When LLMs Meet API Documentation: Can Retrieval Augmentation Aid Code Generation Just as It Helps Developers?

Jingyi Chen*
jchenix@connect.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Songqiang Chen*
i9s.chen@connect.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Jialun Cao†
jcaoap@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Jiasi Shen†
sjs@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Shing-Chi Cheung
scc@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Abstract

Retrieval-augmented generation (RAG) has increasingly shown its power in extending large language models' (LLMs') capability beyond their pre-trained knowledge. Existing works have shown that RAG can help with software development tasks such as code generation, code update, and test generation. Yet, the effectiveness of adapting LLMs to fast-evolving or ***less common API libraries using RAG*** remains unknown. To bridge this gap, we take an initial step to study this unexplored yet practical setting – when developers code with a less common library, they often refer to its API documentation; likewise, ***when LLMs are allowed to look up API documentation via RAG, to what extent can LLMs be advanced?*** To mimic such a setting, we select four less common open-source Python libraries with a total of 1017 eligible APIs. We study the factors that affect the effectiveness of using the documentation of less common API libraries as additional knowledge for retrieval and generation. Our intensive study yields interesting findings: (1) RAG helps improve LLMs' performance by 83%∼220%. (2) Example code contributes the most to advance LLMs, instead of the descriptive texts and parameter lists in the API documentation. (3) LLMs could sometimes tolerate mild noises (typos in description or incorrect parameters) by referencing their pre-trained knowledge or document context. Finally, we suggest that developers pay more attention to the quality and diversity of the code examples in the API documentation. The study sheds light on future low-code software development workflows.

Keywords

Retrieval-Augmented Generation, API Documentation, Library, API Usage Recommendation, Large Language Models

1 Introduction

Retrieval-augmented generation (RAG) [29] has recently been introduced as a method to broaden the pre-trained knowledge of Large Language Models (LLMs) by enriching the initial prompts with relevant documents and knowledge via information retrieval [10]. Since RAG fits perfectly the domains where ***knowledge is constantly refreshed and can hardly be memorized by LLMs*** in time [10], it has increasingly been adopted for software development tasks such as code generation [3, 16, 46, 47, 54] and test generation [45].

However, the effectiveness of adapting LLMs to fast-evolving or less common API libraries using RAG remains unknown. Existing studies on RAG-based code generation were made in settings that either mimic updates of common libraries (*e.g.*, SciPy and TensorFlow) [47] or explore the upper limits of RAG enhancement in programming problems (*e.g.*, HumanEval [8] and LeetCode [22]) [54]. These settings, however, do not study the LLMs' coding capability using less common libraries for software development.

To bridge the gap, this paper discusses an unexplored yet practical setting: ***when LLMs generate code using less common libraries, to what extent can RAG contribute?*** This setting imitates a scenario when human developers use a less common library, they often code by referencing its API documentation [43]. To mimic such a setting, we select four less common open-source Python libraries with a total of **1017** eligible APIs. We study the factors that affect the effectiveness of using the documentation of less common API libraries as additional knowledge for ***retrieval*** and ***generation***. Specifically, we study the following five research questions (RQs).

RQ1: To what extent can the retrieved API documents augment initial prompting and thus enhance the LLMs' generation? We consider three scenarios: ❶ *Worst case:* prompting without API documentation, *i.e.*, imitating developers coding with less common libraries without referencing API documentations; ❷ *Best case:* prompting with only the target API document, *i.e.*, imitating developers using an API by referring to the correct API documentation; and ❸ *Practical case:* prompting with multiple API documents, *i.e.*, imitating developers referencing multiple API documents simultaneously when they have not settled down on a specific API. We also examine the differences in LLM effectiveness across these scenarios and summarize the root causes of LLM failures under each scenario.

RQ2: How do different components in an API document contribute to retrieval and generation? An API document typically includes function descriptions, parameter lists, and code examples. This RQ explores the contribution of each component to retrieval and augmentation. Also, due to varying document quality,

*Co-first authors.
†Corresponding authors.

we introduce mild noises in different sections (*e.g.*, replacing API names in the description and adding a non-existent parameter) to observe the impact of various noises on retrieval and generation results. The study aims to provide suggestions on which components of an API document developers should pay more attention to.

RQ3: What is the impact of retrieval methods on retrieval results? Since RAG acts by searching for relevant queries in external knowledge bases (*i.e.*, API documents in this paper) through information retrieval, the impact of retrieval methods is worth exploring. Intuitively, different retrieval metrics suit different data types (*e.g.*, code, natural language). Yet, it remains unclear which is better for solving coding tasks that involve less common API libraries.

RQ4. How effective is RAG for popular Python libraries? In practice, a developer may not be familiar with all APIs in popular libraries. Likewise, LLMs may not be well trained with all APIs of a popular library, particularly the less commonly used ones. Therefore, this RQ examines RAG's effects on retrieving and generating 139 APIs within a popular library, Pandas.

RQ5. How does the effectiveness of RAG change when the example code provided in the API document fails to adequately cover the intended usage scenario? In many cases, developers may find only a basic code example that cannot cover the intended usage scenario of the interested API [34, 43, 44]. This RQ studies how different examples of an API (*e.g.*, code examples with different parameters or using default parameters) affect the effectiveness of the generation.

Our study yields several interesting findings. First, RAG can effectively advance LLMs to correctly use less common APIs by 83% ~ 220%. Besides, example code contributes the most to augment LLMs API usage, aligning with their helpfulness to human developers' [43]. It is also found that LLMs can tolerate mild noises by referencing common APIs and other contexts in documents. In addition, we identify BM25 as an effective retriever for matching the code completion task to the useful API documents. We reveal that RAG also benefits LLMs on the popular Pandas library. Finally, we suggest that developers prepare diverse code examples in API documentation and enhance LLMs' reasoning to avoid insufficient example codes that mislead LLMs. These findings share the experience of enabling LLMs to code with less common APIs, shedding light on the future low-code software development workflow [3].

Our contribution is summarized as follows:

- **Novelty** – We present the first study on the impact of RAG in the setting where LLMs are leveraged to solve coding tasks involving less common libraries. The setting yields pragmatic observations for AI-assisted software coding.
- **Significance** – We formulate how to use API documentation in RAG for code generation. The intensive experimental results yield interesting findings and actionable advice for both researchers and developers. Our study sheds light on future low-code software development workflows.
- **Usefulness** – We identify the dominating component, *i.e.*, code examples, in the API documentation that contributes the most to the generation. We also suggest prioritizing the importance of different components within the API documentation, allowing developers to focus more on the most useful parts for RAG.

2 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) [29] is proposed to enable LLMs to handle tasks by leveraging external knowledge without fine-tuning LLMs. Specifically, RAG maintains a database of relevant domain-specific or latest information unfamiliar to LLMs. Given a query, RAG first runs *retrieval phase* to dynamically identify multiple pieces of information (also known as entities) relevant to the task from the database. Then, the Top-k entities are included as the context in the prompt to *augment the generation phase.*

In the context of documentation-based API usage recommendation, RAG works for a coding requirement query q on a database of API documents $\mathbb{D} = \{d_1, d_2, ..., d_n\}$, where d_i represents the document of an API unfamiliar to the LLM. It executes two phases:

Retrieval Phase. Given the query q as the input to a RAG system, the retriever computes similarity scores between q and each document $d_i \in \mathbb{D}$ to rank all documents. The RAG system adopts a parameter k that determines the number of top-ranked documents to use as relevant contextual information. The retrieval process can be formally represented as:

$$\textsc{Retriever}\left(\mathbb{D}, q, k\right) = \text{top}\left(\text{sorted}\left(\{d_1, d_2, ..., d_n\}, \text{sim}\left(q, \cdot\right)\right), k\right)$$

Augmented Generation Phase. A prompt is constructed based on top-k relevant documents as the contextual information and q as the query, with an instruction to clarify the logical relationship between them. The prompt is input to the generator LLM to get the recommended API usage, which can be formally represented as:

$$\text{Result} \longleftarrow \textsc{GeneratorLLM}\left(\langle \textsc{Retriever}\left(\mathbb{D}, q, k\right), q\rangle\right)$$

3 Study Design

3.1 Overview of Study

Figure 1 illustrates the pipeline of our study. We first construct a dataset of API usage recommendation tasks for the study. Specifically, we identify eligible Python libraries as subjects from GitHub to represent the libraries unfamiliar to LLMs (Section 3.2.1). Then, we extract API documents of the subject libraries into structural data (Section 3.2.2). Such data are used to construct the RAG database (Section 3.2.4) and formulate code completion tasks (Section 3.2.3).

Then, we leverage LLMs to complete these tasks using RAG and analyze their performance under different scenarios. Specifically, we construct the prompt for each task by specifying the code context and the expected output, with a succinct task instruction and the retrieved documents attached. The prompt follows that proposed by existing studies on code generation and RAG [54, 56, 58]. The generated codes are evaluated against the consistency of their execution results with the ground truth. All the prompts and scripts used by our study can be found in our released artifact [1].

