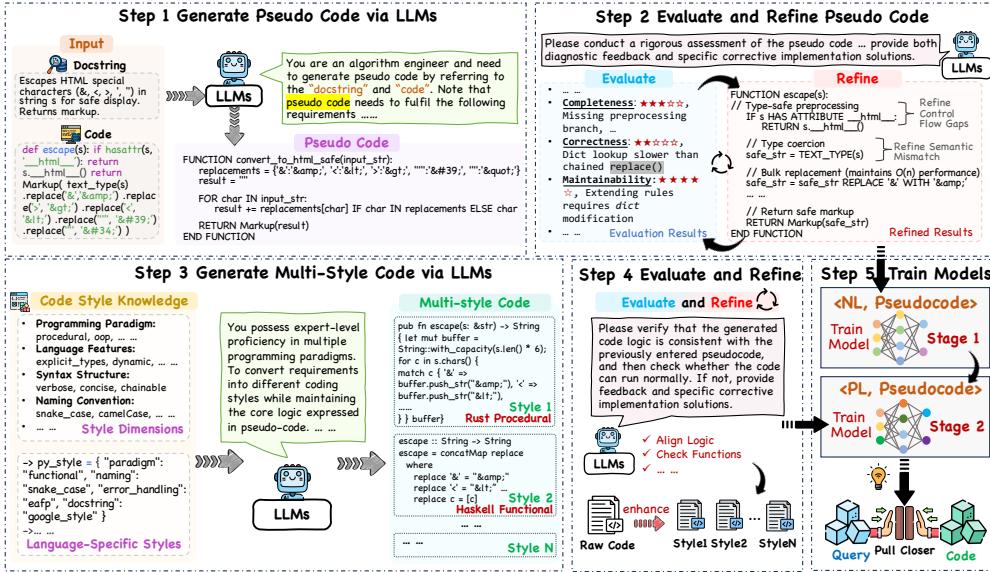


Fig. 3. The framework of PseudoBridge, which comprises three core components: pseudo-code generation, logic-invariant code style enhancement, and model training. Step 1: Synthesize initial pseudo-code using LLMs. Step 2: Assess the quality of the generated pseudo-code and refine it. Step 3: Leverage the refined high-quality pseudo-code to produce syntactically diverse yet functionally equivalent code variants. Step 4: Evaluate the augmented code for logical correctness and quality, performing necessary refinement. Step 5: Utilize the generated pseudo-code, diversified code variants, and corresponding query to jointly train the target model.

struggle to reflect underlying logic and still maintain a significant gap from NL semantics. To address this challenge, we propose using pseudo-code as an intermediate representation. Pseudo-code not only captures the logical flow of PL but also aligns more closely with NL expressions, thereby improving the effectiveness of code retrieval.

However, existing code corpus typically lack pseudo-code annotations [3, 24, 32]. To overcome this limitation, we utilize LLMs to synthesize pseudo-code from existing **<NL, PL>** pairs. In addition, we design a structured prompting framework to enhance both the quality and reliability of the generated pseudo-code. For instance, prompts such as “prioritize key steps” to filter out implementation details, apply “namespace standardization” to maintain conceptual consistency, and specify “standardised indentation” to improve logical readability. Details are provided in Figure 4.

To further validate and refine the quality of the generated pseudo-code, PseudoBridge incorporates an **Evaluate and Refine** mechanism. Inspired by CYCLE [12], this mechanism implements an iterative optimization process driven by LLMs, as illustrated in Step 2 of Figure 3. The process consists of three main steps:

**• Semantics and Logic Verification:** The generated pseudo-code is rigorously verified to ensure accurate alignment with both the NL description and the PL implementation. Semantics verification assesses whether the pseudo-code correctly reflects the intent and key concepts of the NL query, while logic verification checks consistency between the pseudo-code’s algorithmic flow and the actual behavior of the source code. This dual-focus process ensures the pseudo-code reliably bridges NL semantics and PL logic, with any discrepancies triggering immediate refinement.

**Prompt for Pseudo -code Generation**

```

### Task Description
You are an algorithm engineer and you need to generate pseudo-code by referring to a "docstring" and "code". The pseudo code needs to fulfil the following requirements.

### Pseudo Code Generation
1. Algorithm analysis: Analyse the "docstring" and "code", and clarify the query logic, code execution path and core algorithm.
2. Annotation rules: Use // single-line comments to mark the key operation intent and disable multi-line comments.
3. Variable naming: Variable naming should be consistent with "docstring" and "code". If there is an abbreviation, the full name should be added to improve readability.
4. Input/output: Define the inputs and outputs clearly, consistent with the "docstring" description. For complex inputs and outputs (e.g., nested structures, files, network requests, etc.), they should be abstracted to a high-level description.
5. Control structures: Conditions use IF/ELSIF/ELSE, and loops use 2-4 space indentation to declare variables and boundaries.
6. Function Architecture: Declare functions using FUNCTION and specify output using RETURN. Ensure that pseudo-code conforms to the logical hierarchy of the original code to avoid unnecessary process fragmentation.

### Additional Notes
1.Pseudo code should follow common pseudo code standards and avoid language-specific syntax.
2.Keep key algorithmic steps and ignore variable initialisation, language features and implementation details.

### Output Format
Based on the above requirements and the input "docstring" and "code", the generated pseudo code is:{}.

```

Fig. 4. The prompt template example of PseudoBridge generate pseudo-code by LLMs.

- **Multi-Dimensional Evaluation:** According to Knuth [25], pseudo-code serves as a bridge between humans and machines, acting both as a medium for human thought and as a blueprint for execution. Based on this, we evaluate and score the pseudo-code along five key dimensions: readability, correctness, completeness, conciseness, and maintainability.

- **Iterative Refinement:** If the average score falls below a predefined threshold, LLMs are used to diagnose specific issues and refine the pseudo-code accordingly. This evaluate-refine cycle is repeated until the pseudo-code meets the quality standards.

Notably, in our experiments, the number of quality evaluation and refinement iterations for pseudo-code does not exceed three.

### 3.3 Multi-Style Code Generation

Since current code retrieval technologies struggle to understand code logic [13, 29], their retrieval effectiveness decreases significantly when dealing with code written in different styles. In order to solve this problem, we propose a logic-invariant code style enhancement strategy. As shown in Figure 3 Step 3, our approach employs LLMs to generate multiple functionally equivalent yet stylistically diverse code variants from a given pseudo-code. This enables the construction of a mapping relationship between “*one pseudo-code → n implementations*”, which forces the model to identify the invariant logical semantics across different style variants.

To guarantee that these stylistically diverse code variants are strictly functionally equivalent, we employ a structured prompt as a constraint mechanism. Figure 5 presents the LLM to produce  $n$  code variants that are syntactically distinct but logically consistent with the given pseudo-code. Following code generation, the **Evaluate-Refine** process commences. LLMs verify the logical consistency between the generated code variants and the original pseudo-code, and then evaluate whether these code variants can run normally. Problematic code variants undergo a refinement process until they pass verification. It is important to note that all style variants generated during this process are limited to a maximum of three rounds of evaluation and refinement.

### 3.4 Model Training

This paper introduces PseudoBridge, a novel framework for code retrieval. The primary goal is to leverage LLMs to synthesize both pseudo-code and multi-style code variants, thereby enhancing

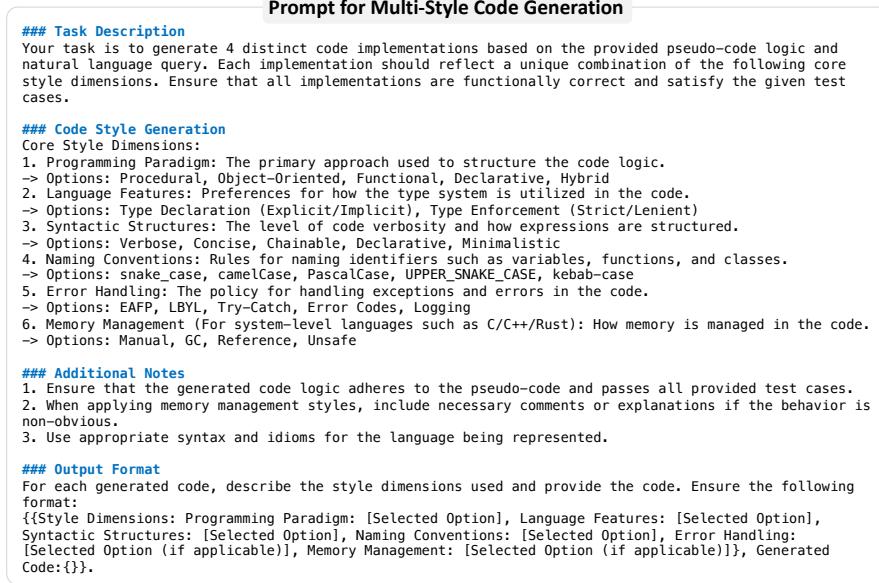


Fig. 5. The prompt template example of PseudoBridge generate multi-style code variants by LLMs.

the model's capacity to align NL semantics with PL logic. This improved alignment translates into better code retrieval performance. The training process of the framework comprises two stages, as illustrated in Figure 3 Step 5.

