Recommending Alternative API Usages for Code Customization based on Code Pre-trained Models

***Abstract*—Developers often need to customize the code searched from online resources or recommended by AI pair programming tools (e.g., Copilot) to meet some needs. To support customization, code alternatives are often needed for reference and comparison, and one of the most important code alternatives is API usage. Existing clone-diff based approaches for code customization cannot harvest code alternatives among the contextually similar code, and existing API usage recommendation approaches cannot support code customization well due to the limitations on library coverage and location flexibility of recommendation. In this paper, we formulate the alternative API usage recommendation problem as a cloze-style fill-in-blank task and leverage code pre-trained models to solve it. The proposed approach, called REALAU, uses a retriever-reranker architecture to recommend alternative API usages for a given code snippet and then reorganizes the results to improve conciseness and understandability. We evaluate REALAU with large amounts of Java code snippets in the CodeSearchNet dataset, the results show that REALAU can accurately predict API usages in code. Based on a user study with 14 participants on 6 code customization tasks, the code alternatives recommended by REALAU are confirmed useful and high-quality.**

I. Introduction

When accomplishing software development tasks, developers often use code snippets searched from online resources like Stack Overflow or recommended by AI pair programming tools such as Copilot [1]. Although the code snippets may provide a good starting point, they cannot always completely suit the needs or concerns (e.g., security concerns) of developers so need to be customized [2], [3], [4], [5]. To be aware of the customization and know how to customize, developers often need code alternatives for reference and comparison [3], [5]. On the other hand, according to the recent work [3] on code adaptation, more than half Stack Overflow code snippets involve *changing a method call* when reused and adapted in GitHub projects. This means that many code customizations involve API usages that are API calls with actual arguments (e.g., *MessageDigest.getInstance("MD5")*). Therefore, API usage is one important kind of code alternative for code customization.

Existing approaches [6], [3] support code customization by extracting code templates and code alternatives based on clone and differential analysis (clone-diff analysis). They match the code snippets in a clone set with each other, using the common part as a code template and different parts as customizable points and code alternatives. These approaches rely heavily on matching-based clone-diff analysis, so cannot harvest code alternatives among the code that are not clones but are contextually similar to each other. For example, given multiple pieces of code that compute checksums for the contents read from different IOs, these approaches fail to obtain code alternatives (e.g., *"SHA-1"* for the API *MessageDigest.getInstance*) from the contextually similar code when they cannot match with each other. On the other hand, there are some existing approaches [7], [8], [9], [10] proposed to recommend API usages based on code context. But they are very limited for code customization since cannot meet the following key requirements: covering alternative APIs in different libraries (e.g., *new SHA256Digest* in the library *bouncycastle*); being able to recommend code alternatives for any locations in the code (e.g., middle of code); recommending alternative APIs and arguments at the same time.

We formulate the alternative API usage recommendation problem as a ***cloze-style fill-in-blank*** task. That is, given an API usage in a code snippet, we mask it with a blank and attempt to fill in the blank with other alternative API usages based on the code context. The advantage of this task formalization is that the recommendation positions (i.e., masked blanks) can be very flexible. On the other hand, code pre-trained models [11], [12], [13], [14] recently achieve promising performance on many tasks, and are also potentially applicable to our alternative API usage recommendation for two main reasons. First, during pre-training the models have seen a large variant of code and likely learned the latent relevance between different API usages and code contexts, so can predict suitable API usages based on code context. Second, because of the pre-training tasks like masked token/span prediction, some pre-trained models such as CodeBERT [11] and CodeT5 [14] can be naturally applied to cloze-style tasks. To figure out which of the pre-trained models are actually applicable and suitable for our recommendation scenario, we contrast them from different perspectives. The summary is that some pre-trained models such as CodeT5 are potentially but not directly applicable, we need to adapt the models properly to solve the problem of *indeterminate length* and *diversity* of alternative API usages.

In this work, we propose an approach, **REALAU**, to **RE**commend **AL**ternative **A**PI **U**sages based on code pre-trained models. REALAU includes an offline phase and an online phase. In the offline phase, REALAU first extracts APIs and API usages from a code corpus to construct an alternative repository. Then, REALAU uses the code corpus and the alternative repository to construct an API usage retriever and an API usage reranker, which are built upon existing code pre-trained models. The retriever can ensure the *diversity* of the API usages at API level and the reranker can assess the fitness for *arbitrary-length* API usages. In the online phase, given a code snippet, REALAU traverses and masks each API usage in it and uses the constructed retriever and reranker to recommend

alternative API usages. Finally, REALAU reorganizes the recommended results into two levels of code alternatives (i.e., API-level and argument-level) to make the results more concise and understandable.

We set up a series of experiments to evaluate REALAU from different aspects. First, the automatic experiment with large amounts of Java code snippets in the CodeSearchNet dataset [15] shows that REALAU can accurately predict the API usages in code. Second, the user study with 14 participants reveals that REALAU can significantly improve the correctness by 35.7%, decrease the completion time by 54.2%, and provide 370.7% more suggestive information than the baseline on 6 code customization tasks. Third, the empirical assessment of the quality of the recommendation results shows that REALAU can produce relevant, concise, and readable code alternatives.

II. Motivating Example

Listing 1 shows the code snippet recommended by Copilot for "calculate the checksum of a file". Although the code is functionally correct, the developer needs to customize it when having other concerns such as security, performance, or maintenance status of the used libraries and APIs. To support the customization, some alternative API usages are needed.

```
// Prompt for Copilot: calculate the checksum of a file
public static String checksum(String filename) throws
     Exception {
    MessageDigest md = MessageDigest.getInstance("MD5");
    FileInputStream fis = new FileInputStream(filename);
    byte[] dataBytes = new byte[1024];
    int nread = 0;
    while ((nread = fis.read(dataBytes)) != -1) {
        md.update(dataBytes, 0, nread);
    }
    byte[] mdbytes = md.digest();
    // convert the byte to hex format
    StringBuffer sb = new StringBuffer("");
    for (int i = 0; i < mdbytes.length; i++) {
        sb.append(Integer.toString((mdbytes[i] & 0xff) +
            0x100, 16).substring(1));
    }
    return sb.toString();
}
```

Listing 1. The Code Snippet Recommended by Copilot

A. Two Levels of Needs

The needs for alternative API usages can be devided into the following two levels according to the developer's awareness of customization.

Developers often actively request some alternative API usages to assess the code quality and customize the current code to meet their needs [5]. For example, they may know the security issues of MD5 algorithm used in *MessageDigest.getInstance("MD5")* so need some alternative API usages to compare and choose suitable digest algorithms. For example, they can use an alternative argument *"SHA-1"* for *"MD5"*, or adopt an alternative API *SHA256Digest* in the third-party library *bouncycastle*. If the input is a GZIP file and the checksum needs to be calculated for the uncompressed content, the developers need to find an appropriate alternative API (e.g., *GZIPInputStream*) for *FileInputStream*.