3.2 Dataset Construction

3.2.1 Library Selection. We collect open-source Python libraries as subjects in our study due to the inaccessibility of representative proprietary libraries. Besides, considering the potential volatility of the new libraries and their documents, we select the relatively less common libraries with certain popularity on GitHub as subjects, whose popularity suggests a certain quality of code and documentation.

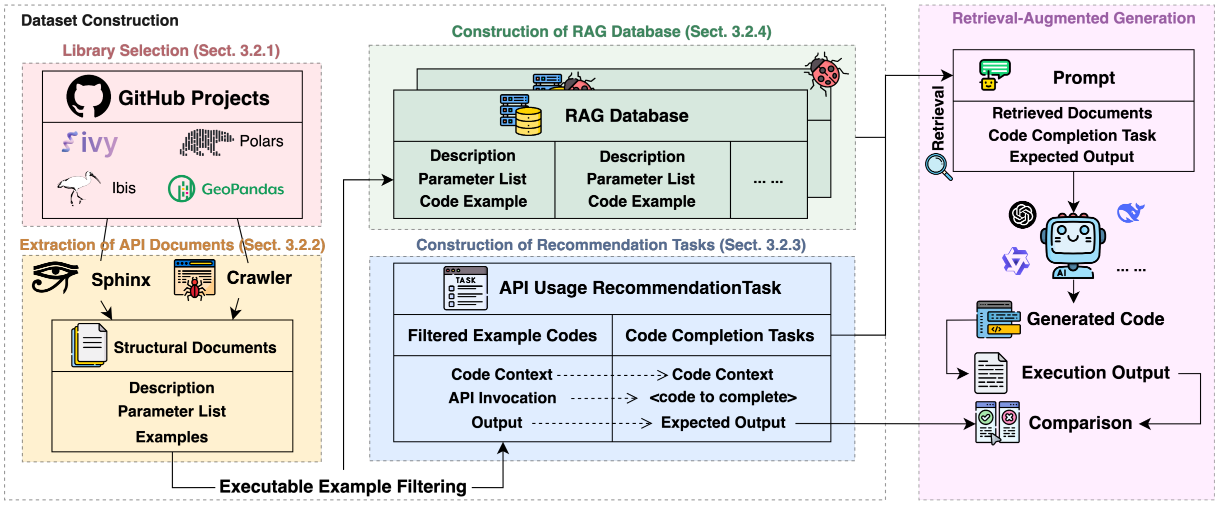


Figure 1: Overview of Study Pipeline

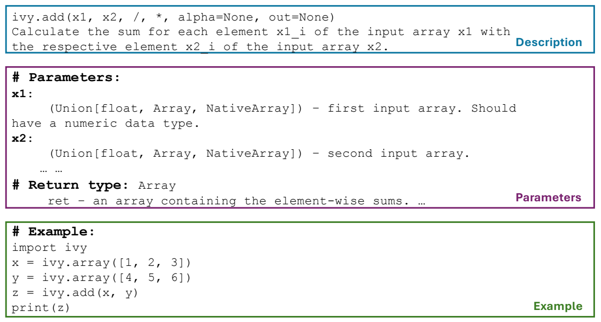
To this end, we use the 3.8k-starred *best-of-python* list [5] and 19.8k-starred *best-of-ml-python* list [4] on GitHub as the library source. They list well-recognized Python open-source libraries for various domains and the popular machine learning area, respectively. Then, we select the subject libraries with the following criteria:

❶ ***The libraries should provide structural documents illustrating API description, parameter list, and example codes.*** We observe that library API documents typically include a natural language description, a parameter list, and example codes [35]. We collect libraries whose documents demarcate the three components with clear sections. Such structural documents facilitate our investigation of the impact of different content in API documents.

❷ ***The libraries should include example codes for APIs.*** We build target coding tasks based on example codes in API documents. To prepare diverse tasks for study, we only keep the libraries that include more than 100 APIs with at least one coding example.

❸ ***The library should not rely on a complex runtime environment and its API outputs can be given in text.*** We prioritize the libraries that can run without a complicated runtime environment like specific hardware and a server to communicate. Besides, in this work, we formulate API usage recommendation as a code completion task and rely on the text representation of the output object to efficiently describe the expected execution result (which will be introduced in Section 3.2.3). Thus, we only keep the libraries whose API output objects have a clear textual representation. We discard the libraries whose API outputs are often hard to represent with clear text (*e.g.*, binary object, images, and no output).

❹ ***The library should be used in less than 100k GitHub code files.*** The criterion helps select libraries less common in open-source codes; thereby, they tend to be unfamiliar to LLMs. Specifically, we use a GitHub regular expression query "`lang:Python /(((import)|(from)).*[\s,]<library-name>[\s,])/`" to search the Python code files importing the library. We keep the libraries with fewer than 100k query results. LLMs tend to be less familiar with such libraries. In comparison, popular libraries used in existing studies [62] can be matched to millions of code files (*e.g.*, *Panads*: 5.3M, *NumPy*: 13.8M, *PyTorch*: 5.5M, *Scikit-learn*: 1.6M).

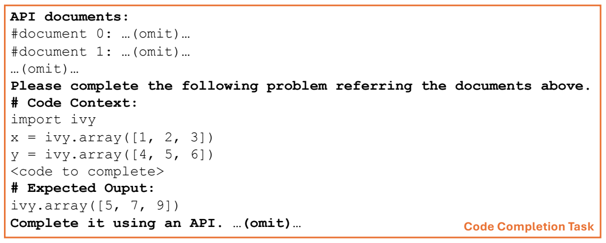


```
ivy.add(x1, x2, /, *, alpha=None, out=None)
Calculate the sum for each element x1_i of the input array x1 with
the respective element x2_i of the input array x2.
```
Description

```
# Parameters:
x1:
     (Union[float, Array, NativeArray]) - first input array. Should
have a numeric data type.
x2:
     (Union[float, Array, NativeArray]) - second input array.
    … …
# Return type: Array
     ret - an array containing the element-wise sums. …
```
Parameters

```
# Example:
import ivy
x = ivy.array([1, 2, 3])
y = ivy.array([4, 5, 6])
z = ivy.add(x, y)
print(z)
```
Example

Figure 2: Extracted API Document Content of `ivy.add` API[1]



```
API documents:
#document 0: …(omit)…
#document 1: …(omit)…
…(omit)…
Please complete the following problem referring the documents above.
# Code Context:
import ivy
x = ivy.array([1, 2, 3])
y = ivy.array([4, 5, 6])
<code to complete>
# Expected Ouput:
ivy.array([5, 7, 9])
Complete it using an API. …(omit)…
```
Code Completion Task

Figure 3: A Code Completion Task for `ivy.add`

The first three criteria help select libraries that provide enough eligible data; the last helps select less common libraries. Following these criteria, we select four libraries, *i.e.*, *Polars* [40], *Ibis* [20], *GeoPandas* [15], and *Ivy* [21]. The first three are data analysis utilities: *Polars* and *GeoPandas* are designed for common and geospatial data, respectively, and *Ibis* works as an abstraction layer over various SQL and DataFrame backends. *Ivy* unifies APIs of machine learning libraries like PyTorch and TensorFlow [28].

3.2.2 *Extraction of API Documents.*

To build the RAG database and formulate code completion tasks, we parse the API documents of

[1] Extracted from the Ivy API documentation at https://www.docs.ivy.dev/docs/functional/ivy/elementwise/ivy.functional.ivy.elementwise.add.html#ivy.add.

the subject libraries and store them in a structural format. Specifically, for each API of a given library, we extract the *description*, *parameter list*, and *example code* from its document. Figure 2 shows an extracted API document including the three components, with each annotated in a box. The *description* contains a function signature of the API and several natural language sentences to describe the API function. The *parameter list* elaborates all the input parameters and output values of the API, with their function, type, and default value (if any) introduced. The *example codes* illustrate typical usage of the API with code snippets, where the code often initializes relevant input values and invokes the API with certain parameter values to realize representative functions. We implement a crawler and a parser to collect the API description, parameter list, and example codes from the official API documentation of the subject libraries.

3.2.3 Construction of Recommendation Tasks. In this work, we formulate API usage recommendation as a code completion task. Specifically, we provide LLMs with the code context (*e.g.*, several code lines to prepare the inputs of invocation) and the expected output, and require LLMs to complete the code snippet with an invocation of the API from a library. LLMs should suggest an API invocation whose execution result conforms with the expected output to pass the code completion task. We consider code completion since it is a common scenario of LLM-assisted development [49]. Besides, code completion tasks can be easily built based on example codes in API documentation and do not require additional information like authentic natural language specifications that may not be easy to collect for less common libraries. The expected output is given in the prompt to clarify the intended function following Lai et al. [25] and the idea of Programming-by-Example [55].

Following the idea, we construct code completion tasks based on example codes we collected from documents. Specifically, given an example code from the API document, we first extract the code line invoking the target API as the goal for LLMs to complete. Then, we collect the codes before the goal as code context. The code lines after the invocation of the target API are removed since the execution result of the target API invocation is generally adequate for assessing the correctness of the suggested API usage on our subjects. Besides, to prepare the expected output for the code completion tasks, we run the whole example code and record the textual representation of the output object. As Section 3.2.1 mentions, the output of our subject libraries' APIs can be represented in text strings. The strings are included in the prompt to LLMs along with the code context.