In the first stage, LLMs are employed to synthesize a pseudo-code dataset. The model learns representations for both NL  $Q$  and pseudo-code  $P$ , and is optimized to improve the alignment between them. The objective is to optimize the alignment between these two modalities by minimizing a similarity loss function  $\mathcal{L}_{\langle Q, P \rangle}$ . The specific computation formula is as follows:

$$\mathcal{L}_{\langle Q, P \rangle} = -\frac{1}{B} \sum_{i=1}^{|B|} \log \left( \frac{e^{\phi(q_i, p_i)}}{e^{\phi(q_i, p_i)} + \sum_{p_j \in N(p_i^-)} e^{\phi(q_i, p_j)}} \right) \quad (1)$$

where  $B$  denotes the batch size,  $q_i$  is the NL query of the  $i$ -th sample,  $p_i$  is the corresponding pseudo-code,  $N(p_i^-)$  contains pseudo-code negatives from other samples in the batch, and  $\phi$  denotes the cosine similarity function.

In the second stage, based on the synthesized pseudo-code, LLMs generate code variants that are functionally equivalent but stylistically diverse. The model is further trained on these diverse code samples along with their corresponding pseudo-code, which strengthens the alignment between PL code representations  $C$  and pseudo-code representations  $P$ . The loss function is formulated as:

$$\mathcal{L}_{\langle C, P \rangle} = -\frac{1}{B} \sum_{i=1}^{|B|} \log \left( \frac{\sum_{c_k^+ \in \mathcal{P}_i} e^{\phi(p_i, c_k^+)}}{\sum_{c_k^+ \in \mathcal{P}_i} e^{\phi(p_i, c_k^+)} + \sum_{c_j \in N(c_i^-)} e^{\phi(p_i, c_j)}} \right) \quad (2)$$

Here,  $\mathcal{P}_i$  denotes the set of positive code variants with stylistic diversity for the  $i$ -th sample,  $c_k^+$  is the  $k$ -th positive code variant in  $\mathcal{P}_i$ ,  $N(c_i^-)$  represents the set of irrelevant code samples in the batch, and  $c_j$  is the  $j$ -th irrelevant code sample.

## 4 Experiments

### 4.1 Experimental Setup

**4.1.1 Dataset.** Table 1 presents the overall statistics of the datasets used for training and evaluating our PseudoBridge. To comprehensively assess the code retrieval capability of PseudoBridge, we adopt CodeSearchNet [24] as the benchmark dataset.

During the training phase, in order to construct a high-quality training set, we follow the data filtering pipeline of CodeXGLUE [32] and perform the following steps sequentially: first, removing code examples that cannot be parsed into abstract syntax trees; second, discarding samples where the documentation contains fewer than 3 or more than 256 tokens; third, excluding examples with special tokens (e.g., ‘<img ...>’ or URLs) in the documentation; and finally, filtering out non-English documentation samples. After this process, we obtain training data covering six programming languages. For the testing phase, to simulate a real-world code retrieval scenario, we directly evaluate the model on the original CodeSearchNet [24] test set. Additionally, to validate the zero-shot generalization performance, we include Solidity [7] as well as C++ and C# datasets from XLCoST [56] for evaluation. The details of the dataset are as follows:

- **CodeSearchNet** [24] corpus is a large-scale function-level dataset comprising code and corresponding documentation collected from open-source GitHub projects. It encompasses six programming languages: Go, Java, JavaScript, PHP, Python, and Ruby. The dataset contains approximately 6 million code snippets, of which 2 million form valid (comment, code) pairs.
- **XLCoST** [56] is collected from the GeeksForGeeks platform and comprises a series of programming problems along with their corresponding solutions, covering seven commonly-used programming languages. To evaluate the zero-shot cross-lingual generalization capability of PseudoBridge, this study selects its C++ and C# subsets for experimentation.
- **Solidity** [7], a high-level programming language designed for smart contract development, is frequently adopted as an evaluation benchmark for cross-domain code retrieval tasks due to its domain-specific nature. We utilize this dataset to assess the zero-shot performance of PseudoBridge in retrieving code written in blockchain programming languages.

Table 1. Statistics of the datasets.

Language	For Training	For Testing	Language	For Training	For Testing
Python	5,914	22,176	PHP	1,000	28,391
Java	5,086	26,909	C++	-	899
JavaScript	5,000	6,483	C#	-	909
Go	1,000	14,291	Solidity	-	1,000
Ruby	1,000	2,279	—	—	—

**4.1.2 Metrics.** We adopt two widely recognized evaluation metrics, Mean Reciprocal Rank (MRR) and *Recall@k* (with k = 1) to assess the performance of code retrieval in the test set. These metrics are commonly used in code search literature [8, 13, 29] for measuring retrieval effectiveness.

**MRR** quantifies the quality of ranked retrieval results by evaluating the reciprocal rank of the first relevant item. The Reciprocal Rank (RR) for a single query and the Mean Reciprocal Rank (MRR) over a query set are defined as:

$$\text{RR} = \frac{1}{\text{rank}_i}, \quad \text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \quad (3)$$

where  $\text{rank}_i$  denotes the rank position (starting at 1) of the first correct result for the  $i$ -th query,  $Q$  represents the query set and  $|Q|$  is the total number of queries.

**Recall@k** measures the proportion of queries for which at least one relevant result appears within the  $top - k$  ranked items. Its query-level and macro-averaged formulations are:

$$\text{Recall}@k^{(i)} = \frac{m_k^i}{M_i} \quad (4)$$

$$\text{Recall}@k = \frac{1}{N} \sum_{i=1}^N \text{Recall}@k^{(i)} \quad (5)$$

where  $m_k^i$  is the number of relevant items in the top- $k$  results for the  $i$ -th query,  $M_i$  is the total relevant items for the  $i$ -th query,  $N$  is the number of queries.

Table 2. Performance of PseudoBridge on code retrieval across different PLMs at MRR (RQ1). Note: Note: Row 1 presents the baseline performance of various PLMs on the code retrieval task. Row 2 shows the results after fine-tuning with PseudoBridge.

