# Optimizing Code Retrieval: High-Quality and Scalable Dataset Annotation through Large Language Models

Anonymous ACL submission

## Abstract

Code retrieval aims to identify code from extensive codebases that semantically aligns with a given query code snippet. Collecting a broad and high-quality set of query and code pairs is crucial to the success of this task. However, existing data collection methods struggle to effectively balance scalability and annotation quality. In this paper, we first analyze the factors influencing the quality of function annotations generated by Large Language Models (LLMs). We find that the invocation of intra-repository functions and third-party APIs plays a significant role. Building on this insight, we propose a novel annotation method that enhances the annotation context by incorporating the content of functions called within the repository and information on third-party API functionalities. Additionally, we integrate LLMs with a novel sorting method to address the multi-level function call relationships within repositories. Furthermore, by applying our proposed method across a range of repositories, we have developed the Query4Code dataset. The quality of this synthesized dataset is validated through both model training and human evaluation, demonstrating high-quality annotations. Moreover, cost analysis confirms the scalability of our annotation method. [1]

## 1 Introduction

Code retrieval aims to find the most relevant code snippet in a database given a user query, facilitating the reuse of programs in the software development process (Bui et al., 2021; Li et al., 2022) and driving recent research on retrieval-augmented code generation (Parvez et al., 2021; Zhou et al., 2022). To ensure good performance in practical applications, the key lies in collecting a wide range of high-quality, dual-modal pairing data between natural language queries and code snippets.

Query
```
How to export the content of a Jupyter Notebook file.
```

Docstring
```
Export content from a Jupyter notebook file.
Parameters: - nb_path : The file path of the
Jupyter notebook to be exported.
```

Code
```python
def export_nb(nb_path):
    exporter = PythonExporter()
    output, res = exporter.from_filename(nb_path)

    if 'outputs' in res:
        for filename, content in res['outputs'].items():
            savefile(filename, content)

    return output
```

Figure 1: Example of code snippet and corresponding query and docstring.

An efficient approach to collect code retrieval datasets involves directly gathering code data from online repositories (e.g., GitHub[2]) and processing it to extract code snippets along with their corresponding docstrings. As depicted in Figure 1, since the docstring serves as a description of the function code, it can be utilized as a query. However, a significant difference exists between the docstring and the user's query, resulting in a deviation from queries encountered in real-world scenarios. To bridge this gap and obtain queries that closely resemble those of actual users, some researchers (Heyman and Van Cutsem, 2020; Yin et al., 2018; Yao et al., 2018) tend to collect user questions and the corresponding code snippets from programming communities such as Stack Overflow[3]. Another approach explored by researchers (Rao et al., 2021; Huang et al., 2021) involves gathering user search queries from browser logs and subsequently enlisting experts to annotate corresponding code snippets based on these queries. Regrettably, the former approach often produces code snippets of inferior quality because of the presence of block and statement-level code within the community. On the other hand, the latter approach allows for the

[1] Our Code and Dataset is available at https://anonymous.4open.science/r/Query4Code-D5C0

[2] https://github.com

[3] https://stackoverflow.com

acquisition of a high-quality dataset but proves to be cost-prohibitive and challenging to scale. Therefore, we pose a question: **Can a more efficient, low-cost method be developed to obtain a high-quality code retrieval dataset?**

The formidable capabilities of Large Language Models (LLMs) present a remarkable opportunity. Firstly, previous research (Rodriguez-Cardenas et al., 2023) has demonstrated the profound code comprehension ability of LLMs in various code understanding tasks, such as code summarization (Geng et al., 2023). Secondly, existing LLMs, employing preference alignment techniques (Ouyang et al., 2022; Geng et al., 2023), can generate content that aligns with human preferences. In the domain of search, some studies (Bonifacio et al., 2022; Dai et al., 2022) have proposed generating the query from the documents, yielding highly promising outcomes. Hence, a straightforward approach is to employ LLMs to generate user-like queries from the code snippets. However, there are some differences between code snippets and traditional documents. For instance, **intra-repository function calls** refer to the calls between different functions within a repository project, as depicted in Figure 1. Function `export_nb` calls function `savefile`, which makes it challenging for LLMs to comprehend function `export_nb` if only provided as input, without considering the function `savefile` it calls. Additionally, **third-party API calls** involve invoking functions from external APIs, as shown in Figure 1. Function `export_nb` calls the third-party API `PythonExporter.from_filename`, and LLM needs to understand the functionality of this API for a better understanding of the function.

In this paper, we first analyze the main factors affecting the quality of annotations for functions in repositories. Through preliminary experiments on a development set from 100 selected repositories, we observe that the presence of intra-repository function calls exerts a substantial influence on the quality of annotations, with a greater number of call relationships resulting in a heightened degree of impact. Additionally, we uncover that infrequent third-party calls have the greatest impact on annotation quality. This observation may be attributed to the limited pretraining knowledge of LLMs regarding these external libraries. Based on these findings, we propose an annotation algorithm aimed at using LLMs for high-quality code retrieval query annotations. We start by parsing the relationships of intra-repository function calls and use a topological sorting approach to guide the LLM annotation sequence. For third-party function calls, we select third-party functions based on popularity and use web scraping to annotate features of unpopular third-party functions, adding this information to the annotation context.