Besides actively looking for alternatives, the developers often over-rely on searched or recommended code snippets [5] and are unaware the code needed to be customized. For example, if *"SHA-1"* and *SHA256Digest* are not provided, they may not think about comparing different digest algorithms so ignore the security issues of *"MD5"*. If they are not provided with *GZIPInputStream* as an alternative API for *FileInputStream*, they probably will not care whether they need to decompress the GZIP file before computing the checksum. We need to automatically and proactively recommend alternative API usages for developers.

Note that, for some alternative APIs are not compatible with the current code (e.g., inconsistent type of return value), developers can replace the entire code solution if an alternative API is particularly suitable for the needs. In this case, the developers can make the decision by searching for code samples containing the API.

B. Results of Different Approaches

To illustrate the capabilities of different approaches for code customization, we give a simple qualitative analysis based on their recommendation results for the Stack Overflow code example [16] that is very similar to the code in Listing 1.

Clone-Diff Based Approaches. The basic ideas of the clone-diff based approaches are similar, thus we choose the tool ExampleStack by Zhang et al. [3] to conduct the analysis. ExampleStack identifies 11 customizable points in the code based on 10 clones of it found in GitHub. The advantage of the clone-diff approaches is that they can recommend different kinds of code alternatives such as *Exception Handling* and *Refactoring*. For example, ExampleStack recommends refactoring the argument *"MD5"* of *MessageDigest.getInstance* as a variable. But many of the identified customizable points and recommended code alternatives are meaningless for users, and some meaningful alternatives are missed. For example, ExampleStack recommends inserting a new variable declaration expression *String datafile = "c:\\INSTLOG.TXT";* at the beginning of the code, the expression is only specific to someone's needs when reusing this code and doesn't make sense to other users. ExampleStack fails to find more alternative digest algorithms besides "SHA-1".

Our approach. REALAU focuses on alternative API usages and can recommend much richer such code alternatives than ExampleStack. For example, for *MessageDigest.getInstance("MD5")* in the code example, REALAU can provide API-level alternatives such as *new SHA256Digest* together with some samples, and argument-level alternatives such as *"SHA-1"*, *"SHA-256"*, *"SHA-384"*, and *"SHA-512"*.

III. Problem Definition & Code Pre-trained Models

In this section, we introduce the problem formalization and analyze the applicability of code pre-trained models.

A. Problem Definition

We formulate the code alternative recommendation problem as a ***cloze-style fill-in-blank*** task. That is, given an API usage in a code snippet, we mask it with a blank (marked as [MASK]) and attempt to fill in the blank with other

alternative API usages based on the code context. For example, Listing 2 shows the code obtained by masking the API usage *MessageDigest.getInstance("MD5")* with a blank [MASK] in the code in Listing 1. We will attempt to fill in the blank with some alternative API usages (e.g., *MessageDigest.getInstance("MD5")* and *new SHA256Digest()*).

```
1 // Prompt for Copilot: calculate the checksum of a file
2 public static String checksum(String filename) throws
      Exception {
3     MessageDigest md = [MASK];
4     FileInputStream fis = new FileInputStream(filename);
5     ...
6 }
```

Listing 2. The Code with a Blank

B. Code Pre-trained Models

In general, the popular code pre-trained models are all based on Transformer [17], a model of encoder-decoder architecture. They can be divided into three categories according to the different combinations of encoder and decoder.

Encoder-Only Models. This category of models only includes the encoder of Transformer, and CodeBERT [11] and GraphCodeBERT [12] are two representatives in this category. They take an entire sequence of tokens as input at once and use *bidirectional context* to generate a vector representation for each token in the sequence [11], [12]. Based on the generated vector representation, different classification heads can be used to support different tasks. One task used to pre-train them is *masked token prediction*, that is, let the models predict a portion of masked tokens in code based on a classification head. Because of the *bidirectional context* encoding capacity and the *masked token prediction* pre-training task, these models are suitable for cloze-style tasks such as variable renaming [18].

But these models cannot be applied to our scenario. The reason is that the blank length (i.e., the number of tokens to predict) must be determined before prediction when applying these models to cloze-style tasks, while the alternative API usages to be filled into the blank are indeterminate-length. When needing to predict multiple API usages for the same blank, the length of each API usage are likely different, making it impossible to determine the blank length in advance.

Decoder-Only Models. Codex [13], which powers Copilot, is a representative of the decoder-only models that only include the decoder of Transformer. The decoder-only models are usually trained by using the autoregressive token prediction task, i.e., predicting the next token based on existing tokens. Therefore, they can only use the *unidirectional context* to predict the following content and are difficult to be applied to fill in the cloze-style tasks where both-side contexts need to be considered.

Encoder-Decoder Models. Encoder-decoder models include the complete architecture of Transformer, a representative of encoder-decoder models is CodeT5 [14]. They take an entire token sequence as input and use *bidirectional context* to generate a vector representation for each token in the sequence [11], [12]. Then, the vector representations of the input tokens are fed into a decoder to generate an output token sequence token-by-token until a [EOS] token is encountered, where [EOS] is a specifical token widely used to represent the end of a sequence. One of the pre-training tasks for pre-training such encoder-decoder models is to predict masked spans in code in a generative way. That is, the models take as input a code where some spans are masked with blanks (here, a span is the continuous tokens), and generate a token sequence by the decoder to fill the blanks. Therefore, the encoder-decoder models pre-trained by the masked span prediction task can be used for the cloze-style tasks and can avoid the problem of indeterminate-length alternatives since they fill in the blank by the token-by-token generation.

However, generation-based filling also brings the problem that the diversity of generated alternatives is usually insufficient. Although multiple alternatives can be obtained by adopting beam search during the decoding process, beam search itself still faces the problem of insufficient diversity and heavy computational overhead [19].

Summary. Some pre-trained models such as CodeT5 are potentially but not directly applicable, we need to adapt the models properly to solve the problem of *indeterminate length* and *diversity* of code alternatives. In this work, we choose the encoder-decoder based code pre-trained model CodeT5 and solve the above problems using a retriever-reranker architecture.

IV. Approach

In this paper, we propose an approach, REALAU, to **RE**commend **Al**ternative **A**PI **U**sages based on code pre-trained models. REALAU includes an offline phase and an online phase, and its approach overview is shown in Figure 1.

In the offline phase, REALAU first extracts APIs and API usages from a code corpus to construct an alternative repository. Then, REALAU uses the code corpus and the alternative repository to construct an API usage retriever and an API usage reranker, which are built upon existing code pre-trained models. Give a code snippet with a blank, the API usage retriever efficiently retrieves the API usages that are relevant to the code and blank, while the API usage reranker reranks the retrieved API usages according to their fitness (i.e., generation probability) to be filled into the blank. With the combination of the retriever and the reranker, the problem of indeterminate length and diversity of API usages can be solved. The retriever can ensure the diversity of the API usages at the API level and the reranker can assess the fitness for arbitrary-length API usages.

In the online phase, given a code snippet, REALAU traverses each API usage in it and uses the trained retriever and reranker to recommend alternative API usages for the traversed API usage. Finally, REALAU reorganizes the recommended alternative API usages into two granularities of code alternatives (i.e., alternative APIs and alternative arguments), makes the recommendation results more concise and understandable.

