# CodeGen4Libs: A Two-Stage Approach for Library-Oriented Code Generation

Mingwei Liu*, Tianyong Yang*, Yiling Lou*†, Xueying Du*, Ying Wang*, and Xin Peng*
Fudan University, Shanghai, China
Email: liumingwei@fudan.edu.cn, 21212010044@m.fudan.edu.cn, yilinglou@fudan.edu.cn,
{21210240012, 22210240051}@m.fudan.edu.cn, pengxin@fudan.edu.cn

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00159

***Abstract*—Automated code generation has been extensively studied in recent literature. In this work, we first survey 66 participants to motivate a more pragmatic code generation scenario, i.e., *library-oriented code generation*, where the generated code should implement the functionally of the natural language query with the given library. We then revisit existing learning-based code generation techniques and find they have limited effectiveness in such a library-oriented code generation scenario.**

**To address this limitation, we propose a novel library-oriented code generation technique, CodeGen4Libs, which incorporates two stages: import generation and code generation. The import generation stage generates import statements for the natural language query with the given third-party libraries, while the code generation stage generates concrete code based on the generated imports and the query. To evaluate the effectiveness of our approach, we conduct extensive experiments on a dataset of 403,780 data items. Our results demonstrate that CodeGen4Libs outperforms baseline models in both import generation and code generation stages, achieving improvements of up to 97.4% on EM (Exact Match), 54.5% on BLEU, and 53.5% on Hit@All. Overall, our proposed CodeGen4Libs approach shows promising results in generating high-quality code with specific third-party libraries, which can improve the efficiency and effectiveness of software development.**

***Index Terms*—Code Generation, Third-party Library, Language Model**

## I. Introduction

In recent years, code generation has gained increasing popularity with the advanced development of deep learning (DL) and large language models (LLM) [1], [2]. Code generation techniques substantially reduce the manual coding effort involved in software development by automatically generating a code snippet (e.g., a method) that implements the desired functionality described in the given natural language requirement. Mainstream code generation techniques first train DL models on a training dataset with natural language queries as input and code as output, and then leverage the trained model to generate code for an unseen natural language query. Recent emerging techniques leverage LLMs (e.g., CodeT5 [3], CodeGPT [4], and PLBART [5]) for code generation, which has been shown to achieve even better efficacy due to the large model scale and being pre-trained on a large code corpus.

* M. Liu, T. Yang, Y. Lou, X. Du, Y. Wang, and X. Peng are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.
† Y. Lou is the corresponding author.

As suggested by the latest survey on the developers' perspective for using code generation tools [6], developers often expect that these tools could be aware of more context/knowledge by using specific frameworks/libraries in the generated code, especially being capable to generate code with using specific third-party libraries (e.g., invocation of an API in a library). However, the majority of existing code generation techniques are designed to only generate a code snippet (e.g., a method) for a standalone natural language description. In other words, these techniques only take the standalone functionality requirement as inputs without considering other context during code generation. Therefore, it remains unclear how existing techniques perform in such a more pragmatic code generation scenario (i.e., *library-oriented code generation*), where the generated code should not only implement the desired functionality but also use the libraries given by the developers. This is underscored by the numerous library-related how-to questions that are frequently encountered on platforms like Stack Overflow [7], [8], [9], [10], [11], [12]. For example, a typical query might be, "How do I read JSON using Gson in Java?"[1].

To fill such a knowledge gap, in this work, we perform an empirical study to (i) first motivate the *library-oriented code generation problem* via a survey from 66 participants, and (ii) then revisit the effectiveness of existing code generation techniques in such a library-oriented code generation task. In particular, our survey results confirm the prevalent demand from developers for library-oriented code generation, i.e., most developers do have personal preference for third-party libraries used in their code. In addition, our survey results further demonstrate the necessity of automated library-oriented code generation techniques, since developers often find it challenging to use the class and methods in their preferred libraries by themselves and often spend a moderate amount of time finding the answers. In summary, the survey results indicate the necessity and motivation for the library-oriented code generation problem. Based on this, we then revisit existing code generation models (e.g., CodeT5 [3], CodeGPT [4], and PLBART [5]) in such a library-oriented code generation scenario, and find that existing models exhibit poor performance.

Inspired by the common practice of developers that first

[1]https://stackoverflow.com/questions/34532431/how-to-read-json-using-gson-in-java

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00159

identify the APIs they want to use (through expert knowledge or search engines) and then write code based on that, we further propose a novel library-oriented code generation technique CodeGen4Libs, which incorporates two stages (i.e., import generation and code generation) to facilitate more powerful library-oriented code generation. The import generation stage first generates import statements for the natural language query with the given library; and then the code generation stage generates the concrete code based on the generated imports and the natural language query. Our main intuition is that the intermediate imports can not only bridge the gap between the specified libraries and the code but also provide more context about the given library during code generation.

We conduct extensive experiments to evaluate the effectiveness of our proposed approach, CodeGen4Libs, on a newly-constructed dataset of 403,780 data items. Our results demonstrate the effectiveness of each stage in CodeGen4Libs, including both the import generation and the code generation stages, which outperformed baseline models. Specifically, compared to the baselines for code generation, our approach achieved improvements of 10.8%-97.4% on EM, 16.1%-54.5% on BLEU, 7.9%-49.8% on CodeBLEU, 8.0%-53.5% on Hit@All, 1.0%-11.0% on Hit@1, 2.8%-16.5% on precision, 63.0%-71.1% on recall, and 3.7%-23.0% on F1. These results demonstrate the effectiveness of our approach in library-oriented code generation, which can generate more accurate and consistent code compared to other models by precisely using APIs from third-party libraries. In addition, we further find that generating imports of higher quality could further improve the performance of the code generation models.

In summary, the contributions of this work are as follows:

- **A survey** involving 66 participants to motivate the library-oriented code generation problem, which demonstrates the prevalent demand of developers in using specific libraries in their code and also the necessity of automated library-oriented code generation techniques.
- **A revisiting study** which demonstrates the limited effectiveness of existing code generation techniques in such a library-oriented code generation scenario.
- **A novel approach** CodeGen4Libs which incorporates two stages (i.e., import generation and code generation) to enable more accurate library-oriented code generation.
- **An extensive evaluation** which demonstrates the effectiveness of the proposed approach CodeGen4Libs in the library-oriented code generation scenario.
- **A new dataset** which is specifically constructed for the library-oriented code generation task. The data could be found at [13].

## II. Motivational Study

In this section, we aim to enhance our comprehension of code generation for third-party libraries. To achieve this, we conducted a survey to investigate developers' familiarity with and preferences for third-party libraries (Section II-A). Moreover, we evaluated the performance of current code generation models on a small-scale dataset, particularly their ability to generate code for specific third-party libraries without specific fine-tuning on library-related data (Section II-B).

Our study aims to answer the following research questions:

**RQ1**: How much do developers prefer specific third-party libraries when coding?

**RQ2**: To what extent are developers familiar with the contextual intricacies of third-party libraries when coding?

**RQ3**: How effective are current code generation models at generating code for specific third-party libraries without specific fine-tuning on library-related data?

### A. Survey

To address RQ1 and RQ2, we conducted an electronic survey targeting computer science students and developers with industrial experience, who were asked to complete a questionnaire. The survey gathered 66 responses from a diverse pool of participants, ranging from undergraduate to doctoral level students, as well as developers with varying levels of experience in the field, ranging from one to five years.

*1) Questionnaire Design:* The questionnaire comprises three questions (Q1, Q2, and Q3), as shown in Table I. Q1 is a multiple-choice question that requires participants to select one or more relevant options as their answer. On the other hand, Q2 and Q3 are ranking question that require respondents to rank the options based on their frequency of occurrence. Furthermore, the questionnaire includes questions about the respondents' backgrounds (such as whether they are undergraduates or graduates) and their experience in the development field (such as their duration of experience).