To substantiate the efficacy of our annotation approach, we initially employed our method to obtain a large-scale code retrieval dataset **Query4Code**, which includes 237.2K queries and code pairs from 12.3K repositories. We use Query4Code a pretraining corpus for various code retrieval models. Subsequently, comprehensive evaluations on multiple real-world benchmarks confirmed that our method significantly enhances the performance of code retrieval models in real scenarios.

## 2 Related Work

### 2.1 Code Retrieval Datasets

The previous methods (Sedykh et al., 2023) of code retrieval data collection can be summarized into three categories: 1). Some researchers (Wang et al., 2023c) parse functions and corresponding docstrings from online repositories to form pairs. For example, Husain et al. (2019) collected 2.1M paired data of 6 programming languages from an open-source repository on GitHub, constituting the CodeSearchNet. 2). Others (Yin et al., 2018) gather questions posted by users on Stack Overflow along with the accepted code snippets to create datasets suitable for code searching. Heyman and Van Cutsem (2020) attempts this by collecting the most popular dataset posts on Stack Overflow and gathering code snippets from highly upvoted responses. 3). The use of manual annotation methods: Huang et al. (2021) initially collects human queries used in code searches from search engines and then manually gathers relevant code snippets from GitHub to match these queries.

However, these methods present a trade-off between data quality and scalability. Therefore, we propose a low-cost and scalable annotation method.

### 2.2 Code Retrieval Models

In token-level pre-training methods, CodeBERT (Feng et al., 2020) attempts to leverage the extensive programming and natural language bimodal data within repositories for pre-training. Building upon this, GraphCodeBERT (Guo et al., 2021) endeavors to incorporate data flow graph signals to devise new pre-training tasks, thereby enhancing

| Calls | Intra-repo | Third-party APIs |
|---|---|---|
| Max nums | 137 | 120 |
| Mean nums | 5.11 | 3.24 |
| Proportion | 46.5% | 53.5% |

Table 1: Statistics on the number and proportion of calls to intra-repository and third-party library APIs.

the understanding of code semantics. UniXcoder (Guo et al., 2022) introduces a unified cross-modal pre-training model specifically designed for programming languages. Recently, some studies have explored the use of contrastive learning approaches to augment code search tasks. ContraCode (Jain et al., 2021) and Corder (Bui et al., 2021) employ semantic-preserving variation techniques for data augmentation and utilize contrastive learning objectives to distinguish between similar and dissimilar code snippets. CodeRetriever (Li et al., 2022) attempts to combine unimodal and bimodal contrastive learning to train code search models.

### 2.3 LLM in Data Annotation

Given the strong generalization capabilities exhibited by Large Language Models (LLMs), they apply across multiple domains (Samuel et al., 2023; Wang et al., 2023a) for data synthesis, facilitating the transfer of rich knowledge from larger models to smaller ones. In Unnatural Instructions (Honovich et al., 2023) and Self-Instruct (Wang et al., 2023b), LLMs utilize to generate the instructional datasets required during the fine-tuning phase. Samuel et al. (2023) utilize a minimal set of original data to guide LLMs in generating datasets required for reading comprehension tasks. West et al. (2022) propose a two-step process for symbolic knowledge distillation rather than the creation of content-related datasets. In the field of information retrieval, Zhang et al. (2023) and Wang et al. (2023a) utilize LLMs to generate positive and negative samples during the training process of contrastive learning.

This paper is the first to use LLMs to annotate code retrieval dataset, focusing on the key factors that affect LLMs in generating queries: library calls and third-party API calls.

## 3 Preliminary Analysis

The direct use of LLMs for annotating functions often results in a lack of contextual information about the annotated functions. Therefore, This section attempts to analyze the impact of intra-repository calls and third-party API calls on LLM annotated queries. Experiments are conducted using the GPT-3.5-turbo (Achiam et al., 2023) and CodeLlama-Instruct 7B (Roziere et al., 2023) models, with all prompts and detailed information being provided in Appendix A.

### 3.1 Setup

Based on the selection of high-quality repositories identified from prior research (Husain et al., 2019), we randomly chose 100 repositories to form our development set. Subsequently, we employ the tree-sitter[4] library to parse code files within these repositories, acquiring all function-level code snippets and their invocation relationships. These relationships are further categorized into intra-repository calls and third-party API calls.

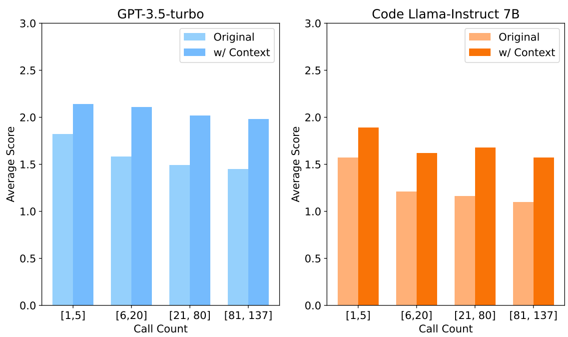


Figure 2: The impact of calls within repositories of varying quantities on the quality of query annotations.