Methods	Python	Java	JavaScript	Go	Ruby	PHP
<b>CDCS [8]</b>	0.0380	0.0168	0.0120	0.0188	0.0182	0.0046
+ <i>PseudoBridge</i>	<b>0.8226</b> ↑ <sup>78.5%</sup>	<b>0.5890</b> ↑ <sup>57.2%</sup>	<b>0.5627</b> ↑ <sup>55.1%</sup>	<b>0.6346</b> ↑ <sup>61.6%</sup>	<b>0.5985</b> ↑ <sup>58.0%</sup>	<b>0.5819</b> ↑ <sup>57.7%</sup>
<b>CodeT5 [47]</b>	0.0494	0.0670	0.0554	0.1106	0.1159	0.0473
+ <i>PseudoBridge</i>	<b>0.2011</b> ↑ <sup>15.2%</sup>	<b>0.1993</b> ↑ <sup>13.2%</sup>	<b>0.1852</b> ↑ <sup>13.0%</sup>	<b>0.1939</b> ↑ <sup>8.3%</sup>	<b>0.2676</b> ↑ <sup>15.2%</sup>	<b>0.1621</b> ↑ <sup>11.5%</sup>
<b>RoBERTa [31]</b>	0.0380	0.0168	0.0120	0.0188	0.0182	0.0046
+ <i>PseudoBridge</i>	<b>0.8191</b> ↑ <sup>78.1%</sup>	<b>0.5824</b> ↑ <sup>56.6%</sup>	<b>0.5559</b> ↑ <sup>54.4%</sup>	<b>0.6404</b> ↑ <sup>62.2%</sup>	<b>0.6082</b> ↑ <sup>59.0%</sup>	<b>0.5681</b> ↑ <sup>56.4%</sup>
<b>CodeBert [14]</b>	0.0052	0.0017	0.0034	0.0026	0.0068	0.0010
+ <i>PseudoBridge</i>	<b>0.8435</b> ↑ <sup>83.8%</sup>	<b>0.6455</b> ↑ <sup>64.4%</sup>	<b>0.6259</b> ↑ <sup>62.3%</sup>	<b>0.6862</b> ↑ <sup>68.4%</sup>	<b>0.6831</b> ↑ <sup>67.6%</sup>	<b>0.6146</b> ↑ <sup>61.4%</sup>
<b>GraphCodeBert [18]</b>	0.0719	0.0643	0.0556	0.0742	0.1085	0.0298
+ <i>PseudoBridge</i>	<b>0.8721</b> ↑ <sup>80.0%</sup>	<b>0.6781</b> ↑ <sup>61.4%</sup>	<b>0.6656</b> ↑ <sup>61.0%</sup>	<b>0.7092</b> ↑ <sup>63.5%</sup>	<b>0.7203</b> ↑ <sup>61.2%</sup>	<b>0.6411</b> ↑ <sup>61.1%</sup>
<b>DistilBert [41]</b>	0.0796	0.0521	0.0498	0.0481	0.1112	0.0347
+ <i>PseudoBridge</i>	<b>0.7848</b> ↑ <sup>70.5%</sup>	<b>0.4974</b> ↑ <sup>44.5%</sup>	<b>0.4895</b> ↑ <sup>44.0%</sup>	<b>0.5368</b> ↑ <sup>48.9%</sup>	<b>0.5907</b> ↑ <sup>48.0%</sup>	<b>0.5106</b> ↑ <sup>47.6%</sup>
<b>UniXcoder [17]</b>	0.6238	0.4970	0.4805	0.5119	0.5446	0.4439
+ <i>PseudoBridge</i>	<b>0.8360</b> ↑ <sup>21.2%</sup>	<b>0.6762</b> ↑ <sup>17.9%</sup>	<b>0.6308</b> ↑ <sup>15.0%</sup>	<b>0.6589</b> ↑ <sup>14.7%</sup>	<b>0.6704</b> ↑ <sup>12.6%</sup>	<b>0.6392</b> ↑ <sup>19.5%</sup>
<b>SentenceBert [37]</b>	0.6310	0.4998	0.5237	0.6363	0.6294	0.4867
+ <i>PseudoBridge</i>	<b>0.8507</b> ↑ <sup>22.0%</sup>	<b>0.6317</b> ↑ <sup>13.2%</sup>	<b>0.6116</b> ↑ <sup>8.8%</sup>	<b>0.6769</b> ↑ <sup>4.1%</sup>	<b>0.6949</b> ↑ <sup>6.6%</sup>	<b>0.6241</b> ↑ <sup>13.8%</sup>
<b>CoCoSoDa [42]</b>	0.7875	0.6336	0.6151	0.6747	0.6714	0.5568
+ <i>PseudoBridge</i>	<b>0.8370</b> ↑ <sup>5.0%</sup>	<b>0.6800</b> ↑ <sup>4.6%</sup>	<b>0.6460</b> ↑ <sup>3.1%</sup>	<b>0.6795</b> ↑ <sup>0.5%</sup>	<b>0.6886</b> ↑ <sup>1.7%</sup>	<b>0.6378</b> ↑ <sup>8.1%</sup>
<b>RAPID [13]</b>	0.7476	0.6144	0.5793	0.6131	0.6363	0.5467
+ <i>PseudoBridge</i>	<b>0.8168</b> ↑ <sup>7.0%</sup>	<b>0.6406</b> ↑ <sup>2.6%</sup>	<b>0.6066</b> ↑ <sup>2.7%</sup>	<b>0.6381</b> ↑ <sup>2.5%</sup>	<b>0.6498</b> ↑ <sup>1.4%</sup>	<b>0.5841</b> ↑ <sup>3.7%</sup>

## 4.2 Baselines

To evaluate the effectiveness of PseudoBridge, we fine-tune ten different pre-trained models as the backbone for the experiment.

- **SentenceBert [37]** enhances BERT with siamese networks to produce comparable semantic sentence embeddings, significantly improving similarity computation efficiency.
- **DistilBert [41]** is a lightweight BERT via distillation, maintaining performance while being smaller and faster.
- **RoBERTa [31]** is built on a multi-layer Transformer encoder and pre-trained with MLM, which is to predict the masked tokens. The pre-trained datasets are natural languages corpus and source code corpus, respectively.

Table 3. Performance of PseudoBridge on code retrieval across different PLMs at Recall@1 (RQ1). Note: Row 1 presents the baseline performance of various PLMs on the code retrieval task. Row 2 shows the results after fine-tuning with PseudoBridge.

Methods	Python	Java	JavaScript	Go	Ruby	PHP
<b>CDCS [8]</b>	0.0292	0.0103	0.0057	0.0096	0.0083	0.0023
+ <i>PseudoBridge</i>	<b>0.7709</b> ↑74.2%	<b>0.4985</b> ↑48.8%	<b>0.4796</b> ↑47.4%	<b>0.5532</b> ↑54.4%	<b>0.5046</b> ↑49.6%	<b>0.4961</b> ↑49.4%
<b>CodeT5 [47]</b>	0.0362	0.0417	0.0299	0.0829	0.0702	0.0297
+ <i>PseudoBridge</i>	<b>0.1586</b> ↑12.2%	<b>0.1403</b> ↑9.9%	<b>0.1248</b> ↑9.5%	<b>0.1456</b> ↑6.3%	<b>0.1887</b> ↑11.9%	<b>0.1157</b> ↑8.6%
<b>RoBERTa [31]</b>	0.0292	0.0103	0.0057	0.0096	0.0083	0.0023
+ <i>PseudoBridge</i>	<b>0.7666</b> ↑73.7%	<b>0.4893</b> ↑47.9%	<b>0.4700</b> ↑46.4%	<b>0.5579</b> ↑54.8%	<b>0.5186</b> ↑51.0%	<b>0.4808</b> ↑47.9%
<b>CodeBert [14]</b>	0.0029	0.0006	0.0011	0.0008	0.0013	0.0002
+ <i>PseudoBridge</i>	<b>0.7946</b> ↑79.2%	<b>0.5583</b> ↑55.8%	<b>0.5439</b> ↑54.3%	<b>0.6152</b> ↑61.4%	<b>0.5919</b> ↑59.1%	<b>0.5346</b> ↑53.4%
<b>GraphCodeBert [18]</b>	0.0487	0.0395	0.0295	0.0442	0.0693	0.0172
+ <i>PseudoBridge</i>	<b>0.8277</b> ↑77.9%	<b>0.5949</b> ↑55.5%	<b>0.5865</b> ↑55.7%	<b>0.6417</b> ↑59.8%	<b>0.6384</b> ↑56.9%	<b>0.5635</b> ↑54.6%
<b>DistilBert [41]</b>	0.0600	0.0321	0.0292	0.0293	0.0706	0.0223
+ <i>PseudoBridge</i>	<b>0.7257</b> ↑76.6%	<b>0.4019</b> ↑37.0%	<b>0.4001</b> ↑37.1%	<b>0.4478</b> ↑41.8%	<b>0.4958</b> ↑42.5%	<b>0.4205</b> ↑39.8%
<b>UniXcoder [17]</b>	0.5600	0.3976	0.3873	0.4199	0.4506	0.3531
+ <i>PseudoBridge</i>	<b>0.7886</b> ↑22.9%	<b>0.5976</b> ↑20.0%	<b>0.5521</b> ↑16.5%	<b>0.5805</b> ↑16.1%	<b>0.5814</b> ↑13.1%	<b>0.5600</b> ↑20.7%
<b>SentenceBert [37]</b>	0.5556	0.3896	0.4211	0.5503	0.5195	0.3905
+ <i>PseudoBridge</i>	<b>0.8028</b> ↑24.7%	<b>0.5411</b> ↑15.2%	<b>0.5275</b> ↑10.6%	<b>0.6006</b> ↑5.0%	<b>0.6042</b> ↑8.5%	<b>0.5414</b> ↑15.1%
<b>CoCoSoDa [42]</b>	0.7306	0.5394	0.5249	0.6028	0.5753	0.4637
+ <i>PseudoBridge</i>	<b>0.7873</b> ↑5.7%	<b>0.5998</b> ↑6.0%	<b>0.5666</b> ↑4.2%	<b>0.6104</b> ↑0.8%	<b>0.5954</b> ↑2.0%	<b>0.5568</b> ↑9.3%
<b>RAPID [13]</b>	0.6884	0.5205	0.4894	0.5287	0.5406	0.4590
+ <i>PseudoBridge</i>	<b>0.7644</b> ↑7.6%	<b>0.5516</b> ↑3.1%	<b>0.5198</b> ↑3.0%	<b>0.5572</b> ↑2.9%	<b>0.5546</b> ↑1.4%	<b>0.4968</b> ↑3.8%