*2) Results:* Based on the survey results of Q1, it can be inferred that only 6 participants (9.1%) had no preference, while the majority of the participants favored using familiar third-party libraries. Out of the 66 respondents, 49 participants (74.2%) expressed a preference for using familiar third-party libraries. Among these participants, 39 (59.1%) preferred well-known third-party libraries, and 38 (57.6%) preferred libraries that have already been used in the project. 14 participants (21.2%) indicated a desire for libraries that meet other non-functional project constraints. A noteworthy finding is that all of the participants who expressed no preference were either undergraduate or graduate students, while all doctoral students and working professionals expressed a preference. This observation highlights that developers, particularly those with professional experience, have specific demands for particular third-party libraries while coding.

The results of the survey on Q2 suggest that developers face the most common issue of uncertainty about which classes and methods to use to achieve a desired functionality, with an average ranking of 1.4. This indicates that developers may lack the necessary knowledge or experience to efficiently utilize third-party libraries in their coding. Additionally, the second most common issue reported was being clear on which classes to use but being uncertain about which methods to use, with an average ranking of 1.6. This finding suggests that even when developers are familiar with the third-party library, they may still encounter difficulties in identifying the

Table I
QUESTIONNAIRE QUESTIONS AND ANSWER OPTIONS FOR THIRD-PARTY LIBRARY PREFERENCE AND FAMILIARITY

| ID | Question | Options | Type |
|---|---|---|---|
| Q1 | Do you have a preferred third-party library when manually implementing a function for a specific feature or when using a code recommendation tool to generate a function? | A. No preference | multiple choice |
| | | B. Prefer a familiar third-party library | |
| | | C. Prefer a well-known third-party library | |
| | | D. Prefer a third-party library already been used in the project | |
| | | E. Prefer a third-party library that satisfies non-functional project requirements (e.g., cross-platform support) | |
| Q2 | If you have a preferred third-party library in the given scenario, which API classes and methods within the library are you familiar with for accomplishing the desired task? (Sort by frequency of occurrence). | A. I know which classes and methods to use in the library to accomplish the desired functionality | sorting |
| | | B. I know which classes in the library to use for the desired functionality, but I am uncertain about which specific methods within these classes to employ | |
| | | C. I am uncertain about which classes and methods in the library to use for achieving the desired functionality, but I typically find the solution by consulting external resources such as library documentation and search engines | |
| Q3 | How much time do you usually spend searching for answers when referring to external resources? (Sort by frequency of occurrence) | A. Less than five minutes | sorting |
| | | B. Between five and ten minutes | |
| | | C. More than ten minutes | |
| | | D. Unable to find the answer | |

most effective methods for their purposes. Finally, developers reported a relatively infrequent occurrence of being clear on which libraries and methods to use, as indicated by its average ranking of 1.8.

Furthermore, the survey results of Q3 reveal that the most frequently reported time spent finding the answer was between 5-10 minutes, with an average ranking of 1.3. This finding suggests that developers may have some level of knowledge about the libraries they are working with, but still require some additional time to find the information they need. The second most commonly reported time spent was over 10 minutes, with an average ranking of 1.6, indicating that some developers may need more time to fully understand and utilize third-party libraries. Additionally, the least commonly reported response was being unable to find an answer, with a ranking of 2.5. This indicates that developers are generally able to find the information they need, even if it may take them some time.

*3) Summary:* In summary, the survey results suggest that developers prefer to use familiar third-party libraries, but they encounter difficulties in using them effectively due to uncertainty about which classes and methods to use. Furthermore, the findings indicate that developers spend a moderate amount of time finding the answers.

### B. Code Generation Model Analysis

To investigate the performance of existing model for generating code for specific thrid-party library, we conduct an experiment on small dataset.

*1) Dataset:* We extracted method-level code snippets related to third-party libraries from open-source projects on GitHub as our code corpus for empirical study and following model training and evaluation for our approach. To obtain the necessary data, we used the GitHub Code dataset [14] provided by the CodeParrot organization, which contains a vast collection of 115 million code files written in 32 different programming languages. In this study, we focus on Java language due to its popularity. After filtering out 5 million Java code files from the GitHub Code dataset, we extracted a preliminary code corpus consisting of code snippet tuples from the code files. A code snippet tuple is in the form of <*NL,Libs,Imports,Code*>. The *Code* field represents a complete method-level code snippet that includes the method declaration and implementation code. The *NL* field provides a natural language description of the programming task corresponding to the *Code*. The *Libs* field contains one or more third-party libraries used in the *Code*, while the *Imports* field indicates the class-level imports from third-party libraries used in the code. For a Java code file, we initially extracted method-level code snippets (*Code*) using the javalang [15] code analysis tool. For each code snippet, we further analyzed the code file and extracted its corresponding method comment as the natural language description of the task (*NL*). We filtered out code snippets without comments. We then extracted the class-level import statements from the code file. For each code snippet, we matched it with the import statements of the file to obtain the related import statements (*Imports*). A code snippet was considered related to an import statement if it contained the imported class name in the code. Finally, we obtained the third-party libraries used in the code snippet (*Libs*) based on the mapping between import statements and libraries. It is worth noting that for convenience, we considered the JDK [16] and Android SDK [17] as third-party libraries. To ensure the quality of the code corpus, we performed a series of data cleaning steps on the NL and code snippets. Specifically, we cleaned the comments extracted from NL by removing annotations such as "@param" and "@return" as well as their content, eliminating non-English content and removing hyperlinks such as "http://" and "https://". We also cleaned the code snippets by removing single-line comments, unifying method names as "function", removing consecutive white spaces, and replacing long string constants with a placeholder token "STR", following similar practices in previous works [18]. As a result, we obtained a code corpus with 2,916,582 code snippet tuples.

To reduce the corpus's size, we filtered the code snippet tuples based on third-party libraries. Initially, we counted the frequency of third-party libraries used in the code snippets and extracted the top 500 most frequently used libraries, excluding the JDK and Android SDK. Subsequently, we retained only the code snippets that utilized these top 500 third-party libraries, resulting in a corpus of 1,215,900 code snippet tuples. This filtering approach allows us to focus on the most commonly used third-party libraries and exclude less commonly used

libraries, reducing the corpus's size while still ensuring it is representative of real-world usage.

We randomly selected 100 libraries from the top 500 most popular ones and chose 5 corresponding code tuples for each library from the code corpus. This resulted in a small-scale testing dataset consisting of 500 code snippet tuples.

*2) Models:* We primarily compared the performance of several existing code generation models that were fine-tuned based on pre-trained language models. The pre-trained models we used were:

**PLBART.** PLBART is based on the BART [19] architecture and is pre-trained on a corpus of natural language and programming language using denoising objectives.

**CodeGPT.** CodeGPT is a GPT-2 [20]-style model that is pre-trained on the CodeSearchNet dataset [4]. For our comparison, we used the Java domain-adaptive model [21], which starts with a GPT-2 model and is continuously trained on Java code from the CodeSearchNet dataset.

**CodeT5.** CodeT5 is adapted from the T5 [22] model and considers crucial token type information from identifiers. It also allows for multi-task learning on downstream tasks.

Zeng et al. [23] evaluated the effectiveness of the three models mentioned above for code generation tasks, but they only provided pre-trained code generation models. To obtain the corresponding code generation models, we applied their associated model fine-tuning code and all hyperparameter settings from the replication package of their work [24]. We used the CONCODE dataset [25], which is a large dataset with over 100,000 examples of Java class files from GitHub repositories, for training. As a result, we obtained three code generation models that can take NL as input and generate corresponding code snippets.