### 3.2 Impact of Intra-Repository Function Calls

Due to the existence of multiple functions in the repository, these functions are usually involved in complex call relationships. After parsing, from Table 1, we can observe the proportion of functions with call relationships, as well as the average and maximum call frequencies. We observe that 46.5% of the code has call relationships, and the maximum number of calls can reach 137 times. This highlights the widespread use of function calls in the repository. Subsequently, we analyze the impact of these call relationships on the quality of final query annotations generated by LLMs. We use two annotation methods: direct annotation and adding calling function context for annotation. After obtaining the final annotated results, we pair annotated queries with code and used the GPT-4-turbo model to score (0-3) and evaluate the quality of generated queries. The final results are shown in Figure 2, from which we observe that including

[4] https://tree-sitter.github.io

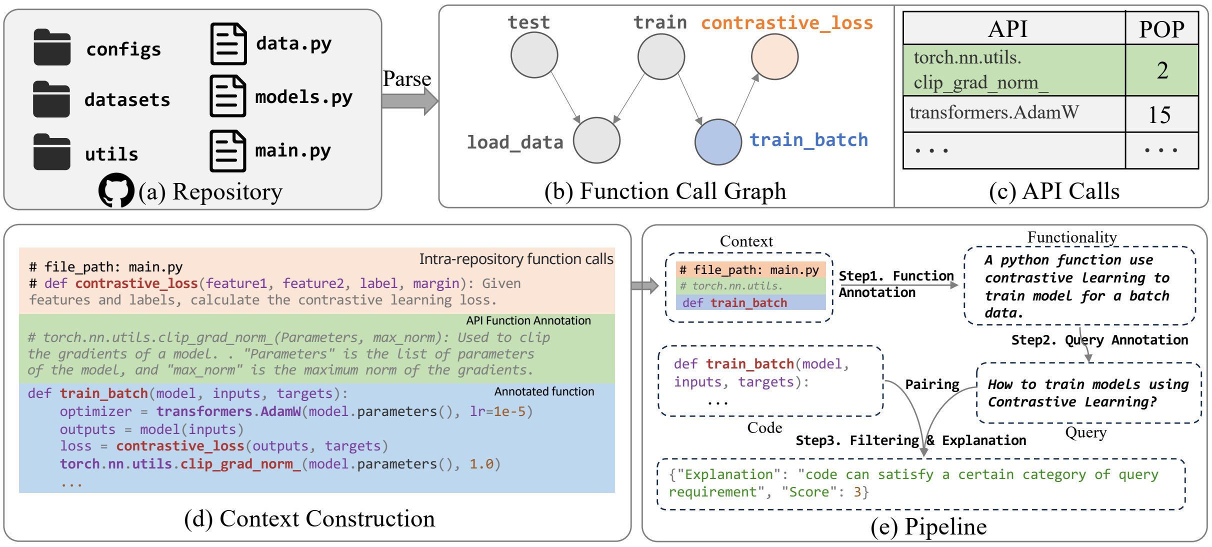


Figure 3: The overview of our annotation method. (a) Files in the repository. (b) Function call graph obtained from parsing. (c) API calls obtained from parsing and their corresponding popularity. (d) Construct annotated context based on call relationships and current API calls. (e) Pipeline for annotation method.

information about called functions significantly affects annotation quality. Furthermore, more call relationships will lead to a greater degree of influence, and model capability also significantly affects the quality of final annotations.

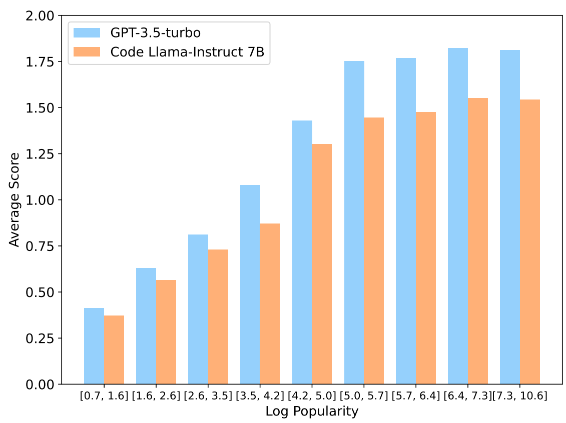


Figure 4: The impact of third-party APIs with Different Popularity Levels on LLM Understanding.

### 3.3 Impact of Third-Party APIs Calls

After analyzing the invocation of third-party APIs in functions, as shown in Table 1, we observe that 53.5% of the functions involve third-party API calls, with the maximum number of calls reaching 120 times. We next examine the impact of third-party APIs on annotation quality. Inspired by previous research (Mallen et al., 2023), we consider that the impact of APIs on annotation quality is closely related to the API's popularity. Therefore, we initially use the frequency of API calls in the repositories as a proxy for API popularity. We then annotate functions in our development set using LLMs, including all available API documentation. GPT-4-turbo is used to compare LLM explanations of API functions against the actual API documentation, with results categorized according to popularity. Our findings, presented in Figure 4, show that LLMs often lack a comprehensive grasp of many API details, particularly for unpopular APIs. This phenomenon adversely affects the quality of LLM annotations for queries. And even for models with stronger performance (e.g., gpt-3.5-turbo), the understanding of low-popularity APIs is also poor.

