Code Search Is All You Need? Improving Code Suggestions with Code Search

Junkai Chen
School of Software Technology,
Zhejiang University
Ningbo, China
junkaichen@zju.edu.cn

Xing Hu*
The State Key Laboratory of
Blockchain and Data Security,
Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Zhenhao Li
Concordia University
Montreal, Canada
l_zhenha@encs.concordia.ca

Cuiyun Gao
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

Xin Xia
Zhejiang University
Hangzhou, China
xin.xia@acm.org

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

ABSTRACT

Modern integrated development environments (IDEs) provide various automated code suggestion techniques (e.g., code completion and code generation) to help developers improve their efficiency. Such techniques may retrieve similar code snippets from the code base or leverage deep learning models to provide code suggestions. However, how to effectively enhance the code suggestions using code retrieval has not been systematically investigated. In this paper, we study and explore a retrieval-augmented framework for code suggestions. Specifically, our framework leverages different retrieval approaches and search strategies to search similar code snippets. Then the retrieved code is used to further enhance the performance of language models on code suggestions. We conduct experiments by integrating different language models into our framework and compare the results with their original models. We find that our framework noticeably improves the performance of both code completion and code generation by up to 53.8% and 130.8% in terms of BLEU-4, respectively. Our study highlights that integrating the retrieval process into code suggestions can improve the performance of code suggestions by a large margin.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**;

KEYWORDS

Code Suggestion, Code Search, Language Model

*Corresponding author.

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.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639085

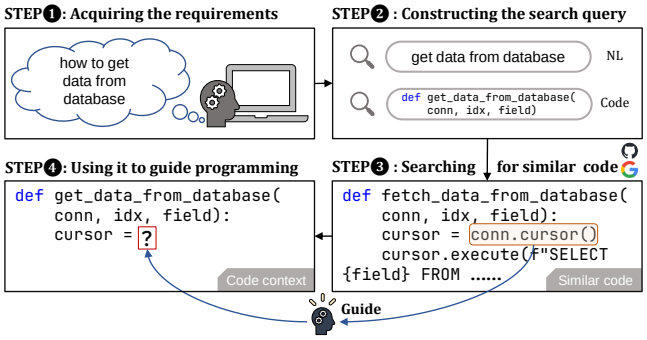

Figure 1: A common programming scenario during software development. A developer first constructs the query in NL or code, and then uses it to find similar code in the search engine or code base, which instructs the developer to program.

ACM Reference Format:
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 *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639085

1 INTRODUCTION

Code suggestion (e.g., code completion and code generation), which aims to suggest code for developers, is an important research topic in the software engineering community. Since the naturalness of software is introduced [21], many prior studies leverage language models (LMs) [18, 57] to improve the quality of code suggestions. In recent years, deep learning (DL) techniques have been widely used in code suggestions. Such DL techniques generally recommend code based on requirements written in natural language (NL). These works utilize deep neural networks (DNNs) to learn different types of code representations, such as token sequences [42, 50, 56, 61], abstract syntax trees (ASTs) [30, 35, 55, 59], and graphs (e.g., control flow graphs) [8, 15, 44], and then provide code suggestions.

However, as shown in Figure 1, developers typically search for similar code according to the requirements and then write the source code by imitating the searched exemplars instead of coding

from scratch [16]. They first construct an appropriate search query using the functional description in NL or using the key information of the desired code snippet (e.g., the header of a method) based on their need. They then use the query to search for similar code from the code base. Given the helpfulness of exemplars in such process, integrating similar code snippets may provide additional information and further improve the performance of code suggestions.

Prior studies [17, 48] proposed approaches to suggesting code based on retrieval and preliminarily investigated the effectiveness of code search in improving code suggestions. However, these approaches are generally designed for specific models and search algorithms. It is difficult to untangle and integrate with other models. Moreover, it still remains unclear for how to effectively combine different code search techniques and code suggestion models. In this paper, we propose a retrieval-augmented code suggestion framework, which has the following characteristics:

- **Various Combinations.** We adopt simple techniques (e.g., concatenation) to combine various types of code search strategies, techniques, and language models, which makes our approach able to efficiently explore the effectiveness of code retrieval in code suggestions under various combinations.
- **Plug-and-play.** Our framework is compatible with various retrieval approaches and language models, which saves the effort of redesigning them. It can integrate well with existing code suggestion systems.

Our framework consists of three components, including a *Retriever*, a *Formulator*, and a *Generator*:

- **Retriever** analyzes information in the *source* query to retrieve similar *target* code snippets. We study three **search strategies**: *Header2Code* (i.e., using method header in *source* and code in *target*), *NL2Code* (i.e., using NL descriptions in *source* and code in *target*), and *NL2NL* (i.e., using NL descriptions in both *source* and *target* as the key information to search). In addition, we adopt both information retrieval-based (IR-based) and deep learning-based (DL-based) approaches to support the retrieval.
- **Formulator** combines similar code obtained from the retriever with the code context. The formulator transforms such information into formulated input that can be processed by the generator.
- **Generator** is generally a language model that receives formatted input from the formulator and provides code suggestions. In this paper, we use both *general DL models* with relatively small parameter sizes (e.g., LSTM) and large language models (LLMs) as the generator.

To evaluate our framework, we combine different kinds of retrieval approaches, search strategies, and language models to evaluate the effectiveness of retrieval in code suggestions. We conduct experiments on the tasks of code suggestion at different levels of granularity: code completion and code generation. To accommodate the capability of LMs in different sizes, we perform code completion with general DL models and code generation with LLMs, respectively. We adopt both Java and Python datasets for the retriever and the generator. Results show that our framework noticeably improves the performance of both code completion and code generation.

The main contributions of our paper are summarized as follows:

- We propose a retrieval-augmented code suggestion framework that can integrate different search strategies, retrieval techniques, and LMs to improve the performance of code suggestions.
- We perform a comprehensive evaluation of our framework and results show that our framework improves Transformer-XL [12] by up to 53.8% and ChatGPT [4] by up to 130.8% in terms of BLEU-4 for the code completion and generation tasks, respectively.
- We investigate the effectiveness of different shot numbers and prompt templates when using retrieval-augmented ChatGPT. We find that a small shot number can sufficiently enhance the performance.

2 BACKGROUND

In this section, we discuss the background information related to our study, including code search and generative language models.

2.1 Code Search

Code search is an important practice in software development and maintenance. We follow previous works [34, 62] and summarize the code search techniques into IR-based and DL-based ones according to the key methodologies leveraged.

IR-based Code Search. Typically, an IR-based code search engine first builds indexes for specific fields of code snippets in the code base and then retrieves similar code that matches the query according to its scoring algorithm. One common similarity measure is BM25 score [53], which is applied to the full-text search engine Lucene [1]. BM25 is based on bag-of-words and considers the word frequency as well as the lengths of both the query and documents. Prior studies proposed various IR-based code search techniques. Liu et al. [34] proposed a code search tool that can understand the sequential semantics in important query words. Lv et al. [43] proposed a code search technique that expands the query with potential APIs. Overall, IR-based code search techniques have been widely used in practice and are relatively convenient to deploy and use.

DL-based Code Search. DL-based code search techniques utilize DNNs to convert queries and code snippets in the code base into feature vectors. After obtaining the vector representations, the common practice is to calculate the cosine similarity between the query and code snippets. Many prior works utilized deep learning techniques to improve the quality of code search. Gu et al. [14] proposed a neural network to learn the unified representations of both source code and natural language queries. Liu et al. [37] proposed a graph-based code search approach that learns the mapping of code and query by capturing structural and semantic information. In addition, pre-trained language models (e.g., CodeBERT [13]) have also been applied to code search and achieve promising results.

2.2 Generative Language Model

Generative language models aim to perform generation tasks such as text generation and code completion. We discuss such models by summarizing them into neural language models, pre-trained language models, and large language models.

