Multi-modal API Recommendation

Anonymous Authors

***Abstract*—Too many options can be a problem, which is the case for Application Programming Interfaces (APIs). As there are many of such APIs, with many more being introduced periodically, it raises a problem of choosing which API to be recommended. Furthermore, numerous APIs are commonly used together with other complementary third-party APIs. It can be challenging for developers to understand how to use each API and to remember all the complementary APIs for the API they want to use. Therefore, an accurate API recommendation approach can improve developers efficiency in implementing a certain functionality. Several approaches have been developed to automatically recommend APIs based on either a natural language query or source code context. However, none of these API recommendation approaches have utilized these two sources of information at the same time (i.e., leveraging natural language query and source code context *together*). In this work, we propose an approach named MULAREC, which leverages the information from natural language query (annotation) and source code context. The results confirm that our approach outperforms state-of-the-art API recommendation approaches which only leverage a single type of information as the input. Our work also demonstrates that multi-modal information can boost the performance of API recommendation approaches by 20%-50% better in terms of BLEU-score than the baselines.**

***Index Terms*—API Recommendation, Multi-modal, Pre-trained Models**

I. INTRODUCTION

Application Programming Interfaces (APIs) are widely used by developers as they can improve development efficiency [1]–[4]. However, appropriate understanding and correct usage of them is challenging as there exists a number of APIs in just a single library (e.g., JDK could contain thousands of them [2]). Consequently, many researchers [2], [3], [5]–[10] design API recommendation approaches to shorten the selection time and assist the correct usage of APIs. These approaches can typically be categorized into two types: (1) query-based and (2) code-based API recommendation. Query-based approaches [2], [5], [6] retrieve related APIs and provide recommendation for users by taking a natural language query that describes the intended functionality. On the other hand, code-based approaches [7]–[10] recommend the next API to use in a code by taking into account the surrounding code as context of the recommendation point.

While many of these query-based and code-based API recommendation approaches have been proposed, they naturally consider only natural language query and source code information, respectively. To the best of our knowledge, none of the existing approaches consider both natural language query and source code information for better API recommendation. As several studies [11]–[13] show that considering multiple modalities of code can be beneficial for source code analysis, this may be missed opportunity in API recommendation for better performance. Such an opportunity may occur when developers implement a method in an IDE. In such a scenario, they annotate the functionality of the method in the comment section (i.e., the natural language query/annotation) and then write the code implementing the functionality (i.e., the surrounding code context). Both of these sources of information are of a different modality (i.e., text vs. code) and are typically used for query-based and code-based API recommendation approaches, respectively.

Furthermore, most of the existing query-based and code-based API recommendation approaches only predict a set of APIs and do not predict the sequence on which the APIs must be invoked (with DeepAPI [2] and PAM [3] being the exceptions). As a consequence, developers need to spend extra efforts to figure out the correct invocation sequence of the given set of APIs. Therefore, there is a need for more approaches that can produce API sequences.

In this work, we deal with the aforementioned shortcomings and propose MULAREC (**Mul**ti-modal **A**PI **Rec**ommendation) to recommend sequences of APIs based on natural language annotation and source code context as inputs. These inputs are made of two different modalities (i.e., text vs. code) and sources (i.e., method comment and body). MULAREC takes both the code annotation and part of the source code to recommend an API sequence. This sequence helps developers to easily identify which APIs should be invoked to complete the method implementation, along with the order of invocation. MULAREC is a deep learning based approach built on top of CodeBERT [14]. It utilizes CodeBERT to semantically encode the annotation and the source code. It then fuses the two generated encodings to produce a single embedding that embodies the annotation and the source code. The embedding is then passed to a decoder that learns how to output the correct API sequence.

Since existing API recommendation approaches are either query-based or code-based, none of the existing datasets contains both natural-language annotation and surrounding source code context as inputs. Hence, we construct a new benchmark that contains multi-source (i.e., both natural language annotation and source code context) as inputs and API sequences as output. Our benchmark is constructed based on the 50K-C dataset [15] shared by Martins et al., which contains 50,000 compilable Java projects. We start with compilable Java projects to ensure that we can extract the correct API sequences (i.e., the API call in the code can be resolved to the correct API). After extracting the API sequences and data cleaning process, the benchmark contains 369,514 methods. Each method has an annotation, a source code context, and an extracted API sequence.

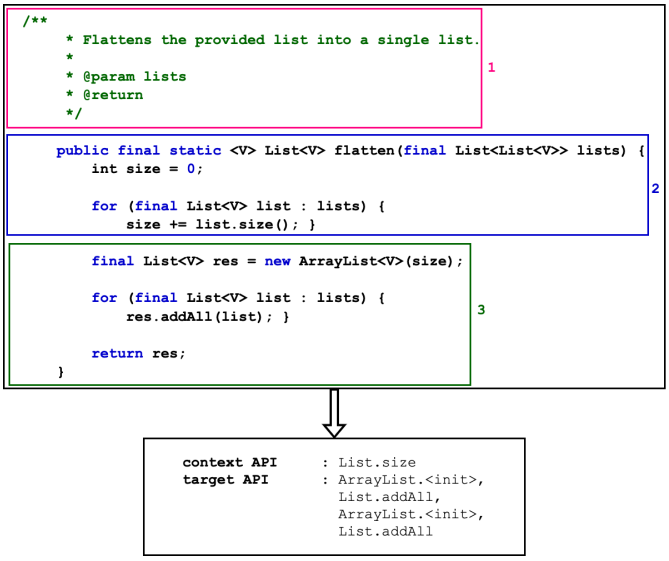

Fig. 1. Motivating example

To evaluate the effectiveness of MULAREC, we select the state-of-the-art approaches from both query-based and code-based API recommendations that recommend API sequences as our baseline approaches and leverage our benchmark. The experimental results show that MULAREC can outperform the state-of-the-art API recommendation approaches by reasonable margins, i.e., 50% for query-based and 20% for code-based, in terms of BLEU-4, respectively.

Our main contributions can be summarized as follows:

- To the best of our knowledge, we are the first to consider multi-modal information for API recommendation.
- We built a new API recommendation benchmark that contains both source code context and annotations.
- We propose a new approach named MULAREC that takes both the natural language and the code context into consideration. The proposed MULAREC outperforms the best-performing query-based, and code-based API recommendation approaches that recommend API sequences.