Figure 3 illustrates a code completion task (and the prompt skeleton) we build based on the example code in Figure 2. Specifically, we collect the first three lines from the example code as the code context and extract the fourth line as the ground truth. Executing the whole example code yields `z`'s value "`ivy.array([5, 7, 9])`", which we include in the prompt as the expected output. The prompt follows the design in existing studies that generate codes based on expected output with LLMs [56]. The prompts used in our study are available at our artifact [1].

During the construction of tasks, we make necessary edits to make the code examples executable, *e.g.*, inserting `import` statements. We remove the tasks where the target APIs are not executable after a quick fix or do not produce any output able to be represented in text. All remaining completion tasks are checked to be executable and deterministic via multiple executions. Besides, when the target API appears in multiple statements in one code example, we manually split them into multiple examples. We formulate one code completion task per API based on a randomly picked example.

Table 1: Statistics on Subject Libraries

Library	Stars	Matched import	Eligible APIs	Eligible Example Codes	Doc Tokens (per API)
Polars [40]	32.3k	42.8k	330	341	206.4
Ibis [20]	5.6k	14.8k	330	469	140.2
GeoPandas [15]	4.7k	84.5k	119	206	309.4
Ivy [21]	14.0k	30.5k	238	1015	279.1
(total)	-	-	*1017*	*2031*	-

3.2.4 Construction of RAG Database. We build an individual RAG database for each library. Each entity in the database is the document of an API in the library. Each API corresponds to only one document (entity) in the database. To study the impact of noises on RAG performance (RQ2), we build a new database with the mutated documents for each library. Besides, as introduced in Section 1, for RQs1-4, we store documents with the example used for the code completion task to explore the upper limits of RAG following Wang et al. [54]; for RQ5, we use another example code that produces an output inconsistent with the code completion task. The documents in RAG do not record the output of example codes.

3.2.5 Statistics. Table 1 summarizes the features of the constructed dataset for our study. It includes four subject libraries with 1017 eligible APIs in total. We extract 2031 example codes for the eligible APIs, with most API documents providing only one example code. As Section 3.2.4 mentions, we provide one example in each document in the RAG database. The dataset is constructed based on the latest version of the libraries in 2025-02. We release the full dataset in our artifact [1].

3.3 Selection of Retrievers and LLMs

Retrievers: We consider three representative retrieval methods, including a sparse retriever, ***BM25*** [42], and two retrievers based on dense embeddings, text-embedding-3-large (*abbr.* ***Text3***) [37] and gte-Qwen2-7B-instruct (*abbr.* ***GTE***) [32]. BM25 identifies relevant documents based on TF/IDF. It has been proposed for code generation and retrieval [58, 59, 62]. Besides, we follow Zhao et al. [62] to retrieve documents with Text3, which embeds a text into a 3072-dimensional vector. We do not use MiniLM as they do as MiniLM can only handle short texts. Instead, we supplement GTE, a state-of-the-art open-source model on the MTEB embedding leaderboard [36]. It embeds a text into a 3584-dimensional vector.

LLMs: We consider four state-of-the-art LLMs from three families on the BigCodeBench leaderboard [65]. For GPT family, we adopt ***GPT-4o-mini*** [38] due to its popularity and efficiency. We do not consider GPT-4o since we use it to implement some mutation. Besides, we consider three open-source LLMs, *i.e.*, Qwen2.5-Coder-32B-Instruct (*abbr.* ***Qwen32B***) and Qwen2.5-Coder-7B-Instruct (*abbr.*

Qwen7B) from Qwen family [19] and DeepSeek-Coder-V2-Lite-Instruct (*abbr.* ***DSCoder***) from DeepSeek family. They perform well on BigCodeBench and are deployable on our machines.

3.4 Mutation Operators for API Documents

RAG shows varying effectiveness on external information of different quality [7]. To understand the importance of each document component and the harm of various noises, we compare RAG performance on mutated documents with certain noise. We design seven mutation operators to simulate noise in practical development.

3.4.1 Operators on Document Content. As Section 3.2.2 mentions, API documents typically record *description*, *parameter list*, and *example code* of APIs. However, missing specific content in API documents is a critical and common practical issue [2]. Besides, different content may provide varying helpfulness for LLMs. Thus, we design three mutation operators to remove each content from documents.

Delete Description *(abbr. DelDesc)*: We delete the *description* from documentation while retaining *parameter list* and *example code.*

Delete Parameter List *(abbr. DelParam)*: We delete the *parameter list*, while retaining *description* and *example code.*

Delete Example Code *(abbr. DelExmpl)*: We delete the *example code*, while retaining *description* and *parameter list.*

3.4.2 Operators on API Names. API documents may accidentally mention a wrong name of the API [27, 63]. To understand the impact of this issue on RAG for API usage recommendation, we design two mutation operators to change the API name in documents. Note that we do not change the API name in library codes.

Add Prefix *(abbr. AddPrefix)*: We prefix original API names with "`func_`" in documents. The new (wrong) API name includes the entire original (correct) API name.

Replace with Synonyms *(abbr. ReplSyn)*: We replace the original API name with its synonym. The synonym is suggested by the widely-used WordNet library [41]. For the words that WordNet fails to provide a synonym, we employ GPT-4o to suggest a synonym as a legitimate API name. All API name synonyms suggested by GPT-4o were manually verified for reliability.

Note that incorrect API names may exist in both natural language components (*i.e.*, *description* and *parameter list*) and code component (*i.e.*, *example code*) [63]. To enable a fine-grained analysis, we separately mutate names in natural language and code components. Take the document in Figure 2 as an example, we apply *AddPrefix* on its natural language components to transform the API name "`ivy.add`" in the function signature into "`ivy.func_add`". We also apply *AddPrefix* on its example code to transform the API invocation "`ivy.add(x, y)`" into "`ivy.func_add(x, y)`".

3.4.3 Operators on API Parameters. Improper parameter setup is a common issue in API usage [17]. Inspired by this, we designed two operators to mutate the parameter information in API documents. The operators need to generate appropriate parameter descriptions or suggest a new appropriate parameter list. Thus, we implement them with GPT-4o instead of a deterministic method to efficiently handle diverse cases. We adopt GPT-4o since it shows strong effectiveness in text revision and is widely used for data preparation[6, 31, 53, 64].

Table 2: API Usage Recommendation Pass Rates of LLMs w/o and w/ referring to API Documents

LLM	Setup	Polars	Ibis	GeoPandas	Ivy	Pandas
Qwen 32B	*w/o doc*	0.2545	0.2303	0.2437	0.5084	0.6187
	w/ top5 doc	0.7667*(+201%)*	0.6697*(+191%)*	0.6639*(+172%)*	0.8151*(+60%)*	0.8921*(+44%)*
	w/ tgt doc	0.8848*(+248%)*	0.8758*(+280%)*	0.7899*(+224%)*	0.8992*(+77%)*	0.9640*(+56%)*
GPT-4o -mini	*w/o doc*	0.2606	0.2758	0.2521	0.4076	0.6187
	w/ top5 doc	0.5758*(+121%)*	0.6273*(+127%)*	0.6273*(+149%)*	0.7647*(+88%)*	0.8273*(+34%)*
	w/ tgt doc	0.7576*(+191%)*	0.8515*(+209%)*	0.8571*(+240%)*	0.9160*(+125%)*	0.9281*(+50%)*
DS Coder	*w/o doc*	0.1061	0.0485	0.1345	0.3487	0.4532
	w/ top5 doc	0.6485*(+511%)*	0.4485*(+825%)*	0.6387*(+375%)*	0.7227*(+107%)*	0.7410*(+63%)*
	w/ tgt doc	0.8818*(+731%)*	0.6152*(+1169%)*	0.8655*(+544%)*	0.9118*(+161%)*	0.9568*(+111%)*
Qwen 7B	*w/o doc*	0.1182	0.2182	0.2353	0.3908	0.5612
	w/ top5 doc	0.3727*(+215%)*	0.5364*(+146%)*	0.5714*(+143%)*	0.7269*(+86%)*	0.7410*(+32%)*
	w/ tgt doc	0.4818*(+308%)*	0.6273*(+188%)*	0.7395*(+214%)*	0.9034*(+131%)*	0.7914*(+41%)*
(avg)	*w/o doc*	0.1848	0.1932	0.2164	0.4139	0.5629
	w/ top5 doc	0.5909*(+220%)*	0.5705*(+195%)*	0.6253*(+189%)*	0.7574*(+83%)*	0.8004*(+42%)*
	w/ tgt doc	0.7515*(+307%)*	0.7424*(+284%)*	0.8130*(+276%)*	0.9076*(+119%)*	0.9101*(+62%)*

Note: Results on less-used Polars, Ibis, GeoPandas, Ivy are discussed in RQ1; Pandas is discussed in RQ4.