A. API & API Usage Extraction

From the code corpus, REALAU extracts APIs, API usages, and the mappings between them to form a code alternative repository.

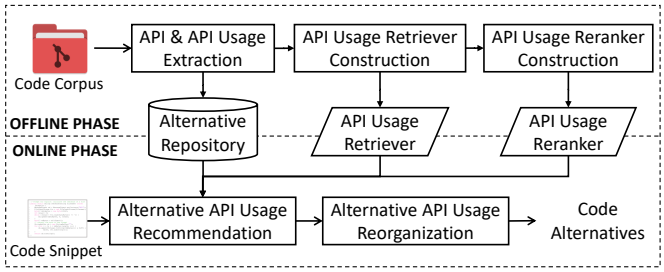


Fig. 1. Approach Overview of REALAU

For each method in the code corpus, REALAU uses a static analysis tool (i.e., javalang [20] in our current implementation) to parse it to an abstract syntax tree (AST) and extract the class instantiation and method call expressions from the AST. REALAU then attempts to map the extracted expressions to the corresponding APIs.

For each expression, if it is a class instantiation, REALAU maps it to the API *new C*, where C is the instantiated class. For example, REALAU maps the expression *new FileInputStream(filename)* to the API *new FileInputStream*. Here, we reserve the keyword "new" in front of the API to keep it consistent with the form that appears in the code. If the expression is a method call (denoted as mi), it can be further divided into three following cases.

- mi calls a custom method of the current class with the form $m(...)$;
- mi calls a static method of class C with the form $C.m(...)$;
- mi calls a member method of the object o with the form $o.m(...)$.

If mi meets the first case, it cannot be mapped to any API and skipped. The latter two cases (i.e., C and o) are distinguished based on the Java naming conventions, i.e., class names should start with a capitalized letter while object names should be the opposite. When mi starts with a capitalized letter (e.g., *MessageDigest.getInstance("MD5")*), it meets the second case and mapped to the API $C.m$ (e.g., *MessageDigest.getInstance*). For the last case, REALAU tries to find the corresponding type T for o by performing the define-use analysis for variables. That is, REALAU tries to trace back to where o is defined to determine the type T of o. If the type T is successfully traced, mi is mapped to the API *T.m*, otherwise, it is skipped. For example, *md.diget()* in Figure 1 is mapped to the API *MessageDigest.digest*.

If an expression can be mapped to an API, it is treated as an API usage. After processing all methods, we obtain the alternative repository. The API usages in the repository can cover a variant of libraries if the code corpus is large enough.

Note that we distinguish APIs only by their names (without parameters), that is, we consider the APIs with the same name to be the same API. For example, the three overloaded versions of *MessageDigest.getInstance* [21] are treated as the same API, thus the two API usages *MessageDigest.getInstance("MD5", provider)* and *MessageDigest.getInstance("MD5")* are both mapped to *MessageDigest.getInstance*. This may confuse unrelated API usages (e.g., the API usages corresponding to same-named APIs in different libraries), but our subsequent API usage reranking can choose the appropriate usages based on the code context and the arguments of the usages.

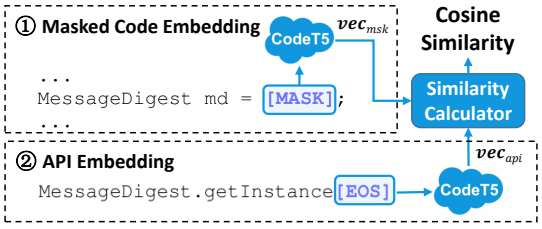


Fig. 2. The Similarity Model in API Usage Retriever

B. API Usage Retriever Construction

Given a code snippet with a blank [MASK], the API usage retriever is responsible for quickly retrieving the API usages relevant to the code context from the alternative repository. To ensure the diversity of the API usages at the API level, REALAU first uses a similarity model to select a list of APIs from the alternative repository, and then retrieves API usages based on the selected APIs and the mappings between APIs and API usages.

1) Similarity Model: The similarity calculation model for API is based on a two-tower architecture that is commonly used for large-scale information retrieval [22]. As shown in Figure 2, the model includes three modules. A blank embedding module produces a vector representation $\boldsymbol{vec}_{msk}$ for the blank [MASK] in the code; An API embedding module produces a vector representation $\boldsymbol{vec}_{api}$ for an API in the alternative repository; A similarity calculation module computes the similarity sim between the two vector representations.

Blank Embedding. The blank embedding module is shown in part ① of Figure 2. For the code with blank, REALAU first uses the tokenizer of CodeT5 to tokenize it into a token sequence and input the sequence into the CodeT5 model. Then, REALAU obtains the vector representations of the tokens by taking the hidden states of the CodeT5 decoder [23] and takes the vector corresponding to [MASK] as the representation of the blank (i.e., $\boldsymbol{vec}_{msk}$). Here, the reason for choosing CodeT5 as the embedding model is that the blanks are actually spans rather than tokens, CodeT5 is more suitable for this kind of blank compared to other models such as CodeBERT and GraphCodeBERT as discussed in Section III-B.

API Embedding. The API embedding module is shown in part ② of Figure 2. We also use CodeT5 in this module to ensure the vector representations of blanks and APIs are in the same feature space. For an API, REALAU first obtains the vector representations of the tokens in it in the same way as in blank embedding module. Then vector representation of the API (i.e., $\boldsymbol{vec}_{api}$) is obtained by taking the vector corresponding to [EOS], which is a common method [23], [14] to obtain a sequence's representation with encoder-decoder models. Here, [EOS] is a special token commonly used to mark the end of the input sequence.

Similarity Calculation. The similarity sim between the blank and the API is the cosine similarity between their vector representations, i.e., $sim = cosine(\boldsymbol{vec}_{msk}, \boldsymbol{vec}_{api})$.

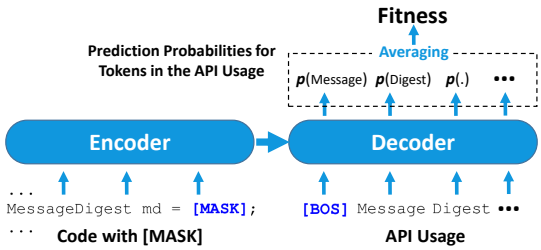


Fig. 3. The Fitness Model in API Usage Reranker

2) Model Fine-tuning: The similarity model is fine-tuned by maximizing the similarities between the blanks and the original APIs in the blanks. For each code snippet in code corpus, REALAU finds all the API usages in it in the same way as in Section IV-A, and randomly masks one of the usages with a blank [MASK]. A training sample is generated by pairing the code with blank and the API corresponding to the masked API usage. The model is fine-tuned by optimizing the loss function $\mathcal{L} = 1 - sim$ based on the generated training example, where sim is the similarity between the blank in the code and the API.

Note that, during fine-tuning we freeze and do not update the parameters of the blank embedding module (i.e., the pre-trained CodeT5 in ①) of the similarity model. There are two reasons. First, CodeT5 can already capture the features of the blanks well since CodeT5 has been trained on large-scale code containing blanks (span) during pre-training. Second, freezing the parameters can reduce the difficulty and overhead of fine-tuning.