*3) Metrics:* For each code snippet tuple <*NL,Libs,Imports,Code*> in the test dataset, we concatenate *NL* and *Libs* using *"using the following libraries: com.google.gson"* as input to the code generation models, e.g., *"read a Json array using the following libraries: com.google.gson"*. We then compare the predicted code snippets generated by the models to the ground truth *Code* and calculate the following metrics to evaluate the performance of the three models:

- **Exact Match (EM)**: This metric measures the percentage of predictions that exactly match the ground truth.
- **Bilingual Evaluation Understudy (BLEU)**: A measure of n-gram overlap between the predicted and ground truth sequences, commonly used in machine translation.
- **CodeBLEU**: A modified version of the BLEU metric designed for code, which is a weighted average of lexical, abstract syntax tree, and data flow match.
- **Hit@All**: This metric measures whether all correct classes belonging to the specified third-party library are included in the generated code. A class is considered correct if it also appears in the ground truth code.
- **Hit@1**: This metric measures whether at least one correct class belonging to the specified third-party library is included in the generated code.
- **Precision**: This metric measures the proportion of correct classes belonging to the specified third-party library that are included in the generated code.
- **Recall**: This metric measures the proportion of correct classes belonging to the specified third-party library that are included in the generated code compared to all correct classes in the ground truth.
- **F1**: The harmonic mean of Precision and Recall, which measures the overall effectiveness of the model in predicting the correct classes of the given libraries.

EM, BLEU, and CodeBLEU are commonly used metrics for evaluating code generation tasks [26], [27], [3]. HIT@All, HIT@1, Precision, Recall, and F1 are specifically designed for the task of generating code for specific third-party libraries, which measure the effectiveness of the generated code in correctly using the specified third-party library API classes.

*4) Results:* As shown in Table II, we can see that the three code generation models performed poorly on generating code for specific third-party libraries. For example, the code generated by CodeGPT only contains 7.7% of API classes from the specified libraries. Among the three models, the CodeT5-based model performed relatively better (9.9%), but still not satisfactory.

There might be two possible reasons for this poor performance. Firstly, the training data for these models did not particularly consider the input of libraries, and the models may not have been fine-tuned on data containing libraries as input. Even if the *Libs* are included as part of the model input, the model may still not understand them well. Secondly, the gap between the libraries included in the input and the actual API classes used in the code might be large. Including some import statements related to the given third-party libraries in the input that are also related to the given *NL* may be helpful for the model (because import statements are related to both Libs and the classes used in the code).

*5) Summary:* In summary, the experiment demonstrated that existing code generation models exhibit poor performance when generating code for specific third-party libraries, suggesting the need for dedicated fine-tuning and design efforts.

## III. Approach

We formulate the problem of library-oriented code generation as generating method-level code snippets (*Code*) from a natural language description (*NL*) and one or more specified third-party libraries (*Libs*), i.e., *NL+Libs->Code*. However, generating code specific to a given library is more challenging than normal code generation due to the restricted generation scenario. To address this task, we propose a two-stage method, CodeGen4Libs, which splits it into import generation and code generation subtasks. The first task generates API class-level import statements (Imports) from *NL* and *Libs* (i.e., *NL+Libs->Imports*), while the second generates *Code* from *NL*, *Libs*, and *Imports* (i.e., *NL+Libs+Imports->Code*). Figure 1 provides an overview of CodeGen4Libs.

Splitting the task into two subtasks is inspired by the practice of developers who, when faced with a task and a third-party

Table II
PERFORMANCE COMPARISON OF EXISTING CODE GENERATION MODELS ON THIRD-PARTY LIBRARY-ORIENTED CODE GENERATION

| Model | EM | BLEU | CodeBLEU | Hit@All | Hit@1 | Precision | Recall | F1 |
|---|---|---|---|---|---|---|---|---|
| PLBART | 0 | 0.046 | 0.097 | 0.060 | 0.126 | 0.123 | 0.089 | 0.097 |
| CodeGPT | 0 | 0.009 | 0.054 | 0.058 | 0.104 | 0.099 | 0.077 | 0.083 |
| CodeT5 | 0 | 0.051 | 0.093 | 0.056 | 0.152 | 0.144 | 0.099 | 0.109 |

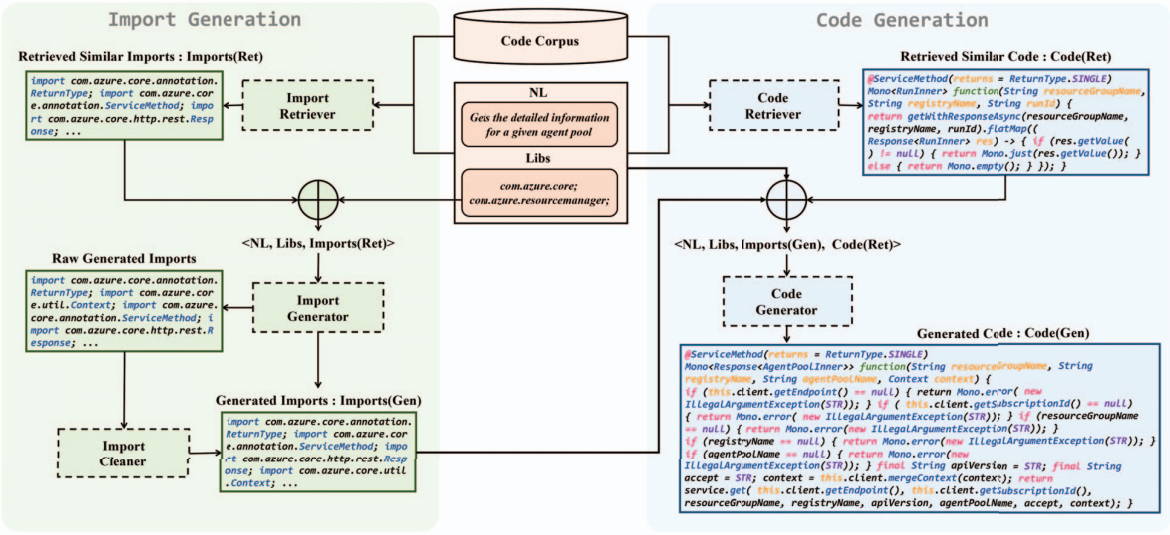


Figure 1. Overview of CodeGen4Libs

library, often first identify the APIs they want to use (through expert knowledge or search engines) and then write code based on that. Separating the task into two steps allows different models to be trained to handle import generation and code generation. Compared to training a single end-to-end model for code generation specific to a given library, splitting the task into two subtasks provides more context about the given library during code generation (provided by the API class imports generated in the first subtask). This is important because it helps bridge the gap between the specified libraries and the generated code, resulting in code that is more limited to the given library. Overall, this two-stage approach enables us to generate code for specific third-party libraries more effectively and efficiently.

In both import generation and code generation, we adopt a retrieval-augmented technique [28] to enhance the performance of our models. We elaborate on our approach in Section III-A and Section III-B, respectively.

### A. Import Generation

We formalize the import generation task as a sequence-to-sequence generation task, similar to code generation tasks. To achieve this, we fine-tune CodeT5, a state-of-the-art model, as it has demonstrated outstanding performance on code-related tasks [3]. To further enhance import generation, we incorporate retrieval-augmented technique to retrieve import statements *Imports(Ret)* related to the given *NL* and *Libs*, which are used as input for the import generation model. Retrieval-augmented techniques have been demonstrated to improve the performance of sequence-to-sequence generation tasks [28], and are widely employed in software engineering-related tasks like code generation [29] and commit message generation [30].