Neural Language Models. Previous studies use neural networks such as RNNs to generate program or natural language contents.

LSTM [22], which introduces the memory cell to vanilla RNN, is widely used to model source code. For example, Li et al. [31, 32] leveraged bi-directional LSTM to provide suggestions for logging code. Recently, language models built on the Transformer [58] architecture such as Transformer-XL [12] have become popular. These neural language models are widely used in code suggestions. Liu et al. [35] utilized both Transformer-XL and LSTM networks to perform multi-task learning for code completion. Svyatkovskiy et al. [56] proposed a Transformer-based approach that provides instant code suggestions in the IDE.

Pre-trained Language Models. Pre-trained language models are usually trained on a large volume of data using Transformer-based architectures. Apart from tasks in NL (e.g., cloze test and question answering), they are also widely used in code suggestions. Wang et al. [60] presented an encoder-decoder pre-trained model leveraging code semantics conveyed from identifiers, which performs well in code generation. Ahmad et al. [6] performed the task of code generation on a language model pre-trained on an extensive collection of functions and NL text.

Large Language Models. LLMs refer to pre-trained models with a very large number of parameters (e.g., billions or more). The state-of-the-art LLMs include ChatGPT [4] and GPT-3.5 [9] which have attracted great attention for their excellent generative abilities. Codex [11], which has enhanced capability in generating the source code, has been integrated into GitHub Copilot [3] to provide automated support for developers in code suggestions.

3 METHODOLOGY

In this section, we present the methodology of our retrieval-augmented code suggestion framework. We first introduce the overview of our framework and then discuss the details of each component, respectively.

3.1 Overview

As shown in Fig. 2, our framework consists of three components: *Retriever*, *Formulator*, and *Generator*. Given the incomplete code snippet that needs suggestions (we refer to such content as *source context*), each of the components performs different roles in the code suggestion tasks:

- **Retriever** searches for similar code using different search strategies (e.g., *Header2Code*, which we will discuss later) and search tools (e.g., Lucene based on IR) according to the given information (e.g., method header) of the source context.
- **Formulator** combines retrieved code with the code context and then formulates them as the input to the generator.
- **Generator** leverages the formulated input to perform the tasks of code suggestions. We use different LMs as our generator.

3.2 Retriever

Developers generally search for the desired code snippets according to the functional descriptions in **natural language** or the **method header** (i.e., the method name, list of parameters, and the return type) that includes the definition information of the method. Therefore, in this paper, we investigate the effectiveness of different search strategies and search techniques on code suggestions.

Table 1: The search strategies and retrieval approaches we use.

Strategy	IR-based	DL-based
Header2Code	Lucene	ReACC-retriever
NL2Code		CodeBERT
NL2NL		Sentence-BERT

Before introducing the search strategies of our retriever, we discuss two common scenarios related to the process of code search. 1) One common scenario for code search is using queries in NL to find similar code snippets matching the corresponding descriptions (we refer to such scenario as *NL2Code*). Prior studies proposed a series of approaches considering this setting [14, 34, 45]. In this paper, we use the term "comment" uniformly to refer to the first sentence of *documentation comment* in Java or *docstring* in Python which usually summarizes the overall functional logic of the method. Given the hypothesis that similar code snippets are likely to have similar NL descriptions, finding code sharing similar comments (which we call *NL2NL*) is also indicated effective in prior studies [17, 29]. 2) The other common scenario for code search is using the definition information of the method (e.g., method header) to find the corresponding code snippets. Motivated by a prior work [19] that considers the method header as the summary of the function, we can use it to find the corresponding similar code (which we call *Header2Code*). Therefore, the retriever searches for code snippets based on these search strategies above, and the retrieved code snippets can then be used to enhance code suggestions.

Search Strategies. We use different search strategies to retrieve similar code snippets based on the information leveraged in the source context and code snippets in the code base. Below, we discuss each search strategy with the example shown in Figure 2.

- **Header2Code.** The retriever takes the method header as the query and uses it to find code snippets. For example, Strategy 1 in Figure 2 that compares the header of the source context with the code snippets in the code base belongs to this search strategy.
- **NL2Code.** The retriever uses the comment as the query to match code in the code base. For example, Strategy 2 in Figure 2 illustrates the process of using the comment *"fetch data from database"* to retrieve the corresponding code snippets.
- **NL2NL.** The retriever searches for comments in the code base with similar meanings and takes the corresponding code as the result of retrieval (i.e., Strategy 3 in Figure 2).

Retrieval Approaches. We use different retrieval approaches for the three search strategies. The search strategies and their corresponding retrieval approaches are shown in Table 1.

- **Lucene** is an IR-based search engine ranking candidates by BM25 score [53]. It supports all three search strategies.
- **ReACC-retriever** [41] is a DL-based retrieval approach that leverages RoBERTa [38] model to perform the code search task. Here we use the method header to find code that matches the functionality of this retriever (i.e., Header2Code search strategy).

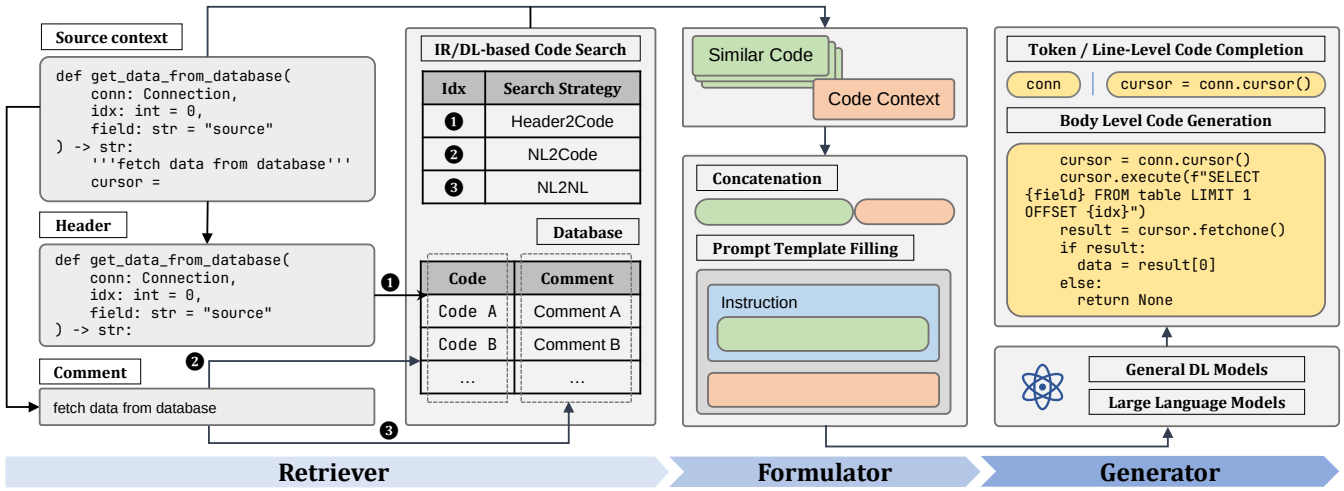

Figure 2: Overview of our code suggestion framework. It consists of a *Retriever*, a *Formulator*, and a *Generator*. The retriever uses queries in NL or code to find similar code, with IR-based or DL-based code search tools utilizing different search strategies. Then the formulator combines retrieved code and the code context according to the specific task (i.e., code completion or code generation). Finally, the generator makes code predictions at different levels using LMs.

- **CodeBERT** [13] is a DL-based language model pre-trained on both NL and code. We fine-tune CodeBERT with pairs of code and comment to adapt it for NL2Code search strategy.
- **Sentence-BERT** [51] is a BERT-like [27] model pre-trained on NL to derive semantically meaningful sentence embeddings. We use it to capture the similarity between comments in the source context and the candidate code (i.e., NL2NL search strategy).