- **CodeBERT [14]** is a pre-trained model for code and natural language, trained with replaced token detection and a hybrid approach combining paired data and unimodal data.
- **GraphCodeBERT [18]** is a pre-trained model for programming languages, specifically accounting for the inherent code structure, including dataflow.
- **CodeT5 [47]** is a sequence-to-sequence pre-trained model for code, trained with three identifier-aware tasks to identify or recover masked identifiers in source code.
- **CDCS [8]** is a meta-learning-based cross-domain code search approach that extends pre-trained models (e.g., CodeBERT) to enable effective knowledge transfer from general-purpose to domain-specific programming languages.
- **UniXcoder [17]** is a multimodal pre-trained model that takes code summaries and simplified ASTs as input, and incorporates language modeling, denoising, and contrastive learning for training.
- **CoCoSoDa [42]** enhances code search via contrastive learning with soft data augmentation and momentum-based negative samples, optimizing multimodal representation alignment.
- **RAPID [13]** is a zero-shot domain adaptation framework for code search that leverages pseudo-labeled synthetic data and a mixture sampling strategy for hard negatives, significantly improving cross-domain performance without labeled training data. The RAPID used in this paper is pre-trained using UniXcoder.

### 4.3 Implementation Details

We employ GPT-4o (*gpt-4o-2024-08-06*) for pseudo-code generation and synthesis of multi-style code variants, while utilizing DeepSeek-R1 [19] to assess pseudo-code quality and multi-style code variations. All experiments are conducted in a Python 3.10.16 and PyTorch 2.6.0 environment with CUDA 12.4 acceleration, running on 2 NVIDIA A100-SXM4 GPUs with 80GB memory.

Our training pipeline comprises two stages. In Stage 1, we bridge the gap between natural language and pseudo-code. In Stage 2, we enhance model generalization by aligning pseudo-code with stylistic variations of code. Specifically, each code sample is augmented into four functionally

equivalent stylistic variants, which are jointly trained with their corresponding pseudo-code representations. Hyperparameter settings include: batch size 48, training epochs 3, learning rate 5e-5 (with 10% warm-up), maximum sequence length 512 tokens, and temperature parameter 0.05.

#### 4.4 Evaluation

**4.4.1 RQ1: How effective is PseudoBridge in bridging the gap between natural language semantics and programming language logic?** To evaluate the performance of PseudoBridge, we conduct code retrieval experiments on six programming languages: Python, Java, JavaScript, Go, Ruby, and PHP. We compare the performance of the proposed method PseudoBridge with ten baseline PLMs models. After fine-tuning each model on CodeSearchNet, performance evaluation is conducted using MRR and Recall@1 metrics.

As shown in Tables 2 and 3, PseudoBridge consistently outperforms all baselines across six PLs. This advantage is particularly pronounced with lower-performing models, such as RoBERTa [31], CodeBert [14], GraphCodeBert [18], DistilBert [41]. While such models possess basic code comprehension capabilities, they fail to align logical and semantic representations between NL and PL. PseudoBridge addresses this obstacle through training with pseudo-code and multi-style code synthesized by LLMs, thereby improving understanding of semantic and logic relationships and significantly enhancing code retrieval performance.

When applied to high-performance models [13, 17, 37, 42], PseudoBridge yields limited improvements in retrieval capabilities. This is largely because these models already exhibit advanced semantic understanding. For example, UniXcoder [17] inherently integrates AST and data-flow information with additional supervised signals for alignment. Similarly, SentenceBERT [37] refines the semantic embedding space using a siamese network architecture. Moreover, RAPID [13] and CoCoSoDa [42] employ synthetic data and momentum contrastive learning to enable domain adaptation. These models have already utilized different technologies and large amounts of data during the pre-training stage to narrow the semantic gap between NL and PL, leaving very limited room for improvement during the fine-tuning stage. Nevertheless, PseudoBridge is still able to further enhance the performance of these models, although the improvement is less pronounced compared to that observed with lower-performing models.

**Conclusion 1:** PseudoBridge effectively aligns NL semantics and PL logic through pseudo-code and a logic-invariant code style enhancement strategy, thereby improving code retrieval performance. The poorer the model performance, the more significant the improvement provided by PseudoBridge.

**4.4.2 RQ2: How does the proportion of pseudo-code affect PseudoBridge’s alignment between NL semantics and PL logic?** To systematically investigate the impact of different proportions of pseudo-code on the performance of PseudoBridge code retrieval, we set up a controlled experiment using five training sets with varying pseudo-code ratios (20%, 40%, 60%, 80%, 100%). We train and assess models under the same conditions. For capability analysis, we select three representative baseline models from each category: lower-performance models (CodeBERT, GraphCodeBERT, DistilBERT) and higher-performance models (UniXcoder, SentenceBERT, CoCoSoDa). All models are tested on Python and Ruby (from CodeSearchNet dataset) datasets to examine: (1) performance impact of pseudo-code ratio variation; (2) differential sensitivity to pseudo-code across model capabilities.

The experimental results are presented in Tables 4 and 5. It indicate that low-performance models require more than 60% of pseudo-code training data to achieve optimal results. Specifically, GraphCodeBERT [41] and DistilBERT [41] achieve peak performance at 100% pseudo-code ratio on

Table 4. Impact of pseudo code proportions on code retrieval performance of PLMs at MRR (RQ2). Note: Python and Ruby CodeSearchNet [24]. Bold numbers indicate the best performance metrics.

Language	Methods	20%	40%	60%	80%	100%
Python	CodeBert [14]	0.8386	0.8377	0.8359	<b>0.8439</b>	0.8435
	GraphCodeBert [18]	0.8619	0.8681	0.8708	0.8570	<b>0.8721</b>
	DistilBert [41]	0.7599	0.7772	0.7803	0.7818	<b>0.7848</b>
	UniXcoder [17]	0.8352	0.8308	0.8328	<b>0.8369</b>	0.8360
	SentenceBert [37]	0.8481	<b>0.8534</b>	0.8445	0.8523	0.8507
	CoCoSoDa [42]	0.8350	<b>0.8381</b>	0.8360	0.8377	0.8370
Ruby	CodeBert [14]	0.6779	0.6844	<b>0.6845</b>	0.6826	0.6831
	GraphCodeBert [18]	0.8619	0.8681	0.8708	0.8570	<b>0.8721</b>
	DistilBert [41]	0.7599	0.7772	0.7803	0.7818	<b>0.7848</b>
	UniXcoder [17]	0.8352	0.8308	0.8328	<b>0.8369</b>	0.8360
	SentenceBert [37]	0.8481	<b>0.8534</b>	0.8445	0.8523	0.8507
	CoCoSoDa [42]	0.8350	<b>0.8381</b>	0.8360	0.8377	0.8370

Table 5. Impact of pseudo code proportions on code retrieval performance of PLMs at Recall@1 (RQ2). Note: Python and Ruby CodeSearchNet [24]. Bold numbers indicate the best performance metrics.