The remainder of the paper is organized as follows. Section II introduces the problem formulation and the benchmark creation process. We describe our approach in Section III. Section IV describes our experimental setup, including the dataset building, baseline approaches comparison, evaluation metrics, and the research questions investigated in our work. Experimental results are shown in Section V. We further discuss and analyze our results in Section VI. Section VII talks about the most related works. We conclude our work and show some potential future work in Section VII.

II. PROBLEM FORMULATION AND BENCHMARK CREATION

In this section, we formulate the problem and describe in detail how we construct our benchmark.

A. Problem Formulation

We formulate the API recommendation problem as a sequence generation task. Given the input of natural language and part of the source code, we aim to generate API sequence recommendations to be used in the latter part of the code. To achieve this, we leverage open source Java projects from Github to build a dataset that is tailored to our experiment.

Each data point in our dataset is built upon a Java method's structure, representing natural language (NL) and programming language (PL) part of the method. We divided every method into several components, as shown in Fig1. There are 3 items highlighted in this example:

1) Javadoc comment / Annotation
   The first sentence of the Javadoc comment is used as annotation.
2) Code context
   The method declaration and first 3 lines of the method's source code. API sequence extracted from this code context is called *context API*.
3) Target
   This is the part where we extract the *target API* as our reference. The target API will serve as training target and example's reference in our experiment.

For the example showed in Figure1, the annotation for this example is *"Flattens the provided list into a single list."*, while the code context is

```
int size = 0;
for (final List<V> list : lists) {
    size += list.size(); }
```

Based on the code context, the *context API* extracted for this example is `List.size` and the *target API* is made of 4 API calls, namely *ArrayList.⟨init⟩, List.addAll, ArrayList.⟨init⟩, List.addAll*

B. Benchmark Creation

To evaluate our approach, we need a benchmark that consists of annotation, source code, and API sequence extracted from the source code. Yet, there is no existing dataset that contains all of this information. Therefore, we decided to build our own benchmark. Initially, we aimed to start from an existing dataset from DeepAPI [2], which contains annotations and API sequences, and add the corresponding source code contexts. However, the dataset does not provide the source projects from which the annotations and API sequences were extracted. Thus, we built the benchmark from scratch.

As described in II-A, one data point in our benchmark represents a method implementation. The first sentence of the Javadoc comment is used as *annotation*, while the source code in the method's body is split into 2 parts. Method declaration and the first 3 lines of the body are used as the input, i.e., *context code* and *context API*, while the later part of the source code is used to extract the *target API*. Given the annotation and context code, our model aims to recommend the target API sequence.

To ensure the quality of extracted API sequence (i.e., the API call refers to the correct API), we built our benchmark based on the 50K-C dataset provided by Martins et al. [15],

which contains 50,000 compilable Java projects. The dataset building process is broken down as follows. First, we downloaded the project source code from the 50K-C website[1]. Second, we passed each Java file in the project through the Eclipse JDT parser[2] to extract methods in the project. For each method identified by the parser, we extract the method declaration, the method body, and the Javadoc annotation. Method declaration and body are mandatory components, while Javadoc annotation is an optional component and not every method is accompanied by a Javadoc annotation. For such methods, we discard them from our dataset because we cannot derive the first sentence of the Javadoc annotation to use as the natural language query.

Note that we only consider English annotation and thus discard methods that contain non-English annotation. Finally, we extracted the API sequence from the method body using the Eclipse JDT parser. For this API sequence extraction, we utilized the implementation provided by DeepAPI in their Github repository[3]. In total, we collected 624,487 annotation and source code context pairs.

Further post-processing was conducted to filter out the methods that are unsuitable for our task, as stated below:

1) Removal of methods with an insufficient number of lines of code and API calls. We removed methods having less than 4 lines of code and methods containing only 1 API call.
2) Removal of methods with no target APIs. After extracting the first 3 lines of the source code as *code context* and splitting the API sequence into *context API* and *target API*, we removed methods with no *target APIs*. This typically occurs when the method body is short, so all of the API calls are concentrated on the first 3 lines of the code.
3) Outlier removal based on the number of API calls, i.e., length of API sequence in the method body. After calculating the mean μ and standard deviation σ for the number of API calls per method in our benchmark, we removed the outlier methods based on 3-σ rule [16]. Methods that contain smaller than $\mu - 3\sigma$ or larger than $\mu - 3\sigma$ API calls are considered outliers and removed from the benchmark.

Following the processing of our benchmark, we removed 254,973 unqualified methods, and ended up with a total of 369,514 methods in our benchmark. We further split the benchmark into training, validation, and testing with a ratio of 8:1:1.

III. Approach

In this section, we first introduce the architecture of our proposed approach. After that, we discuss the details of each component.

[1]https://mondego.ics.uci.edu/projects/jbf/
[2]https://www.eclipse.org/jdt/
[3]https://github.com/guxd/deepAPI/issues/2

A. Architecture

Figure 2 shows the overall framework of MULAREC. MULAREC follows the typical Encoder-Decoder architecture [17]. It mainly consists of two modules. The first module is an *Encoder*, which encodes the source code context and the annotation into vectors, while the second module is a *Decoder*, which maps input and the encoded vector into output tokens until the end of the sequence token is reached.

The input of our proposed approach MULAREC is composed of two components: annotation and source code context. The output is an API sequence, which serves as a recommendation of API calls that should be used following the source code context. We initialize the *Encoder* with CodeBERT [14], and encodes annotation and source code context separately. After getting the two encoded vectors from the *Encoder*, *Normalization Layer* normalizes these two vectors and passed them to the *Fusion Layer*. Furthermore, the *Fusion Layer* combines these two vectors to obtain a feature vector that encodes the information from the two vectors. Next, the feature vector is passed to the Decoder Module, which decodes the vector into an API sequence.

The details of each module and layer are introduced as follows.