3.3 Formulator

The formulator combines the retrieved code snippets with code of the source context and then transfers it to the generator as its input. Considering the differences in general DL models and LLMs, we use two different methods to formulate the input information for the generator:

- For **general DL models**, we formulate the generator's input x by concatenating the similar code s and code of source context c: $x = s \oplus c$.
- For **LLMs**, which perform well by prompting, we formalize the process as the task of prompt template filling. Given prompt template t, set of similar code snippets S, and given code of the source context c, our prompt P for LLMs is formalized as $P = fill(t, S, c)$, where *fill* means the procedure of inserting content into corresponding positions of the template. The prompt template we use consists of the introduction (i.e., the task description), exemplar(s) (i.e., retrieved similar code), and code context (i.e., method header). We will discuss the details of template designing in Section 5.3.

The code context mentioned above depends on the specific task of code suggestions (i.e., code completion or code generation), and it will be discussed in *Generator* below.

3.4 Generator

The generator leverages results returned from the formulator and performs the tasks of code suggestions accordingly. In our work, we use different LMs as the generator including general DL models (e.g., Transformer-XL [12]) and state-of-the-art LLMs (e.g., ChatGPT [4]). Specifically, given the formulated input that combines the retrieved code snippets and code of the source context, the generator learns from the retrieved code snippets and gives code suggestions. We use different LMs for different tasks of code suggestion:

- **General DL models** for **code completion**: We perform code completion at both token level and line level. For token-level code completion, given the formulated input including the retrieved code snippets and code tokens of source context, we use the generator to predict the next token following the input. For line-level code completion, we use the generator to predict the full statement in the function.
- **LLMs** for **code generation**: The objective of code generation is to generate the entire method body utilizing the formulated input, which includes the prompt instruction, retrieved code snippets, and method header of the source context.

4 EXPERIMENTAL SETUP

In this section, we present the metrics of code suggestions, the datasets we use in this paper, as well as the implementation details of our framework.

4.1 Metrics

For code completion tasks at different levels, we use different metrics to evaluate the performance of our framework:

- **Accuracy (Acc.)** (token-level) is the percentage of sequentially generated tokens that are exactly the same as the target tokens.

Table 2: Statistics of the datasets.

	Generator		Retriever	
Dataset	CodeXGLUE		CodeMatcher	PyTorrent
Language	Java	Python	Java	Python
# Methods in train	147,418	82,143		
# Methods in valid	5,150	4,169	10,482,463	2,841,300
# Methods in test	10,729	5,894		
# Avg. tokens in code	97.9	92.2	77.2	90.3
# Avg. tokens in comment	13.2	11.3	10.4	8.1

- **BLEU** score [47] (line-level) is a metric originally used for machine translation, and many works use it to evaluate code generation [17, 25]. In this task, it is computed based on the similarity of n-gram between generated code and ground truth. We use the metric where $n = 4$ (i.e., BLEU-4) following prior studies [33, 63].
- **Edit Similarity (ES)** measures the similarity between two code snippets based on the editing operations.

Following metrics are used for evaluation of code generation:

- **CodeBLEU** score [52] is a variant of BLEU score. On top of the textual similarity, it further leverages AST and data-flow structures to evaluate the grammatical correctness and logic correctness of the source code.
- **BLUE** is also used in the evaluation of code generation.

4.2 Datasets

Table 2 presents the overall statistics of datasets we use for the retriever and generator in our framework.

Datasets for Retriever. We use CodeMatcher [34] and PyTorrent [7] as the code base for the retrieval of code written in Java and Python, respectively.

- **CodeMatcher** [34] is a collection of Java code originally used for the evaluation of code search. It contains around 10.5M methods of Java code extracted from over 40,000 GitHub repositories with at least five stars.
- **PyTorrent** [7] is a corpus of Python code collected from around 220,000 Python packages in PyPI and Anaconda. This dataset contains over 2M methods in total.

Besides, we follow CodeXGLUE [42] and fine-tune CodeBERT to perform NL2Code search strategy with CodeSearchNet dataset [23].

Datasets for Generator. We use CodeXGLUE [42] as the dataset for our generator and follow its filtering and pre-processing process. We use the training portion of the dataset for the generator that requires training, including 147,418 Java methods and 82,143 Python methods, respectively. We use the testing portion to evaluate the performance of our framework on code suggestions. In total, the testing datasets contain 10,729 Java methods and 5,894 Python methods, respectively. For general DL models, we use the complete testing datasets for evaluation. For LLMs, we utilize a fixed random seed to select a sample of 500 examples for RQ2 and 100 examples for RQ3 due to the costs of utilizing OpenAI API [5].

4.3 Implementation Details

Information Extraction. In order to implement three search strategies (i.e., Header2Code, NL2Code, and NL2NL), we need to extract specific information (i.e., method header and comment) from the datasets of both the retriever and generator. We use the language parser tree-sitter [2] to extract the method header. In terms of the comment, we extract the first sentence of the documentation comment for the dataset CodeMatcher and use the given field such as "summary" in CodeXGLUE and PyTorrent.

Data Leakage Prevention and Preprocessing. In order to alleviate data leakage between the retriever and the generator (e.g., the generator directly "copies" the searched exemplar from the same source), we detect the same and similar names of the repositores and packages (e.g., "pypackage" and "package-py") between the datasets of retriever and generator. We then exclude such data from the generator dataset to remove the overlaps. We deduplicate the code base and remove the methods that cannot be parsed by tree-sitter correctly. We follow the preprocessing procedures of CodeXGLUE to normalize uncommon literals when we use general DL models. For all the data, we remove the comments inside the methods to avoid using information apart from code. We remove all illegal characters and convert comments to lowercase. When using general DL models, we leverage Byte Pair Encoding (BPE) [54] algorithm to tokenize code and comments.

Implementation Details. The implementation details of our framework are as follows:

- **Retriever**: We use Lucene as the IR-based retrieval approach and three different techniques shown in Table 1 as the DL-based retrieval approaches. We fine-tune CodeBERT for NL2Code search strategy and use released pre-trained models for Header2Code and NL2NL. We use Faiss [26] to accelerate DL-based retrievers. Note that ReACC-retriever only supports Python and we do not apply Header2Code search strategy to DL-based retrievers in Java.
- **Generator**: For general DL models, we use the corresponding official implementations to build our generators. For LLMs, we invoke OpenAI API to use them.
- **Training & Testing:** For general DL models and CodeBERT (for NL2Code search strategy), we use NVIDIA GTX 3090 to train and test them. We only use Top-1 search results in code retrieval for general DL models. We use early stopping and set the maximum training steps to limit the cost of training. For LLMs, we set the maximum number of output tokens to 300 in order to ensure consistency when using different numbers of similar codes.

5 RESULTS

In this section, we discuss the results by proposing and answering three research questions.

5.1 RQ1: Does Code Search Improve the Performance of Code Completion Using General Deep Learning Models?

Motivation. General DL models (e.g., LSTM) are widely used in code completion tasks by prior studies [35, 36, 42]. In this RQ, we

Table 3: Results for general DL models on token-level code completion (RQ1). "Ori.": original models (w/o retrieval). Bold numbers: best values of the corresponding metric. "↑X": relative improvements over the original models. We omit some non-highest improvements for conciseness. Percentage before token type: the proportion of tokens belonging to this type.