3) API Usage Retrieval: Give a code snippet with a blank [MASK], REALAU can calculate the similarities between the blank and all APIs in the alternative repository based on the similarity model. Then, REALAU ranks the APIs and selects the top M with the highest similarities, and finds the API usages for each of these APIs by using the mappings in the alternative repository. REALAU collects all the found API usages together as the retrieved API usages.

Note that, REALAU can offline generate and store the vector representations of all the APIs in the alternative repository using the API embedding module. When performing the similarity calculation online, REALAU only needs to use the blank embedding module to obtain vector representation of the blank and calculate similarities, so the retrieval is time-efficient. Because the API usages corresponding to all the M APIs are collected, the diversity of the retrieved API usages can be ensured at the API level.

C. API Usage Reranker Construction

Given the code with a blank [MASK], REALAU reranks the API usages retrieved by the API usage retriever. To achieve this, we build a fitness model to assess the fitness of the API usages to be filled into the blank and rerank the API usages based on their fitness scores.

1) Fitness Model: For an API usage, the fitness model takes its generation probability for [MASK] as its fitness to the blank. As shown in Figure 3, the model is based on a pre-trained CodeT5 and follows the same generation process as the masked span prediction task in CodeT5 pre-training. That is, the encoder takes as input the code with mask at once, and the decoder accepts the API usage token-by-token and predicts a probability distribution for the next token based on already accepted tokens. Here, in the generation process, the decoder starts with [BOS], a special token commonly used to represent the beginning of a sequence. In the token-by-token generation process, REALAU can obtain the probability of each token in the API usage from the probability distributions predicted by the decoder. For example, in Figure 3, when the decoder accepts the token "Message" in the API usage *MessageDigest.getInstance("MD5")*, the decoder predicts a probability distribution for the next token such as the probability of "Digest" (i.e., $\boldsymbol{p}(Digest)$) can be obtained from the distribution. REALAU computes the fitness of the API usage to the blank by averaging the prediction probabilities of all the tokens in it.

2) Model Fine-tuning: We fine-tune the CodeT5 in the fitness model to better support the computation of fitness. The fine-tuning is necessary since the spans are masked randomly in the masked span prediction pre-training task [14], instead of specific to API usages. For example, suppose that the code in Listing 1 is used to pre-train CodeT5, "*MessageDigest md =*" may be a masked span. The fine-tuning process is the same as the masked span prediction pre-training task, except the spans masked here are all API usages. The training samples are pairs of code with blank and masked API usage, which are obtained similarly as in Section IV-B2.

3) API Usage Reranking: Given the code with blank, REALAU reranks a given set of API usages according to their fitnesses computed based on the fitness model.

D. Alternative API Usage Recommendation

Based on the offline constructed alternative repository, API usage retriever, and API usage ranker, REALAU recommends alternative API usages for a code snippet in the online phase.

1) Customizable Point Generation: Given a code snippet, REALAU first identifies all the API usages in the same way as Section IV-A and traverses them. For each traversed API usage, REALAU masks it with a blank [MASK] as a customizable point. Note that, we only generate one customizable point for the code snippet at a time.

2) Alternative API Usage Retrieval & Reranking: For each customizable point, REALAU recommends alternative API usages for it as follows. REALAU first retrieves API usages for the blank [MASK] using the retrieval method described in Section IV-B3 and then reranks these retrieved API usages by the reranking method described in Section IV-C3. After the retrieval and reranking, the API usages are recommended as alternatives.

E. Alternative API Usage Reorganization

For each customizable point, REALAU reorganizes the recommended API usages by grouping them by their corresponding APIs and generates the two levels of code alternatives, i.e., alternative APIs and alternative arguments. This reorganization

can also make the recommendation results more concise and understandable.

1) Grouping: For a customizable point, we denote the masked API and API usage that are originally in the customizable point as api_{msk} and $usage_{msk}$. REALAU puts the recommended alternative API usages corresponding to the same API into a group named the API. For example, all the API usages corresponding to the API *MessageDigest.getInstance* are grouped. If there is no group correspondinng to api_{msk}, the customizable point and all the alternatives are omitted. For each group, REALAU reserves the top N API usages having the highest fitness scores and sum the fitness scores of the reserved API usages as the group's score. REALAU ranks all the groups based on their scores and reserves the top G groups.

2) API-Level Alternatives: REALAU directly treats the groups whose name is not api_{msk} as alternative APIs. For example, for the *MessageDigest.getInstance("MD5")* in the code in Listing 1, *new SHA256Digest* are alternative APIs recommended by REALAU. Note that, the recommended alternative APIs may not directly replace the original API in the customizable point due to control and data dependencies. Developers can reference the API usages in the groups corresponding to these alternative APIs and search related code samples, to determine whether the APIs are suitable and how to customize the current code if adopting the APIs (e.g., assembling some parts from the searched samples into the current code). For example, some code samples can be searched from online code search services such as Tabnine [24] for the API *SHA256Digest*. Developers can compare the current code snippet to these code samples and determine the customization solution.

3) Argument-Level Alternatives: The group whose name is api_{msk} is the alternative arguments for this customizable point, i.e., the alternatives in this group are different usages of api_{msk} (e.g., *MessageDigest.getInstance*) and just differ in the argument lists (e.g., [*digestAlgorithm*] and [*"SHA-1"*]). REALAU categorizes and abstracts the alternatives in the group at the argument level to make the results more concise and understandable. The basic idea is to categorize arguments into identifiers and constants, and then extract abstract labels with conceptual semantics from the identifiers and treat the constants as concrete examples. The details are as follows.

First, REALAU categorizes each argument in the group into identifiers and constants based on the tokenization module of javalang [20]. Given an argument, REALAU inputs it into the javalang tokenization and treats it as a constant if identified as *Literal* by javalang (e.g., *"SHA1"*); otherwise, it is treated as an identifier (e.g., *algorithm*).

Second, according to the number of arguments, REALAU divides the argument lists in the group into different N-arg sets, where each argument list in an N-arg set contains N arguments. For example, the argument lists [*digestAlgorithm*] and [*"SHA-1"*] belong to an 1-arg set, but the argument lists [*algorithm*, *provider*] and [*"SHA1"*, *"ZBC"*] belong to a 2-arg set. Then, for an N-arg set, REALAU aligns the argument lists in it according to the argument position, that is, the i-th arguments of these lists correspond to each other. For example, for the two argument lists [*algorithm*, *provider*] and [*"SHA1"*, *"ZBC"*], REALAU aligns *algorithm* with *"SHA1"* and *provider* with *"ZBC"* at the 1st argument position and the 2nd argument position, respectively.