Add a Parameter *(abbr. AddParam)*: We prompt GPT-4o to add a new parameter into the function signature and explain it in the parameter list accordingly. For example, GPT-4o inserts a parameter "`beta`" for the API in Figure 2, creating the signature `ivy.add(x1, x2, /, *, beta, alpha=None, out=None)`" and insert the explanation "`beta: (Union[int, float])-additional...`" to the parameter list.

Rename Parameters *(abbr. RenmParam)*: We prompt GPT-4o to redesign and rename all the API parameters across the function signature and parameter list. For instance, for the API in Figure 2, GPT-4o transforms its signature to `ivy.add(array1, array2, /, *, scalar=None, output=None)`" and updates the parameter list into (`array1: ... array2: ... scalar: ... output: ...`") accordingly.

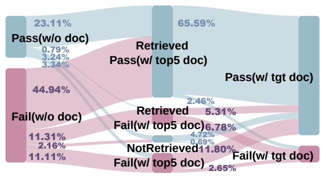
Even if parameters exist in *description*, *parameter list* and *example code*, in this work, we only rename parameters in natural language *description* and *parameter list* with GPT-4o. Codes may not tolerate the noise of LLM hallucination and need more precise mutation.

3.5 Experimental Setup

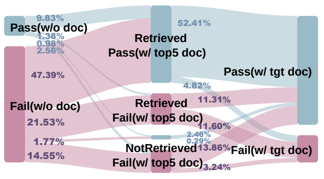
3.5.1 LLM Configuration. We run LLMs with a temperature of 0.0 to collect their most-confident response. We invoke GPT-4o-mini via its API and run the other open-source LLMs on servers equipped with NVIDIA RTX3090 and RTX4090 GPUs with 24GB VRAM.

3.5.2 Metrics. We assess the retrieval result based on the widely-used Recall@k rate [62]. The metric reflects the ratio of tasks where the correct document is ranked within Top-k.

For the final recommendation result, we measure the pass rate, *i.e.*, the ratio of tasks where RAG yields a correct API usage [62]. To assess the correctness of the recommended API usage, we extract the code segment enclosed within triple backticks (‴) from LLM response and concatenate it to the code context of the code completion task. Then, we execute the whole code snippet and collect the execution result. The LLM is considered to pass the task only if the execution result is consistent with the result of ground truth [14].



(a) QWen32B



(b) DSCoder

Figure 4: Transition from Fail to Pass on *w/o doc, top5 doc, corr doc* on Four Less-Common Libraries

4 Study Results

4.1 RQ1: Overall Effectiveness of RAG

4.1.1 Setup. We compare the pass rates of LLMs when referencing no external information (*w/o doc*), the top-5 documents retrieved by BM25 (*w/ top5 doc*), and the document of the target API (*w/ tgt doc*). As introduced in Section 1, setup *w/o doc* simulates the *worst case* where LLMs directly suggest recommendations based on pre-trained knowledge. Setup *w/ top5 doc* simulates the *practical case* of an end-to-end RAG. We consider Top-5 documents followed by the setup suggested in earlier studies [23, 54, 62]. We use BM25 in this RQ since it is robust in domain adaptation [42] and found most effective in our study (discussed in RQ3 in Section 4.3). We also include setup *w/ tgt doc* to simulate the *best case* where we give LLMs the document of target API to reveal the performance based on an ideal retriever that accurately retrieves correct documents following Wang et al. [54].

4.1.2 Usefulness of Documents in RAG. As the last three rows in Table 2 show, the average pass rate of LLMs improves significantly by 83%~220% after referring to the Top-5 documents retrieved with BM25 (*w/ top5 doc*) compared to not using RAG (*w/o doc*) on Polars, Ibis, GeoPandas, and Ivy. Specifically, the stronger Qwen32B and GPT-4o-mini suggest correct API usage for 66%~82% and 58%~76% tasks, respectively. Qwen7B works worst in general, but it still solves more than 53% of tasks except on *Polars*. The results demonstrate that **RAG allows LLMs to learn API usage from the retrieved documents and complete code correctly.**.

Upon Ideal Retrieval: We also analyze the pass rates under setup *w/ tgt doc*, which reveal RAG performance with an ideal retriever. It is found that Qwen32B, GPT-4o-mini, and DSCoder effectively solve around or over 85% of tasks most of the time. Such results confirm the helpfulness of RAG for LLMs to learn the correct invocation of unfamiliar library APIs to solve coding tasks.

4.1.3 Tracing Improvement. We trace the passes and failures on different setups to understand how RAG helps LLMs. As shown in Figure 4, RAG enables LLMs to work out numerous tasks that they fail *w/o doc*. Specifically, on 44.94% and 47.39% tasks, Qwen32B and DSCoder fail *w/o doc* while succeed *w/ top5 doc*. On setup *w/ tgt doc*, Qwen32B and DSCoder work out another 11.80% and 13.86% tasks, respectively, where they fail due to missing retrieval of correct documents *w/ top5 doc*. **The results again confirm the significant helpfulness of RAG and suggest enhancing retrievers as a direction to enlarge benefit.** Meanwhile, RAG hinders successful recommendation on only a few tasks, *e.g.*, 4.13% (0.79%+3.34%) and 3.94% (1.38%+2.56%) where Qwen32B and DSCoder can solve *w/o doc*, respectively. The results on GPT-4o-mini and Qwen7B show consistent conclusions and can be found in our artifact [1].

4.1.4 Failure Cases. LLMs still fail to solve a few tasks with RAG. For example, Figure 4 shows that failure of RAG with Qwen32B and DSCoder on 11.80% and 13.86% tasks is due to ***missing retrieval of correct documents***. Besides, even if the correct API documents have been retrieved within Top-5, LLMs still fail on some tasks. We manually summarized three failure patterns based on these cases, expecting that they may inspire enhancement of RAG on our task.

❶ *LLMs are confused by the retrieved documents and select a wrong API.* The issue causes most failures when the correct document is retrieved within Top-5. For example, for a task of Polars `starts_with` API, BM25 additionally retrieves `ends_with` and another three APIs and Qwen32B wrongly selects `ends_with`. But it solves the task on setup *w/ tgt doc*. The issue may be tackled by enhancing retrieval accuracy to reduce noisy context and improving LLMs in API selection.

❷ *LLMs hallucinate a wrong API usage.* Even on setup *w/ tgt doc*, weaker LLMs may hallucinate an incorrect API usage. For example, for a Polars task expecting `s.rolling_quantile(quantile=0.33,window_size=3)`, DSCoder wrongly returns `s.rolling_quantile(quantile=0.33, window_size=3, min_periods=1)`; while Qwen32B succeeds. Enhancing LLMs in document understanding may help solve the issue.

❸ *LLMs struggle to use APIs with complicated parameters.* Even on setup *w/ tgt doc*, LLMs may not set parameters correctly when the API has a few complex parameters. For example, `polars.Expr.ewm_std` has eight parameters. Qwen32B wrongly uses `spam` but not the desired `com` parameter even if `com` is already illustrated in the example. In fact, all studied LLMs cannot use this API correctly. A method to help LLMs distinguish complex parameters may solve the issue.

4.1.5 Implications. We summarize insights based on the observed overall effectiveness of RAG on API usage recommendation.

❶ RAG enables LLMs to solve code completion tasks by correctly invoking unfamiliar library APIs based on API documents.

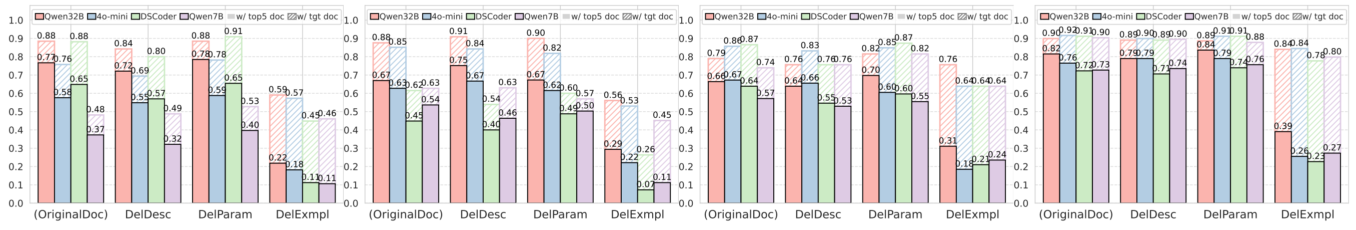
❷ LLMs often use APIs correctly when target API documents are retrieved. A few wrong API uses of RAG are due to failing to retrieve documents of correct APIs or being misled by high-ranking documents of irrelevant APIs. Improving retrieval accuracy to rank the target API document higher should effectively enhance RAG.

❸ LLMs may still be confused by candidate APIs and complicated parameters. They should be enhanced towards better discrimination on APIs and understanding of complex API parameter setup.