As shown in Figure 1, the entire import generation process comprises three main modules: the import retriever, import generator, and imports cleaner. The import retriever is responsible for retrieving relevant imports *Imports(Ret)* from a large-scale corpus based on the given *NL* and *Libs*. The import generator takes the concatenated input of *NL*, *Libs*, and *Imports(Ret)* as input and employs a pre-trained import generation model to generate raw imports statements. Finally, the imports cleaner is responsible for cleaning the generated imports to obtain higher-quality imports statements, *Imports(Gen)*, to serve as input to the subsequent code generation model.

We will now delve into each module in more detail.

*1) Import Retriever:* To retrieve relevant imports for a given *NL* and *Libs*, we employ the BM25 algorithm, which is widely used in text similarity tasks [31]. BM25 is a popular bag-of-words retrieval function that estimates the lexical-level similarity between two sentences. The higher the BM25 score, the more similar the sentences are.

To retrieve relevant imports, we apply BM25 to a pre-collected code corpus (e.g., the Java code corpus we collected in Section II-B1) that contains a series of code snippet tuples in the form of *<NL,Libs,Imports,Code>*. We retrieve the top-k (e.g., 1,000) code snippets with the most similar *NL* to the given *NL* and then filter them in order of decreasing similarity until we find one that uses all the given *Libs*. Next, we remove any imports statements from non-specified *Libs* and sort the remaining imports alphabetically to obtain the final set of relevant imports *Imports(Ret)*.

For instance, consider the *NL Gets the detailed information for a given agent pool"* and the two libraries *com.azure.core* and *com.azure.resourcemanager* as *Libs*, shown in Figure 1. The BM25-based retriever may retrieve a code snippet tuple with the most similar *NL* as *Gets the detailed information for a given run"*. The tuple has four import statements as *Imports*, i.e., *"import com.azure.core.annotation.ReturnType; import com.azure.core.annotation.ServiceMethod; import com.azure.core.http.rest.Response; import com.azure.resourcemanager.containerregistry.fluent.models.RunInner;"*, which covers all the two given *Libs*. After the sorting and filtering steps, the final set of relevant imports *Imports(Ret)* is obtained.

We applied the filtering step to avoid introducing imports for non-specified third-party libraries, which could mislead the code generation model. We also applied the sorting step to normalize the imports from different sources.

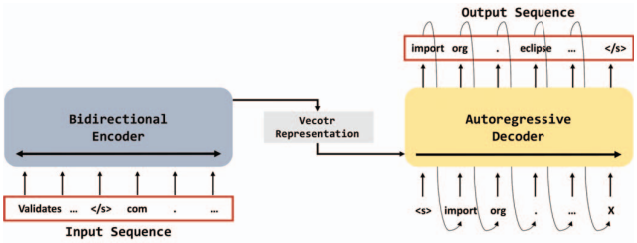


Figure 2. Architecture of the Import Generator

*2) Import Generator:* To implement our import generator, we employed an encoder-decoder neural network based on CodeT5, which has shown excellent performance on code-related tasks [3], [32]. As shown in Figure 2, the model architecture consists of a bidirectional encoder and an autoregressive decoder. In our approach, we fine-tuned CodeT5 for import generation, which we modeled as a sequence-to-sequence generation task.

First, we concatenate the input *NL*, *Libs*, and *Imports(Ret)* together with a special separator token *[SEP]* to form a single input sequence. Then, we tokenize the input sequence and encode the tokenized input sequence into a vector representation using a bidirectional transformer-based encoder, which captures the contextual information of the input sequence. Then, we use a transformer-based autoregressive decoder to generate the target sequence of *Imports*. The decoder is autoregressive, meaning that it generates one token at a time based on the previous tokens generated. It takes the vector representations of the input sequence as its initial input. At each decoding step, the decoder generates a probability distribution over the possible next tokens in the sequence, conditioned on the previously generated tokens. The next token is then sampled from this distribution and used as input to the next decoding step. This process is repeated until the end-of-sequence token (e.g., </s>) is generated.

During training, we use a cross-entropy loss to optimize the model's parameters to minimize the difference between the generated *Imports* and the ground truth *Imports*. We fine-tune the pre-trained CodeT5 model on our import generation task using the training data described in Section IV-A1. The details of our implementation are described in Section IV-A2.

*3) Import Cleaner:* The imports generated by the model may suffer from noise, such as duplicates and incomplete statements, which can have a negative impact on the effectiveness of the code generation process. For instance, the model might output import statements like *"import com"* or *"import com.google.gson.Gson; import com.google.gson.Gson;"*, which contain duplicates or are incomplete.

This issue arises from the fact that the import generator is based on encoder-decoder architectures, and certain content may have a higher decoding probability, leading to repeated generation. Additionally, the generated content may exceed the length limit, leading to incomplete or truncated statements. To mitigate these issues, we apply several criteria to clean up the generated import statements.

We first split the generated import statements based on semicolons to obtain individual import statements. We then apply several criteria to clean up each import statement:

- Remove any duplicate import statements to eliminate redundancy in the final list of imports.
- Filter out any import statements that were incomplete, meaning they did not end with a semicolon or did not start with the keyword "import".
- Split the fully qualified class names in the import statements into a list of strings representing the package and class names, and then remove any import statements containing duplicate package or class names.
- Compare the generated imports with the given *Libs* and filter out any imports that do not belong to the given *Libs*.

Lastly, we alphabetically sort the remaining import statements and combine them to form a final set of clean import statements named *Imports(Gen)*.

### B. Code Generation

We formalize the code generation task as a sequence-to-sequence generation task as well. Similar to the import generation, we fine-tune CodeT5 as the code generation model and incorporate retrieval-augmented technique to retrieve code snippets *Code(Ret)* related to the given *NL* and *Libs* as the input for the code generation model. The overall process is illustrated in Figure 1, and consists of two main modules: code retriever and code generator. We will now delve into each module in more detail.

*1) Code Retriever:* To retrieve relevant code snippets for a given *NL* and *Libs*, we utilize the BM25 algorithm, which is similar to the import retrieval process discussed in Section III-A1. We begin by applying BM25 to a pre-collected code corpus, such as the Java code corpus collected in Section II-B1, which contains a series of code snippet tuples in the form of <*NL,Libs,Imports,Code*>. We then retrieve the top-k (e.g., 1,000) code snippets with the highest similarity score to the given *NL*, and filter them in order of decreasing similarity until we find one that uses all the given *Libs*.

As shown in Figure 1, the retrieved code snippet *Code(Ret)* uses the two given *Libs* and has the highest similarity score with the given *NL*, "Gets the detailed information for a given agent pool."

*2) Code Generator:* We employ the same model architecture for our code generation module as our import generator, using CodeT5 as the core model. The task of generating code is modeled as a sequence-to-sequence generation task, where the input sequence includes the natural language description *NL*, required libraries *Libs*, generated imports *Import(Gen)*, and relevant code snippets *Code(Ret)* retrieved using the BM25 algorithm (as described in Section III-B1). The target sequence is the generated code *Code(Gen)*.

To prepare the input sequence for the model, we first concatenate the input fields with the special separator token *[SEP]*, creating a single input sequence. This input sequence is then tokenized and encoded into a vector representation using the bidirectional transformer-based encoder. The decoder generates the target sequence of *Code(Gen)*, conditioned on the encoded vector representation of the input sequence.

Importantly, the combination of *Imports(Gen)* and *Code(Ret)* offers several benefits to our code generation task. *Imports(Gen)* provides the model with key third-party library APIs that may be required to generate the code, reducing the need for extensive search through libraries. Meanwhile, *Code(Ret)* provides templates, such as loop control structures, and usage patterns for specific third-party library APIs that can be used as references during code generation. Although neither *Imports(Gen)* nor *Code(Ret)* can ensure correctness, their combination enables the model to concentrate on the key APIs that are frequently present in both *Imports(Gen)* and *Code(Ret)* and are essential for the task. Together, these two inputs help to reduce the noise and interference in the final code generation.