Third, for each argument position of an N-arg set, REALAU produces some abstract labels based on the arguments categorized as identifiers at this position. For each identifier, REALAU uses Sprial [25] to split it into words by camel case, and then extracts nouns from the words using POSSE [26] (we re-implement the method in Python). Then REALAU chooses labels from all the nouns at this argument position by using word embeddings and PageRank [27]. REALAU constructs a similarity graph for the nouns by adding an edge between two nouns if the cosine similarity between their embeddings is above a threshold. Then, the PageRank algorithm is performed on the graph and the top 3 nouns are chosen as labels. If two labels (e.g., *digest*, and *algorithm*) appear together in some identifiers at this argument position, REALAU merges them as a new label (e.g., *digest algorithm*). The word embeddings are trained on the code corpus by extracting identifier sequences from the methods using javalang and running fastText [28] on the identifier sequences, where the identifiers are split as words by camel case. The threshold of cosine similarity when constructing similarity graph is set to 0.5 in our implementation according to some trials on a validation set. The abstract labels can help developers intuitively understand the conceptual meanings of the API arguments (e.g., *digest algorithm*). The reason for not directly using the formal parameters of an API as the abstract labels is that it is difficult to infer the full qualified name of the API according to the code snippet only.

Fourth, for each argument position of an N-arg set, the identifiers are removed and the constants are reserved as concrete examples for this argument position. For example, for the 1-arg set for the group of *MessageDigest.getInstance*, REALAU produces many different algorithm instances such as *"SHA-1"*, *"SHA-384"*, and *"SHA-512"*.

Finally, the abstract labels and concrete examples are recommended together with the API usages in this group.

V. Evaluation

We conduct a series of experiments to evaluate the accuracy, usefulness, and quality of REALAU by answering the following research questions.

- **RQ1 (Accuracy for API Usage Prediction):** How accurately REALAU can predict the masked API usages in code?
- **RQ2 (Usefulness for Code Customization):** Are the code alternatives recommended by REALAU useful for code customization tasks?
- **RQ3 (Quality of Recommended Alternatives):** What is the quality of the code alternatives recommended by REALAU?

All the data and results can be found in our replication package [29].

A. Implementation

The code corpus is collected from the training set of CodeSearchNet [15] dataset, which is used to pre-train CodeT5. Based on the code corpus, the constructed alternative repository includes 390,981 APIs and 1,155,504 API usages. The API usage retriever & reranker are constructed based on HuggingFace transformers [30], which is a wide used library for Transformer based models. The pre-trained CodeT5 used in these modules is CodeT5-base and the training hyperparameters, such as learning rate, batch size, and epoch, follow the settings in the fine-tuning tutorials provided by HuggingFace [31]. In the final implementation, the hyperparameters M (the number of retrieved APIs), N (the number of reranked API usages in each group), and G (the number of reversed groups) are set to 30, 30, and 5 based on some trail experiments on a small validation set.

B. RQ1: Accuracy for API Usage Prediction

1) Motivation: REALAU uses a retriever-reranker architecture including an API usage retriever and API usage reranker to recommend alternative API usages (see Section IV-D). We evaluate the accuracy of the REALAU to predict API usages, that is, how accurate REALAU can predict the masked API usages in code.

2) Test Samples: We use the test set of CodeSearchNet dataset to evaluate the accuracy. For each code snippet in the test set, we find all the API usages in it in the same way as in Section IV-A, and randomly mask one of the usages with a blank [MASK]. If the masked API usage is not in the alternative repository, we skip this code. Otherwise, a test sample ($code^{blk}$, api, usg) is generated by pairing the code with blank c_{blk}, the masked API usage usg, and the masked API api that usg corresponds to. After processing all code in the test set, we obtain 7,520 test samples in total.

3) Protocol: For each i-th test sample ($code_i^{blk}$, api_i, usg_i), we feed $code_i^{blk}$ into REALAU and let the retriever and reranker predict API usages in the same way as in Section IV-D2. We denote the API list retrieved by the API usage retriever and the API usage list reranked by the API usage reranker as $apis_i^{rtr}$ and $usgs_i^{rrk}$, respectively.

In the prediction process, the accuracy of the API usage retriever and API usage reranker need to be evaluated. For API usage retriever, we care about whether the masked API api_i is retrieved because this affects whether the retrieval results provided for the subsequent API usage reranker are complete. Therefore we use the hit rate (hit) to evaluate the accuracy of the retriever. For API usage reranker, we are more concerned about whether the masked API usage usg_i high in the results, so mean reciprocal rank (mrr) is used to evaluate the accuracy of the reranker.

When considering top k results, the hit and mrr are calculated as follows.

$$hit@k = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(rank(api_i, apis_i^{rtr}) <= k)$$

TABLE I
ACCURACY OF MASKED API & API USAGE PREDICTION

Module	k=1	k=5	k=10	k=50	k=100
Retriever			$\boldsymbol{hit@k}$ (%)		
	15.4	29.2	35.7	51.6	57.3
Reranker			$\boldsymbol{mrr@k}$		
	0.431	0.450	0.451	0.452	0.452
			$\boldsymbol{mrr^{hit}@k}$		
	0.752	0.785	0.787	0.789	0.789

$$mrr@k = \frac{1}{N}\sum_{i=1}^{N}\frac{\mathbb{I}(rank(usg_i, usgs_i^{rrk}) <= k)}{rank(usg_i, usgs_i^{rrk})}$$

In the above equations, N is the number of all the test samples and $\mathbb{I}(\cdot)$ is a function that returns 1 if the input is true else 0. The function $rank(tgt, list)$ returns the ranking of the target tgt in the results $list$. For example, if tgt is the second result in $list$, then $rank(tgt, list) = 2$. Here, if tgt does not appear in $list$, the function returns ∞ (infinity).

Note that, because the API usage reranker is based on the results of the API usage retriever, the hit rate of the retriever affects the result of the reranker. When evaluating the mrr of the reranker, we fix the hyperparameter M (i.e., API number) involved in API usage retriever (see Section IV-B3) as 100 to ensure the number of retrieved API usages. Moreover, in order to exclude the influence of the retriever on the reranker, we also calculate a variant of mrr mrr^{hit} to reflect the individual accuracy of the reranker on the premise that the retriever hits the masked APIs. That is, $\boldsymbol{mrr^{hit}@k} = mrr@k/hit@100$.

4) Result: The evaluation results are shown in Table I. It can be seen that the API usage retriever achieves a relatively high hit rate of masked API when k is 50 or 100 (i.e., $hit@50$ and $hit@100$ exceed 50%). The goal of API usage retriever is to According to the results for API usage reranker, the $mrr^{hit}@1$ have reached 0.752, indicating that the reranker can make most of the masked API usages rank at the first in the results.

We analyze the results of the API usage retriever and find that the following main cause of the unhit masked APIs. Some masked APIs are loosely coupled with the code context in which they are located, and the combination between the two depends on the specific business requirements. This makes it difficult to determine what the masked api is by context. From the point of view of data dependency, this situation is usually reflected in that the return values of the masked APIs are very general types such as *int* and *String*, and there are too many APIs that can return these types of data. Therefore, in the absence of business constraints, there is a lot of freedom based only on the context and it is difficult to determine what the masked API is. For example, for the statement with a blank *String key = bundleName + '_' + [MASK];*, the original API in [MASK] is *Locale.toLanguageTag*. Just looking at the code context in which this statement is placed is difficult to determine what the API in [MASK] is because it depends on the specific business requirements.

