Hybrid API Migration: A Marriage of Small API Mapping Models and Large Language Models

Bingzhe Zhou
State Key Laboratory of Novel Software Technology, Nanjing University, China
bingzhezhou@smail.nju.edu.cn

Xinying Wang
State Key Laboratory of Novel Software Technology, Nanjing University, China
pangding@smail.nju.edu.cn

Shengbin Xu
State Key Laboratory of Novel Software Technology, Nanjing University, China
kingxu@smail.nju.edu.cn

Yuan Yao
State Key Laboratory of Novel Software Technology, Nanjing University, China
y.yao@nju.edu.cn

Minxue Pan
State Key Laboratory of Novel Software Technology, Nanjing University, China
mxp@nju.edu.cn

Feng Xu
State Key Laboratory of Novel Software Technology, Nanjing University, China
xf@nju.edu.cn

Xiaoxing Ma
State Key Laboratory of Novel Software Technology, Nanjing University, China
xxm@nju.edu.cn

ABSTRACT

API migration is an essential step for code migration between libraries or programming languages, and it is a challenging task as it requires detailed comprehension of both source and target APIs. The existing work either recommends mapped API names only and requires developers to select specific parameters and return value, or uses encoder-decoder models to directly "translate" the source API code into the target API code without considering the characteristics of APIs. In this paper, we propose a hybrid approach that combines small API mapping models with Large Language Models (LLMs). Specifically, the small API mapping model is employed to embed API semantics through their usages and declarations, enabling accurate inference of API mappings across different libraries and programming languages. The inferred mappings are subsequently used as part of the prompts to guide LLMs to generate the target API code corresponding to the source API code. Experimental evaluations demonstrate the effectiveness of our approach in comparison to existing approaches w.r.t. both cross-library and cross-language API migration.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

API migration, API mapping, large language models

ACM Reference Format:
Bingzhe Zhou, Xinying Wang, Shengbin Xu, Yuan Yao, Minxue Pan, Feng Xu, and Xiaoxing Ma. 2023. Hybrid API Migration: A Marriage of Small API Mapping Models and Large Language Models. In *14th Asia-Pacific Symposium on Internetware (Internetware 2023), August 04–06, 2023, Hangzhou, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3609437.3609466

Internetware 2023, August 04–06, 2023, Hangzhou, China

ACM ISBN 979-8-4007-0894-7/23/08…$15.00
https://doi.org/10.1145/3609437.3609466

1 INTRODUCTION

API migration is critical to code migration and has been widely studied [34, 36]. It refers to the process of searching for matching APIs from different libraries or different programming languages, and using them to replace the current APIs in cases when we have to migrate APIs from outdated libraries or adapt the software to different platforms.

Manually migrating APIs is time-consuming and error-prone. It requires not only a thorough understanding of the source API's functionality but also the identification of a compatible match from an often substantial ensemble of candidate APIs. To improve the efficiency and quality of API migration, various automated API migration approaches have been proposed. Most of the existing studies [11, 20, 29, 35, 35, 37, 39, 49] focus on recommending the potential mapping API to developers. These approaches reduce the difficulty of API migration but still require developer involvement in generating the complete API code during the subsequent migration process (*e.g.*, determining parameter names and return value names). In this paper, we refer to recommending API as *API mapping*, and generating the complete API code as *API migration*. Some other work [1, 6, 18, 27, 41, 57] does not distinguish between API migration and code migration, and proposes to directly "translate" source code into target code. However, these work suffers

when translating API code as APIs usually contain much richer information compared to normal code.

Recently, large language models (LLMs) [55] like GPT-3 [7] and PaLM [14] exhibit some surprising emergent abilities [50] and can perform various software engineering tasks [2, 40, 43] very well without fine-tuning. This raises a natural question: **can LLMs help for API migration**?

With this question in head, we first choose GPT-3 as the object of our study, design the task prompt, input it along with source API code, and let the model generate the target API code. However, we observe that the results of GPT-3 are not always satisfactory.[1] On the other side, existing work [13, 32, 44] shows that LLMs benefit from prompts containing more detailed information (*e.g.*, input-output examples). To this end, another questions rises: **how to better leverage LLMs in API migration**?