B. Encoder Module

As mentioned earlier, the *Encoder* maps the input sequence to a contextualized encoding sequence. We adopt CodeBERT [14] as our encoding backbone since it is a bimodal pre-trained model for both of our targets (i.e., source code and NL). Moreover, its large-scale (2.1M) bimodal data points across 6 programming languages are known as beneficial [18]. It is specifically designed with a Transformer [19]. It has 12 layers, 768-dimensional hidden states, and 12 attention heads. CodeBERT has been employed in different tasks [20]–[22] to encode the source code and natural language tokens, and it has shown to be more than capable of solving a range of different types of programming understanding and generation tasks [23]. We fine-tune CodeBERT as the *Encoder* on our API recommendation task and generate two different vectors (i.e., annotation and code context) as the input of the next layer.

C. Normalization Layer

After getting the two vectors that represent the annotation and the code context, i.e., E_{ant} and E_{code}, respectively, we normalize them to ensure they lie in the same scope. Inspired by the prior works [24], [25], we adopt the L_2 normalization on the annotation vector and the code context vector. The formula to get the normalized vectors $\overrightarrow{V_{ant}}$ and $\overrightarrow{V_{code}}$ are as follows.

$$\overrightarrow{V_{ant}} = \frac{E_{ant}}{\|E_{ant}\|} = \frac{E_{ant}}{\sqrt{\sum_{i=1}^{n} {E_{ant}}_i^2}} \tag{1}$$

$$\overrightarrow{V_{code}} = \frac{E_{code}}{\|E_{code}\|} = \frac{E_{code}}{\sqrt{\sum_{i=1}^{n} {E_{code}}_i^2}} \tag{2}$$

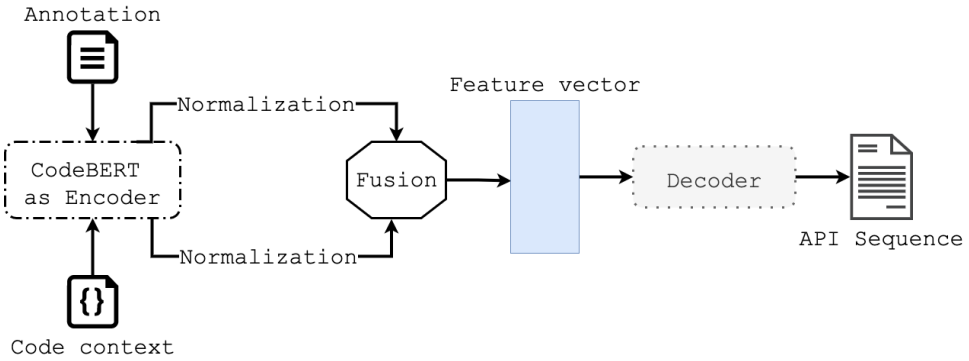

Fig. 2. The architecture of MULAREC

where n is the dimension of the vector E.

D. Fusion Layer

There exists a wide range of use-case scenarios where unifying the obtained embeddings would be inevitable. We define the unification of embeddings to include concatenation, combination, and fusion. Although generated embeddings already represent software elements, these embeddings tend to contain inadequate information. For example, multi-modal analysis tasks that encode each unimodal data to get different embeddings, or a task that requires sub-embeddings, such as embedding of each method to represent a class, etc [26]–[28]. To decode them, embedding unification is generally necessary to represent the multi-modality or a high-dimensional embedding. Towards this issue, multiple studies [29], [30] focus on the automation of this unification process.

To obtain a comprehensive understanding of different (multi-modal) vectors without losing the necessary information, we apply a fusion layer. Yang et al. [31] discovered that the similarities and differences between multi-modal vectors affect the representation learning performance. Following the prior works [25], [31], we fuse both (i.e., the annotation and code context) vectors following the formula below:

$$\overrightarrow{O_1} = F_1\left(\left[\overrightarrow{V_{ant}}; \overrightarrow{V_{code}}\right]\right) \tag{3}$$

$$\overrightarrow{O_2} = F_2\left(\left[\overrightarrow{V_{ant}}; \overrightarrow{V_{ant}} - \overrightarrow{V_{code}}\right]\right) \tag{4}$$

$$\overrightarrow{O_3} = F_3\left(\left[\overrightarrow{V_{ant}}; \overrightarrow{V_{ant}} \circ \overrightarrow{V_{code}}\right]\right) \tag{5}$$

$$\overrightarrow{O} = F\left(\left[\overrightarrow{O_1}; \overrightarrow{O_2}; \overrightarrow{O_3}\right]\right) \tag{6}$$

where $\overrightarrow{V_{ant}}$ and $\overrightarrow{V_{code}}$ represents the annotation and code context vectors, respectively. $\circ$ denotes element-wise multiplication. Each of F_1, F_2, F_3, and F consists of single-layer feedforward networks with independent parameters. The concatenation process generates three embeddings first considering the similarities and differences between the individual vectors. Finally, we concatenate these three vectors into a feature vector such that it can represent both the annotation and the source code context without losing information.

E. Decoder Module

The *Decoder* is used to generate API sequences from the feature vector. Following the setting used by Lu et al. [20], we use the randomly initialized Transformer [19] with 6 layers, 768-dimensional hidden states, and 12 attention heads. Each layer contains three sublayers: decoder self-attention, encoder-decoder attention, and position-wise feed-forward networks [32]. These sublayers employ a residual connection around them followed by a layer normalization to generate one word at a time, from left to right. They attend to both the previously generated APIs and the final representations generated by the *Encoder*.

IV. EXPERIMENTAL SETUP

In this section, we elaborate on our experiment setup, including the dataset building, description of the baseline approaches, and evaluation metrics. Finally, we raise the research questions that are investigated in this work. Our replication package is publicly available.[4]

A. Baseline Approaches

We selected the state-of-the-art approaches that recommend API sequences as the baseline approaches. We compare the performance of MULAREC with three baseline approaches as follows:

PAM [3] (Probabilistic API Miner) is the best performing code-based API sequence recommendation approach, as evaluated by Peng at al. [1]. It is a near parameter-free probabilistic algorithm that is used to mine the *most interesting* API sequence patterns. It leverages a probabilistic model of sequences based on generating a sequence by interleaving a group of sub-sequences. PAM is a context-insensitive approach, which means that PAM does not take into account any input from the test data to provide a recommendation. Instead, it directly utilizes the top-N highest probability sequence mined from the training data as a recommendation for each of the test data. Compared to other pattern miner algorithms such as MAPO or UPMiner, PAM evidently provides better pattern to supplement developer-written API sequence as it generates more diverse and less redundant API in the result.