The above cause is for masked API prediction, but we don't think it is necessarily a problem for alternative API usage

recommendation. On the contrary, this may provide a basis for us to filter out some inappropriate customizable points. As described in Section IV-E1, we filter out customizable points whose original API does not appear in the recommended groups. The rationale behind this is that if the recommended groups do not include the original API, it means that the degree of freedom of the customizable point may be too high, so it is difficult to recommend meaningful alternatives. For example, recommending the API *ResourceBundle.toBundleName* for the statement *String key = bundleName + '_' + [MASK];* is meaningless.

5) Summary: REALAU can accurately predict the masked API usages in code. The API usage retriever and API usage reranker are confirmed generally accurate by the evaluation results.

C. RQ2: Usefulness for Code Customization

1) Motivation: To evaluate whether the code alternatives recommended by REALAU are really useful for customizing code, we conduct a user study on some code customization tasks.

2) Tasks: We construct the code customization tasks based on the *adaptation dataset* collected by Zhang et al. [3], which is used to study the adaptation of online code snippets. The dataset includes 629 Java code snippets that are posted in Stack Overflow answers and explicitly reused in GitHub projects. The code snippets provide a basic solution to the corresponding Stack Overflow questions. We randomly sample 100 code snippets from the dataset and manually check whether there are needs for API usages for customizing the code snippets.

For each sampled code snippet, we obtain the Stack Overflow question and answer corresponding to this code, and search Stack Overflow for some similar questions and answers. Then, we try to collect possible API or argument customization requirements for this code snippet from the comments under these questions and answers. If such requirement is collected, based on it, we generate two questions following the templates below:

- To solve the ### problem/issue, you can change the API/argument ### to ___. (answer some options)
- If you didn't know the problem before, do you think the results recommend by the tool help you to realize the problem? (answer yes/no)

The first question is used to directly assess the usefulness of the tools when the customization requirements are relatively clear; The second question is designed to figure out whether the participants are unaware of some important potential customization requirements and whether the tools can help them realize the problems. A code customization task is constructed by taking the code snippet and its corresponding Stack Overflow question title as task context together with the two questions. For example, the code snippet [32] contains an API usage *MessageDigest.getInstance("MD5")*, one related comment reads: *Keep in mind that according to the recent research "MD5 should be considered cryptographically broken and unsuitable for further use"* [33]. We construct a code customization task that consists of the code and the following two questions:

- To solve the secure issue of the "MD5" digest algorithm in the code, you can modify the algorithm "MD5" to ___.
- If you didn't know the problem before, do you think the results recommend by the tool help you to realize the problem? (answer yes/no)

In this way, we collect 6 (including 2 API-level and 4 argument-level) code customization tasks (i.e., T1-T6).

3) Baseline: We choose the code customization tool, ExampleStack, proposed by Zhang et al. [3] as the baseline. The reason is that this tool itself is proposed with the *adaptation dataset* and can work well on the code snippets in the constructed code customization tasks.

4) Protocol: We randomly divide the tasks into two roughly equivalent groups T_A and T_B in terms of difficulty. We invite 14 MS students with 1-5 years' Java programming experience to participate in this experiment. We pair the 14 participants into 7 pairs according to their programming expertise and randomly divide them into two "equivalent" groups (G_A and G_B). That is, for each two paired participants, we randomly assign one of them to G_A and the other to G_B.

We ask the participants to complete the code customization tasks with REALAU and the baseline by adopting a balanced treatment distribution for the groups. Participants in G_A are asked to complete the tasks in T_A with REALAU and complete the tasks in T_B with the baseline. Conversely, participants in G_B are asked to complete the tasks in T_B with REALAU and complete the tasks in T_A with the baseline. Overall, each participant is asked to complete all 6 tasks, 3 with REALAU and 3 with the baseline. Before starting the tasks, we give these participants a 20-minute training session and provide some examples to familiarize them with REALAU and the baseline. When completing the tasks, the participants are allowed to use external resources like Google and GitHub. For each participant, the tasks are done by interleaving REALAU and the baseline. At the same time, the order in which tasks are completed is completely random.

For each task, a participant is asked to answer the two questions and record the screen. For the first question, the participant is allowed to use other resources such as Google search engine to find the answer. The participants have a time limit of 10 minutes on this question, after which they are considered to submit an empty answer. For the second question, the participant is required to give a *yes or no* answer. When all participants have completed their tasks, for each task we assess the *correctness* of the answer for the first question and count the *suggestiveness* for the second question (i.e., the answer is *yes*). Two authors independently perform the correctness assessment and a third author resolves conflicts based on the major-win strategy. Finally, we also conduct interviews to collect participants' feedback on REALAU. The participants are asked to answer whether REALAU is useful and why.

5) Results: Table II shows the correctness, suggestiveness, and completion time over the tasks when completed with REALAU and the baseline respectively. According to the

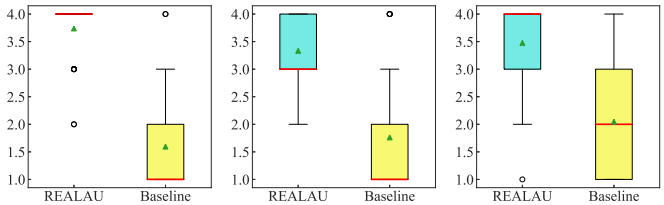
TABLE II
RESULTS OF USER STUDY

Task	REALAU			Baseline		
	Correctness	Suggestiveness	Time (s)	Correctness	Suggestiveness	Time (s)
T1	6/7	4/7	140.3	5/7	2/7	167.9
T2	7/7	7/7	38.6	5/7	0/7	210.9
T3	7/7	7/7	104.0	4/7	1/7	277.4
T4	7/7	7/7	29.0	4/7	3/7	215.4
T5	6/7	4/7	141.1	5/7	1/7	158.0
T6	5/7	4/7	46.7	5/7	0/7	62.4
Avg.	90.5%	78.6%	83.3	66.7%	16.7%	182.0

results, compared to using the baseline, using REALAU the participants complete the tasks 35.7% more correctly (i.e., 90.5% vs. 66.7% on average) in 54.2% less time (i.e., 83.3s vs. 182.0s on average), and obtain 370.7% more suggestive information (i.e., 78.6% vs. 16.7% on average). We use Welch's T-test [34] for verifying the statistical significance of the differences. The differences in the correctness, suggestiveness, and completion time between REALAU and the baseline are statistically significant ($p = 0.002, 0.0004, 0.02$ respectively).