Language	Methods	20%	40%	60%	80%	100%
Python	CodeBert [14]	0.7901	0.7865	0.7858	<b>0.7960</b>	0.7946
	GraphCodeBert [18]	0.8156	0.8244	0.8261	0.8108	<b>0.8277</b>
	DistilBert [41]	0.6990	0.7182	0.7217	0.7236	<b>0.7257</b>
	UniXcoder [17]	0.7869	0.7807	0.7832	<b>0.7887</b>	0.7886
	SentenceBert [37]	0.8004	<b>0.8061</b>	0.7953	0.8055	0.8028
	CoCoSoDa [42]	0.7858	<b>0.7903</b>	0.7868	0.7890	0.7873
Ruby	CodeBert [14]	0.5858	0.5946	<b>0.5950</b>	0.5915	0.5919
	GraphCodeBert [18]	0.8156	0.8244	0.8261	0.8108	<b>0.8277</b>
	DistilBert [41]	0.6990	0.7182	0.7217	0.7236	<b>0.7257</b>
	UniXcoder [17]	0.7869	0.7807	0.7832	<b>0.7887</b>	0.7886
	SentenceBert [37]	0.8004	<b>0.8061</b>	0.7953	0.8055	0.8028
	CoCoSoDa [42]	0.7858	<b>0.7903</b>	0.7868	0.7890	0.7873

both Python and Ruby tasks, while CodeBERT [14] reaches optimal performance at 80% pseudo-code for Python but requires 100% for Ruby. This discrepancy can be attributed to the distinct characteristics of these models. GraphCodeBERT [18], while capable of modeling variable-level data flows, struggles to capture algorithm-level semantic intent effectively. DistilBERT [41], as a pure text-based model, compensates for its lack of cross-modal understanding by explicitly learning the logical structure of both NL and PL. Consequently, both models heavily rely on a high proportion of pseudo-code to enhance code retrieval accuracy. On the other hand, CodeBERT [14], as a bimodal model trained only on code and natural language queries and lacking structured understanding capabilities, exhibits sufficient performance only with a moderate amount of pseudo-code.

For high-performance models, SentenceBERT [37] and CoCoSoDa [42] achieve optimal performance with only 40% of pseudo-code training data in Python and Ruby tasks. This is because these two models already have cross-modal alignment capabilities, so pseudo-code data has a limited impact on further improving their logical alignment capabilities. UniXcoder [17] performs best at a 60% pseudo-code ratio. This result can be attributed to its use of a prefix adapter to control model behavior, combined with cross-modal information such as ASTs to enhance code representation capabilities. Despite this, such models can still be further enhanced by providing a small amount of pseudo-code.

**Conclusion 2:** Different amounts of pseudo-code have different effects on model enhancement for models with different performance levels. Specifically, low-performance models require a higher proportion of pseudo-code, while high-performance models can achieve optimal alignment between NL semantics and PL logic even with a lower proportion.

Table 6. Impact of code style enhancement quantity on code retrieval performance of PLMs (RQ3). Note: Python and Ruby CodeSearchNet [24]. *Bold numbers indicate the best performance metrics.*

Language	Methods	1		2		3		4	
		MRR	Recall@1	MRR	Recall@1	MRR	Recall@1	MRR	Recall@1
Python	CodeBert [14]	0.8377	0.7865	0.8359	0.7858	<b>0.8439</b>	<b>0.7960</b>	0.8435	0.7946
	GraphCodeBert [18]	0.8681	0.8244	0.8708	0.8261	0.8719	0.8270	<b>0.8721</b>	<b>0.8277</b>
	DistilBert [41]	0.7772	0.7182	0.7803	0.7217	0.7818	0.7236	<b>0.7848</b>	<b>0.7257</b>
	UniXcoder [17]	0.8308	0.7807	0.8328	0.7832	<b>0.8369</b>	<b>0.7887</b>	0.8360	0.7886
	SentenceBert [37]	<b>0.8534</b>	<b>0.8061</b>	0.8445	0.7953	0.8523	0.8055	0.8507	0.8028
	CoCoSoDa [42]	<b>0.8381</b>	<b>0.7903</b>	0.8360	0.7868	0.8377	0.7890	0.8370	0.7873
Ruby	CodeBert [14]	0.6844	0.5946	<b>0.6845</b>	<b>0.5950</b>	0.6826	0.5915	0.6831	0.5919
	GraphCodeBert [18]	<b>0.7303</b>	<b>0.6477</b>	0.7291	0.6463	0.7195	0.6341	0.7203	0.6384
	DistilBert [41]	0.5811	0.4822	0.5878	0.4928	0.5887	0.4963	<b>0.5907</b>	<b>0.4958</b>
	UniXcoder [17]	0.6698	0.5792	0.6652	0.5753	<b>0.6749</b>	<b>0.5875</b>	0.6704	0.5814
	SentenceBert [37]	0.6937	0.6055	0.6903	0.5989	0.6940	0.6025	<b>0.6949</b>	<b>0.6042</b>
	CoCoSoDa [42]	0.6888	0.5954	<b>0.6896</b>	<b>0.5994</b>	0.6828	0.5862	0.6886	0.5954

**4.4.3 RQ3: How does the number of code styles generated based on logic-invariant strategy impact PseudoBridge’s ability to align NL semantics and PL logic?** To assess the impact of code style diversity on PLMs approaches retrieval performance, we test how PseudoBridge performed with different numbers of code style variants (1, 2, 3, 4). This experiment focuses on understanding whether increasing stylistic diversity in logically equivalent code enhances the model’s ability to bridge the semantic and logic gap between NL and PL. To ensure a comprehensive evaluation, we select three representative baseline models from both the high/low-performing baselines, and evaluate them on the Python and Ruby datasets.

As shown in Table 6, there are marked differences in the number of code style improvements required to achieve optimal retrieval results across different models and programming languages. For **Python** language, low-performance models exhibit continuous improvement with increasing code style variants, indicating multi-style training effectively fills the gap between NL semantics and PL logic. In contrast, high-performance models such as SentenceBERT [37] and CoCoSoDa [42] achieve peak performance with only one code variant. For **Ruby** language, there is no clear pattern in the number of code variants corresponding to low-performance and high-performance PLMs methods when attaining optimal retrieval. This is likely due to the limited number of Ruby training datasets, resulting in a limited performance of the foundation model and thus an unstable requirement for the number of style samples.

**Conclusion 3:** The code style variants in PseudoBridge enhance the model’s ability to align natural language semantics with programming language logic by exposing it to diverse code expressions. For widely used languages like Python, stronger models require fewer variants, as their pretraining already captures much of the language’s structural and idiomatic patterns.

**4.4.4 RQ4: How does PseudoBridge facilitate zero-shot knowledge transfer across domains and programming languages?** To rigorously assess the zero-shot knowledge transfer capability of PseudoBridge across previously unseen programming languages and application

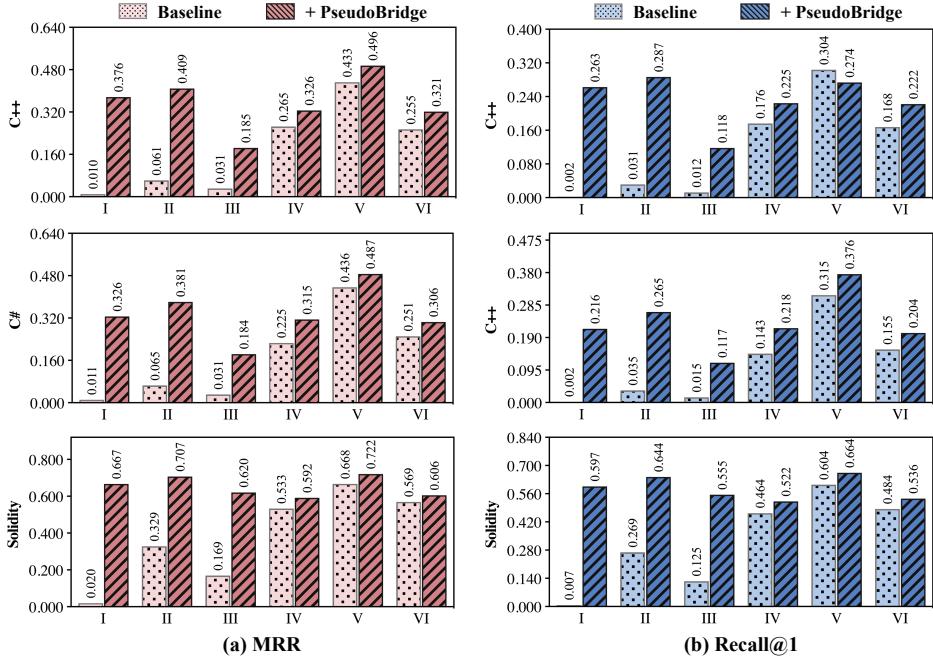


Fig. 6. Zero-shot learning capability comparison across domains and tasks for baseline models with/without PseudoBridge fine-tuning (RQ4). Note: I: CodeBert; II: GraphCodeBert; III: DistilBert; IV: UniXcoder; V: SentencenBert; VI: CoCoSoDa.