[4]https://anonymous.4open.science/r/MulaRec/README.md

In our task, we leverage context API in the test data to choose the best sequence patterns from mined patterns from PAM. First, we filter out any mined patterns that do not overlap with the context API. The remaining patterns are then sorted by the probability of them appearing in the training set. After removing the irrelevant patterns, we then pick the first pattern that begins with the context API. In the event that there are no patterns that start with the context API, we choose the pattern with the highest probability score to be used as the recommendation. The recommendation is generated by excluding the context API from the chosen pattern.

DeepAPI [2] is the best performing query-based API sequence recommendation approach, as evaluated by Peng at al. [1]. It is a query-based API recommendation approach proposed by Gu et al., and it is the first deep learning-based approach that generates API sequence recommendation for a given natural language query. They formulate the API sequence generation as a machine translation problem, where the aim of the model is to translate natural language description, i.e., annotation into an API sequence. It adapts the neural language model called RNN Encoder-decoder in their approach. DeepAPI shows dominance over the traditional bag of words approach because of the usage of word embedding that helps the model to recognize semantically similar words. To generate the API sequence, DeepAPI employs the beam search strategy.

In this experiment, we modified the beam search strategy to incorporate context API into the API sequence generation. Further details about the beam search and how we incorporated the context API are explained in Section IV-C1. As DeepAPI is trained to predict a full sequence of API given an annotation, it raises an issue of unfairness when the model is expected to predict only target API in the evaluation. For the sake of fair evaluation, instead of predicting full API sequence for a given annotation, we provide the context API in the beginning of the beam search to compensate for the absence of the code context information. We reckon this will enable a fair evaluation when we compare the generated API sequence with target API.

CodeBERT [14]. Recently, Martin and Guo have shown that CodeBERT is more capable of recommending API sequence than DeepAPI [33]. Their study first reproduced the results of DeepAPI. Other than the original Java dataset, they also curated a Python dataset. They evaluated DeepAPI and CodeBERT on these two datasets. The experimental results show that CodeBERT outperforms DeepAPI to a large extent. Note that both datasets belong to the query-based API recommendation: generating the API sequence given a natural language description. In our work, we treat annotation and context code as the query separately and run CodeBERT as our baseline. Thus, we have two variants of CodeBERT. The first is CodeBERT-annotation, a baseline for query-based approach, which accepts the annotation as input. The second is CodeBERT-code, a baseline for code-based approach, which accepts the code context as input.

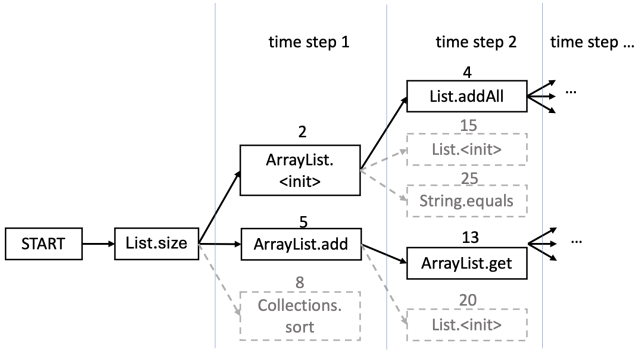

Fig. 3. Illustration of beam search, beam width=2

B. Hyperparameter Setting

We used the default hyperparameters as mentioned in the replication package for DeepAPI[5] and PAM[6]. As for CodeBERT and our proposed method MULAREC, we set the maximum token length of both the source sequence and the target sequence to 256. All the models were trained for 30 epochs.

C. Evaluation

1) Beam Search: To conduct a fair evaluation for model that only uses annotation as input, i.e., CodeBERT-annotation and DeepAPI, we incorporate the context API information via beam search [34].

Beam search is a heuristic search strategy that produces API sequences with the least cost value (i.e., the highest probability) given by the language model. At each time step, it takes top-*n* API branch with the least cost value to continue the search, with *n* is the size of the beam width. It then prunes the other branches that are not chosen and continue the search from the selected branches. We illustrate our modified beam search with an example in Figure3. In this example, the input annotation is *"Flattens the provided list into a single list."*. There is a *context API* introduced in the search process, which is `List.size`. For the context API, we disregard the cost value and directly choose the context API as the start of the sequence. The time step starts after all of the context API(s) have been selected in the search pipeline. The search continues from the last context API to calculate the cost of all tokens appearing after the context API. In the example, at the first time step, `ArrayList.<init>` and `ArrayList.add` are chosen because these APIs have the least cost value of 2 and 5, respectively. The search process ignores other branches and only expand the search for these two branches in the next time step. The branch expansion continues until the end-of-sequence symbol is reached or the maximum length of sequence has been generated. In total, beam search will generates *n* sequences that we can consider to use. Even though we aim to provide only one result for each

[5]https://github.com/guxd/deepAPI
[6]https://github.com/mast-group/api-mining

test data, we use $n>1$ for the search process in order to expand more branches. This will prevent the branch to get stuck in the local minima situation. Finally, we the API sequence with the least cost value is selected as the recommended sequence.

2) Metrics: As we are aiming to generate API sequence, BLEU score [35] is adopted to evaluate the performance of our approach. This metric measures the ability to generate accurate sequence by comparing the recommended sequence with the target sequence mined from the method source code (i.e., ground truth). Though it is commonly used in the machine translation problem, we regard BLEU score as a suitable metric to gauge how close the generated sequence compared to the human-written API sequence. This metric has also been used in past work that generate API sequence [2], [33]. BLEU score is expressed mathematically as below:

$$BLEU = BP * exp\left(\sum_{n=1}^{N} w_n log\left(P_n\right)\right) \tag{7}$$

$$BP = \begin{cases} 1 & c \geqslant r \\ exp\left(1 - \frac{r}{c}\right) & c < r \end{cases} \tag{8}$$

In the above equation, BP is the brevity penalty, which aims to penalize the generated sequence that is shorter than the reference. r and c refer to the number of words in ground truth and candidate respectively. w_n refers to weights given at n-gram point, and P_n is the number of precision of n-gram.