Token-Level Code Completion: Java													
Model	**Retriever**	**Overall Acc.(%)**				**~32% Identifier Acc.(%)**				**~46% Separator Acc.(%)**			
		Ori.	Header2Code	NL2Code	NL2NL	Ori.	Header2Code	NL2Code	NL2NL	Ori.	Header2Code	NL2Code	NL2NL
CodeGPT	IR-based	68.6	70.7	69.1	**71.0 ↑3.5**	48.9	51.9	49.0	**52.8 ↑8.0**	83.6	84.7	84.3	**84.8 ↑1.4**
	DL-based		——	70.2	**71.3 ↑3.9**		——	51.3	**53.4 ↑9.2**		——	84.6	**84.8 ↑1.4**
Tr-XL	IR-based	62.1	65.3	62.9	**65.5 ↑5.5**	35.7	41.4	37.4	**42.4 ↑18.8**	80.7	**81.9 ↑1.5**	81.0	81.8
	DL-based		——	64.2	**65.5 ↑5.5**		——	39.5	**41.8 ↑17.1**		————	81.8	**82.1 ↑1.7**
TFM	IR-based	61.5	64.8	62.3	**65.0 ↑5.7**	34.8	40.6	36.4	**41.7 ↑19.8**	80.4	**81.5 ↑1.4**	80.6	**81.5 ↑1.4**
	DL-based		——	63.6	**64.9 ↑5.5**		——	38.8	**41.2 ↑18.4**		——	81.3	**81.4 ↑1.2**
LSTM	IR-based	52.1	**53.1 ↑1.9**	52.3	**53.1 ↑1.9**	13.3	**14.8 ↑11.3**	13.9	14.7	77.3	**78.3 ↑1.3**	77.6	78.0
	DL-based		——	**53.2 ↑2.1**	53.0		——	**14.7 ↑10.5**	**14.7 ↑10.5**		——	**78.1 ↑1.0**	77.7
Avg.	IR-based	61.1	63.5 ↑3.9	61.7 ↑1.0	**63.7 ↑4.3**	33.2	37.2 ↑12.0	34.2 ↑3.0	**37.9 ↑14.2**	80.5	**81.6 ↑1.4**	80.9 ↑0.5	81.5 ↑1.2
	DL-based		——	62.8 ↑2.8	**63.7 ↑4.3**		——	36.1 ↑8.7	**37.8 ↑13.9**		——	**81.5 ↑1.2**	**81.5 ↑1.2**
Token-Level Code Completion: Python													
Model	**Retriever**	**Overall Acc.(%)**				**~36% Identifier Acc.(%)**				**~26% Separator Acc.(%)**			
		Ori.	Header2Code	NL2Code	NL2NL	Ori.	Header2Code	NL2Code	NL2NL	Ori.	Header2Code	NL2Code	NL2NL
CodeGPT	IR-based	60.6	63.5	61.5	**65.2 ↑7.6**	43.3	47.8	44.7	**50.5 ↑16.6**	77.5	78.8	78.3	**80.3 ↑3.6**
	DL-based		64.3	63.1	**65.8 ↑8.6**		48.9	47.2	**51.4 ↑18.7**		79.2	79.1	**80.4 ↑3.7**
Tr-XL	IR-based	53.7	58.1	55.7	**60.2 ↑12.1**	31.2	39.5	34.7	**42.5 ↑36.2**	74.1	75.2	75.3	**77.9 ↑5.1**
	DL-based		57.3	56.6	**60.0 ↑11.7**		37.8	36.6	**41.5 ↑33.0**		73.7	75.2	**75.9 ↑2.4**
TFM	IR-based	52.7	57.4	54.8	**59.2 ↑12.3**	29.3	37.7	33.6	**40.6 ↑38.6**	73.2	75.5	74.2	**76.8 ↑4.9**
	DL-based		56.3	55.8	**59.0 ↑12.0**		36.1	35.3	**40.2 ↑37.2**		73.9	73.6	**76.0 ↑3.8**
LSTM	IR-based	43.0	44.1	43.9	**44.2 ↑2.8**	15.0	**16.0 ↑6.7**	**16.0 ↑6.7**	15.9	64.6	66.3	67.0	**67.6 ↑4.6**
	DL-based		44.2	44.2	**44.3 ↑3.0**		15.8	**16.1 ↑7.3**	15.9		65.8	67.2	**67.4 ↑4.3**
Avg.	IR-based	52.5	55.8 ↑6.3	54.0 ↑2.9	**57.2 ↑9.0**	29.7	35.3 ↑18.9	32.3 ↑8.8	**37.4 ↑25.8**	72.4	74.0 ↑2.2	73.7 ↑1.8	**75.7 ↑4.6**
	DL-based		55.5 ↑5.7	54.9 ↑4.6	**57.3 ↑9.1**		34.7 ↑16.8	33.8 ↑13.8	**37.3 ↑25.4**		73.2 ↑1.1	73.8 ↑1.9	**74.9 ↑3.6**

investigate the effectiveness of our framework on code completion with general DL models as the generator.

Approach. For the retriever, we use both the IR-based retriever Lucene and DL-based retrievers to perform code search with three search strategies (i.e., Header2Code, NL2Code, and NL2NL). For the generator, we use the following general DL models which have been applied to code completion by prior studies [35, 36, 42]:

- **LSTM** [22] is a variant of classic RNN that adds the memory cell to vanilla RNN architecture to maintain long-period information.
- **Transformer Decoder (TFM)** [58] is the decoder part of vanilla Transformer, which is suitable for generative tasks.
- **Transformer-XL (Tr-XL)** [12] is a Transformer-based, decoder-only language model, which utilizes the recurrence mechanism and relative positioning embeddings to capture the dependency beyond a fixed-length context.
- **CodeGPT** [42] is a Transformer-based, decoder-only language model. It is pre-trained on code and shares the same architecture with GPT-2 [49].

We conduct experiments of code completion at token level and line level, respectively. Particularly, for token-level code completion, we evaluate the accuracy by different types of tokens (i.e., identifier and separator, which are the top two most common types of tokens in both Java and Python) motivated by prior studies [24, 35]. We use the results generated by each model without code retrieval as baselines.

Results. Table 3 shows the results of token-level code completion and Table 4 shows the results of line-level code completion, respectively. In each table, *Ori.* is the original result of each model without code retrieval. Overall, we find that **our retrieval-augmented framework improves the performances of code completion for general DL models at both token level and line level.** We discuss the detailed results of this RQ from different aspects.

Table 4: Results for general DL models on line-level code completion (RQ1).

Line-Level Code Completion: Java									
Model	**Retriever**	**BLEU-4 (%)**				**ES (%)**			
		Ori.	Header2Code	NL2Code	NL2NL	Ori.	Header2Code	NL2Code	NL2NL
CodeGPT	IR-based	26.3	30.9	26.2	**33.3 ↑26.6**	52.5	55.7	52.4	**57.2 ↑9.0**
	DL-based		——	28.7	**33.8 ↑28.5**		——	54.3	**57.5 ↑9.5**
Tr-XL	IR-based	20.1	25.5	21.3	**28.5 ↑41.8**	47.1	51.4	48.2	**53.2 ↑13.0**
	DL-based		——	23.6	**27.4 ↑36.3**		——	50.3	**52.4 ↑11.3**
Avg.	IR-based	23.2	28.2 ↑21.6	23.8 ↑2.6	**30.9 ↑33.2**	49.8	53.6 ↑7.6	50.3 ↑1.0	**55.2 ↑10.8**
	DL-based		——	26.2 ↑12.9	**30.6 ↑31.9**		——	52.3 ↑5.0	**55.0 ↑10.4**
Line-Level Code Completion: Python									
Model	**Retriever**	**BLEU-4 (%)**				**ES (%)**			
		Ori.	Header2Code	NL2Code	NL2NL	Ori.	Header2Code	NL2Code	NL2NL
CodeGPT	IR-based	22.0	26.6	22.6	**31.7 ↑44.1**	48.5	51.8	48.8	**55.0 ↑13.4**
	DL-based		26.9	24.6	**32.1 ↑45.9**		51.7	50.3	**55.3 ↑14.0**
Tr-XL	IR-based	18.4	23.5	18.8	**28.3 ↑53.8**	45.1	49.1	45.7	**52.5 ↑16.4**
	DL-based		22.6	20.6	**28.1 ↑52.7**		48.6	47.1	**52.2 ↑15.7**
Avg.	IR-based	20.2	25.1 ↑24.3	20.7 ↑2.5	**30.0 ↑48.5**	46.8	50.5 ↑7.9	47.3 ↑1.1	**53.8 ↑15.0**
	DL-based		24.8 ↑22.8	22.6 ↑11.9	**30.1 ↑49.0**		50.2 ↑7.3	48.7 ↑4.1	**53.8 ↑15.0**

Comparison among the models. In general, our framework enhances all the general DL models to varying degrees. For example, for IR-based retriever and NL2NL search strategy (represented in the format of *IR-based + NL2NL* in the remaining paper), the accuracy of Tr-XL improves by 5.5% in Java and 12.1% in Python at token level. In contrast, the increases observed in LSTM at token level are relatively lower, ranging from 0.4% to 3.0%. We consider that the suboptimal performance of augmented-LSTM may be due to the input length increased by concatenated exemplars, which presents a challenge to learn similarities between similar code and the code context.