domains, we select three representative datasets, each aligned with a distinct and practically relevant scenario. Specifically, we evaluate performance on C++, which is commonly associated with hardware resource management; C#, which is frequently used in software testing framework development and adaptability; and Solidity, a domain-specific language integral to smart contract implementation and security. These selections ensure coverage across general-purpose, object-oriented, and domain-specific programming paradigms.

We conduct zero-shot evaluations on PLMs both before and after PseudoBridge fine-tuning. As shown in Figure 6, the models fine-tuned using PseudoBridge demonstrate notable improvements in code retrieval accuracy, even when applied to programming languages and domains not encountered during training. This indicates a substantial enhancement in the generalizability of the learned representations. Notably, retrieval performance improvements are particularly pronounced on PLMs with weaker foundational capabilities. For instance, CodeBERT achieves a 37.5% enhancement in C++ retrieval performance and a 31.5% improvement in C#. The gains for Solidity are even more significant, with a relative improvement reaching 64.7%. These empirical results underscore the effectiveness of PseudoBridge in bridging the semantic gap between NL and PL. By improving the alignment between linguistic and programmatic representations, our method enhances the robustness and adaptability of PLMs in diverse and previously unseen environments. This capability is essential for enabling practical and scalable solutions to real-world code search and retrieval tasks in heterogeneous software engineering contexts.

**Conclusion 4:** PseudoBridge exhibits strong zero-shot transfer capabilities across both programming languages and application domains. This strong generalization performance suggests that the core mechanism of aligning NL semantics with PL logic plays a pivotal role in enabling effective cross-domain knowledge transfer.

## 4.5 Ablation Studies

**4.5.1 Ablation Study on PseudoBridge Components.** To better understand the contribution of each component in PseudoBridge to code retrieval performance, we conduct comprehensive ablation studies on six baseline models tested by Python and Ruby. The results are presented in Figure 7. We set three ablation experiments, each with the following objectives:

- (1) **w/o Stage 1:** In this setting, we fine-tune the model using only multi-style code, completely removing the pseudo-code stage. This experiment evaluates whether the model can learn meaningful code representations directly from natural language queries without the support of pseudo-code, thereby assessing the role of pseudo-code as a semantic bridge between queries and code.
- (2) **w/o Stage 2:** This setting trains the model only on natural language and pseudo-code pairs, without introducing real code or multi-style code. It is designed to examine whether aligning queries with pseudo-code alone, without further grounding pseudo-code to real code, leads to a loss of semantic continuity and results in weaker retrieval performance.
- (3) **w/o Code Style:** In this configuration, the model is trained with pseudo-code and original code snippets but without multi-style code. This setting aims to assess the impact of multi-style code on the generalization ability of models, especially in scenarios where code exhibits diverse stylistic variations.

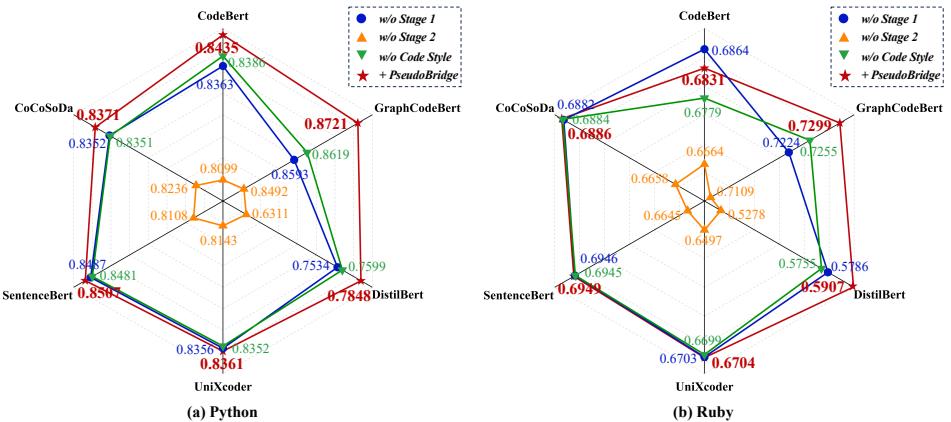


Fig. 7. Ablation study on the component analysis based on Python and Ruby. Note: (1) **w/o Stage 1:** removes <NL,Pseudo-code> pairs and directly trains <Pseudo-code, Multi-style code variants>; (2) **w/o Stage 2** uses <NL,Pseudo-code> pairs and <Pseudo-code, Original code> training, without multi-style code variants; (3) **w/o Code Style** trains on <NL,Pseudo-code>, then on <Pseudo-code, Original code>, without style-enhanced variants; (4) **+ PseudoBridge** employs training with pseudo-code to natural language pairs, then on pseudo-code to style-enhanced code variants.

The results show that under the **w/o Stage 1** setting, all models perform significantly worse than the full +PseudoBridge configuration. This indicates that skipping **Stage 1** prevents the model from effectively learning the semantic alignment between natural language queries and pseudo-code,

which in turn harms retrieval accuracy. These observations confirm the importance of pseudo-code as an intermediate representation for bridging query and code semantics. Under the **w/o Stage 2** setting, model performance is consistently the lowest across both datasets. This is because the model does not gain exposure to real code with diverse styles, limiting its ability to learn deeper structural and semantic patterns from pseudo-code. Without **Stage 2**, the model remains at a superficial semantic level and struggles to generalize to real-world code retrieval tasks.

In the **w/o Stage Code Style** setting, all models still perform worse than the full PseudoBridge configuration, which further demonstrates the importance of multi-style code in improving robustness and generalization. Interestingly, on the Python dataset, this setting outperforms **w/o Stage 1**, while on the Ruby dataset, it performs slightly worse. This difference suggests that the structural characteristics of different programming languages influence the relative importance of the two stages. Based on these results, we further analyze the complementary nature of the two-stage design in PseudoBridge. **Stage 1**, which maps natural language to pseudo-code, builds the foundation for understanding query intent. This stage is especially critical for languages with regular syntax and clear structure, such as Python. **Stage 2**, which connects pseudo-code to multi-style code, enhances the model's ability to generalize across diverse coding styles. This stage is particularly effective for languages with flexible syntax and multiple programming paradigms, such as Ruby. Together, the two stages enable the model to both accurately interpret queries and adapt to stylistic variations in code, leading to significantly improved retrieval performance.

**Conclusion 5:** PseudoBridge introduces pseudo-code and code style enhancements based on logical invariant strategy, both of which are crucial for improving the code retrieval performance of baseline models. This two-stage design effectively bridges logic and semantic alignment between NL and PL, enabling the model to capture functional intent through pseudo-code alignment before training to enhance its generalization capabilities.