BLEU score ranges from 0-1. BLEU score of 1 means that the generated API sequence matches perfectly with the ground truth. We measure the performance of the approach on each pair of annotation and code with cumulative n-gram BLEU score with n=1,2,3,4. The cumulative scores refer to the calculation of individual n-gram scores at all orders from 1 to n and weighting them by calculating the weighted geometric mean. For 2-gram BLEU score, i.e., BLUE-2, we set the weight to [1/2, 1/2], while for the BLUE-3 the weight should be set to [1/3, 1/3, 1/3]. Finally, we set the weight to be [1/4, 1/4, 1/4, 1/4] while calculating the BLEU-4 score.

D. Research Questions

In this work, we aim to answer three Research Questions (RQs) as follows:

- **RQ1:** How effective and efficient is our proposed approach MULAREC compared to state-of-the-art API sequence recommendation approaches?
  We compare the effectiveness of MULAREC with baseline approaches (i.e., DeepAPI, PAM, CodeBERT-annotation, and CodeBERT-code) in terms of BLEU score. For MULAREC, DeepAPI, CodeBERT-annotation, and CodeBERT-code, we use training set to train their models and the validation set to estimate the effectiveness of the learned models. For PAM, we use the training set to mine the sequence patterns. We then evaluate the effectiveness of the approaches in the testing data. Moreover, we also compare their efficiency by measuring their evaluation time.

TABLE I
BLEU SCORES OF DIFFERENT API SEQUENCE RECOMMENDATION APPROACHES

Approach	BLEU-1	BLEU-2	BLEU-3	BLEU-4
PAM	0.17	0.06	0.02	0.01
DeepAPI	0.30	0.23	0.19	0.12
CodeBERT-annotation	0.52	0.47	0.42	0.24
CodeBERT-code	0.59	0.56	0.53	0.30
MULAREC	**0.66**	**0.62**	**0.58**	**0.36**

- **RQ2:** What is the contribution of each modality to the effectiveness of MULAREC?
  In this research question, we investigate the contribution of each modality to the effectiveness of MULAREC. We aim to show that each modality is important. To do so, we report the effectiveness of MULAREC when any of the two modalities do not exist. We omit the modality from the MULAREC inputs. We run MULAREC with the input having only a single source of information, i.e., annotation only and code only. For the input with only annotation, we set the code part as empty; similarly, for the input with only code, we set the annotation part as empty.
- **RQ3:** What is the effect of different concatenation strategies to the effectiveness of MULAREC?
  Since we have two types of sequences (i.e., the natural language and the source code context), we need to combine these two types of inputs. The first strategy we experiment with is following the CONCODE dataset [36]. The CONCODE dataset is used for text-to-code generation task, specifically, generating class member functions from a natural language queries and a class environment. In our API recommendation setting, we have a natural language annotation and a source code context. Considering the similarity between the text-to-code generation task and our task, we combine the annotation and the code context together as the input, where the two are separated by a special token. We refer to this concatenation strategy as *concatenating sequences*. Other than concatenating the two sequences before feeding them into the model, another strategy that we experiment with is feeding the sequences into the model separately and then combining the generated vectors. We refer to this concatenation strategy as *concatenating vectors*. This strategy is the one employed by the *Fusion Layer* of MULAREC. We investigate how these concatenation strategies perform and analyze why one concatenation strategy may be better than the other.

V. RESULTS

In this section, we show the experimental results and answer the RQs.

A. Effectiveness and Efficiency of MULAREC

In this research question, we aim to understand how MULAREC compares with state-of-the-art API sequence recommen-

TABLE II
BLEU SCORES OF MULAREC UNDER WITH DIFFERENT SOURCES OF INFORMATION

Source	BLEU-1	BLEU-2	BLEU-3	BLEU-4
annotation	0.21	0.10	0.06	0.02
code	0.57	0.52	0.47	0.27
annotation + code	**0.66**	**0.62**	**0.58**	**0.36**

dation approaches. Table I shows that MULAREC consistently outperforms the baseline approaches, with CodeBERT-code as the best performing baseline. In terms of BLEU-4, MULAREC outperforms CodeBERT-code by 20%.

Moreover, MULAREC can perform the recommendation reasonably fast in an average time of 380ms. DeepAPI, CodeBERT-annotation, and CodeBERT-code also take the similar amount of time to generate a recommendation with subpar performance compared to MULAREC. PAM is the fastest and on average, it generates a recommendation within 150ms. However, this does not come without a cost in accuracy. As shown by the experiment result in Table I, PAM is more than 30x less effective compared to MULAREC, with only 25% improvement in terms of recommendation time. The reason that PAM could give a recommendation quickly is that it is context-insensitive and it stops searching once it has found a pattern that begins with the context API.

Answer to RQ1: MULAREC achieves the highest BLEU-4 score of 0.36, which outperforms the second best performer, i.e., CodeBERT-code by 20%. PAM performs the worst with the BLEU-4 of 0.01.

B. Contribution of Each Modality

Table II shows the effectiveness of MULAREC when considering different types of information. When both modalities are considered, the effectiveness of MULAREC is the best. We find that the source code context plays a more profound role than the annotation. When the annotation is absent, the value of BLEU-4 drops by 25%, while in the absence of code context, the value of BLEU-4 drops by 94% instead. However, when combined, the performance of bimodal input outperforms the performance of annotation and code if used individually. This implies that each modality has its own characteristics and the benefit of leveraging both modality would be multiplied as compared of leveraging only one of them. These results also present the capability of MULAREC to learn the information carried by the bimodal input and leverage the joined information to provide better recommendations.

Answer to RQ2: When both modalities are utilized, MULAREC performs the best. When only a single modality is leveraged, the code context contributes more than the annotation.

TABLE III
COMPARISON OF DIFFERENT STRATEGIES IN MULAREC

Strategy	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Concatenating sequences	0.55	0.51	0.47	0.27
Concatenating vectors	**0.66**	**0.62**	**0.58**	**0.36**

C. Effect of Different Concatenation Strategies

Table III shows the performance of MULAREC when considering different concatenation strategies. We find that when simply concatenating the two types of input (i.e., concatenating sequences), the performance is worse than the performance of concatenating vectors. There are two potential causes. The first potential cause is the model cannot fully distinguish the annotation and the source code context. Following prior work on text-to-code generation task [36], to create the input, we also directly combine the annotation and the code context together as the input. The other potential cause is the length limitation. When concatenating sequence, the maximum sequence length of CodeBERT is shared among the two types of input. On the other hand, when concatenating vectors, the maximum sequence length can be fully utilized for each type of input. Thus, more information can be encoded when using concatenating vectors strategy.