We fine-tune the pre-trained CodeT5 model on our code generation task using the training data described in Section IV-A1. During training, we use *Imports(Gen)* generated by the model for each code tuple, rather than relying on the ground truth *Imports*. This approach allows us to minimize the gap between the input at training time and the input during inference, as both inputs use the same import generator to produce *Imports(Gen)*. By reducing this gap, we can better simulate the real-world use case and improve the model's performance on actual tasks. More implementation details are described in Section IV-A2.

## IV. Evaluation

In this section, we evaluate the effectiveness of CodeGen4Libs by addressing the following research questions:

**RQ1 (Effectiveness of Library-oriented Imports Generation)**: How effective is CodeGen4Libs in generating high-quality library-oriented imports?

**RQ2 (Effectiveness of Library-oriented Code Generation)**: How effective is CodeGen4Libs in generating high-quality library-oriented code?

**RQ3 (Imports Generation Quality Impact)**: To what extent does the quality of import generation affect the quality of code generation results?

Table III
Statistics of the Benchmark

| Dataset | Size | NL/token | Code/token | Imports | Libs |
|---|---|---|---|---|---|
| Train | 391,811 | 19.0 | 87.2 | 2.8 | 1.7 |
| Validation | 5,967 | 18.3 | 72.9 | 2.1 | 1.3 |
| Test | 6,002 | 18.4 | 77.3 | 2.4 | 1.5 |

### A. Experimentation Setup

*1) Benchmark:* We created a benchmark for training and evaluation using a code corpus described in Section II-B1. Initially, we randomly sampled 600,000 code snippet tuples <*NL,Libs,Imports,Code*> from the corpus. We filtered out tuple samples whose tokenized *Code* length exceeded 512 tokens and samples with inputs (*NL+Libs+Imports*) exceeding 512 tokens. This is because our model has a maximum input length limitation. Additionally, we removed samples that had the same *NL* and *Libs* but different *Code*, as they could potentially interfere with the model's learning. To standardize the benchmark, we sorted libraries and import statements alphabetically. To ensure the balance of the dataset, we include a maximum of 5,000 corresponding code snippet tuples for each library. The resulting benchmark included 403,780 code snippet tuples for 500 libraries. We split the benchmark randomly into training, validation, and test datasets and partitioned the tuples to ensure balance and include at least 1.5% of relevant code snippets for each library in the training and validation datasets. Table III shows statistics for the datasets.

*2) Implementation:* To build the import retriever and code retriever, we utilized the open-source search engine Elasticsearch [33] and built an index on the *NL* of the code corpus, which contains 1,215,900 code snippet tuples (See Section II-B1). This allowed us to efficiently retrieve relevant code snippets and imports for a given natural language query.

We trained the import generation model and code generation models on the benchmark dataset using the training set, and validated their performance using the validation set. The models were implemented using the Python library transformers [34], initialized with the CodeT5-base [35] model. For model optimization, we used the cross-entropy loss and the Adam optimizer, with a learning rate of 4e-5 and a batch size of 8. Early stopping based on validation loss was used during the 30 epochs of training, which were conducted on a single Nvidia 3090 GPU. We followed the same hyperparameters and training procedure as in previous work [23].

### B. RQ1: Effectiveness of Library-oriented Imports Generation

To evaluate the effectiveness of CodeGen4Libs in import generation, we compared our approach with multiple baselines on the benchmark dataset.

*1) Baselines:* We refer to our approach for import generation as *Import(Gen)*, which represents the import statements obtained through our import generator and cleaner. We compared it with the following baseline methods:

- **Imports(Ret)**. The simplest method for the *NL+Libs->Imports* task is retrieval-based. In Section III-A1, we used the BM25 algorithm to retrieve the most relevant imports for a given *NL* and *Libs*. We used this BM25-based

Table IV
COMPARISON OF IMPORTS GENERATION PERFORMANCE BETWEEN DIFFERENT METHODS

| Method | EM | BLEU | Hit@All | Hit@1 | Precision | Recall | F1 |
|---|---|---|---|---|---|---|---|
| Imports(Ret) | 0.380 | 0.640 | 0.457 | 0.675 | 0.677 | 0.581 | 0.625 |
| Imports(Gen)-NL+Libs | 0.495 | 0.772 | 0.554 | 0.816 | 0.718 | 0.709 | 0.713 |
| Imports(Gen)∪Imports(Ret) | 0.394 | 0.626 | **0.643** | **0.869** | 0.659 | **0.778** | 0.714 |
| Imports(Gen)∩Imports(Ret) | 0.395 | 0.445 | 0.423 | 0.649 | **0.887** | 0.545 | 0.675 |
| Imports(Gen) | **0.536** | **0.782** | 0.602 | 0.848 | 0.746 | 0.748 | **0.747** |

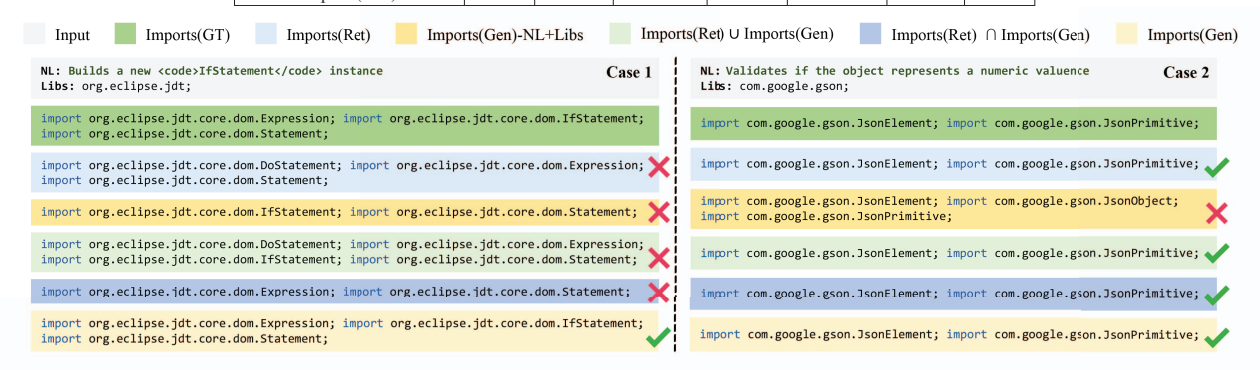


Figure 3. Import Generation Test Cases

import statement retriever as a baseline and compared it with our generation-based approach.

- **Import(Gen)-NL+Libs**. To investigate whether retrieval-enhancement technology is helpful for the import generation task, we trained a new import generation model using the same dataset and hyperparameters, but with only *NL* and *Libs* as input, denoted as *Import(Gen)-NL+Libs*. This comparison allows us to assess whether retrieving relevant imports as input for the import generation model truly improves the effectiveness of import generation.
- **Import(Gen)∩Imports(Ret)**. One possible conjecture is that combining the imports generated by the generation-based method and the retrieval-based method can further improve the effectiveness. *Import(Gen)∩Imports(Ret)* represents taking the intersection of the import statements generated by the two methods as the final import generation results, which can reduce the noise in the import generation result.
- **Import(Gen)∪Imports(Ret)**. *Import(Gen)∪Imports(Ret)* is another way to combine the two methods, representing taking the union of the import statements obtained by the two methods, which may improve the coverage of generated imports.