The usefulness of REALAU in the code customization tasks can be reflected in two aspects. First, users can easily obtain some alternative APIs or arguments and corresponding sample API usages from the recommendation results. This will allow them to complete tasks more correctly and faster. For example, for the task T1 is about the alternatives of MD5, the users can directly find many other digest algorithms such as *"SHA-1"*, *"SHA-256"*, *"SHA-384"*, *"SHA-512"* from the recommendation results of REALAU, but the baseline only provides a single alternative *"SHA-1"*. Second, the recommended code alternatives can give users some suggestive information that helps them to realize some *unaware* needs of customization (see Section II-A), which will make them better understand and trust the code. For example, task T5 asks the users to figure out the protocol used in the API *Socket* and find an alternative API for it when UDP protocol is needed. According to the feedback of some participants, the recommendation results can suggest they compare the recommended APIs (e.g., *DatagramSocket*) with the original API (i.e., *Socket*) when they do not know their differences before. This will give them more confidence to use or customize the code, and prevent them from relying too much on the searched or recommended code and ignoring some potentially critical problems. This also reminds us to provide some comparisons with the original code to the recommended results in the future, thereby reducing the effort of users to check the information by themselves.

In fact, after analyzing the screen recordings, we find that when using the baseline to complete tasks, users often cannot get help from the recommendation results and rely entirely on external resources such as google for answers. So they take more time and get less suggestiveness from the tool. This is because the results recommended by the baseline are mined from the code adaptation history for some specific projects, and may not be meaningful to other users. For example, remove a variable declaration expression (*MessageDigest md = MessageDigest.getInstance("MD5")*) from the current code snippet for some refactoring needs.



(a) Relevance (b) Conciseness (c) Readability

Fig. 4. Quality of Recommendation Results

6) Summary: REALAU can significantly improve the correctness by 35.7%, decrease the completion time by 54.2%, and provide 370.7% more suggestive information than baseline in code customization tasks.

D. RQ3: Quality of Recommended Alternatives

1) Motivation: REALAU reorganizes the recommended alternative API usages by grouping them and dividing them into alternative APIs and arguments (see Section IV-E), and the quality of the final recommendation results greatly affects the usability of REALAU. To this end, we perform an empirical evaluation for assessing the code alternatives recommended by REALAU, similar to previous research [35], [36], [37].

2) Protocol: After finishing the code customization tasks in Section V-C (RQ2), for each task the participants are asked to evaluate the recommended results of REALAU and the baseline (i.e., ExampleStack) in terms of relevance, conciseness, and readability on a 4-points Likert scale [38] (1-disagree; 2-somewhat disagree; 3-somewhat agree; 4-agree) by the following statements.

Relevance. The recommended results are completely (or mostly) relevant to the current task contexts.

Conciseness. The recommended results contain no (or very little) unnecessary or redundant information.

Readability. The recommended results are well-organized and easy to understand.

Note that in order to reduce bias, the second statement is phrased negatively to maintain the interpretation of the answers similar to all three statements. After the participants finish the evaluation, we ask them to explain the low ratings (1 or 2).

3) Results: For *relevance* of REALAU, 78.6% of the answers are 4 (agree), 16.7% are 3 (somewhat agree), 4.8% are 2 (somewhat disagree), and there are no 1 (disagree). For *conciseness* of REALAU, 42.9% of the answers are 4 (agree), 47.6% are 3 (somewhat agree), 9.5% are 2 (somewhat disagree), and there are no 1 (disagree) answers. For *readability* of REALAU, 57.1% of the answers are 4 (agree), 35.7% are 3 (somewhat agree), 4.8% are 2 (somewhat disagree), and 2.4% are 1 (disagree) answers. We use Welch's T-test [34] for verifying the statistical significance of the differences between the ratings of REALAU and the ratings of the baseline. The results show that for each property the statistical difference is significant ($p << 0.05$). The main reason for the baseline having lower relevance, conciseness, and readability is that the code alternatives recommended by it are obtained by clone-diff analysis so maybe noise to the users in current task context.

4) Summary: REALAU can recommend high-quality alternatives for code customization, in terms of relevance, conciseness, and readability.

E. Threats to Validity

RQ2 involves data sampling and data labeling. Therefore, common threats to internal validity involved, including sampling randomness and judgment subjectivity of the annotators. We follow common sampling and data analysis techniques, such as involving multiple annotators and conflict resolution steps, to minimize such threats. There are two main threats to external validity. First, we only use a limited number of tasks (i.e., 6 tasks) constructed from the *adaptation dataset* in the evaluation. Therefore, our findings may not be generalizable to other code customization tasks. To minimize such threats, we sample the code snippets from the dataset and try to ensure the diversity of tasks as much as possible by covering different topics. Second, although the baseline used for RQ2 and RQ3 is proposed for code customization, it is not originally intended for alternative API usages. Therefore, there may be some biases in our comparison results. To minimize such threats, we use the adaptation dataset from the baseline itself to make sure it works well.

VI. Related Work

Reliability Issues of Code Snippets. Some researchers analyze the reliability issues of code snippets on online Q&A forums and recommended by Copilot. Zhou et al. [39] investigate the deprecated API issues on Stack Overflow and find that out of 200 accepted answers 86 contain deprecated APIs and only three of them are reported by other users. Fischer et al. [40] find that 29% of security-related code on Stack Overflow is insecure and may have been reused in over 1 million Android apps on Google Play. Zhang et al. [2] mine usage patterns for 100 Java APIs based on a code corpus from GitHub and use the patterns to check code examples on Stack Overflow. They find that 31% of the analyzed 217,818 Stack Overflow posts may have potential defects. Ragkhitwetsagul et al. [41] survey 87 Stack Overflow visitors and show that Stack Overflow answers often suffer from mismatched solutions, outdated solutions, incorrect solutions, and buggy code. Some other research works investigate code compilability problems and present corresponding solutions [42], [43], [44], [45]. Vaithilingam et al. [5] conduct an empirical study to analyze the usability of the code snippets recommended by Copilot. They find that the code snippets usually face the reliability issues so the developers sometimes do not trust the code and completely rewrite the implementation. Pearce et al. [4] systematically investigate the secure issues of the code snippets recommended by Copilot via prompting Copilot to generate code in scenarios relevant to high-risk cybersecurity weaknesses. They find that about 40% of the 1,689 programs produced by Copilot are vulnerable. These works report the reliability issues in online code examples and recommended code snippets, which may cause that developers need to modify the code when reusing them. The code alternatives recommended by REALAU can help developers to understand and modify the code.

Code Customization. There are some research works that focus on the customization of the code snippets. Lin et al. [6] propose a clone-diff based approach to recommend modification options for a copy-pasted code snippet. Zhang et al. [3] analyze the typical categories of the adaptation of Stack Overflow code snippets and propose a clone-diff based approach to support the adaptation. Given a code clone set, these approaches apply matching-based diff analysis (sequence or tree matching) to find the common and different parts of the code in the clone set. The common parts are treated as a code template, and the positions of the differences are treated as customizable points where the contents of the differences are considered as customization options. Compared to these clone-diff based approaches, REALAU focuses on alternative API usage recommendation and is more flexible.