Answer to RQ3: Concatenating the annotation and code vectors can achieve better performance than directly concatenating the annotation and code sequences as the input.

VI. DISCUSSION

A. Failure Analysis

Although our proposed method MULAREC outperforms state-of-the-art API sequence recommendation approaches, it still has limitations. To understand why MULAREC fails, we conduct a qualitative analysis of the failure cases as follows.

Correct domain but wrong target API. Consider the following example:

annotation: *"updates the frame with the attributes from the given PLP image"*

code context:

```
PLPFrame.updateFrameState() {
Insets insets=getInsets();
int w=im.getWidth();
w=w + insets.left + insets.right; }
```

The API sequence generated by MULAREC for the above annotation and code context is `BufferedImage.getHeight`. This generated API differs from the target API, which should be `ImageIcon.setImage`. However, we can see that

the model could infer the intent of this query, which is to use API that is related to image domain.

Recommending more APIs than necessary. While most of the result of combining the annotation and context code, i.e., bimodal input, improves the overall sequence generation up to 33.3% compared to unimodal input, there is a case where the result is further than the target AP, as shown in the example below:

annotation: *"This method First build the Query and pass to SQLUTIL for execution"*

code context:

```
SQLMaker.executeDelete(String
tableName, String whereClause) {

StringBuffer fullQuery=new
StringBuffer();
fullQuery.append(
DatabaseOperationConstant.DELETE_FROM
+ );
fullQuery.append(tableName + WHERE +
whereClause+ ;);
```

MULAREC that is based on either annotation or source code context only predicted the correct API, which is `StringBuffer.toString`, as the recommended API sequence. However, MULAREC outputs `String.isEmpty`, `StringBuffer.append`, `StringBuffer.toString`. Even though MULAREC has `StringBuffer.toString` in the recommended API sequence, its BLEU score is smaller. This suggests that further research in leveraging multimodal input may be needed to explore the best way to combine different modality for better understanding. The understanding should also consider how many APIs should be recommended.

Semantic gap in translation. Although MULAREC shows a significant improvement over the baselines, there exist some cases where the MULAREC's prediction is far from the target API both lexically and semantically. Consider the following example:

annotation: *"Updates all of the positions of the carousel does not do a repaint just does the math ready for the next one"*

code context:

```
CaroselLayout.recalculateCarosel() {
numberOfItems=recalculateVisibleItems();
try { boolean animate=false;
```

For the above example, the result given by MULAREC is `Label.setText`, while the correct API sequence is `Component.isVisible`, `Timer.start`. As comparison, this test data yield even worse result when passed to the DeepAPI. It gave a longer but repetitive sequence, such as `map<object>.values`, `map<object>.values`, `map<object>.entryset`, `entry<object>.getvalue`, `entry<object>.getkey`, `entry<object>.getvalue` as the API sequence recommendation. This example shows that there is still a gap in translating the annotation and context code to API sequence. Even though DeepAPI and MULAREC used language model that enables them to leverage semantic information of the input data, it is clear that more attention is needed to be given to the representation model, possibly by incorporating more semantic knowledge. Some possible directions is to experiment with other models to represent the input data. This is motivated from the fact that DeepAPI is able to generate a better API sequence due to the usage of word embeddings, and our approach MULAREC that leverage pre-trained model CodeBERT yields even better result than deepAPI, even on its variant that only use annotation as input. Each step incorporates more semantic knowledge that contributes to the effectiveness improvement. Another possibility to incorporate more semantic knowledge is to obtain knowledge from other sources such as Stack Overflow.

B. Lessons Learned

Multi-modal information can improve effectiveness. Based on the results of our experiments, multi-modal input improves the effectiveness of MULAREC as compared to using only uni-modal input. Even though using the natural language annotation yields lower performance compared to using source code context alone, they produce the best results when used together. This indicates that, in the case of API sequence recommendation, multi-modality brings more advantages than using only one modal of input. It may suggest that, while both the annotation and the source code context express what the code is doing, they may express different parts of it, and listening to both would improve the performance. It may also be possible that some parts are easier to understand from the annotation and some other parts are easier to understand from the source code context.

Another lesson that can be taken from our results is that, when CodeBERT is used for a single input, i.e., annotation or code context only, it yields better results than MULAREC using only a single input. This may happen because MULAREC is designed to handle bimodal input. Nevertheless, this emphasizes the fact that we could improve the performance by enriching the code with natural language information. Future work can potentially further enrich the information from other resources, such as API documentation and Stack Overflow.

Strategy to leverage multi-modal information matters. Directly concatenating sequences of natural language annotation and source code context only bring little improvement compared to the concatenating vectors of their encoding. This

shows that the natural language annotation and the source code content are best treated using a separate encoding layer. While both strategies are plausible, they lead to a significant performance difference. It highlights the importance of selecting the correct strategy when leveraging multi-modal information.

Our findings suggest that future work should explore different ways to leverage multi-modal information. In this work, we explore different concatenation strategies, focusing on whether concatenating the raw sequences or encoded vectors would bring more performance gain. However, another possible direction is to combine the API sequence recommendations output with a model built on a unimodal input. The sequences generated from the two models (i.e., one for each modality) can be strategically merged to achieve better performance.

C. Threats to Validity

Threats to internal validity relate to several aspects in our experiment, such as baseline implementation and evaluation strategy. We implemented DeepAPI and PAM with our dataset by utilizing the replication package made available by the authors of both papers. Thus, the risk of incorrect baseline implementation should be mitigated. Moreover, since DeepAPI outputs the full API sequence based on the given annotation, there is a risk of unfairness if we directly compare the generated API sequence with the target API. To mitigate this, we modified its beam search to introduce the information from context API when generating the API sequence. The same modification is also applied to CodeBERT-annotation. Hence, we believe the threats to internal validity should be minimal.

Threats to external validity relate to the generalizability of our results. In this study, we only experiment with the API sequence generation task on Java programming language, raising a concern of whether the result of this study could be generalized towards other programming languages. However, we believe the impact of the programming language should not be significant because Java is one of the most used programming languages and the same setting has been used in the past API recommendation work [2], [3], [33]. Yet, the results of our experiment motivate us to evaluate our API recommendation system on other popular programming languages such as Python.