*2) Metrics:* We evaluate our approach and the baselines on the test dataset of import generation. The evaluation metrics used include EM, BLEU, HIT@All, HIT@1, Precision, Recall, and F1. These metrics have been introduced in Section II-B3. We do not use CodeBLEU to evaluate the quality of generated imports because imports do not contain the additional information like data flow. To compute the metrics HIT@All, HIT@1, Precision, Recall, and F1, we split the generated imports and ground truth imports into individual import statements by semicolon and compare them at the statement level.

*3) Results:* Table IV provides a comprehensive comparison of various methods for import statement generation, and the results clearly indicate that the two generation-based methods, *Imports(Gen)* and *Imports(Gen)-NL+Libs*, outperform the retrieval-based method, *Imports(Ret)*, across all seven metrics. This suggests that generating import statements directly from natural language descriptions and relevant libraries is a more effective approach than retrieving them solely based on the given description and libraries. Moreover, the performance of *Imports(Gen)* is significantly better than *Imports(Gen)-NL+Libs*, demonstrating the benefits of incorporating relevant libraries during the generation process.

We also evaluate the effectiveness of combining the generation-based and retrieval-based methods. The method of taking the union of the import statements obtained by the two methods, *Import(Gen)∪Imports(Ret)*, achieves a lower EM and BLEU score compared to *Import(Gen)*, but with higher Hit@All, Hit@1, and recall scores, indicating a better coverage of the import statements. The method of taking the intersection of the import statements obtained by the two methods, *Import(Gen)∩Imports(Ret)*, has lower performance in all metrics except for precision, which is higher than other methods. This suggests that combining the two methods through taking the intersection of their import statements may reduce noise but with lower coverage.

Overall, the results demonstrate the effectiveness of our proposed generation-based method for import generation and the potential benefits of combining it with retrieval-based methods. However, we ultimately choose *Imports(Gen)* as our method for code generation because it achieves a good balance between coverage and precision (highest F1 score) and significantly outperforms the method of combining generation-based and retrieval-based methods in terms of EM and BLEU scores.

Figure 3 illustrates two test cases for import statement generation, where the results of all methods are marked with different colors. The term *Imports(GT)* refers to the ground truth import statements. In case 1, we observe that only *Imports(Gen)*

Table V
COMPARISON OF CODE GENERATION PERFORMANCE BETWEEN DIFFERENT METHODS

| Model | Input | EM | BLEU | CodeBLEU | Hit@All | Hit@1 | Precision | Recall | F1 |
|---|---|---|---|---|---|---|---|---|---|
| CodeGPT | NL+Libs | 0.190 | 0.316 | 0.339 | 0.395 | 0.822 | 0.684 | 0.597 | 0.637 |
| PLBART | NL+Libs | 0.114 | 0.308 | 0.309 | 0.327 | 0.807 | 0.638 | 0.551 | 0.591 |
| CodeT5 | NL+Libs | 0.177 | 0.401 | 0.385 | 0.406 | 0.852 | 0.687 | 0.630 | 0.657 |
| CodeT5 | NL+Libs+Imports(Gen) | 0.191 | 0.388 | 0.396 | 0.447 | 0.887 | 0.723 | 0.676 | 0.699 |
| CodeT5 | NL+Libs+Code(Ret) | 0.203 | 0.410 | 0.429 | 0.465 | 0.876 | 0.722 | 0.681 | 0.701 |
| CodeT5 | NL+Libs+Imports(Gen)+Code(Ret) | **0.225** | **0.476** | **0.463** | **0.502** | **0.896** | **0.743** | **0.711** | **0.727** |

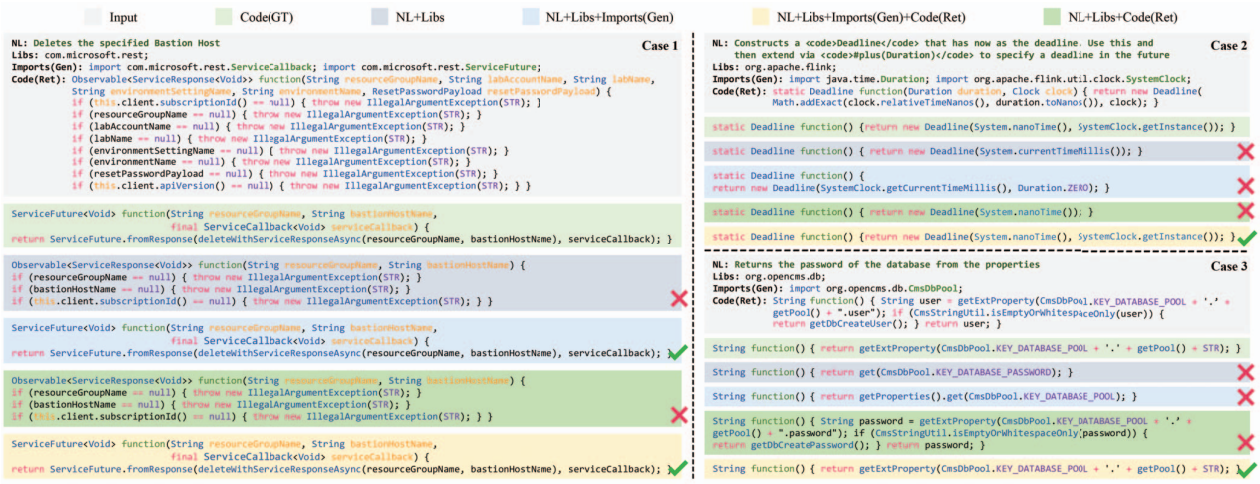


Figure 4. Code Generation Test Cases

correctly predicts all import statements in the ground truth, while even *Imports(Ret)* contains an incorrect import statement (*"import org.eclipse.jdt.core.dom.DoStatement"*) and misses a correct one (*"import org.eclipse.jdt.core.dom.IfStatement"*). However, the import generator is not affected by this and still predicts the correct imports. Moreover, comparing *Imports(Gen)* to *Imports(Gen)-NL+Libs*, we can see that incorporating relevant libraries during generation helps the model generate more correct import statements and improves the model's effectiveness. In case 2, we observe that *Imports(Gen)-NL+Libs* generates one extra import statement compared to the ground truth, but with the help of retrieval-augmented technique, this error disappears. Overall, the retrieval-augmented technique has greatly improved import generation by increasing both precision and recall.

*4) Summary:* In summary, our method demonstrates superior effectiveness in import generation, compared to the baselines.

### C. RQ2: Effectiveness of Library-oriented Code Generation

*1) Baselines:* To serve as baselines for our approach, we fine-tuned the pre-trained language models that we introduced in Section II-B – namely CodeGPT, PLBART, and CodeT5 – using the benchmark dataset. In contrast to our approach, these models take only the *NL+Libs* as input and generate the corresponding code *Code* as output, without any additional input. We fine-tuned these models by providing the input sequence of *NL+Libs* to the models and training them to generate the corresponding code $c$. The fine-tuning process was carried out in accordance with previous work [23], same as in Section II-B2.

To demonstrate the effectiveness of our approach in incorporating additional inputs, we trained two variants of our code generation model using CodeT5. The first variant took *NL+Libs+Import(Gen)* as input, while the second variant took *NL+Libs+Code(Ret)* as input. We followed the same hyperparameters and training procedures as those detailed in Section IV-A2. This approach allowed us to compare the performance of our method with and without incorporating import statements generated through our method, as well as with the performance of using retrieved code as input.

*2) Metrics:* The evaluation metrics include EM, BLEU, CodeBLEU, Hit@All, Hit@1, Precision, Recall, and F1. They evaluate the quality of generated code and the matching between generated code and the ground truth.

*3) Results:* Table V presents the experimental results of code generation for different models and input variations. Among these models, our code generation model, i.e., CodeT5 with *NL+Libs+Import(Gen)+Code(Ret)* input, achieved the best performance in all evaluation metrics.