## 4 Approach

### 4.1 Overview

In the preceding analysis, we demonstrate how the invocation relationships within a repository and those in third-party libraries can impact the quality of Large Language Models (LLMs) in annotating queries. As shown in Figure 3, we attempt to propose an annotation method to address these issues. We endeavor to collect information about functions with invocation relationships, as well as functionalities of unpopular APIs, and incorporate them into the annotation context. Then, we use this context to prompt LLMs to generate queries (see the prompt in Appendix B).

### 4.2 Task Decomposition

Inspired by previous research work (Wei et al., 2022), a complex task can be simplified by decomposing it into multiple simpler tasks, thereby easing the model's inference load. For the task of query annotation, we consider that the model first needs to understand the code of the currently annotated function and then generate queries that a user might write during the development process based on this understanding of code semantics. As shown in Figure 3 (e), we initially use LLMs for code interpretation and then proceed to annotate queries based on the interpretation and the content of the code snippets:

$$s = LLM(c), q = LLM(s, c). \tag{1}$$

In the code interpretation stage, we mainly rely on the LLM's understanding of the code, while in the query generation stage, the alignment capability of LLMs with human intent is primarily utilized.

### 4.3 Analyzing Function and API Calls

Since in Section 3, we have analyzed that the main factors affecting the quality of LLM annotations for queries are function calls within the repository and third-party API calls. Therefore, as shown in the upper of Figure 3, for a given repository, we first use the tree-sitter tool to parse all functions in the code files within the repository. Then, we analyze each function's calls to other intra-repository functions and third-party APIs separately.

### 4.4 Annotation Algorithm Based on Function Call Graph

Having established the function invocation relationships within the repository, a straightforward approach would be to include the relevant context of the function to be annotated along with the query into the LLM's input context. However, as shown in Figure 3 (b), there are multi-level call relationships between functions in the repository. Understanding the train function requires knowing the `train_batch` function because it calls the `train_batch` function, which then calls the `contrastive_loss` function. Similarly, to grasp the `train_batch` function properly, it's essential to understand the `contrastive_loss` function. Directly incorporating all functions into the context would pose challenges associated with multi-level reasoning.

Thus, we propose a novel annotation algorithm based on topological ordering. The intuition behind this algorithm is the decoupling of multi-level invocation relationships into single-level relationships. Specifically, we first construct a directed graph $G(V, E)$ of function calls, where each node $v \in V$ represents a function in the repository. If function A is called by function B, there will be a directed edge $e \in E$ from $v_A$ to $v_B$. Based on topological sorting, we first annotate functions without dependency relationships. During the annotation process, when encountering recursive calls, we randomly delete an edge to continue with the annotation. Subsequently, we annotate functions with invocation relationships, thus breaking down multi-level invocation relationships into single-level relationships. For the annotation context of the function currently being annotated, it is only necessary to include information about its directly called functions. We summarized the algorithm in Appendix C.

### 4.5 Collection of Third-Party API Documentation Based on Popularity

In Section 3, our analysis indicates that LLMs struggle to understand unpopular APIs. Therefore, we aim to add descriptions of unpopular third-party API functionalities in the annotation context. As shown in figure 3 (c), first, we need to assess the popularity of APIs, using the frequency of API calls in the repository as a basis for popularity. Our analysis concludes that LLMs understand APIs better if they exceed a popularity threshold. Therefore, we set a popularity threshold and for third-party APIs below this threshold in the function, we use the DuckDuckGo[5] search engine to look up documentation and employ LLM to summarize the API functionalities. Then, we add this information into the annotation context.

### 4.6 Data Filtering

To further enhance the quality of generated queries and improve the explainability of the annotation process, we attempt to incorporate a reverse validation and an explanation phase for the query and code snippet pairs into the annotation framework. Specifically, as shown in figure 3 (e), after completing the annotation to obtain aligned query and code snippet pairs, we first use LLMs for reverse validation. Inspired by Huang et al. (2021), we notice that the code in the annotated query-code pairs cannot fully answer the query. It may exceed, partially satisfy, or completely fail to meet the query requirements. Specifically, we focus on the following four scenarios: 1) If the code can answer and exceed

[5]https://duckduckgo.com

| Dataset | Training | Validation | Test |
|---|---|---|---|
| CoSQA | 19.0K | 0.5K | 0.5K |
| SO-DS | 14.2K | 0.9K | 1.1K |
| StaQC | 20.4K | 2.6K | 2.7K |
| CoNaLa | 2.8K | - | 0.8K |
| WebQueryTest | - | - | 1.0K |

Table 2: The statistics of benchmark datasets.

the query requirements, it is considered a correct answer. 2) If the code can satisfy certain categories of query requirements, it is also deemed a correct answer. 3) If the code satisfies less than 50% of the query requirements, it cannot correctly answer the query. 4) The code has almost no relevance to the query. Based on this principle, we construct the `CLS` prompt language model to obtain classification results:

$$f(q, c) = LLM(q, c, \texttt{CLS}). \quad (2)$$

Then, we will filter out the code snippets of categories 1 and 2 from the original constructed dataset $C$ to obtain $C_{\text{filtered}}$:

$$C_{\text{filtered}} = \{c \in C \mid f(q, c) \in \{1, 2\}\}. \quad (3)$$

## 5 Experiment

### 5.1 Annotation

To facilitate comparison, we followed the selection of GitHub repositories in **CodeSearchNet** (Husain et al., 2019), choosing only Python repositories for cost reasons. We then applied a certain method to filter high-quality functions within these repositories. Subsequently, we used the GPT-3.5-turbo model to generate queries using the annotation method mentioned above. Ultimately, we successfully annotated a total of 237.2K pairs of natural language and code snippets, forming the **Query4Code** dataset.