VII. Related Work

In this section, we review three lines of research: (1) API recommendation, (2) API-related empirical studies, and (3) software artifact generation.

A. API Recommendation

There are mainly two types of API recommendation approaches, namely code-based and query-based. These two types are categorized based on the input of the API recommendation model.

Query-Based Approaches: Aside from DeepAPI, many other query-based approaches have been proposed. Most of them work on an API call recommendation instead of an API sequence recommendation. We selected DeepAPI and CodeBERT as our query-based API recommendation baselines based on the recent study conducted by Martin and Guo [33]. Other than proposing different models for query-based API recommendation, query reformulation, which changes the query instead, also draws research interest [1]. Peng et al. [1] show that, query reformulation methods such as query expansion and query modification are helpful in API recommendation. Query expansion aims to introduce relevant tokens that are not included in the original query, while query modification aims to mitigate both the lexical and knowledge gap between the natural language query and the target sequence. Furthermore, domain-specific knowledge representation techniques usually play a role in representing the query better. One such example is Post2Vec [37]. Xu et al. show that better representation of information from external resources such as Stack Overflow could improve the performance of the API recommendation technique such as BIKER [5]. In the future, we also plan to incorporate the query-reformulation techniques or domain-specific representation techniques in MulaRec to better represent queries, which may further boost the API sequence recommendation performance.

Code-Based Approaches: In the realm of code-based API recommendation, there are two types of approaches to handle the problem [1]: i.e., pattern-based and learning-based. We compare MulaRec with pattern-based approaches because the learning-based approaches typically treat the API recommendation as a next-token prediction problem. Learning-based approaches such as GraLan [38] leverage the statistical language model to obtain the next token prediction. GraLan is a graph-based statistical language model that can be used to generate an API call recommendation given a code context. API calls that are present in the training data are modeled into a graph where the nodes represent actions (i.e., classes, method calls) while the edges represent control and data flow dependency between the nodes. It leverages Bayesian statistical inference to calculate the probability that an API would be used given a subgraph as input. Differing from our works, GraLan also utilizes the information obtained from the AST such as while loop and branches in their graph. In the API generation process, the recommendation algorithm extracts the code context to be used as input. Then, GraLan is used to compute the probabilities of the children graphs given the context graph. Each child has a probability value, which is used to compute the scores for ranking the API suggestion. In this study, we choose PAM as one of the baseline for code-based recommendation along with CodeBERT because it generates API sequence instead of API call. Moreover, a sequence is preferred for encoding API usage compared to graph due to its simplicity and it is easy to understand [39].

Other Approaches: Aside from the above families, other approaches have been proposed. We briefly discuss the two recent works; one works on the cross-library API recommendation, while the other utilizes Stack Overflow posts.

APIRecX [40] is the first approach that focuses on handling the Out-Of-Vocabulary (OOV) issue on the task of cross-

library API recommendation. APIRecX is able to recommend API calls for new libraries. To relieve the OOV problem at the API level, it first splits each API call into a sequence of subwords. Then, a GPT-based sub-word language model was pre-trained on a large number of API usage data. In the fine-tuning stage, APIRecX first predicts subwords and then incorporates beam search to compose an API call.

CLEAR [41] is an API recommendation approach that leverages Stack Overflow posts. CLEAR leverages BERT sentence embedding to preserve the semantic information in queries and Stack Overflow posts. Furthermore, CLEAR utilizes contrastive learning to distinguish further the queries which are semantically dissimilar while lexically similar.

MULAREC has a different type of input from these two approaches. In the future, we also plan to consider leveraging Stack Overflow to boost the performance of MULAREC further.

B. Empirical Study on APIs

Several empirical studies have been performed to understand API usages [42], [43] and API usability [39].

Zhong and Mei [42] conducted an empirical study on API usage, providing nine important findings. The most relevant findings with our approach are related to (1) the best format to define API usage and (2) how developers use APIs from different libraries. To accurately define the API usage, extracting the API graphs from the source code is recommended. However, the sequence is more popular for encoding API usage because of its simplicity which supports our motivation.

To benefit from a deeper understanding of developer behaviors/activities, Xu et al. [43] investigated the reuse and re-implementation of libraries/APIs. They conducted two types of surveys (i.e., individual survey to corresponding developers of the validated instances and another open survey). They received 36/139 and 13/71 responses for library reuse and re-implementation, respectively, from the contacted developers, as well as 56 responses from the open survey. The results demonstrated that library reuse mainly occurs due to the lack of initial knowledge of their existence, and re-implementation occurs due to the complexity of the library dependencies, deprecation, or lack the partial implementations.

Another empirical study [39] has been conducted investigating API usability. They collected and interpreted the responses from the developers regarding the cognitive dimensions. Specifically, they studied usability tokens, an original classification of the developers' reactions such as "surprise" or "incorrect choice" as they try to understand and use API functionalities to perform specific tasks. The overall results revealed that accurate and complete documentation is a crucial issue for API usability; most usability defects discovered in our study trace back to unsatisfactory documentation.

C. Software Artifact Generation

Our work can be considered as one type of generating software artifacts, specifically, API sequence. Recently, pre-trained Transformer models have been applied to generating many different types of software artifacts, except for an API sequence in our work. Generating other artifacts has also been investigated, such as pull request titles [44], code snippets based on the natural language description, and repair of the buggy code [23].

Zhang et al. [44] proposed the task of automatic pull request title generation. Their experimental results show that BART, a type of pre-trained Transformer model, can achieve the best performance in both automatic and manual evaluation. In a recent extensive study conducted by Zeng et al. [23], they evaluated the effectiveness of different types of pre-trained Transformer models in several program understanding and generation tasks. Among the generation tasks investigated in their work, code generation [36] the closest to our setting. Instead of generating an API sequence, the code generation task would generate code based on the given natural language description. In other words, code generation would generate all types of code tokens. Since our work only focuses on generating API sequence, generating other types of code tokens is outside of the scope of our work. In this sense, it would not be fair to compare our approach with theirs as they would have to generate more tokens.