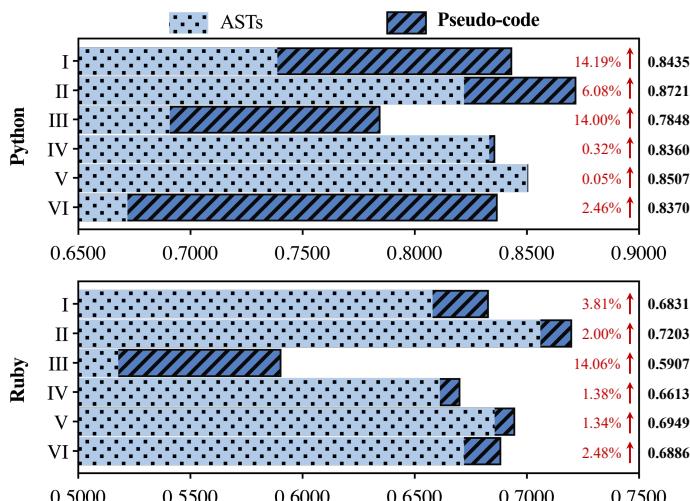


Fig. 8. Ablation study on Pseudo-code vs. ASTs intermediate representations for code retrieval. Note: I: CodeBert; II: GraphCodeBert; III: DistilBert; IV: UniXcoder; V: SentenceBert; VI: CoCoSoDa.

**4.5.2 Representation Impact: Pseudo-code vs. ASTs.** To investigate the impact of different intermediate representations on retrieval performance, we carry out ablation experiments using ASTs as the baseline method. As shown in Figure 8, we compare the retrieval effectiveness of AST-based representations with our proposed PseudoBridge approach across two programming languages: Python and Ruby. Experimental results show that models fine-tuned on ASTs generated by large language models and trained using a two-stage process consistently perform worse in retrieval tasks than those trained with pseudo-code under the PseudoBridge framework. Although ASTs accurately capture the syntactic hierarchy and structural dependencies within code, their rigid structural nature tends to bias the model toward syntactic composition, making it difficult to learn higher-level semantic alignments and functional intent from natural language queries. In contrast, pseudo-code abstracts away language-specific syntactic details, enabling the model to better represent and align the semantic content of natural language with programming logic. This abstraction helps the model overlook superficial implementation differences and instead focus on cross-modal alignment between intent expressed in natural language and the corresponding program behavior, thereby enhancing retrieval performance.

**Conclusion 6:** Pseudo-code, in contrast to ASTs, provides a more effective bridge between NL semantics and PL logic in code retrieval. By abstracting away language-specific syntactic structures and emphasizing high-level functional intent, it enables more accurate semantic and logic alignment between queries and code.

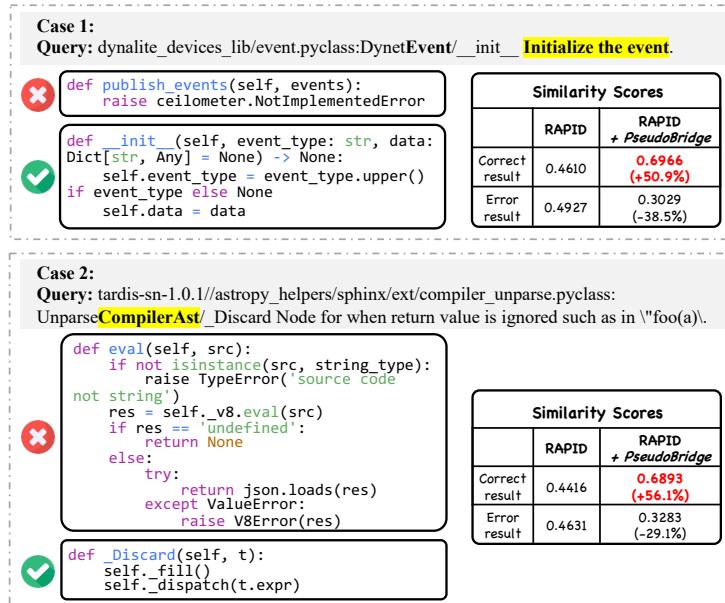


Fig. 9. Comparative analysis of code retrieval performance between base and fine-tuned models.

#### 4.6 Qualitative Analysis

To empirically validate the effectiveness of PseudoBridge, we execute qualitative case studies on Python code retrieval in two representative scenarios: *Event Initialization* and *AST Parse*. As shown in Figure 9, RAPID without fine-tuning tends to retrieve code that is syntactically similar but

functionally incorrect, with similarity scores of 0.4927 and 0.4631. In contrast, the similarity to the correct implementations is lower, at 0.4610 and 0.4416. After fine-tuning with PseudoBridge, the similarity to correct code increases substantially to **0.6966** ( $\uparrow 50.9\%$ ) and **0.6893** ( $\uparrow 56.1\%$ ) while incorrect matches decrease to **0.3029** ( $\downarrow 38.5\%$ ) and **0.3283** ( $\downarrow 29.1\%$ ). These results demonstrate that PseudoBridge enables effective alignment between NL semantics and PL logic, rather than achieving only superficial semantic matching. Consequently, it improves code retrieval accuracy in practical scenarios.

To further validate the effectiveness of PseudoBridge in improving code retrieval performance, we computed the similarity scores between natural language queries and code snippets using the RAPID model fine-tuned with PseudoBridge. The distribution of these scores is visualized in Figure 10. In this visualization, a higher degree of overlap between the centers of “**● Docstring**” and “**+** Code” embeddings indicates stronger semantic alignment, which corresponds to more accurate retrieval results. The experimental results demonstrate that PseudoBridge significantly reduces the embedding distance between natural language queries and their corresponding relevant code snippets. This leads to a more compact distribution in the embedding space, providing an intuitive reflection of the model’s enhanced semantic alignment capability. Examples marked with green lines in the figure highlight cases where there is a large discrepancy in similarity between queries and code snippets. After fine-tuning with PseudoBridge, the number of such retrieval errors is notably reduced. These findings indicate that PseudoBridge, by incorporating pseudo-code generation and semantics-preserving contrastive learning, effectively enhances the alignment between natural language and code semantics. The method captures the true intent of natural language queries and maps it to the underlying program logic of code snippets. Consequently, it achieves a deep integration of natural language semantics with program semantics, thereby significantly improving the effectiveness and robustness of code retrieval systems.

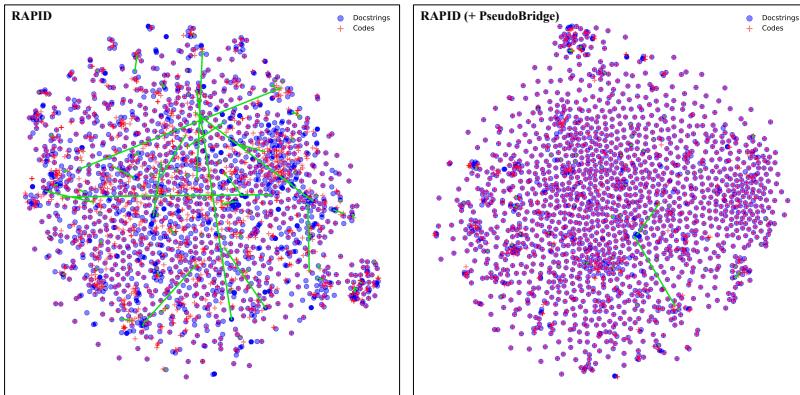


Fig. 10. T-SNE visualization of docstrings (NL) and code (PL) embeddings for base and fine-tuned models. Green lines connect docstrings to matching code ( $t\text{-SNE}$  distance  $> 0.1$ ).

## 5 Discussion

### 5.1 Cost Analysis

To quantify the economic cost of synthesizing code data using large language models, this study estimates the expense based on API call fees for GPT-4o. Empirical results indicate that the average cost of generating a single pseudo-code sample is **\$0.0076**, while the cost of producing a multi-style

code set corresponding to one pseudo-code sample amounts to **\$0.037**. These findings demonstrate that the PseudoBridge framework offers considerable cost efficiency, thereby providing an economically viable technical pathway for large-scale instruction generation tasks.

## 5.2 Threats to Validity

Although PseudoBridge effectively aligns natural language semantics with programming language logic to enhance code retrieval efficiency, the following potential challenges remain to be addressed in future work:

**5.2.1 Programming Languages.** Our experiments cover a limited set of programming languages. Due to the significant human and computational resources required for a comprehensive evaluation across all languages, this study select six representative, commonly-used languages based on existing datasets for validation. Experimental results indicate noticeable performance variations across different languages, suggesting that language-specific characteristics, such as syntactic structure and programming paradigms, may influence the method’s effectiveness. In the future, we will systematically incorporate a wider variety of languages to further validate the universality and robustness of PseudoBridge across diverse language ecosystems.