### 5.2 Model Validation

To validate the quality of the Query4Code dataset, which we obtain through our final annotation process, we pre-train existing pre-trained code representation models using both the CodeSearchNet and Query4Code. We aim to evaluate model performance across multiple real-world code retrieval benchmarks in a zero-shot setting. Furthermore, we fine-tune the models on real-world datasets to assess the adaptability of the Query4Code dataset to downstream benchmarks.

#### 5.2.1 Baseline

To compare the performance differences when pre-training with the CodeSearchNet and Query4Code datasets, we pre-trained the following code representation models using different datasets and conducted a performance comparison:

- CodeBERT (Feng et al., 2020) is a bimodal pre-trained model that is pre-trained through two tasks: Masked Language Modeling (MLM) and Replaced Token Detection (RTD).
- GraphCodeBERT (Guo et al., 2021) proposes two structure-based pre-training tasks (data flow edge prediction and node alignment) to enhance code representation.
- UniXcoder (Guo et al., 2022) proposes to enhance code representation using cross-modal content such as AST and code comments.
- StarEncoder (Li et al., 2023) is pre-trained on The Stack (Kocetkov et al., 2022) dataset, using MLM and Next Sentence Prediction (NSP) as the pretraining tasks.

#### 5.2.2 Benchmark and Metric

In order to evaluate the performance of the model in real-world code retrieval scenarios, we have selected a wide range of benchmarks for validation. Among them, the datasets CoNaLa (Yin et al., 2018), SO-DS (Heyman and Van Cutsem, 2020), and StaQC (Yao et al., 2018) are collected from Stackoverflow questions, and queries in CoSQA (Huang et al., 2021) and WebQueryTest (Lu et al., 2021) are collected from web search engines. Therefore, the queries in these datasets are closer to real code search scenarios. The statistics of benchmark datasets are listed in Table 2. Following prior research works (Kanade et al., 2020; Li et al., 2022), we employed Mean Reciprocal Rank (MRR) (Hull, 1999) as the evaluation metric:

$$MRR = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{rank_i}, \quad (4)$$

where $rank_i$ is the rank of the correct code snippet related to the i-th query.

#### 5.2.3 Training Objective

Given a paired query $q$ and code $c^+$ pair, we adopt the contrastive learning InfoNCE objective function commonly used in existing code retrieval tasks

| Model | CoNaLa | | SO-DS | | StaQC | | CoSQA | | WebQueryTest | |
|---|---|---|---|---|---|---|---|---|---|---|
| | CSN | Q4C | CSN | Q4C | CSN | Q4C | CSN | Q4C | CSN | Q4C |
| ***Zero-Shot*** | | | | | | | | | | |
| CodeBERT | 21.65 | **25.45** | 18.42 | **18.98** | 14.26 | **15.74** | 56.34 | **59.80** | 32.43 | **35.61** |
| GraphCodeBERT | 23.70 | **28.88** | 19.01 | **21.56** | 16.90 | **18.72** | 56.83 | **60.24** | 31.83 | **35.97** |
| UniXCoder | 25.47 | **29.07** | 18.78 | **19.85** | 16.45 | **19.07** | 55.22 | **58.87** | 30.18 | **34.42** |
| StarEncoder | 25.72 | **28.14** | 17.31 | **19.65** | 15.55 | **18.59** | 54.27 | **58.41** | 31.46 | **35.80** |
| ***Fine-Tuning*** | | | | | | | | | | |
| CodeBERT | 22.41 | **26.83** | 23.24 | **25.76** | 23.75 | **25.39** | 67.72 | **72.91** | - | - |
| GraphCodeBERT | 25.01 | **29.15** | 24.05 | **25.92** | 24.41 | **25.84** | 67.35 | **73.64** | - | - |
| UniXCoder | 26.27 | **29.96** | 23.59 | **25.90** | 23.38 | **26.10** | 68.47 | **73.30** | - | - |
| StarEncoder | 26.05 | **29.58** | 24.31 | **26.83** | 24.07 | **25.29** | 67.41 | **72.65** | - | - |

Table 3: Compare the zero-shot and fine-tune performance of code representation models pre-trained on CodeSearchNet (CSN) and Query4Code (Q4C) datasets.

for model training. Furthermore, we employ an in-batch negative sampling approach for selecting negative samples $c^-$ in contrastive learning:

$$\mathcal{L} = -\mathbb{E}\left[\log \frac{\exp\left(q \cdot c^+\right)}{\exp\left(q \cdot c^+\right) + \sum_{j=1}^{N} \exp\left(q \cdot c_j^-\right)}\right], \quad (5)$$

where $N$ represents batch size.

#### 5.2.4 Implementation details

All experiments are implemented using PyTorch. During the pre-training phase, for all settings related to model architecture and hyperparameters, we follow the original paper. During the fine-tuning phase, to adapt to variations between different datasets, we conducte a grid search on the downstream dataset to find the learning rate, setting the range in our experiments as {1e-5, 2e-5, 5e-5}, and utilize the AdamW optimizer (Loshchilov and Hutter, 2017). The options for batch size included {32, 64, 128}. Training is set for 10 epochs and to prevent overfitting, we adopte an early stopping strategy. The experiments described in this paper are conducted with three random seeds: 0, 1, and 2, and we will report the average results in the paper. All experiments meet the p<0.01 significance threshold. Experiments are conducted on a GeForce RTX 4090 GPU.