Comparison among the search strategies. The retriever improves the performance of code completion in general, while different search

strategies bring different enhancements. As shown in Table 3 and Table 4, for each model and retriever, the search strategy with the highest improvement is marked in bold associated with the relative change. The results indicate that NL2NL search strategy improves the accuracy of code completion to the largest margin, while NL2Code search strategy bring the most limited improvements in general. The finding emphasizes the effectiveness of NL2NL search strategy, which leverages text similarity to find similar code. On the other hand, the under-performance of NL2Code strategy highlights the inherent difficulty in cross-modality retrieval (i.e., NL and code). Besides, DL-based retrievers generally perform better than IR-based retrievers when using NL2Code search strategy as illustrated in Table 4. It suggests that deep learning techniques reduce the disparities between NL and code, and enhance DL-based retriever for the understanding of NL queries.

Token types and languages. In general, we find that the improvement on identifiers is more considerable than other type of tokens. For example, identifiers receive a higher improvement in *Java + CodeGPT + IR-based + NL2NL*, and the improvement obtained by separators is more subtle (i.e., 8.0% v.s. 1.4%). This discrepancy arises because separators in code are commonly used and follow regular patterns, and the results of original models are already high (i.e., over 70% for most of the models). The models may hardly capture additional knowledge for seperators from the retrieved code snippets. Additionally, the proportion of separators in Java is noticeably higher than in Python (i.e., ∼46% v.s. ∼26%). This disparity can be attributed to that Java requires more symbols like commas and brackets to separate code fragments, whereas Python relies on indents. We consider this difference might be related to the superior performance of Java code completion compared to Python.

Line-level code completion. As shown in Table 4, we find that our framework can also improve the performance of line-level code completion. Similar to token-level code completion, NL2NL performs the best among all the search strategies. For example, the ES metric is improved by 15.7% in *Python + Tr-XL + DL-based* using NL2NL, while the other two search strategies only slightly outperform the original model (i.e., 48.6% and 47.1% v.s. 45.1%).

Summary of RQ1: Our retrieval-augmented code suggestion framework observably improves the performance of code completion for general DL models. Among the search strategies, NL2NL achieves the best overall improvements.

5.2 RQ2: Does Code Search Improve the Performance of Code Generation Using Large Language Models?

Motivation. Recently, the outstanding performance of LLMs on code generation has attracted widespread attention from the software engineering community. Hence, in this RQ, we further combine our code suggestion framework with LLMs to verify the effectiveness of our approach on code generation.

Approach. LLMs leverage prompting to perform the tasks specified by users [39]. In our framework, the formulator for LLMs combines code of the source context (i.e., method header) and the corresponding similar code snippets with a specifically designed prompt template, which is formulated as the task of prompt template filling. We choose the "detailed, implicit, two-step" prompt template which is discussed in Section 5.3. For the generator, we conduct our experiments with two GPT-3.5 series models to verify the effectiveness of our framework on code generation:

- **ChatGPT** (*gpt-3.5-turbo*) is a variant of the GPT-3.5 series model which shows great performance on generation tasks.
- **GPT-3.5** (*text-davinci-003*) is a GPT-3.5 series model which is trained via reinforcement learning.

Different from general DL models, we evaluate our framework using code generation tasks to accommodate LLM's enhanced generation capabilities. We use vanilla LLMs as baselines to compare with LLMs using our framework.

Table 5: Results for LLMs on code generation (RQ2).

Body-Level Code Generation: Java									
Model	Retriever	BLEU-4 (%)				CodeBLEU (%)			
		Ori.	Header2Code	NL2Code	NL2NL	Ori.	Header2Code	NL2Code	NL2NL
ChatGPT (gpt-3.5-turbo)	IR-based	20.1	23.9	20.6	**29.6 ↑47.3**	34.2	36.9	35.3	**41.8 ↑22.2**
	DL-based		––	22.7	**29.9 ↑48.8**		––	35.3	**41.6 ↑21.6**
GPT-3.5 (text-davinci-003)	IR-based	24.2	24.5	23.3	**32.0 ↑32.2**	35.3	37.5	36.3	**41.9 ↑18.7**
	DL-based		––	24.5	**31.5 ↑30.2**		––	35.8	**42.1 ↑19.3**
Avg.	IR-based	22.2	24.2 ↑9.0	22.0	**30.8 ↑39.1**	34.8	37.2 ↑6.9	35.8 ↑2.9	**41.9 ↑20.4**
	DL-based		––	23.6 ↑6.3	**30.7 ↑38.6**		––	35.6 ↑2.3	**41.9 ↑20.4**
Body-Level Code Generation: Python									
Model	Retriever	BLEU-4 (%)				CodeBLEU (%)			
		Ori.	Header2Code	NL2Code	NL2NL	Ori.	Header2Code	NL2Code	NL2NL
ChatGPT (gpt-3.5-turbo)	IR-based	10.7	18.1	14.3	**24.7 ↑130.8**	23.6	29.7	26.5	**33.7 ↑42.8**
	DL-based		16.2	16.3	**24.2 ↑126.2**		27.0	27.7	**33.5 ↑41.9**
GPT-3.5 (text-davinci-003)	IR-based	11.9	18.8	15.1	**25.0 ↑110.0**	23.2	31.1	28.4	**35.5 ↑53.0**
	DL-based		15.8	16.3	**23.8 ↑100.0**		26.4	27.7	**34.5 ↑48.7**
Avg.	IR-based	11.3	18.5 ↑63.7	14.7 ↑30.1	**24.9 ↑119.9**	23.4	30.4 ↑29.9	27.5 ↑17.5	**34.6 ↑47.9**
	DL-based		16.0 ↑41.6	16.3 ↑44.2	**24.0 ↑112.4**		26.7 ↑14.1	27.7 ↑18.4	**34.0 ↑45.3**

Results. Table 5 presents the results of this RQ. Overall, we find that **large language models in the retrieval-augmented framework we propose achieve significant improvements compared to baselines without retrieval.** The performance of code generation is enhanced across all metrics with the combination of almost any retrieval approaches and search strategies. For instance, retrieval-augmented ChatGPT outperforms vanilla ChatGPT by over 100% (i.e., *Python + BLEU-4*).

We find our framework brings more noticable improvement on LLMs than general DL models. Compared to general DL models, LLMs in the framework exhibit a larger margin of relative improvements. For example, the BLEU-4 score of general DL models on line-level Python code completion improves from 44.1% to 53.8%, while the improvement for LLMs on Java code generation ranges from 100.0% to 130.8% in BLEU-4 score. The results indicate that LLMs excel at capturing the relationship between the method header and similar code snippets through proper prompts. Given that attaching retrieved code snippets can considerably increase the length of input, the results show that LLMs may have superior capability in leveraging long inputs than general DL models. Such capability enables them to imitate searched exemplars like developers and generate the corresponding method body given the context.