Compared to the baseline models, our proposed method achieved significant improvements in all evaluation metrics, with EM improvement ranging from 10.8% to 97.4%, BLEU improvement ranging from 16.1% to 54.5%, CodeBLEU improvement ranging from 7.9% to 49.8%, Hit@All improvement ranging from 8.0% to 53.5%, Hit@1 improvement ranging from 1.0% to 11.0%, precision improvement ranging from 2.8% to 16.5%, recall improvement ranging from 63.0% to 71.1%, and F1 improvement ranging from 3.7% to 23.0%. These results demonstrate the effectiveness of our approach

Table VI
THE IMPACT OF IMPORTS QUALITY ON CODE GENERATION RESULTS IN CODEGEN4LIBS

| Imports | EM | BLEU | CodeBLEU | Hit@All | Hit@1 | Precision | Recall | F1 |
|---|---|---|---|---|---|---|---|---|
| Imports(Ret) | 0.172 | 0.420 | 0.412 | 0.406 | 0.828 | 0.664 | 0.618 | 0.640 |
| Imports(Gen)∪Imports(Ret) | 0.210 | 0.438 | 0.454 | 0.495 | 0.895 | 0.710 | 0.710 | 0.710 |
| Imports(Gen)∩Imports(Ret) | 0.179 | 0.433 | 0.407 | 0.406 | 0.830 | 0.715 | 0.615 | 0.662 |
| Imports(Gen) | 0.225 | 0.476 | 0.463 | 0.502 | 0.896 | 0.743 | 0.711 | 0.727 |
| Imports(GT) | **0.249** | **0.504** | **0.484** | **0.603** | **0.969** | **0.866** | **0.819** | **0.842** |

in library-oriented code generation, which can generate more accurate and consistent code compared to other models by precisely using APIs from third-party libraries.

The two variants of CodeGen4Libs, i.e., CodeT5 with *NL+Libs+Import(Gen)* and *NL+Libs+Code(Ret)*, also achieve good results, but are outperformed by the CodeGen4Libs (CodeT5 with *NL+Libs+Import(Gen)+Code(Ret)*). This suggests that both incorporating generated import statements and using retrieved code snippets can improve the code generation performance, but combining them leads to even better results.

Figure 4 illustrates three test cases for code generation, where the results of different methods are marked with different colors. The term *Code(GT)* refers to the ground truth code for the input. In case 1, we observe that both the code generation models with *NL+Libs+Imports(Gen)* and *NL+Libs+Imports(Gen)+Code(Ret)* generate the correct results, while the models with only *NL+Libs* and *NL+Libs+Code(Ret)* generate incorrect code. Although the retrieved code *Code(Ret)* is unrelated to the given task *NL* and *Libs*, our approach can still generate correct code based on the help of generated imports *Imports(Gen)*, even in the presence of noise from retrieved code. In case 2, only our approach with *NL+Libs+Imports(Gen)+Code(Ret)* generates the correct code. This is because it combines the information provided by *Imports(Gen)* and *Code(Ret)* together, and the noise in the *Code(Ret)* (using some irrelevant APIs) does not affect the generated effect since the model uses the code structure provided by *Code(Ret)*. Similarly, in case 3, only the information provided by the *Imports(Gen)* is not enough, and combining *Imports(Gen)* and *Code(Ret)* leads to the best result. These results demonstrate that *Imports(Gen)* and *Code(Ret)* can complement each other in library-oriented code generation tasks. By combining them, our approach can leverage the strengths of both of them and produce more accurate and consistent code.

In this study, we fine-tuned CodeT5 to develop import generation and code generation models due to its superior performance on code-related tasks compared to other existing pre-trained language models [3], [32]. However, it's important to note that our approach can serve as a foundational framework, and in the future, more advanced models could replace CodeT5 for enhanced outcomes.

*4) Summary:* Our experiments demonstrate that our approach, which combines generated import statements and retrieved code snippets, is effective in improving the accuracy and consistency of library-oriented code generation.

### D. RQ3: Imports Generation Quality Impact

In this section, we investigate the impact of import quality on code generation results in CodeGen4Libs.

*1) Design:* Specifically, we compare the performance of CodeGen4Libs using different imports as inputs on test dataset of the benchmark, i.e., the different imports shown in Table IV (see Section IV-B1). We also try to use the *Imports(GT)* that is the ground truth imports from the benchmark as the input. We evaluate the performance the same metrics as Section IV-C2.

*2) Results:* Table VI presents the impact of import quality on code generation results in CodeGen4Libs. The study examined five different import strategies. The results show that using *Imports(GT)* as input achieved the best performance across all metrics, followed by *Imports(Gen)*. *Import(GT)* resulted in a Hit@1 of 0.969 and F1 of 0.842, which is 8.15%-17.03% and 15.82%-31.56% higher than other strategies, respectively. The study demonstrates that the quality of imports used as input has a significant impact on the performance of CodeGen4Libs. It is worth noting that using *Imports(Gen)* also performed well, indicating that the CodeT5 model can generate high-quality imports. However, the performance of *Imports(Gen)* is still lower than that of *Imports(GT)*, suggesting that there is still room for improvement in the imports generation capability of the model.

*3) Summary:* In conclusion, the experiment results underscore the significance of using high-quality imports as input for code generation models and imply that enhancing imports generation capabilities can further improve the performance of code generation models.

### E. Threats to Validity

Our study may face three validity threats. The first pertains to the subjectivity and lack of representativeness in our survey. To address this, we invited participants from diverse backgrounds to ensure the generalizability of our conclusions.

The second validity threat relates to the construction of our dataset from scratch, as there is no existing dataset specifically designed for code generation from third-party libraries. To mitigate this threat, we followed similar practices as previous works and ensured that our dataset covers a diverse range of third-party libraries [18].

The third validity threat concerns the implementation of our proposed model and baseline methods. To address this, we adopted existing fine-tuning scripts and hyperparameters from related works [23] and made our source code and dataset publicly available for validation [13]. Moreover, our dataset contains 6,002 code snippet pairs, significantly larger than the widely-used CONCODE benchmark's 2,000 test cases, to improve the robustness of our model. Despite our focus on

Java, our method is not language-specific and can apply to any object-oriented language involving a significant amount of third-party library APIs. We plan to expand our dataset to support multiple programming languages in the future.

## V. Related Work

Code generation aims to produce source code from given natural language descriptions or requirements, and it has long been a central focus of software engineering research [36], [37], [38], [39]. Traditional approaches to code generation include sequence-based and tree-based methods. Sequence-based models utilize neural networks to generate source code token by token based on the input description, whereas tree-based models construct a parse tree of the program from the natural language description and subsequently convert it into corresponding code [40], [41].

In recent times, the landscape of code-related tasks has been revolutionized by pre-trained language models, which have outperformed conventional sequence-based and tree-based methods. Some prominent large-scale pre-trained models in this domain include CodeBERT [42], CodeT5 [3], InCoder [43], CodeGPT [4], and PLBART [5]. Fine-tuning these models has emerged as a new paradigm for code generation. In this study, we fine-tune CodeT5 to develop import generation and code generation models. Diverging from general code generation, our focus lies in library-oriented code generation within a specific scenario. In fact, there has been a surge of interest in code generation related to libraries [44], [45], [46]. While existing efforts on library-oriented code generation mainly support a few specific third-party libraries (e.g., Numpy) or focus on generating code involving one single external library, our work proposes a novel two-stage approach which is able to generate code for multiple arbitrary libraries.