4.2 RQ2: Effectiveness of RAG on Documents of Different Quality

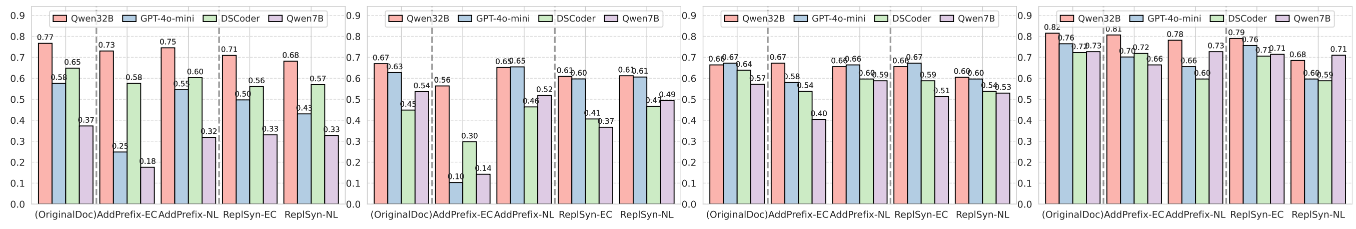
4.2.1 Setup. In this RQ, we compare API usage recommendation pass rates of RAG on documents with different noise. We still report the result with the BM25 retriever since it is found to be most effective in general (based on RQ3 results in Section 4.3).

4.2.2 Impact of Lacking Content. We first study the performance on the BM25-retrieved Top-5 documents with certain content deleted (solid bars in Figure 5). Obviously, the pass rates of LLMs decrease most after we remove example codes (*DelExmpl*) from documents. For example, the pass rates of Qwen32B decrease from 0.66~0.82 to 0.22~0.39 on the four libraries. The result shows the essential



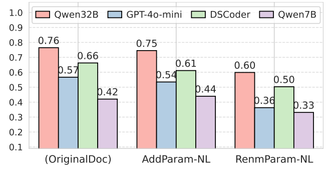
(a) Polars (b) Ibis (c) GeoPandas (d) Ivy

Figure 5: API Usage Recommendation Pass Rates of RAG on Documents Lacking Different Contents (Solid and shaded bars illustrate results under setups w/ top5 doc and w/tgt doc, respectively, as introduced in Section 4.1.1).

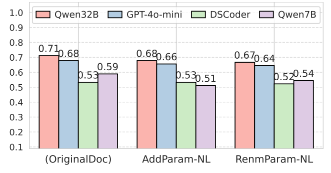


(a) Polars (b) Ibis (c) GeoPandas (d) Ivy

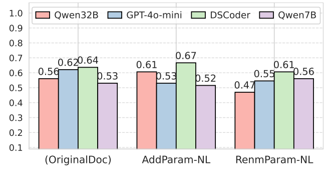
Figure 6: API Usage Recommendation Pass Rates of RAG on Documents with Different Noise on API Names (Postfixes "-EC" and "-NL" refer to the mutation on example codes and natural language description and parameter list, respectively.)



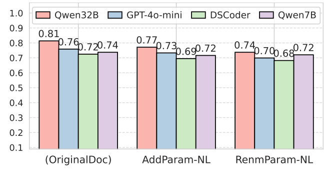
(a) Polars



(b) Ibis



(c) GeoPandas



(d) Ivy

Figure 7: RAG Performance on Documents with Different Noise on Parameters (on the APIs with at least one parameter)

impact of example codes on the success of the whole RAG system. *Interestingly, the finding aligns with human developers' heavy reliance on example codes to learn API usage [43].*

To understand the pure impact on LLM code generation regardless of retrieval phase, we further check LLM performance on target API documents (shaded bars in Figure 5). *DelExmpl* also leads to the largest performance drop in this situation, *e.g.*, the pass rates of Qwen32B decrease from 0.79~0.90 to 0.56~0.84. The results confirm the importance of examples for LLMs to learn API usage.

Meanwhile, lacking description (*DelDesc*) causes only slight performance change. Interestingly, removing the parameter list (*DelParam*) even slightly improves generation (as well as retrieval, which will be discussed in RQ3). The reason may be that LLMs can already learn API usage based on the example codes and descriptions, and removing the parameter list alleviates distraction.

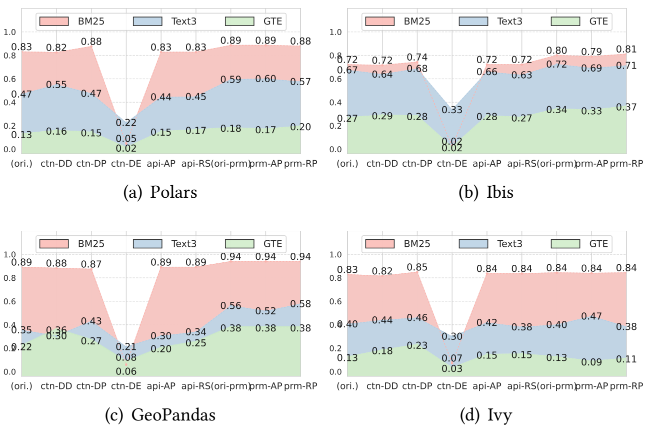
Project-wise: We notice that *DelExmpl* has a slighter hindrance of RAG on Ivy than on the other three libraries. This is likely because Ivy APIs mirror the functions in popular libraries like PyTorch and TensorFlow since Ivy aims to unify the deep learning interfaces [28]. As a result, once the correct Ivy API is identified, LLMs may infer its usage by referencing similar APIs in familiar libraries like PyTorch and TensorFlow. This suggests LLMs tend to be more robust when using unfamiliar APIs that resemble popular ones.

4.2.3 *Noises at API Names and Parameters.*

In this section, we study RAG performance on documents with noises at API names and parameters. We mainly discuss the performance on the BM25-retrieved top-5 documents (*w/ top5 doc*) in this RQ since the noisy API names and parameter information disturb BM25 slightly (discussed in RQ3) and the results on target API documents show similar trends.

Noise at API Names: We first compare the impacts of adding a prefix to the API names in the example code (*AddPrefix-EC*) and in the natural language description and parameter list (*AddPrefix-NL*). It is found that noisy API names in example codes (-EC) lead to a larger decrease in pass rates. The results suggest LLMs prefer to follow EC when there is a minor inconsistency between API names in NL and EC. Meanwhile, the impacts differ slightly between *ReplSync-EC* and *ReplSync-NL*, with noise in NL components being slightly more destructive. This indicates that LLMs do not manifest a clear preference for API names in NL and EC when the inconsistent names are quite different in syntax. LLMs may wisely select the correct name based on other descriptive contexts in documents.

Noise at Parameter Info: We observe that the impact of two mutations on parameters in NL components is minor in general.



(a) Polars (b) Ibis (c) GeoPandas (d) Ivy

Figure 8: Recall@5 of Retrievers against Different Noise

Table 3: Relative Performance Change against Different Noise on Less-Used Libraries and Popular Pandas Library

LLM	Library	ctn-DE	ctn-DP	ctn-DD	api-AP (nl)	api-AP(ec)	api-RS(nl)	api-RS(ec)	prm-AP	prm-RP	(avg)
Qwen 32B	*avg.* 4 libs	-58%	3%	0%	-3%	-5%	-11%	-5%	-2%	-13%	-11%
	Pandas	-36%	-2%	-4%	2%	0%	1%	-2%	-2%	-6%	-5%
GPT-4o -mini	*avg.* 4 libs	-68%	0%	3%	-3%	-37%	-14%	-3%	-6%	-14%	-16%
	Pandas	-44%	3%	5%	-3%	-1%	-8%	-6%	-3%	-4%	-7%
DS Coder	*avg.* 4 libs	-75%	1%	-10%	-8%	-13%	-12%	-8%	-2%	-10%	-15%
	Pandas	-51%	1%	7%	1%	-7%	1%	-7%	5%	0%	-6%
Qwen 7B	*avg.* 4 libs	-67%	0%	-7%	-3%	-37%	-7%	-13%	-4%	-5%	-16%
	Pandas	-43%	-4%	0%	-1%	-20%	-2%	-5%	-11%	-6%	-10%

Relative Change Rate = $(Perf_{NoisyDoc} - Perf_{OriDoc})/Perf_{OriDoc}$. Detailed values available at artifact [1]. "avg. 4 libs" refers to average results on Polars, Ibis, GeoPandas, and Ivy. Full results available at [1].
Prefixes *ctn-*, *api-*, *prm-* refer to operators on contents, API names, and parameters, respectively; the last two letters are the initials of operator names.

This suggests the robustness of LLMs in tolerating such noise. Specifically, even if we introduce a new parameter, LLMs may wisely learn from the example code that the new parameter is optional to realize the goal. Therefore, *AddParam-NL* may not seriously mislead LLMs. Meanwhile, Python does not necessarily require explicitly mentioning parameter names in invocation; thus the impact of parameter renaming is not dramatic either. However, we observe that LLMs sometimes explicitly mention the names of adopted parameters, leading to a more obvious performance drop on *RenmParam-NL*.

4.2.4 Implications. Based on findings on various noisy documents, we summarize advice on API document preparation for RAG.

❶ Example codes are vital to the success of RAG in API usage recommendation, aligning with their helpfulness to human developers [34, 43]. RAG users should provide examples in API documentation to teach LLMs API usage. (RQ5 will further discuss the preparation of example codes.)