We find that ChatGPT and GPT-3.5 receive a significantly higher boost in Python than in Java (i.e., 119.9% v.s. 39.1% in *Avg. + IR-based + NL2NL*). To further explore this, we use several prompts such as "Give the implementation of the quick sort algorithm."

Table 6: Results of different prompt templates for ChatGPT with and without our framework in RQ3. Bold numbers: Values ranking top 3 in the column.

Feature			BLEU-4 (%)		CodeBLEU (%)	
Det.	Exp.	Ts.	w/	w/o	w/	w/o
			33.4	20.6	**47.0**	37.0
		✓	32.9	20.6	**47.5**	37.6
	✓		**34.5**	21.5	45.8	34.0
	✓	✓	30.1	21.6	43.6	33.8
✓			29.7	22.3	44.5	36.6
✓		✓	**34.1**	22.0	**47.5**	36.4
✓	✓		**33.7**	20.8	44.9	34.5
✓	✓	✓	32.5	21.1	44.3	34.3

Det.: Using detailed (with "✓") or brief instruction (without "✓")
Exp.: Using explicit (with "✓") or implicit instruction (without "✓").
Ts.: Using two-step (with "✓") or one-step instruction (without "✓").

without specifying the programming language to prompt ChatGPT to generate code snippets. Through testing a dozen of samples, we observe that ChatGPT consistently generates Python code snippets by default. It may indicate that ChatGPT has more proficiency in Python, and it is consistent with Codex [3] which suggests Python code more accurately and shares similar code capabilities with ChatGPT. Additionally, **we find that the observations related to search strategies in RQ1 still hold for this RQ** (e.g., NL2NL outperforms other search strategies).

Summary of RQ2: LLMs in the retrieval-augmented code suggestion framework we propose exhibit improvements of up to 130.8% in terms of BLEU-4 in code generation, surpassing the margin of improvement on general DL models.

5.3 RQ3: How does Retrieval-Augmented ChatGPT Perform with Different Shot Numbers and Prompt Templates ?

Motivation. In RQ2, we find that our framework can improve the capability of LLMs on code generation. Considering the impressive performance of LLMs in few-shot learning and prompting, it is important to investigate the effect of varying shot numbers and prompt templates. Hence, in this RQ, we first examine the performance of LLMs on code generation using different numbers of similar code snippets and subsequently analyze the impact of different prompt templates.

Approach. We use the combination with the best performance in RQ2 (i.e., ChatGPT and NL2NL) to conduct the experiments using a range of different shot numbers (i.e., [0, 1, 2, 4, 8]). We do not experiment with a larger number of similar code snippets due to the imposed upper limit of input length. We use the evaluation dataset written in Java discussed in Section 4 to conduct the experiments. To study the impact of the prompt template, we examine three distinct features that differentiate various prompt templates, as illustrated in Figure 3:

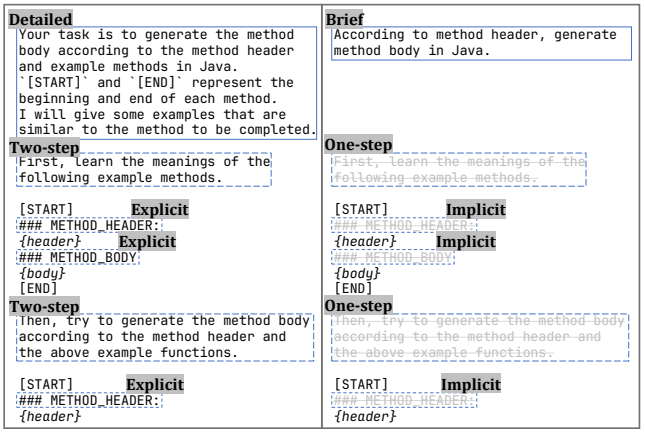

Figure 3: The illustration of the different prompt templates. The left prompt template is "detailed, explicit and two-step", and the right one is "brief, implicit and single-step."

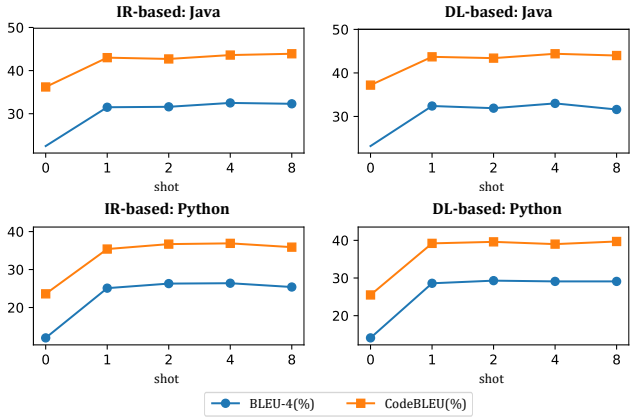

Figure 4: The performance of ChatGPT with NL2NL search strategy using different numbers of exemplars.

- **Detailed / Brief** means whether or not to provide a detailed instruction of the task. For example, the left instruction in Figure 3 is "Detailed" since it provides specific information regarding the formats of the input. On the contrary, the "Brief" instruction on the right only provides the basic requirements.
- **Explicit / Implicit** means whether or not to indicate the method header and body explicitly. For instance, the left prompt in Figure 3 (i.e., "Explicit") uses *"###METHOD_HEADER"* to explicitly state the position of the method header, while the right prompt (i.e., "Implicit") do not state such position in an explicit way.
- **Two-step / One-step** means whether or not to add additional instruction before the prediction. For example, the left prompt (i.e., "Two-Step") further specifies the model to generate code based on the above exemplars, while the right prompt (i.e., "One-Step") does not include this information.

Results. Overall, we find that **as the shot number increases, the performance of retrieval-augmented ChatGPT on code**

generation initially rises and then fluctuates. Figure 4 illustrates the results of code generation with different shot numbers. From the line chart, we find that the performance of *ChatGPT + NL2NL* consistently improves when the number of shots is in the range of 1, 2, 4, and 8. However, the improvement is not linear as the shot number increases. There is a significant boost of the performance when the number of shots is increased from 0 to 1, while only a slight improvement is generally observed from 1 to 2 shots. However, when the number of shots reaches 4 and 8, the performance start to fluctuate and show minor improvement. This trend is observed across different combinations of retrievers and languages. Overall, we find that one and two shots can enhance the performance of code generation compared to zero-shot. This can be attributed to the increased exposure to similar code examples, enabling LLMs to learn more information from the exemplar. We do not find an obvious improvement when the number of shots increases to 4 and 8. The potential reasons might be: 1) The quality of similar code starts declining, becoming less relevant to the query, thereby introducing interference to the model; 2) The increase in prompt length makes it challenging for the model to comprehend the context.

ChatGPT demonstrates robust performance on retrieval-augmented code generation tasks with various types of prompt templates. We conduct experiments using all combinations of prompt templates features (e.g., brief/detailed), and the results are presented in Table 6. We find that ChatGPT achieves improvements in code generation by all prompt templates with various features, The "brief, explicit, two-step" and "detailed, implicit, single-step" templates perform poorly among the template combinations, while the "detailed, implicit, two-step" prompt template ranks in the top 3 in both metrics. The performances of the other prompt templates are relatively similar. These results indicate that ChatGPT is generally robust in generating code using our retrieval-augmented framework and might be less affected by changes in prompt templates.

Summary of RQ3: Different numbers of searched exemplars can enhance ChatGPT's performance in code generation. Incorporating a small yet high-quality set of similar code snippets yields more substantial improvements. Retrieval-augmented ChatGPT performs well with various prompt templates.