**5.2.2 Pre-trained Models.** To evaluate the improvement in code retrieval performance facilitated by PseudoBridge for pre-trained language models (PLMs), we conduct empirical analyses on ten different PLMs, including general-purpose natural language models and models specialized for code. However, current evaluations have yet to encompass certain emerging or specialized PLMs, such as architectures designed specifically for zero-shot code understanding (e.g., CodeBridge [29]). These models may exhibit differences in parameter initialization, training objectives, or structural design. The transferability of the PseudoBridge approach and its adaptation mechanisms on such models warrant further exploration to comprehensively gauge the framework’s broad applicability.

**5.2.3 Large Language Models.** The proposed PseudoBridge framework leverages large language models to synthesize pseudocode and code in multiple styles, aiming to enhance the model’s ability in natural language understanding and programming language reasoning. Currently, all data synthesis is conducted using the GPT-4o model. However, different large language models may produce variations in synthesis quality. This study does not include comparison experiments with other models. Future work plans to incorporate additional advanced models, such as Deepseekcoder [20], to further evaluate and improve the generality and effectiveness of the PseudoBridge framework.

**5.2.4 Inaccuracy Accumulation Issues.** PseudoBridge effectively enhances the performance of existing code retrieval models in data-limited training scenarios by introducing pseudo-code and code style enhancement techniques, with particularly notable gains on large-scale retrieval tasks. While LLMs are fundamentally probabilistic generative models and lack a definitive mechanism to correct generation errors. PseudoBridge addresses this challenge by employing an LLM-based automatic filtering mechanism after each generation step. This mechanism scores generated content, such as the logical correctness of pseudo-code and the alignment between multi-style code and pseudo-code, and removes low-quality outputs. Despite this, the filtering process still depends on the discriminative capabilities of LLMs, which cannot completely prevent errors from propagating through the generation process. In future work, we plan to explore the integration of reinforcement learning methods such as GRPO [19] and DAPO [53] with LLMs for code retrieval, aiming to guide the generation of more reliable intermediate representations through the development of more robust and efficient reward functions.

## 6 Conclusion

In this paper, we propose PseudoBridge, a novel code retrieval framework leveraging pseudo-code as an intermediate semi-structured modality, enabling more precise alignment between NL semantics and PL logic. In addition, we design a logic-invariant code style augmentation strategy, using LLMs to generate code implementations that are stylistically diverse yet logically equivalent and aligned with the same pseudo-code. We perform extensive evaluations across ten baselines and six mainstream programming languages, confirming substantial improvements in retrieval accuracy with PseudoBridge enhancement. Results show that PseudoBridge consistently outperforms existing methods, particularly in challenging zero-shot domain transfer scenarios such as Solidity and XLCoST datasets. Future research will conduct tests on more programming languages and compare the impact of different advanced LLMs on the framework.

## References

- [1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Moján Javaheripi, Piero Kauffmann, et al. 2024. Phi-4 technical report. *arXiv preprint arXiv:2412.08905* (2024).
- [2] Shushan Arakelyan, Anna Hakverdyan, Miltiadis Allamanis, Luis Garcia, Christophe Hauser, and Xiang Ren. 2022. NS3: Neuro-symbolic semantic code search. *Advances in Neural Information Processing Systems* 35 (2022), 10476–10491.
- [3] Mehdi Bahrami, NC Shrikanth, Shade Ruangwan, Lei Liu, Yuji Mizobuchi, Masahiro Fukuyori, Wei-Peng Chen, Kazuki Munakata, and Tim Menzies. 2021. Pytorrent: A python library corpus for large-scale language models. *arXiv preprint arXiv:2110.01710* (2021).
- [4] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. 2014. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming* 79 (2014), 241–259.
- [5] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.
- [6] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. 2021. Automated query reformulation for efficient search based on query logs from stack overflow. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1273–1285.
- [7] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-domain deep code search with meta learning. In *Proceedings of the 44th international conference on software engineering*. 487–498.
- [8] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-domain deep code search with meta learning. In *Proceedings of the 44th international conference on software engineering*. 487–498.
- [9] Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
- [10] Yi Cheng and Li Kuang. 2022. CSRS: code search with relevance matching and semantic matching. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 533–542.
- [11] Luca Di Grazia and Michael Pradel. 2023. Code search: A survey of techniques for finding code. *Comput. Surveys* 55, 11 (2023), 1–31.
- [12] Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. 2024. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 392–418.
- [13] Guodong Fan, Shizhan Chen, Cuiyun Gao, Jianmao Xiao, Tao Zhang, and Zhiyong Feng. 2024. Rapid: Zero-shot domain adaptation for code search with pre-trained models. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–35.
- [14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Dixin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
- [15] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th international conference on software engineering*. 933–944.
- [16] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Moján Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644* (2023).
- [17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

- [18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
- [19] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* 645, 8081 (2025), 633–638.
- [20] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu 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).
- [21] Patrick Halupczok, Matthew Bowers, and Adam Tauman Kalai. 2022. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502* (2022).
- [22] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 524–527.
- [23] Fan Hu, Yanlin Wang, Lun Du, Xirong Li, Hongyu Zhang, Shi Han, and Dongmei Zhang. 2023. Revisiting code search in a two-stage paradigm. In *Proceedings of the sixteenth ACM international conference on Web search and data mining*. 994–1002.
- [24] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
- [25] Donald Ervin Knuth. 1984. Literate programming. *The computer journal* 27, 2 (1984), 97–111.
- [26] Jia Li, Chongyang Tao, Jia Li, Ge Li, Zhi Jin, Huangzhao Zhang, Zheng Fang, and Fang Liu. 2023. Large language model-aware in-context learning for code generation. *ACM Transactions on Software Engineering and Methodology* (2023).
- [27] Kaixin Li, Qisheng Hu, James Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Michael Shieh, and Junxian He. 2024. InstructCoder: Instruction Tuning Large Language Models for Code Editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*. 50–70.
- [28] Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. Coderetriever: A large scale contrastive pre-training method for code search. In *Proceedings of the 2022 conference on empirical methods in natural language processing*. 2898–2910.
- [29] Keyu Liang, Zhongxin Liu, Chao Liu, Zhiyuan Wan, David Lo, and Xiaohu Yang. 2025. Zero-Shot Cross-Domain Code Search without Fine-Tuning. *arXiv preprint arXiv:2504.07740* (2025).
- [30] Chao Liu, Xin Xia, David Lo, Zhiwei Liu, Ahmed E Hassan, and Shanping Li. 2021. Codematcher: Searching code based on sequential semantics of important query words. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–37.
- [31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).
- [32] 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).
- [33] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *ICLR*.
- [34] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.
- [35] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. 111–120.
- [36] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1066–1082.
- [37] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).
- [38] 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.
- [39] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

- [40] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*. 31–41.
- [41] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).
- [42] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Cocosoda: Effective contrastive learning for code search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2198–2210.
- [43] Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, et al. 2024. Learning Performance-Improving Code Edits. In *The Twelfth International Conference on Learning Representations*.
- [44] Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. 2024. UniCoder: Scaling Code Large Language Model via Universal Code. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1812–1824.
- [45] Zhensu Sun, Li Li, Yan Liu, Xiaoming Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*. 1609–1620.
- [46] 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).
- [47] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [48] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: empowering code generation with OSS-INSTRUCT. In *Proceedings of the 41st International Conference on Machine Learning*. 52632–52657.
- [49] Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Yewen Pu, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, et al. 2025. InverseCoder: Self-improving Instruction-Tuned Code LLMs with Inverse-Instruct. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 25525–25533.
- [50] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22 (2017), 3149–3185.
- [51] Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. 2023. Survey of code search based on deep learning. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–42.
- [52] Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. 2025. An Empirical Study of Retrieval-Augmented Code Generation: Challenges and Opportunities. *ACM Transactions on Software Engineering and Methodology* (2025).
- [53] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. 2025. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476* (2025).
- [54] 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*. 2471–2484.
- [55] Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Expanding queries for code search using semantically related api class-names. *IEEE Transactions on Software Engineering* 44, 11 (2017), 1070–1082.
- [56] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474* (2022).