❷ LLMs prefer following examples when natural language components conflict with examples slightly. RAG users should carefully examine the quality of example code. Besides, explicitly mentioning optional information in examples, *e.g.*, correct parameter names and values, may guide LLMs to learn correct API usage.

❸ LLMs can tolerate noises in documents by referencing context or APIs familiar to them. Prompting LLMs to verify the answer with various information may enhance their robustness against noise in API documents.

4.3 RQ3: Performance of Different Retrievers

4.3.1 Setup. We compare retrievers using recall@k rate as mentioned in Section 3.5.2. We set k to be 5 as in earlier RQs. Besides, we also study the Mean Reciprocal Rank (MRR), which provides a single-score summary of retrieval performance across different ks. The MRR results show consistent findings and can be found at [1].

4.3.2 Comparison of Retrievers. As shown in Figure 8, BM25 shows a generally much higher Recall@5 rate than Text3 and GTE except against deleting examples (*ctn-DE*). Specifically, except on *ctn-DE*, BM25 shows 0.72~0.94 recall@5 rates; while Text3 scores only 0.21~0.72 and GTE performs worse. The result aligns with findings in [18, 58] that BM25 is effective on code. The reason may be that for the requirement described with code context and expected output, matching codes literally with BM25 is more helpful than considering the semantics Text3 and GTE compress in a vector.

4.3.3 Noise-wise Comparison. As shown in Figure 8, among different noise, deletion of example code content (*ctn-DE*) decreases retrieval accuracy a lot. In this case, three retrievers can retrieve correct documents for no more than 33% tasks within Top-5. The results reveal the importance of example codes for accurate retrieval. Besides, deleting the parameter list (*ctn-DP*) slightly increases the accuracy of all retrievers. The reason may be that the parameter list offers little information about the general features of APIs and may introduce noise by contrast.

4.3.4 Implications. In general, BM25 works most effectively in our setup, demonstrating that "Code Context + BM25-based RAG" as a potentially promising solution to API usage recommendation when there are reliable example codes in documents. This is reasonable since codes are often concise and structured, and thus easy to match to example codes and API information in documents literally.

4.4 RQ4: Comparison with Popular Library

4.4.1 Setup. We compare the effectiveness of RAG on the four less common libraries and the popular Pandas library. Pandas matches 5.4M GitHub codes with our query mentioned in Section 3.2.1. It is used as a subject in an existing study on RAG [62] and for code generation [25]. We extract 139 eligible Pandas APIs for study.

4.4.2 Consistent and Generalizable Findings. We observe some consistent findings on the less common libraries and Pandas. Specifically, Table 2 shows that RAG also helps improve pass rates on Pandas. For example, Qwen32B works out another 44% and 56% tasks with the help of *top5 doc* and *tgt doc*, respectively. Besides, Table 3 shows that the removal of example codes (*ctn-DE*) also causes large performance drops of RAG on Pandas, demonstrating the importance of example codes to the success of RAG on Pandas.

The similar findings show the generalizability of our conclusions and implications. RAG together with well-prepared examples also effectively helps LLMs use APIs from well-learned libraries.

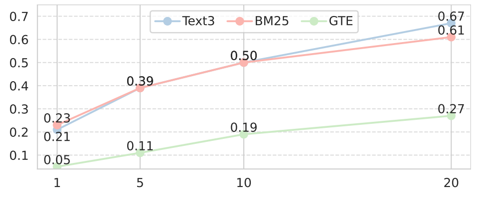


Figure 9: Recall@k on Tasks Different From Example Code

Table 4: Pass Rates on Tasks Different From Example Code

	w/o doc	*w/ tgt doc (Original Doc)*	*w/ tgt doc (DelExmpl)*	*w/ tgt doc (OnlyExmpl)*
Qwen32B	0.5564	0.8346	0.8421	0.7744
GPT-4o-mini	0.4361	0.7895	0.8271	0.7444
DSCoder	0.4060	0.6917	0.7970	0.6165
Qwen7B	0.4436	0.6617	0.7820	0.6241
R1-Qwen32B	0.5789	0.8496	0.8346	0.7444

4.4.3 Unique Observations. Besides consistent findings, we observe differences between results on Pandas and four less common libraries. Specifically, Table 2 shows that LLMs without referencing documents (*w/o doc*) can already solve 56.29% tasks on Pandas on average, which is much higher than 18.48%~41.39% on four less common libraries. Besides, Table 3 shows that the pass rates of RAG decrease by 11%~16% on four less common libraries while only 5%~10% on Pandas. In particular, RAG is much more robust to noise of wrong API names, notably, *api-AP(ec)* and *api-RS(nl)*.

The results show that studying RAG on only popular libraries may hide some problems of RAG on less common libraries. They echo our selection of less common libraries as subjects.

4.5 RQ5: Effectiveness on Documents with Only Example Codes Mismatching Target Tasks

4.5.1 Setup. This RQ explores the effectiveness of RAG on API documents whose example code differs from the code completion task, aiming to shed light on the preparation of example codes in API documents for RAG. However, existing documents seldom provide multiple examples for each API [43]. Among our subjects, only Ivy offers more than one example for over 100 APIs. Thus, we study this RQ on the 133 Ivy APIs with multiple examples, by ensuring the example codes in documents produce an output inconsistent with the code completion task as mentioned in Section 3.2.4.

4.5.2 Retrieval Performance. As shown in Figure 9, BM25 still outperforms GTE and performs on par with Text3. However, its Recall@5 rate drops a lot on such documents compared to the documents including the target code completion task as the example.

4.5.3 Generation Performance. Considering the low retrieval rate, we study the generation performance on setup *w/ tgt doc*. As shown in Table 4, RAG still effectively increases the pass rates of four LLMs from 0.4060~0.5564 (*w/o doc*) to 0.6617~0.8346 (*w/ tgt doc*) on documents with examples mismatching the code completion tasks.

However, we observe deleting examples (*DelExmpl*) conversely enhances performance, notably for the less capable DSCoder and Qwen7B. Our analysis of failure cases reveals that LLMs often fail to identify differences between the code completion tasks and example codes and copy wrong usage patterns from the provided examples.

To understand whether example codes mismatching the completion task always hinder RAG performance, we first evaluate RAG on documents with only example code (*OnlyExmpl*, *i.e.*, removing description and parameter list from documents). Besides, we study the performance of DeepSeek-R1-Distill-Qwen-32B (*abbr.* R1-Qwen32B), which shows exceptional programming ability through reasoning.[2] Results show that all five LLMs achieve higher pass rates on *OnlyExmpl* than without document (*w/o doc*). Besides, R1-QWen32B performs better on *Original Doc* than on *DelExmpl*. By studying the reasoning process of R1-Qwen32B, we notice that it demonstrates the ability to adapt example codes to target tasks.

In fact, the mismatch between examples in documents and user goals has been identified as a hindrance for humans to learn API usage [43, 44]. Our results reveal that ***the issue also affects LLMs***.

4.5.4 Implications. Results confirm the usefulness of RAG on documents with examples mismatching the target task. Meanwhile, to enhance its effectiveness, we highly suggest RAG users illustrate more usage of each API by **introducing diverse example codes**, *e.g.*, to cover more parameter combinations. Then, retrievers may be able to identify target APIs based on similar code contexts shared by example codes and completion tasks. Meanwhile, diverse examples may force the generator LLMs to distinguish usages in examples as they help humans [43]. Besides, it is found that human developers can write a correct API usage by copying and adapting example codes [44]. Thus, enhancing reasoning ability may also help LLMs distinguish examples and learn API usage wisely.

We demonstrate a task where new diverse examples help LLMs solve it. The task expects `y = ivy.conv1d_transpose(x,filters,2,'SAME')`. Given the API description, parameter list, and a single example using `ivy.conv1d_transpose(x,filters,1,'VALID',out=x)`, Qwen32B incorrectly generates `ivy.conv1d_transpose(x,filters,strides=2,padding='VALID')` by copying "`VALID`". By adding four new examples where two use "`VALID`" and two use "`SAME`", Qwen32B successfully distinguishes between the two values and gives a correct solution `ivy.conv1d_transpose(x,filters,2,'SAME')`. This demonstrates the potential of diverse examples to help LLMs learn API usage wisely rather than directly copying. Furthermore, initially, Qwen32B also fails to recognize `strides` as a positional-only parameter (already specified in the API description) and incorrectly sets it by name. This error is also fixed after new examples are given. This further demonstrates the help of diverse examples in guiding LLMs to learn API usage, which may also address the failures mentioned in RQ1 (Section 4.1.4). The complete task and document can be found in our artifact [1].

5 Discussion

5.1 RAG for Code Generation from Natural Language Requirements

In our study, we recommend API usage in the code completion setup. In this discussion, we further explore the usefulness of RAG in another LLM-assisted development scenario, code generation,

[2]We didn't study R1-QWen32B in earlier RQs due to its much longer reasoning time and unique challenges, *e.g.*, overthinking without producing final answers. Studying reasoning LLMs' behavior in our task remains an interesting future work.