6 DISCUSSION

6.1 Discussion on Existing Retrieval-based Code Suggestion Approaches

Prior studies proposed some retrieval-based code suggestion approaches such as ReCode [17] and REDCODER [48]. Compared to these approaches, our framework has the following main differences or improvements:

- **Usage scenario and mechanism.** (1) Our framework is compatible with various language models, code search strategies, and tools without redesigning new model architectures. Its simplicity and modularity facilitate easy integration with state-of-the-art code suggestion models. However, existing approaches typically employ complicated models and algorithms, rendering them difficult to transfer to other systems. (2) Our framework employs a separate large-scale retrieval codebase which allows the generator to learn semantic relationships between similar code pairs in the training phrase. (3) Our framework only requires the generator to handle the code-to-code generation task, whereas existing approaches typically need to learn both code and NL. Such process may also weaken their generality.
- **Dataset and evaluation.** (1) The popular NL-to-code datasets (e.g., Hearthstone [33] and Django [46]) are not generic enough compared to datasets like CodeXGLUE that have a wide variety of code sources. For example, the Hearthstone dataset (which comes from a card game) contains highly structured NL descriptions of cards such as "Acidic Swamp Ooze NAME_END 3 ATK_END 2..." and significantly repetitive code snippets for similar attributes of cards. (2) We conduct the evaluation on various generation granularities, code retrieval setups, and different scales of LMs, which is commonly absent in prior studies.

Table 7: Results of NL-to-code generation. We use the IR-based code search tool and NL2NL strategy for retrieval. Bold numbers: best values in the columns. EM: exact match, which is the proportion of predictions that exactly match the ground truth.

Model	Hearthstone		Django	
	EM	BLEU-4	EM	BLEU-4
ReCode [17]	19.6	78.4	72.8	84.7
REDCODER [48]	**21.2**	80.1	——	——
CodeGPT (w/o retrieval)	15.2	**80.9**	76.5	82.9
CodeGPT (w/ retrieval)	19.6	78.4	**79.2**	**85.4**

Preliminary comparison with our framework. Due to the significant differences between the prior approaches and our framework discussed above, we do not include these approaches in our evaluation in Section 5. Instead, we conduct a preliminary comparison of the existing approaches (i.e., ReCode and REDCODER) with our framework. Specifically, we follow the setup in previous works [17] [29] and conduct NL-to-code generation experiments on Hearthstone [33] and Django [46] datasets. For our approach, we utilize IR-based retriever Lucene and NL2NL search strategy to retrieve similar code and use CodeGPT as the generator.

Table 7 presents the results of our experiments. Overall, we find that our approach (i.e., CodeGPT w/ retrieval in Table 7) shows competitive performance compared with the baselines, which demonstrates the effectiveness of our approach. Specifically, our approach achieves the best performance in terms of EM and BLEU-4 (i.e., 79.2 and 85.4) on the Django dataset. For the results of the Hearthstone dataset, CodeGPT without retrieval (i.e., CodeGPT w/o retrieval in Table 7) fails to generate satisfactory code snippets compared to ReCode (i.e., 15.2 v.s. 19.6 in EM), but it is improved using our framework. For the results of the Django dataset, CodeGPT with retrieval obtains a 3.5% performance boost in EM (i.e., 76.5→79.2) and a 0.8% performance boost in BLEU-4 (i.e., 84.7→85.4). We observe that our approach falls slightly behind the existing approaches in

Table 8: The time costs of different retrievers to search for similar code.

Tool	Time cost		
	Training	Indexing	Searching
Lucene	——	3m 5s	0.02s
ReACC-retriever	N/A	5h 45m 26s	0.03s
CodeBERT	2h 33m 4s	5h 36m 14s	0.03s
Sentence-BERT	N/A	55m 5s	0.02s

N/A: The details of the time for training are not available and we do not retrain or fine-tune these retrievers.

terms of the BLEU-4 metric on the Hearthstone dataset, which may suggest that our approach is not suitable for a sturctured context like card game.

6.2 Time Efficiency of Different Retrievers

In real-world scenarios, the latency of code suggestions significantly impacts developers' user experience. For example, the time taken to search for similar code could lead to high time costs for retrieval-based code suggestion systems. Therefore, we discuss the time efficiency of different retrievers (i.e., IR-based and DL-based tools) employed in our framework. We choose Python for analysis because all search strategies are available for Python. We use PyTorrent [7] as the code base and CodeXGLUE as the source of queries, which is consistent with the RQs above.

Table 8 presents the details of time efficiency for different retrievers in our study. For each retriever, we use 100 queries to conduct the search and report the average time. On the whole, Lucene excels in the training and indexing phases, and the difference in search time between the retrievers is not significant. As Lucene is based on IR techniques rather than learnable approaches, we do not need to train it before usage. In contrast, DL-based retrievers require much time to adapt the models. In the indexing phase, the sparse bag-of-words representation makes Lucene take significantly less time than other DL-based approaches (i.e., 3m 5s v.s. at least 55m 5s). Dense retrievers such as CodeBERT need to convert the queries and code into high-dimensional embeddings, and the calculations on them are time-consuming. Since Sentence-BERT uses embedding vectors of lower dimensions than the other two DL-based retrievers, it has a relatively short indexing time. The retrievers we use in our framework for the searching phase can perform code retrieval in a very short period of time (i.e., less than 0.05 seconds), and their performance can be considered at the same level.

It is worth noting that although building indexes and training models consume a lot of time for DL-based retrievers, these processes are typically performed offline and only need to be done once in actual usage. Therefore, we pay more attention to the actual search time of the retriever which has a substantial impact on the time efficiency of retrieval-based code suggestion systems. Given that all the retrievers we employed in our framework can retrieve efficiently (i.e., no more than 0.03s per search), we consider that the time consumption of similar code retrieval may not be a noticeable concern for the user experience of code suggestions.

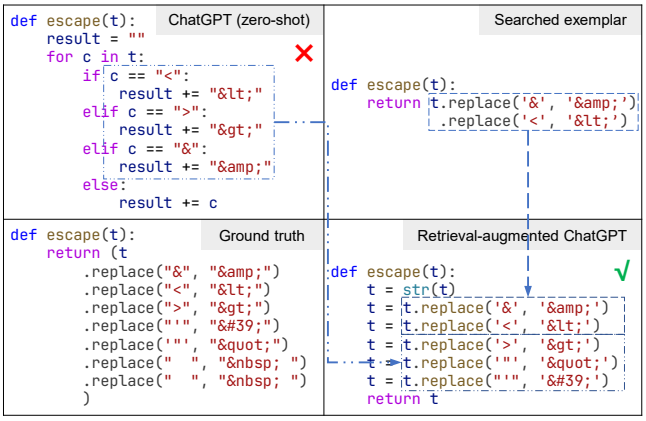

Figure 5: Examples of code snippets generated by zero-shot ChatGPT and retrieval-augmented ChatGPT. We omit the non-code text in the prompts for conciseness.

6.3 Case study

Figure 5 shows the contents of our case study, which includes code snippets generated by ChatGPT with zero-shot and ChatGPT with our retrieval-augmented framework. From these examples, we have the following findings: (1) ChatGPT under the zero-shot setup may generate logically incorrect code in the absence of sufficient context. For example in Figure 5, we find that ChatGPT misunderstands the meaning of the code context (i.e., this function simply concatenates some special characters such as "<" and "&" into a string rather than "escape" them). (2) The searched exemplar obtained through code retrieval serves as a valuable source of context to assist ChatGPT in generating better code. By comparing the code snippets generated by retrieval-augmented ChatGPT with the exemplar, we find that LLMs with retrieval can correct logical errors of code in the zero-shot setting (i.e., the generated function replaces special tokens instead of concatenating them, which is consistent with the meaning of "escape"). (3) Our approach could help ChatGPT combine its own knowledge with exemplars to generate code. In other words, the model does not simply copy code from the exemplar but learns the useful code pattern in it. For instance, retrieval-augmented ChatGPT not only learns to replace "&" to "&" but also combines its knowledge of special characters in HTML to generate code that is closer to the ground truth.