#### 5.2.5 Results

**Zero-shot Performance** The final zero-shot experimental results, as shown in Table 3, indicate that pre-training on the Query4Code dataset significantly enhances performance compared to pre-training on the CodeSearchNet dataset, with improvements observed across multiple code representation models. Additionally, we note substantial performance gains on both the CoSQA and WebQueryTest datasets. We attribute this improvement to the fact that the queries in these two datasets were extracted from logs of real-world search engines, which closely match the distribution of our annotated queries. Conversely, the improvement on the SO-DS dataset was minimal, likely due to a greater disparity between the code snippets in the SO-DS dataset and our annotated dataset.

**Fine-tuning Performance** In the fine-tuning experiment, it is worth noting that since the WebQueryTest dataset is specifically designed for assessing real-world code retrieval task performance without available training data, its related results were not reported. The final experiments demonstrate that pretraining with the Query4Code dataset before fine-tuning yielded superior performance across all other datasets, confirming that models pretrained through Query4Code exhibit enhanced adaptability in real-world code retrieval scenarios.

### 5.3 The potential of the dataset

| | $C_{qc}$ | $C_{sc}$ | $C_{qc}$+$C_{sc}$ |
|---|---|---|---|
| CoNaLa | 25.45 | 23.28 | **26.39** |
| SO-DS | 18.98 | 19.35 | **20.17** |
| StaQC | 15.74 | 15.92 | **16.51** |
| CoSQA | 59.80 | 58.46 | **61.93** |
| WebQueryTest | 35.61 | 35.07 | **36.55** |

Table 4: Using different data pairs with Query4Code to train CodeBERT for zero-shot performance.

Although this paper mainly focuses on generating annotations for query retrieval of code, our two-stage annotation method can obtain functional summaries of functions. We are interested in whether

**Code**

```python
def escape_shell_arg(shell_arg):
    if isinstance(shell_arg, six.text_type):
        msg = "ERROR: escape_shell_arg() expected string argument but " \
            "got '%s' of type '%s'." % (repr(shell_arg), type(shell_arg))
        raise TypeError(msg)
    return "'%s'" % shell_arg.replace("'", r"'\''")
```

**Docstring**

```
"""Escape shell argument shell_arg by placing it within single-quotes. Any single quotes found within the shell argument string will be escaped.
@param shell_arg: The shell argument to be escaped.
@type shell_arg: string
..."""
```

**Query**

```
Python code for shell argument escaping with single quotes
```

Figure 5: Example of code snippet with docstring and annotated query.

the functional summary of functions can enhance the ability of the current code retrieval model. As shown in Table 4, compared with only using $(q, c)$ pairs (denoted as $C_{qc}$) for contrastive learning, using only $(s, c)$ pairs (denoted as $C_{sc}$) achieved comparable performance and performed better on the SO-DS and CoSQA datasets. Furthermore, utilizing both annotated query $q$ and summary $c$ data achieved the best performance. For detailed experimental settings, please refer to Appendix D. This demonstrates the potential of the our annotation method.

### 5.4 Human Evaluation

To evaluate the quality of the data generated by the annotation algorithm we proposed, we employed a manual assessment approach. We extracte 200 pairs of queries and code snippets from the Query4Code dataset and invited three experts to score them according to the four types mentioned in Section 4.6. We then calculate the Pearson's $r$ and Kendall's $\tau$ correlation coefficients between the scores and the results generated by the model. The results are summarized in Table 5. Observation reveals that the query-code pairs we annotate demonstrate a strong correlation, confirming the effectiveness of our filtering method.

| Expert | $r$ | $\tau$ | score |
|---|---|---|---|
| Expert1 | 0.652 | 0.483 | 2.47 |
| Expert2 | 0.630 | 0.469 | 2.65 |
| Expert3 | 0.623 | 0.471 | 2.58 |

Table 5: Results of human evaluation.

### 5.5 Cost Analysis

Our annotation algorithm surpasses traditional expert annotation methods in both cost-effectiveness and time efficiency. The API call cost for the GPT-3.5-turbo model we used generally ranges from \$0.001 to \$0.004, allowing for the processing of approximately 3K requests per minute. In contrast, based on crowdsourcing platform rates, the cost for pairing a query with a code snippet is around \$0.2; meanwhile, the time required for an expert to annotate, including reading the query and finding a matching code snippet, typically takes about 3 minutes. This demonstrates the superior scalability of our method.

### 5.6 Case Study

As illustrated in Figure 5, there exists a discrepancy between the docstring of the code snippet and the query annotated by us. Docstrings are typically employed to elucidate the function's purpose and usage, possibly encompassing descriptions of input and output parameters. In contrast, a query represents the functionality requirements described by users in natural language.