Table 5: Pass Rates of RAG for Code Generation

	w/o doc	*w/ top5 doc (Original Doc)*	*w/ top5 doc (DelExmpl)*	*w/ top5 doc (DelParam)*	*w/ top5 doc (DelDesc)*
Qwen32B	0.2697	0.4667	0.3394	0.4394	0.3879
GPT-4o-mini	0.2576	0.4667	0.3758	0.4364	0.3939
Qwen7B	0.1667	0.3939	0.2242	0.3364	0.2576
DSCoder	0.1636	0.3818	0.1667	0.3636	0.2606
(average)	*0.2144*	*0.4273*	*0.2765*	*0.3939*	*0.3250*
BM25 Recall@5	-	0.3091	0.3273	0.2303	0.1515

where users provide a natural language (NL) requirement to prompt LLMs to generate whole code snippets. The exploration aims to broaden insights on RAG for unfamiliar API use.

We did not find off-the-shelf NL requirements for programming tasks of unfamiliar APIs. Thus, we use GPT-4o to summarize an NL requirement for the example code (preprocessed as Section 3.2.3 mentions) in API documents considering its effectiveness in code summarization [50]. The whole API document is given as context during summarization. The prompt is available in our artifact [1]. We conduct the exploration on Polars and use BM25 retriever since it is also found effective in NL-based code generation [58].

As shown in Table 5, using RAG increases the pass rates of four LLMs from 0.2144 (*w/o doc*) to 0.4273 (*w/ top5 doc (Original Doc)*), demonstrating that RAG also enables LLMs to use unfamiliar APIs for code generation. Meanwhile, Recall@5 is only 0.3091, showing the known gap between NL requirement and API documentation [58]. Deleting examples (*Del Exmpl*) increases the Recall@5 from 0.3091 to 0.3273, but it decreases the final success rate from 0.4273 to 0.2765. This suggests that example code is vital for LLMs to learn API usage but may not benefit retrieval. *DelParam* and *DelDesc* hinder successful retrieval and decrease the final success rates.

To summarize, RAG also enables LLMs to use unfamiliar APIs for code generation. The generator LLMs still benefit from example codes; meanwhile, the retrievers face challenges such as bridging the semantic gap between NL requirements and API documents.

5.2 Threats to Validity

We identified potential threats to the validity of our study and have taken measures to mitigate them as follows:

Representativeness of subject libraries. This study explores the usefulness of RAG in recommending usages of APIs unfamiliar to LLMs. To balance the number of reliable documents and the unfamiliarity with LLMs, we collect four less common Python libraries used fairly infrequently in GitHub codes with four criteria. They are utilities of two major application domains of Python, data science and machine learning. They include 1017 eligible APIs. We start with Python libraries following existing studies [23, 54, 62]. Study results on these libraries show consistent conclusions with diversity (*e.g.*, better robustness on Ivy). Besides, they reveal both consistent and unique findings compared to the popular Pandas library. Based on these, we consider our study results representative and able to generalize to the practical libraries unfamiliar to LLMs.

Representativeness of studied RAG setups. We build a typical RAG pipeline with the popular *langchain* [26] framework [24]. RAG typically relies on a retriever and a generator LLM. We consider three popular retrievers found effective in existing studies [23, 54, 62] and leaderboards [36]. We study four LLMs from three families based on their outstanding performance on the BigCodeBench leaderboard [65]. The setup should be representative.

Representativeness of mutation. We design mutation operators to reveal RAG performance on documents with varying quality. All operators are inspired by noise in API documents identified by existing studies as Section 3.4 introduces. We adopt GPT-4o to suggest synonyms and prepare new parameters. Despite the uncertainty of LLMs, GPT-4o is widely used in task preparation due to its strong natural language processing ability [6, 12, 52, 53, 64]. Our manual check also suggests satisfying mutation results of GPT-4o.

6 Related Work

6.1 RAG for Code Generation and Completion

Several studies focus on enhancing retrieval accuracy for better RAG performance. Zhang et al. [61] designed a tailored retriever aware of syntax constraints. Li et al. [30] trained a retriever to select examples that the generator LLM prefers. RepoCoder [60] iteratively optimizes its retrieval results. ProCC [48] leverages three styles of code contexts to benefit retrieval. In our study, enhancing retrieval accuracy also proves beneficial for RAG on API documentation as we discuss in RQ1. Enhancing retrieval of proper documents remains an interesting future work.

Another line of studies diversifies information sources for RAG, e.g., in-domain code bases [13], cross-code-file context [13], web resources [47], and Android development feature [59]. Dutta et al. [14] generated code for low-resource languages based on language documentation and code examples, which act as a good knowledge base for uncommon domains. Unlike these works, we focus on RAG on API documentation to use less common APIs. In particular, we evaluate RAG in different setups. Our study results share experience in enabling LLMs to use unfamiliar library APIs.

6.2 Empirical Study on RAG Performance

To learn experience on the optimal setup for these essential configurations of RAG, researchers evaluated RAG under diverse document types [11], document positions [11, 33], and choice of retrievers [51]. There are also studies on the retrieval granularity [9], length of contexts [57], and integration with training-based methods [39]. Besides configuration, Chen et al. [7] evaluated RAG against noise and reveal that RAG may easily be misled by incorrect information in retrieved documents. These studies are mainly conducted on general applications like question answering. Inspired by them, we explore RAG for API usage recommendation and share experience on enabling LLMs to code with less common library APIs.

Recently, Zhao et al. [62] studied the optimal setup of retrievers and prompts for question answering and code generation with popular libraries like Pandas. In comparison, we study the impact of API documentation quality on both the retrieval and generation phases. Besides, we focus on the less common libraries. We reveal both consistent helpfulness and unique challenges of RAG on the less common libraries and the popular Pandas library.

Besides, Wang et al. [54] built CodeRAGBench to benchmark RAG on eight coding tasks. They mainly construct RAG databases with canonical solutions of coding tasks to explore the upper limits of RAG, as well as explore the limitation of RAG on open-domain resources. We follow their idea to benchmark RAG on the API documents that include the code completion task and its ground truth as examples in RQs1-4. With the setup, we identify the bottleneck in retrieval and LLM understanding ability. We also explore RAG performance on documentation with only an example mismatching the coding task in RQ5, and reveal the importance of diverse examples and reasoning ability to effectively guide LLMs.

7 Conclusion

In this paper, we study an unexplored yet practical question: *To what extent can RAG contribute when LLMs generate code using less common libraries?* To answer this question, we select four less common open-source Python libraries with a total of 1017 eligible APIs and conduct experiments on them. Our study yields several interesting findings, including the determining factors in improving LLMs' performance using RAG, the essential role of code examples to augment LLMs, and LLMs' noise-tolerant ability. Based on the results, we suggest developers pay more attention to the quality and diversity of the code examples in API documentation and enhance LLMs in reasoning to help them better learn API usage.

References

[1] [n. d.]. Artifact of this paper.

[2] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. 2020. Software documentation: the practitioners' perspective. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020.* ACM, 590–601. doi:10.1145/3377811.3380405

[3] Orlando Marquez Ayala and Patrice Béchard. 2024. Generating a Low-code Complete Workflow via Task Decomposition and RAG. *arXiv preprint arXiv:2412.00239* (2024).

[4] best-of-ml-python. [n. d.]. best-of-ml-python GitHub repository. https://github.com/ml-tooling/best-of-ml-python

[5] best-of-python. [n. d.]. best-of-python GitHub repository. https://github.com/ml-tooling/best-of-python

[6] Jialun Cao, Yaojie Lu, Meiziniu Li, Haoyang Ma, Haokun Li, Mengda He, Cheng Wen, Le Sun, Hongyu Zhang, Shengchao Qin, Shing-Chi Cheung, and Cong Tian. 2025. From Informal to Formal - Incorporating and Evaluating LLMs on Natural Language Requirements to Verifiable Formal Proofs. *CoRR* abs/2501.16207 (2025). doi:10.48550/ARXIV.2501.16207 arXiv:2501.16207

[7] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking Large Language Models in Retrieval-Augmented Generation. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada.* AAAI Press, 17754–17762. doi:10.1609/AAAI.V38I16.29728

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[9] Tong Chen, Hongwei Wang, Sihao Chen, Wenhao Yu, Kaixin Ma, Xinran Zhao, Hongming Zhang, and Dong Yu. 2024. Dense x retrieval: What retrieval granularity should we use?. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing.* 15159–15177.

[10] Florin Cuconasu, Giovanni Trappolini, Federico Siciliano, Simone Filice, Cesare Campagnano, Yoelle Maarek, Nicola Tonellotto, and Fabrizio Silvestri. 2024. The Power of Noise: Redefining Retrieval for RAG Systems *(SIGIR '24).* Association for Computing Machinery, New York, NY, USA, 719729. doi:10.1145/3626772.3657834

[11] Florin Cuconasu, Giovanni Trappolini, Federico Siciliano, Simone Filice, Cesare Campagnano, Yoelle Maarek, Nicola Tonellotto, and Fabrizio Silvestri. 2024. The power of noise: Redefining retrieval for rag systems. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval.* 719–729.

[12] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing Chat Language Models by Scaling High-quality Instructional Conversations. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023.* Association for Computational Linguistics, 3029–3051. doi:10.18653/V1/2023.EMNLP-MAIN.183

[13] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007* (2022).