Retrieval-augmented techniques have gained attention in natural language text generation tasks [28] and software engineering tasks like code generation, summarization, and completion [29], [47], [48], [49], [50]. Parvez et al. [29] proposed the REDCODER framework that retrieves relevant code/summaries using dense embedding retrieval and supplements them to code generation/summarization models. Our approach also uses retrieval to obtain import statements/code snippets for specific libraries that are similar to the given description. To the best of our knowledge, this is the first application of retrieval-augmented techniques to the import generation task.

Researchers have created benchmarks for various software engineering tasks, including code generation, code search, and defect repair, to facilitate evaluation on the same benchmark [18], [27], [51], [52], [38]. CONCODE [18], created by Iyer *et al.*, is a widely-used benchmark for natural language to code generation, consisting of over 100,000 examples of Java classes from online code repositories. However, it focuses on general code generation rather than specifically targeting the task of library-oriented code generation. Our work is the first to construct a large-scale dataset for the task of library-oriented code generation.

## VI. Conclusions

In this work, we proposed CodeGen4Libs, a novel approach which incorporates two stages (i.e., import generation and code generation) to enable more accurate library-oriented code generation. Our experiments demonstrated its superior performance compared to existing approaches, and our questionnaire provided insights into the demand for library-oriented code generation. Our work highlights the importance of considering import statements in code generation tasks, with potential to significantly improve software development efficiency and effectiveness. Future work includes expanding to other programming languages and libraries and improving import generation performance.

## Acknowledgment

This work is supported by National Natural Science Foundation of China under Grant No. 61972098.

## References

[1] Z. Yang, S. Chen, C. Gao, Z. Li, G. Li, and R. Lv, "Deep learning based code generation methods: A literature review," *arXiv preprint arXiv:2303.01056*, 2023.

[2] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," *arXiv preprint arXiv:2308.01240*, 2023.

[3] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 2021, pp. 8696–8708.

[4] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "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*, 2021.

[5] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[6] M. Ciniselli, L. Pascarella, E. Aghajani, S. Scalabrino, R. Oliveto, and G. Bavota, "Source code recommender systems: The practitioners' perspective," *arXiv preprint arXiv:2302.04098*, 2023.

[7] M. Liu, X. Peng, A. Marcus, S. Xing, C. Treude, and C. Zhao, "Api-related developer information needs in stack overflow," *IEEE Trans. Software Eng.*, vol. 48, no. 11, pp. 4485–4500, 2022.

[8] M. Liu, X. Peng, A. Marcus, C. Treude, J. Xie, H. Xu, and Y. Yang, "How to formulate specific how-to questions in software development?" in *30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2020, November 14-18, 2022, Virtual Event, Singapore*. ACM, 2020, pp. 1015–1026.

[9] M. Liu, X. Peng, Q. Jiang, A. Marcus, J. Yang, and W. Zhao, "Searching stackoverflow questions with multi-faceted categorization," in *Proceedings of the Tenth Asia-Pacific Symposium on Internetware, Internetware 2018, Beijing, China, September 16-16, 2018*. ACM, 2018, pp. 10:1–10:10.

[10] J. Liu, S. Baltes, C. Treude, D. Lo, Y. Zhang, and X. Xia, "Characterizing search activities on stack overflow," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 2021, pp. 919–931.

[11] M. Liu, S. Yu, X. Peng, X. Du, T. Yang, H. Xu, and G. Zhang, "Knowledge graph based explainable question retrieval for programming tasks," 2023.

[12] C. Wang, X. Peng, Z. Xing, Y. Zhang, M. Liu, R. Luo, and X. Meng, "Xcos: Explainable code search based on query scoping and knowledge graph," *ACM Transactions on Software Engineering and Methodology*, 2023.

[13] (2023) Replication package. [Online]. Available: https://github.com/FudanSELab/codegen4libs

[14] (2023) Github code dataset. [Online]. Available: https://huggingface.co/datasets/codeparrot/github-code

[15] (2023) javalang. [Online]. Available: https://github.com/c2nes/javalang

[16] (2023) Jdk 8 documentation. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/overview-summary.html

[17] (2023) Android api reference. [Online]. Available: https://developer.android.com/reference

[18] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *arXiv preprint arXiv:1808.09588*, 2018.

[19] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. Association for Computational Linguistics, 2020, pp. 7871–7880.

[20] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[21] (2023) microsoft/codegpt-small-java-adaptedgpt2. [Online]. Available: https://huggingface.co/microsoft/CodeGPT-small-java-adaptedGPT2

[22] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020.

[23] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 2022, pp. 39–51.

[24] (2023) Zzr0/issta22-codestudy. [Online]. Available: https://github.com/ZZR0/ISSTA22-CodeStudy

[25] (2023) Ahmedssoliman/codexglue-concode. [Online]. Available: https://huggingface.co/datasets/AhmedSSoliman/CodeXGLUE-CONCODE

[26] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[27] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *ArXiv*, vol. abs/2102.04664, 2021.

[28] H. Li, Y. Su, D. Cai, Y. Wang, and L. Liu, "A survey on retrieval-augmented text generation," *arXiv preprint arXiv:2202.01110*, 2022.

[29] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," *ArXiv*, vol. abs/2108.11601, 2021.

[30] E. Shi, Y. Wang, W. Tao, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Race: Retrieval-augmented commit message generation," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022, pp. 5520–5530.

[31] S. E. Robertson and S. Walker, "Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval," in *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*. ACM/Springer, 1994, pp. 232–241.

[32] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2136–2148.

[33] (2023) Elasticsearch. [Online]. Available: https://github.com/elastic/elasticsearch

[34] (2023) Transformers. [Online]. Available: https://github.com/huggingface/transformers

[35] (2023) Salesforce/codet5-base. [Online]. Available: https://huggingface.co/Salesforce/codet5-base

[36] C. Yang, Y. Liu, and C. Yin, "Recent advances in intelligent source code generation: A survey on natural language based studies," *Entropy*, vol. 23, 2021.

[37] J. Shin and J. Nam, "A survey of automatic code generation fromnatural language," *J. Inf. Process. Syst.*, vol. 17, pp. 537–555, 2021.

[38] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[39] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.

[40] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kociský, F. Wang, and A. W. Senior, "Latent predictor networks for code generation," *ArXiv*, vol. abs/1603.06744, 2016.

[41] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," *ArXiv*, vol. abs/1911.09983, 2019.

[42] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "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*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[43] D. Fried, A. Aghajanyan, J. Lin, S. I. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *ArXiv*, vol. abs/2204.05999, 2022.

[44] D. Zan, B. Chen, Y. Gong, J. Cao, F. Zhang, B. Wu, B. Guan, Y. Yin, and Y. Wang, "Private-library-oriented code generation with large language models," *arXiv preprint arXiv:2307.15370*, 2023.

[45] D. Zan, B. Chen, Z. Lin, B. Guan, Y. Wang, and J. Lou, "When language model meets private library," in *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*. Association for Computational Linguistics, 2022, pp. 277–288.

[46] D. Zan, B. Chen, D. Yang, Z. Lin, M. Kim, B. Guan, Y. Wang, W. Chen, and J. Lou, "CERT: continual pre-training on sketches for library-oriented code generation," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. ijcai.org, 2022, pp. 2369–2375.

[47] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," *arXiv preprint arXiv:2006.05405*, 2020.

[48] S. Lu, N. Duan, H. Han, D. Guo, S. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," 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*. Association for Computational Linguistics, 2022, pp. 6227–6240.

[49] J. Li, Y. Li, G. Li, Z. Jin, Y. Hao, and X. Hu, "Skcoder: A sketch-based approach for automatic code generation," *arXiv preprint arXiv:2302.06144*, 2023.

[50] F. Zhang, B. Chen, Y. Zhang, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," *arXiv preprint arXiv:2303.12570*, 2023.

[51] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *ArXiv*, vol. abs/1909.09436, 2019.

[52] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.