## 6 Conclusion

In this paper, we addressed the trade-off between quality and scalability inherent in the construction methods of previous code retrieval datasets by attempting to generate queries based on Large Language Models (LLMs). Initially, we analyzed the key factors affecting the annotation of queries by LLMs and identified that both intra-repository function calls and third-party API calls significantly impacted annotation quality. Based on this understanding, we had designed an annotation algorithm that constructed appropriate contexts by parsing call relationships to generate function queries. Moreover, we had utilized existing code snippets to create the Query4Code dataset. Through model validation and manual assessment, the high quality of the Query4Code dataset was confirmed, and cost analysis had demonstrated the scalability of our annotation approach.

## Limitations

This study primarily focuses on utilizing Large Language Models (LLMs) for the construction of code retrieval datasets and demonstrates the significant impact of call relations on the understanding of function-level code snippets in repositories by language models. However, this paper has certain limitations. Due to cost considerations, we only analyzed and annotated a Python dataset. Although our analytical method is adaptable across different programming languages, we cannot guarantee that our conclusions will perform consistently across various languages. Therefore, we aim to explore the construction of code retrieval datasets for other programming languages using LLMs in future work.

## Ethical consideration

This paper considers using LLMs for code retrieval data synthesis tasks. Previous studies have shown that LLMs may have hallucination problems, and using synthetic data may lead to potential biases in the retrieval process.

## References


Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Luiz Bonifacio, Hugo Abonizio, Marzieh Fadaee, and Rodrigo Nogueira. 2022. Inpars: Data augmentation for information retrieval using large language models. *arXiv preprint arXiv:2202.05144*.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521.

Zhuyun Dai, Vincent Y Zhao, Ji Ma, Yi Luan, Jianmo Ni, Jing Lu, Anton Bakalov, Kelvin Guu, Keith Hall, and Ming-Wei Chang. 2022. Promptagator: Few-shot dense retrieval from 8 examples. In *The Eleventh International Conference on Learning Representations*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2023. An empirical study on using large language models for multi-intent comment generation. *arXiv preprint arXiv:2304.11384*.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Geert Heyman and Tom Van Cutsem. 2020. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *arXiv preprint arXiv:2008.12193*.

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2023. Unnatural instructions: Tuning language models with (almost) no human labor. 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*, pages 14409–14428. Association for Computational Linguistics.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5690–5700.

David A Hull. 1999. Xerox trec-8 question answering track report. In *TREC*.

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*.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5954–5971.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121.

Denis Kocetkov, Raymond Li, LI Jia, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, et al. 2022. The stack: 3 tb of permissively licensed source code. *Transactions on Machine Learning Research*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Haotong Zhang, and I. Stoica. 2023. Efficient memory management for large language model serving with pagedattention. *Symposium on Operating Systems Principles*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. Coderetriever: A large scale contrastive pre-training method for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2898–2910.

Ilya Loshchilov and Frank Hutter. 2017. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101.

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. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9802–9822.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

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*, pages 2719–2734.

Nikitha Rao, Chetan Bansal, and Joe Guan. 2021. Search4code: Code search intent classification using weak supervision. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 575–579. IEEE.

Daniel Rodriguez-Cardenas, David N Palacio, Dipin Khati, Henry Burke, and Denys Poshyvanyk. 2023. Benchmarking causal study to interpret large language models for source code. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 329–334. IEEE.

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

Vinay Samuel, Houda Aynaou, Arijit Ghosh Chowdhury, Karthik Venkat Ramanan, and Aman Chadha. 2023. Can llms augment low-resource reading comprehension datasets? opportunities and challenges. *arXiv preprint arXiv:2309.12426*.

Ivan Sedykh, Dmitry Abulkhanov, Nikita Sorokin, Sergey Nikolenko, and Valentin Malykh. 2023. Searching by code: a new searchbysnippet dataset and snipper retrieval model for searching by code snippets. *arXiv preprint arXiv:2305.11625*.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023a. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023b. 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*, pages 13484–13508. Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023c. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Peter West, Chandra Bhagavatula, Jack Hessel, Jena Hwang, Liwei Jiang, Ronan Le Bras, Ximing Lu, Sean Welleck, and Yejin Choi. 2022. Symbolic knowledge distillation: from general language models to commonsense models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4602–4625.

Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. 2018. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pages 1693–1703.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories*, pages 476–486.

Junlei Zhang, Zhenzhong Lan, and Junxian He. 2023. Contrastive learning of sentence embeddings from scratch. *arXiv preprint arXiv:2305.15077*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

# A Analysis Settings

We use the CodeLlama-Instruct 7B and GPT-3.5-turbo, where we load the checkpoint for CodeLlama-Instruct 7B from huggingface. For GPT-3.5-turbo, we chose to experiment with the *gpt-3.5-turbo-0613* version. And we use the GPT-4-turbo model for scoring, where we select the *gpt-4-1106-preview* version for experimentation. For GPT model, we use the official OpenAI API and employ the default temperature parameters and sampling methods.

## A.1 LLM Inference Details

In the inference process of CodeLlama-Instruct 7B, we adopt a sampling method with a temperature parameter of 0.2 and top-p of 0.95. Additionally, we utilize the vLLM (Kwon et al., 2023) inference library, which integrates various decoding techniques to accelerate sampling during generation.