[14] Avik Dutta, Mukul Singh, Gust Verbruggen, Sumit Gulwani, and Vu Le. 2024. RAR: Retrieval-augmented retrieval for code generation in low resource languages. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, 21506–21515.

[15] GeoPandas Developer Team. [n. d.]. GitHub Repository of GeoPandas Library. https://github.com/geopandas/geopandas

[16] Akhilesh Deepak Gotmare, Junnan Li, Shafiq Joty, and Steven CH Hoi. 2023. Efficient Text-to-Code Retrieval with Cascaded Fast and Slow Transformer Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 388–400.

[17] William G. J. Halfond and Alessandro Orso. 2008. Automated identification of parameter mismatches in web applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008.* ACM, 181–191. doi:10.1145/1453101.1453126

[18] Pengfei He, Shaowei Wang, Shaiful Chowdhury, and Tse-Hsun Chen. 2024. Exploring Demonstration Retrievers in RAG for Coding Tasks: Yeas and Nays! *CoRR* abs/2410.09662 (2024). doi:10.48550/ARXIV.2410.09662 arXiv:2410.09662

[19] Binyuan Hui and Jian Yang et al. 2024. Qwen2.5-Coder Technical Report. *CoRR* abs/2409.12186 (2024). doi:10.48550/ARXIV.2409.12186 arXiv:2409.12186

[20] Ibis Developer Team. [n. d.]. GitHub Repository of Ibis Library. https://github.com/ibis-project/ibis

[21] Ivy Developer Team. [n. d.]. GitHub Repository of Ivy Library. https://github.com/ivy-llc/ivy

[22] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *CoRR* abs/2403.07974 (2024). doi:10.48550/ARXIV.2403.07974 arXiv:2403.07974

[23] Nihal Jain, Robert Kwiatkowski, Baishakhi Ray, Murali Krishna Ramanathan, and Varun Kumar. 2024. On Mitigating Code LLM Hallucinations with API Documentation. *CoRR* abs/2407.09726 (2024). doi:10.48550/ARXIV.2407.09726 arXiv:2407.09726

[24] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *CoRR* abs/2406.00515 (2024). doi:10.48550/ARXIV.2406.00515 arXiv:2406.00515

[25] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202).* PMLR, 18319–18345.

[26] langchain. [n. d.]. Langchain homepage. https://www.langchain.com/

[27] Seonah Lee, Rongxin Wu, Shing-Chi Cheung, and Sungwon Kang. 2021. Automatic Detection and Update Suggestion for Outdated API Names in Documentation. *IEEE Trans. Software Eng.* 47, 4 (2021), 653–675. doi:10.1109/TSE.2019.2901459

[28] Daniel Lenton, Fabio Pardo, Fabian Falck, Stephen James, and Ronald Clark. 2021. Ivy: Templated Deep Learning for Inter-Framework Portability. *CoRR* abs/2102.02886 (2021). arXiv:2102.02886

[29] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.* doi:10.5555/3495724.3496517

[30] 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).

[31] Rui Li, Qi Liu, Liyang He, Zheng Zhang, Hao Zhang, Shengyu Ye, Junyu Lu, and Zhenya Huang. 2024. Optimizing Code Retrieval: High-Quality and Scalable

Dataset Annotation through Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*. Association for Computational Linguistics, 2053–2065.

[32] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards General Text Embeddings with Multi-stage Contrastive Learning. *CoRR* abs/2308.03281 (2023). doi:10.48550/ARXIV.2308.03281 arXiv:2308.03281

[33] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.

[34] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay Spinuzzi. 1998. Building More Usable APIs. *IEEE Softw.* 15, 3 (1998), 78–86. doi:10.1109/52.676963

[35] James Mertz. [n. d.]. Documenting Python Code: A Complete Guide. https://realpython.com/documenting-python-code/

[36] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2023. MTEB: Massive Text Embedding Benchmark. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*. Association for Computational Linguistics, 2006–2029. doi:10.18653/V1/2023.EACL-MAIN.148

[37] Arvind Neelakantan and Tao Xu et al. 2022. Text and Code Embeddings by Contrastive Pre-Training. *CoRR* abs/2201.10005 (2022). arXiv:2201.10005

[38] OpenAI GPT-4o-mini. [n. d.]. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[39] Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Elisha. 2023. Fine-tuning or retrieval? comparing knowledge injection in llms. *arXiv preprint arXiv:2312.05934* (2023).

[40] Polars Developer Team. [n. d.]. GitHub Repository of Polars Library. https://github.com/pola-rs/polars

[41] Princeton University. [n. d.]. WordNet homepage. https://wordnet.princeton.edu/

[42] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (2009), 333–389. doi:10.1561/1500000019

[43] Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Softw.* 26, 6 (2009), 27–34. doi:10.1109/MS.2009.193

[44] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empir. Softw. Eng.* 16, 6 (2011), 703–732. doi:10.1007/S10664-010-9150-8

[45] Jiho Shin, Reem Aleithan, Hadi Hemmati, and Song Wang. 2024. Retrieval-Augmented Test Generation: How Far Are We? *arXiv preprint arXiv:2409.12682* (2024).

[46] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. Arks: Active retrieval in knowledge soup for code generation. *arXiv preprint arXiv:2402.12317* (2024).

[47] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. EvoR: Evolving Retrieval for Code Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 2538–2554.

[48] Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *arXiv preprint arXiv:2405.07530* (2024).

[49] Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. 2024. How and Why LLMs Use Deprecated APIs in Code Completion? An Empirical Study. *CoRR* abs/2406.09834 (2024). doi:10.48550/ARXIV.2406.09834 arXiv:2406.09834

[50] Guoqing Wang, Zeyu Sun, Zhihao Gong, Sixiang Ye, Yizhou Chen, Yifan Zhao, Qingyuan Liang, and Dan Hao. 2024. Do Advanced Language Models Eliminate the Need for Prompt Engineering in Software Engineering? *CoRR* abs/2411.02093 (2024). doi:10.48550/ARXIV.2411.02093 arXiv:2411.02093

[51] Xiaohua Wang, Zhenghua Wang, Xuan Gao, Feiran Zhang, Yixin Wu, Zhibo Xu, Tianyuan Shi, Zhengyuan Wang, Shizheng Li, Qi Qian, et al. 2024. Searching for best practices in retrieval-augmented generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 17716–17736.

[52] Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. 2023. How Far Can Camels Go? Exploring the State of Instruction Tuning on Open Resources. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

[53] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*. Association for Computational Linguistics, 13484–13508. doi:10.18653/V1/2023.ACL-LONG.754

[54] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. CodeRAG-Bench: Can Retrieval Augment Code Generation? *CoRR* abs/2406.14497 (2024). doi:10.48550/ARXIV.2406.14497 arXiv:2406.14497

[55] Jiarong Wu, Lili Wei, Yanyan Jiang, Shing-Chi Cheung, Luyao Ren, and Chang Xu. 2024. Programming by Example Made Easy. *ACM Trans. Softw. Eng. Methodol.* 33, 1 (2024), 4:1–4:36. doi:10.1145/3607185

[56] Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Hengcheng Zhu, and Jialun Cao. 2024. MR-Adopt: Automatic Deduction of Input Transformation Function for Metamorphic Testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 557–569. doi:10.1145/3691620.3696020

[57] Peng Xu, Wei Ping, Xianchao Wu, Lawrence McAfee, Chen Zhu, Zihan Liu, Sandeep Subramanian, Evelina Bakhturina, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Retrieval meets long context large language models. In *The Twelfth International Conference on Learning Representations*.

[58] 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. *CoRR* abs/2501.13742 (2025). doi:10.48550/ARXIV.2501.13742 arXiv:2501.13742

[59] Xinran Yu, Chun Li, Minxue Pan, and Xuandong Li. 2024. DroidCoder: Enhanced Android Code Completion with Context-Enriched Retrieval-Augmented Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. ACM, 681–693. doi:10.1145/3691620.3695063

[60] 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. *arXiv preprint arXiv:2303.12570* (2023).

[61] Xiangyu Zhang, Yu Zhou, Guang Yang, and Taolue Chen. 2023. Syntax-aware retrieval augmented code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 1291–1302.

[62] Shengming Zhao, Yuheng Huang, Jiayang Song, Zhijie Wang, Chengcheng Wan, and Lei Ma. 2024. Towards Understanding Retrieval Accuracy and Prompt Quality in RAG Systems. *CoRR* abs/2411.19463 (2024). doi:10.48550/ARXIV.2411.19463 arXiv:2411.19463

[63] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, 803–816. doi:10.1145/2509136.2509523

[64] Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. 2024. DOMAINEVAL: An Auto-Constructed Benchmark for Multi-Domain Code Generation. *CoRR* abs/2408.13204 (2024). doi:10.48550/ARXIV.2408.13204 arXiv:2408.13204

[65] Terry Yue Zhuo and Minh Chien Vu et al. 2024. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *CoRR* abs/2406.15877 (2024). doi:10.48550/ARXIV.2406.15877 arXiv:2406.15877