VIII. CONCLUSION AND FUTURE WORK

There exist too many APIs that we can leverage. This causes issues in the selection of appropriate APIs for specific tasks, and the eco-system of APIs became more complex. Despite several recommendation approaches are proposed for APIs, they still lack the performance for practical usage. Inspired by recent studies that leveraged multi-modality to boost the recommendation performance, we proposed an approach by leveraging multi-modal information, i.e., source code as code context and natural languages as annotation, to generate API sequences. We performed a comparative study against several state-of-the-art API recommendation approaches with a new benchmark that contains multiple modalities. The results show that our approach outperformed the baseline API recommendation techniques by 20% - 50% better in terms of BLEU-score. It indicated that multi-modal information affects in positive ways, and an appropriate strategy is needed to optimize the performance of the technique with multi-modality.

In the future, we plan to utilize more types of data, such as Stack Overflow posts, in order to further improve the performance of MULAREC. We also plan to work on different methods of combining different modalities, such as beam search cost value modification, to provide better API sequence generation.

REPLICATION PACKAGE

To facilitate future research, we released our replication package: https://anonymous.4open.science/r/MulaRec/README.md.

REFERENCES

[1] Y. Peng, S. Li, W. Gu, Y. Li, W. Wang, C. Gao, and M. Lyu, "Revisiting, benchmarking and exploring api recommendation: How far are we?" *IEEE Transactions on Software Engineering*, 2022.

[2] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 631–642.

[3] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 254–265.

[4] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 319–328.

[5] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 293–304.

[6] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 349–359.

[7] R. Xie, X. Kong, L. Wang, Y. Zhou, and B. Li, "Hirec: Api recommendation using hierarchical context," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 369–379.

[8] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 254–265.

[9] X. He, L. Xu, X. Zhang, R. Hao, Y. Feng, and B. Xu, "Pyart: Python api recommendation in real-time," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1634–1645.

[10] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1050–1060.

[11] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," *arXiv preprint arXiv:2108.04556*, 2021.

[12] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 1–12.

[13] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[15] P. Martins, R. Achar, and C. V. Lopes, "50k-c: A dataset of compilable, and compiled, java projects," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 1–5.

[16] M. Van Selst and P. Jolicoeur, "A solution to the effect of sample size on outlier elimination," *The Quarterly Journal of Experimental Psychology Section A*, vol. 47, no. 3, pp. 631–650, 1994.

[17] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," in *EMNLP*, 2014.

[18] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[20] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[21] F. Zhang, X. Yu, J. Keung, F. Li, Z. Xie, Z. Yang, C. Ma, and Z. Zhang, "Improving stack overflow question title generation with copying enhanced codebert model and bi-modal information," *Information and Software Technology*, vol. 148, p. 106922, 2022.

[22] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 505–509.

[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 *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 39–51. [Online]. Available: https://doi.org/10.1145/3533767.3534390

[24] A. Henry, P. R. Dachapally, S. S. Pawar, and Y. Chen, "Query-key normalization for transformers," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 4246–4253.

[25] C. Yu, G. Yang, X. Chen, K. Liu, and Y. Zhou, "Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert," *2022 IEEE 38th International Conference on Software Maintenance and Evolution (ICSME)*, 2022.

[26] R. Sun, X. Cao, Y. Zhao, J. Wan, K. Zhou, F. Zhang, Z. Wang, and K. Zheng, "Multi-modal knowledge graphs for recommender systems," in *Proceedings of the 29th ACM international conference on information & knowledge management*, 2020, pp. 1405–1414.

[27] D. Francis, P. Anh Nguyen, B. Huet, and C.-W. Ngo, "Fusion of multimodal embeddings for ad-hoc video search," in *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, 2019, pp. 0–0.

[28] N. Shvetsova, B. Chen, A. Rouditchenko, S. Thomas, B. Kingsbury, R. S. Feris, D. Harwath, J. Glass, and H. Kuehne, "Everything at once-multi-modal fusion transformer for video retrieval," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 20 020–20 029.

[29] X. Wang, Y. Jiang, N. Bach, T. Wang, Z. Huang, F. Huang, and K. Tu, "Automated concatenation of embeddings for structured prediction," *arXiv preprint arXiv:2010.05006*, 2020.

[30] X. Wang, Z. Jia, Y. Jiang, and K. Tu, "Enhanced universal dependency parsing with automated concatenation of embeddings," *arXiv preprint arXiv:2107.02416*, 2021.

[31] R. Yang, J. Zhang, X. Gao, F. Ji, and H. Chen, "Simple and effective text matching with richer alignment features," in *Association for Computational Linguistics (ACL)*, 2019.

[32] "11.7. the transformer architecture — dive into deep learning 1.0.0-alpha1.post0 documentation," https://d2l.ai/chapter_attention-mechanisms-and-transformers/transformer.html#model, (Accessed on 10/23/2022).

[33] J. Martin and J. C. Guo, "Deep api learning revisited," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 321–330. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3524610.3527872

[34] P. Koehn, "Pharaoh: a beam search decoder for phrase-based statistical machine translation models," in *Conference of the Association for Machine Translation in the Americas*. Springer, 2004, pp. 115–124.

[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[36] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Association for Computational Linguistics, 2018, pp. 1643–1652. [Online]. Available: https://doi.org/10.18653/v1/d18-1192

[37] B. Xu, T. Hoang, A. Sharma, C. Yang, X. Xia, and D. Lo, "Post2vec: Learning distributed representations of stack overflow posts," *IEEE Transactions on Software Engineering*, 2021.

[38] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 858–868.

[39] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 5–14.

[40] Y. Kang, Z. Wang, H. Zhang, J. Chen, and H. You, "Apirecx: Cross-library api recommendation via pre-trained language model," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 3425–3436.

[41] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, "Clear: contrastive learning for api recommendation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 376–387.

[42] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, p. 319–334, Apr 2019.

[43] B. Xu, L. An, F. Thung, F. Khomh, and D. Lo, "Why reinventing the wheels? an empirical study on library reuse and re-implementation," *Empirical Software Engineering*, vol. 25, no. 1, pp. 755–789, 2020.

[44] T. Zhang, I. C. Irsan, F. Thung, D. Han, D. Lo, and L. Jiang, "Automatic pull request title generation," in *2022 IEEE 38th International Conference on Software Maintenance and Evolution (ICSME)*, 2022, p. Research Track.