## A.2 Prompts for Analysis

**System Prompt for Directly Generating Query**

```
Please act as a query generator.
For the given function-level code
snippet in the repository, please
provide a query that the user might use.
This query should be able to search for
that function in a search engine.
Note that you should not provide any
other information.
```

**User Input**

```
Code: {code snippet}
```

**System Prompt for Generating Query (w/ Context)**

```
Please act as a query generator.
For the given function-level code
snippet in the repository and the
information about functions called
within those code snippets, please
provide a query that the user might use.
This query should be able to search for
that function in a search engine.
Note that you should not provide any
other information.
```

**User Input**

```
Code: {code snippet}
Called Function: {called code snippet}
```

**Verification System Prompt for Query**

```
Please play the role of a programming
expert. For the given user queries and
function pairs, please judge whether
the code can meet the needs of the
user's query based on the following
principles:
1. The code can answer and exceed the
requirements for query needs (3 points);
2. The code can satisfy a certain
category of query needs (2 points);
3. The code only meets less than 50% of
query needs (1 points);
4. The code is only minimally related
to the query (0 point).
Please provide an explanation along
with corresponding scores, noting that
you need to output in JSON format as
follows: `{"Explanation": <explanation>,
"Score": <score>}`, without providing
any other information
```

**User Input**

```
Code: {code snippet}
Query: {query}
```

**System Prompt for API Explanation**

```
Please provide a detailed explanation
of the functionality of the third-
party library API and the role of its
mandatory parameters. Please note that
you do not need to provide any
additional output.
```

**User Input**

```
API: {API}
```

**System Prompt for API Explanation (w/ Document)**

```
Please summarize the functions of the
API and the roles of its mandatory
parameters based on the API and
document information. Please note that
you do not need to provide any
additional output.
```

**User Input**

```
API: {API}
Document : {doc}
```

**System Prompt for Rating APIs**

```
Please play the role of a programming
expert.
For a given API and its corresponding
documentation explanation, as well as a
user's description of the API's
functionality, please help me confirm
the degree to which the user-provided
description of the API's functionality
matches with what is described in the
documentation. If it completely matches
semantically, award 2 points; if it
partially matches, give 1 point; if
there is no match, give 0 points.
Please provide an explanation along with
corresponding scores, noting that you
need to output in JSON format as follows:
`{"Explanation": <explanation>, "Score":
<score>}`, without providing any other
information.
```

**User Input**

```
API Documentation Explanation: {function}
User-Provided description: {description}
```

## B Method Settings

### B.1 Prompts for Method

In the method, for summarizing the functions of API documentation, see prompt in section A.2; for scoring prompts used in Data Filtering, refer to section A.2.

**System Prompt for Generating Query (w/ Summary)**

```
Please act as a query generator.
For a function-level code snippet and its
functional summary (to help you
understand the function's purpose)
provided by the user, please provide a
query that can be used to find the
function on search engine.
Note, do not provide any additional
information.
```

**User Input**

```
Code: {Code}
Code Summary: {summary}
```

## C Annotation Algorithm

## D Experimental settings for dataset potential performance

To test the potential performance of our annotation method, we used CodeBERT to initialize the

System Prompt for Generating Summary

```
Please play the role of a programming
expert.
For the functions in a given repository
and the description of third-party API
functionalities called within those
functions, as well as summaries of
functionalities for functions called
within the repository, please provide a
summary of the specified code's
functionality. Note that you need to
offer a concise summary of the code
rather than step-by-step explanations,
and there is no need to reply with any
additional information.
```

User Input

```
Code: {Code}
API Explanation: {explanation}
Called Function Summary: {summary}
```

**Algorithm 1** Annotation Algorithm

**Input:** A directed function call graph, $G(V, E)$;
**Output:** The annotation order of functions, $L$;

Initialize sorted elements list $L \leftarrow \varnothing$
Compute in-degrees $d_{in}(v), \forall v \in V$
Initialize a queue $Q \leftarrow \{v \in V : d_{in}(v) = 0\}$
**while** $Q \neq \varnothing$ **or** $|A| \neq |V|$ **do**
  **while** $Q = \varnothing$ **and** $|A| \neq |V|$ **do**
    $e \leftarrow \text{RandomSelect}(E)$
    $E \leftarrow E \setminus \{e\}$
    $Q \leftarrow \{v \in V : d_{in}(v) = 0\}$
  **end while**
  $v \leftarrow \text{Dequeue}(Q)$
  $L \leftarrow L \cup \{v\}$
  **for** $u \in Adjacent(v)$ **do**
    $d_{in}(u) \leftarrow d_{in}(u) - 1$
    **if** $d_{in}(u) = 0$ **then**
      $Q \leftarrow Q \cup \{u\}$
    **end if**
  **end for**
**end while**
**return** $L$

model and constructed three pre-training data settings through the Query4Code dataset:

- Using only query $q$ and code $c$ pair data (original data setting)
- Using only function summary $s$ and code $c$ pair data
- Construct a triplet (q, c, s) using data from query, code, and summary simultaneously. At this point, consider the summary corresponding to the code as a positive sample. Finally, compare the learning loss function as shown in Equation 6.

$$\mathcal{L}_{tri} = -\mathbb{E}\left[\log \frac{\exp\left(q \cdot c^{+}\right) + \exp\left(s \cdot c^{+}\right)}{\sum_{j=1}^{N} \exp\left(q \cdot c_j\right) + \sum_{j=1}^{N} \exp\left(s \cdot c_j\right)}\right]. \tag{6}$$