6.4 Implications

According to the findings of our research questions for the proposed framework, we discuss the implications for the community:

The strategy of code search. According to the results of RQ1 and RQ2, we observe a strong correlation between the effectiveness of code suggestions and the search strategies employed by retrievers. Specifically, Header2Code and NL2NL code search strategies demonstrate significant improvements in code suggestions, which may indicate that the search results are more accurate. Therefore, we recommend using the same modality for both the query and the target (e.g., NL2NL) during the process of code search. Additionally, when constructing a code base for code search, it is advisable to

gather bimodal corpora (i.e., NL-code pairs) whenever possible, as this allows for flexible adjustments to the search strategy.

The technique of code search. As introduced in Section 2, DL and IR are two commonly used techniques in code search tools. In our study, we find that there is no significant difference in the improvements of code suggestion tasks between different code search techniques (i.e., IR and DL). However, it is worth noting that DL-based code search tools generally require more time before available compared to IR-based tools due to the training and indexing processes involved as discussed before. Therefore, in most cases, we believe that IR-based code search tools (e.g., Lucene) are a better choice in terms of "cost-effectiveness" for retrieval-augmented code suggestions. In the meantime, we suggest employing DL-based code search tools when NL2Code search strategy is applied due to the fact that DL models possess superior capabilities in capturing cross-modality relationships compared to traditional IR approaches.

The number of shots in the prompt. Exemplars in the prompt can aid LLMs in accomplishing code generation according to the results. However, an excessive shot number is not necessarily advantageous. Our study reveals that shot numbers ranging from 1 to 2 result in notable improvements, whereas an increased number introduces instability and even regression in performance. Therefore, we suggest employing a limited number of examples in the prompt for optimal performance of LLMs.

6.5 Threats to Validity

Internal Validity. One potential threat to internal validity is the randomness of LLMs' outputs. During the experiments, we control the parameters of the API to remain unchanged, varying only the prompts necessary for the evaluation to mitigate this threat. Due to the costs and rate limits of invoking OpenAI APIs, we conduct the experiments only once. Future studies may consider repeating the experiments at certain times to investigate the randomness of LLMs if available.

External Validity. One potential threat to external validity is the generalizability of our findings which involves datasets, languages, baselines, and LMs. We conduct our study on both Java and Python datasets widely used in prior works related to code suggestions [13, 42, 60]. In terms of LMs, we use both general DL models and LLMs which have different parameter sizes and generation capabilities. For the baseline models, we compare our framework with the vanilla models to verify the effectiveness. However, we cannot confirm that our results can be generalized to other settings. Further studies could verify the findings of our work.

Construct validity. One potential threat to construct validity is the evaluation tasks and metrics. We conduct experiments on both code completion and code generation tasks which are common and practical in real-world development. The corresponding metrics for different tasks are commonly used in prior works [24, 28, 33]. Using other metrics may have different results, which can be further validated by future studies.

7 RELATED WORK

Retrieval-based Code Suggestions. Prior works presented various approaches that integrated similar code into code suggestions and initially verified that code suggestion tasks could benefit from code retrieval. Hayati et al. [17] introduced retrieval methods to neural code generation models. Parvez et al. [48] presented a retrieval-based approach for code generation that could utilize both unimodal (NL or code) and bimodal (NL-code pairs) data. Li et al. [29] proposed a sketch-based code generation approach by generating and editing code sketches from similar code snippets. These works primarily emphasized the advanced algorithms and sophisticated models, while our framework explores different code search techniques and strategies for code suggestion tasks.

Code Search. A number of works investigated how to search for more accurate code given the query. For IR-based code search techniques, related studies mainly focused on utilizing the queries effectively by expansion and reformulation [20, 40, 43]. For DL-based code search techniques, these studies generally focused on constructing various neural networks to capture the semantic relationship between the query and code snippets [10, 14, 37]. In this paper, we emphasize how to leverage existing code search tools to find similar code as the exemplar for LMs. Specifically, we explore the impact of different search techniques and strategies on the improvement of code suggestions.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose a retrieval-augmented framework for code suggestions. We integrate different code search techniques, search strategies, and language models into our framework and evaluate the performance. The results show that our framework noticeably improves the performance of code suggestions by a large margin. In addition, we also study the impact of shot numbers and prompt templates for retrieval-augmented ChatGPT. In the future, we may explore the profound impact of retrieval on code suggestions, and further unleash the potential of retrieval-augmented approaches to more tasks in software engineering.

DATA AVAILABILITY

Our replication package[1] is publicly available.

ACKNOWLEDGMENTS

This research is supported by the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), National Natural Science Foundation of China (No. 62141222), and the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] 1999. Apache Lucene. https://lucene.apache.org/. Last accessed May. 2023.
[2] 2017. Tree-Sitter. https://github.com/tree-sitter/tree-sitter. Last accessed May. 2023.
[3] 2021. GitHub Copilot, your AI pair programmer. https://github.com/features/copilot. Last accessed May. 2023.
[4] 2022. ChatGPT, OpenAI. https://chat.openai.com/chat. Last accessed May. 2023.
[5] 2023. Introduction - OpenAI API. https://platform.openai.com/docs/guides/completion/introduction. Last accessed May. 2023.
[6] WU Ahmad, S Chakraborty, B Ray, and KW Chang. 2021. Unified pre-training for program understanding and generation.. In *Proceedings of the 2021 Conference*

[1] https://github.com/iCSawyer/CodeSuggestion

of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.

[7] Mehdi Bahrami, N. C. 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. https://arxiv.org/abs/2110.01710. arXiv:2110.01710 [cs.SE]

[8] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *International Conference on Learning Representations*.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[10] 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.

[11] 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).

[12] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2978–2988.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[14] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. 933–944.

[15] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[16] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. 2008. Code reuse in open source software. *Management science* 54, 1 (2008), 180–193.

[17] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-Based Neural Code Generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*.

[18] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.

[19] 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.

[20] Emily Hill, Manuel Roldan-Vega, Jerry Alan Fails, and Greg Mallet. 2014. NL-based query refinement and contextualized code search results: A user study. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 34–43.

[21] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[23] 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).

[24] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering*. 401–412.

[25] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*. PMLR, 54–63.

[26] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[27] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT*. 4171–4186.

[28] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems* 35 (2022), 21314–21328.

[29] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *Proceedings of the ACM/IEEE 45nd International Conference on Software Engineering*.

[30] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 4159–25.

[31] Zhenhao Li, An Ran Chen, Xing Hu, Xin Xia, Tse-Hsun Chen, and Weiyi Shang. 2023. Are They All Good? Studying Practitioners' Expectations on the Readability of Log Messages. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[32] Zhenhao Li, Heng Li, Tse-Hsun (Peter) Chen, and Weiyi Shang. 2021. DeepLV: Suggesting Log Levels Using Ordinal Based Neural Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1461–1472.

[33] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 599–609.

[34] Chao Liu, Xin Xia, David Lo, Zhiwe 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.

[35] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 37–47.

[36] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.

[37] Shangqing Liu, Xiaofei Xie, Jingkai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search. *IEEE Transactions on Software Engineering* (2023).

[38] 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).

[39] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1355–1367.

[40] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via wordnet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 545–549.

[41] 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.

[42] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[43] 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.

[44] Chen Lyu, Ruyun Wang, Hongyu Zhang, Hanwen Zhang, and Songlin Hu. 2021. Embedding API dependency graph for neural code generation. *Empirical Software Engineering* 26 (2021), 1–51.

[45] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2011. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering* 38, 5 (2011), 1069–1087.

[46] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[47] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[48] Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2719–2734.

[49] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[50] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*. 419–428.

[51] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 3982–3992.

[52] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[53] 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.

[54] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1715–1725.

[55] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[56] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1433–1443.

[57] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 269–280.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[59] Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 14015–14023.

[60] 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. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[61] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 334–345.

[62] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 344–354.

[63] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.