API Usage Recommendation. Some approaches are proposed to recommend API usages for developers based on code context. Nguyen et al. [7] propose an approach, FOCUS, to recommend the next API usages based on context-aware collaborative filtering, but cannot recommend for other places in code instead of at the end of code. Chen et al. [46] propose a multi-view heterogeneous graph learning based approach named MEGA to improve the API usage recommendation accuracy, especially for the low-frequency APIs. Moreover, the API usages recommended by these approaches are API signatures without actual arguments (e.g., *HelperFormatter.printHelp(String, Options)*). Chen et al. [9], [8] propose learning based approaches to recommend APIs and concrete arguments for holes in code, but these approaches are bound to specific libraries so cannot cover alternative API usages in different libraries (e.g., *MessageDigest.getInstance* in JDK and *SHA256Digest* in bouncycastle). In this paper, we recommend alternative API usages with actual arguments for any places in code, and the recommended API usages cover a wide range of APIs in different libraries.

VII. Conclusions and Future Work

In this paper, we propose an approach (called REALAU) to support code customization by recommending alternative API usages based on code pre-trained models. It can recommend rich code alternatives at API and argument levels. Our evaluation confirms the accuracy of REALAU for predicting API usages and the quality of the recommended code alternatives. It also shows the usefulness of REALAU in code customization tasks. In the future, we will improve the approach from the following aspects. First, we will provide explanations such as comparisons of alternatives for the recommendation results to make REALAU more usable. Second, we will incorporate code search techniques to enrich the recommended code alternatives with sample code. Third, we will incorporate program repairing techniques to make the recommended code alternatives can be directly integratied into the given code snippets. Fourth, we will support other types of code alternatives for code customization such as adding exception handling.

REFERENCES

[1] (2022) Copilot. [Online]. Available: https://github.com/features/copilot/

[2] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow," in *Proceedings of 40th IEEE/ACM International Conference on Software Engineering*, 2018, pp. 886–896.

[3] T. Zhang, D. Yang, C. Lopes, and M. Kim, "Analyzing and supporting adaptation of online code examples," in *Proceedings of 41st IEEE/ACM International Conference on Software Engineering*, 2019, pp. 316–327.

[4] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *Proceedings of 2022 IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 754–768.

[5] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Proceedings of CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–7.

[6] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao, "Clone-based and interactive recommendation for modifying pasted code," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 520–531.

[7] P. T. Nguyen, J. D. Rocco, C. D. Sipio, D. D. Ruscio, and M. D. Penta, "Recommending API function calls and code snippets to support software development," *IEEE Trans. Software Eng.*, vol. 48, no. 7, pp. 2417–2438, 2022.

[8] C. Chen, X. Peng, J. Sun, Z. Xing, X. Wang, Y. Zhao, H. Zhang, and W. Zhao, "Generative API usage code recommendation with parameter concretization," *Sci. China Inf. Sci.*, vol. 62, no. 9, pp. 192 103:1–192 103:22, 2019.

[9] C. Chen, X. Peng, Z. Xing, J. Sun, X. Wang, Y. Zhao, and W. Zhao, "Holistic combination of structural and textual code information for context based API recommendation," *IEEE Trans. Software Eng.*, vol. 48, no. 8, pp. 2987–3009, 2022.

[10] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds., 2012, pp. 826–836.

[11] 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*, 2020, pp. 1536–1547.

[12] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *Proceedings of the 9th International Conference on Learning Representations*, 2021.

[13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[14] 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*, 2021, pp. 8696–8708.

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

[16] (2022) Stack overflow code example about calculating checksum. [Online]. Available: https://stackoverflow.com/questions/14098236

[17] 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.

[18] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[19] A. K. Vijayakumar, M. Cogswell, R. R. Selvaraju, Q. Sun, S. Lee, D. Crandall, and D. Batra, "Diverse beam search: Decoding diverse solutions from neural sequence models," *CoRR*, vol. abs/1610.02424, 2016.

[20] (2022) javalang. [Online]. Available: https://github.com/c2nes/javalang

[21] (2022) Api documentation for messagedigest.getinstance. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html

[22] X. Yi, J. Yang, L. Hong, D. Z. Cheng, L. Heldt, A. Kumthekar, Z. Zhao, L. Wei, and E. H. Chi, "Sampling-bias-corrected neural modeling for large corpus item recommendations," in *Proceedings of the 13th ACM Conference on Recommender Systems, RecSys 2019, Copenhagen, Denmark, September 16-20, 2019*, 2019, pp. 269–277.

[23] 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*, 2020, pp. 7871–7880.

[24] (2022) Tabnine code search service. [Online]. Available: https://www.tabnine.com/code/

[25] (2022) Sprial. [Online]. Available: https://github.com/casics/spiral

[26] S. Gupta, S. Malik, L. Pollock, and K. Vijay-Shanker, "Part-of-speech tagging of program identifiers for improved text-based software engineering tools," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 3–12.

[27] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," in *WWW 1999*, 1999.

[28] (2022) fasttext. [Online]. Available: https://fasttext.cc/

[29] (2022) Replication package. [Online]. Available: https://codealternatives-replication.github.io/

[30] (2022) Huggingface transformers. [Online]. Available: https://github.com/huggingface/transformers

[31] (2022) Fine-tuning tutorial. [Online]. Available: https://huggingface.co/docs/transformers/training

[32] (2022) Stack overflow code example using md5. [Online]. Available: https://stackoverflow.com/questions/20670

[33] (2022) Stack overflow comment about md5. [Online]. Available: https://stackoverflow.com/questions/24740361/

[34] B. L. Welch, "The generalization of student's problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1/2, pp. 28–35, 1947.

[35] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and V. Shanker, "Automatic Generation of Natural Language Summaries for Java Classes," in *21st IEEE International Conference on Program Comprehension (ICPC'13)*. San Francisco, USA: IEEE, 2013, pp. 23–32.

[36] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards Automatically Generating Summary Comments for Java Methods," in *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, Antwerp, Belgium, 2010, pp. 43–52.

[37] M. Liu, X. Peng, A. Marcus, Z. Xing, W. Xie, S. Xing, and Y. Liu, "Generating query-specific class API summaries," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, August 26-30, 2019, Tallinn, Estonia*, 2019, pp. 120–130.

[38] R. Likert, "A technique for the measurement of attitudes." *Archives of psychology*, 1932.

[39] J. Zhou and R. J. Walker, "Api deprecation: a retrospective analysis and detection method for code examples on the web," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 266–277.

[40] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on android application security," in *Proceedings of 2017 IEEE Symposium on Security and Privacy*, 2017, pp. 121–136.

[41] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto, "Toxic code snippets on stack overflow," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 560–581, 2019.

[42] S. Subramanian and R. Holmes, "Making sense of online code snippets," in *Proceedings of the 10th IEEE/ACM Working Conference on Mining Software Repositories*, 2013, pp. 85–88.

[43] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: an analysis of stack overflow code snippets," in *Proceedings of the 13th IEEE/ACM Working Conference on Mining Software Repositories*, 2016, pp. 391–401.

[44] B. Dagenais and M. P. Robillard, "Recovering traceability links between an api and its learning resources," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 47–57.

[45] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 643–652.

[46] Y. Chen, X. Ren, C. Gao, Y. Peng, X. Xia, and M. R. Lyu, "API usage recommendation via multi-view heterogeneous graph representation learning," *CoRR*, vol. abs/2208.01971, 2022.