In this work, we introduce a Hybrid API Migration approach HAPIM that is based on a marriage of "small" API mapping models and "large" language models. Specifically, we propose to use the API mapping results as part of the prompts for LLMs. Our API mapping method adopts a BERT-like architecture [47] and is carefully re-designed to accommodate API mapping tasks. It is fully unsupervised and pre-trains an API embedding model using API usages as training data; it adopts three pre-training tasks, *i.e.*, Bigram Shift Prediction, Masked Subword Prediction, and Masked Word Prediction, to learn joint API embeddings. The final API mapping results are obtained via computing the similarities between API embeddings. We then design a prompt template that integrates the matched APIs, along with the source API code. The prompt is then fed into LLMs guiding the generation of the target API code. To evaluate the effectiveness of HAPIM, we collect both cross-library (Java libraries) and cross-language (Java to C#) API migration datasets, and the results show that HAPIM boosts the performance of directly querying LLMs for API migration by 43.7% in the cross-library case, and 27.4% in the cross-language case.

In summary, the main contributions of this paper include:

- We propose a hybrid approach HAPIM for API migration. HAPIM borrows the capability of LLMs and further augments it in the API migration task via inputting the API mapping results into the prompt.
- We propose a learning-based API mapping approach. It learns the joint API embeddings of both source APIs and target APIs via encoding their API usage semantics and API name semantics.
- Experimental evaluations show that the proposed approaches outperform the existing competitors, and that the combination of small API mapping models and LLMs can further boost the API migration effectiveness.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes our approach HAPIM for API migration, and Section 4 presents the experimental results. Section 5 discusses threats to validity, and Section 6 concludes the paper.

[1]Experimental results show that directly using LLMs does not perform as well as the other competitors as shown in Table 2.

2 RELATED WORK

In this section, we review prior work in the fields of API migration, code migration, and Large Language Models(LLMs).

API migration. Most of the past API migration research recommends potential candidate API names to developers by constructing precise API mappings. They are either rule-based methods [15, 23, 51] or learning-based methods [9–11, 20, 21, 29, 33, 35, 37, 39, 46, 49, 54]. In this work, we mainly focus on the learning-based methods due to their empirical performance.

Earlier learning-based methods mainly infer analogical APIs across languages by comparing their usages [19, 34, 56]. However, these work relies on the existence of bilingual projects (essentially the same project with different languages), which are not always available. Later, representation learning methods have been applied in learning API mapping [9, 37]. These methods learn separate API embeddings for the source domain and target domain respectively, and link them with a set of pre-collected analogical API pairs. However, collecting such API pairs is labor-intensive and the corresponding models could be overfitted to their belonging projects. To mitigate the issues of the above methods, several researchers propose to use API documentation to link two domains [33, 35, 54]. They map APIs relying on the computation of text similarities between accompanied API documents, since natural language words are shared across libraries and languages. However, these methods ignore the API usage semantics which is crucial for API mapping. In view of this, some other work embeds both API documentation and API usages [10, 11, 21]. However, all the above work still suffers from the situation when API documentation is not available in many cases [20, 46].

Another limitation of the above work is that they often simply recommend the API names only, and still require developers to read complex documentation to learn how to implement the recommended API. In contrast, our method does not require developers to have knowledge about the source and target APIs, and we aim to provide fully automated API migration where the complete target API code corresponding to the source API code is generated.

Code Migration. Similar to our settings, the code migration approaches [1, 6, 18, 27, 41, 57] attempt to translate the source code to the target code. Some approaches [1, 18, 27] directly migrate natural language processing models to the programming language domain. Some approaches [6, 41] make use of structure information such as abstract syntax tree (AST) or control flow graph (CFG). MuST-CoST [57] is an encoder-decoder based method that leverages multilingual snippet denoising auto-encoding and multilingual snippet translation pre-training. However, MuST-CoST is concentrated on general code snippets without considering the characteristics of APIs. Therefore, such methods may be not as effective in our API migration task as they are in code migration.

Large Language Models. (LLMs) [7, 12, 31, 42] have shown strong performance to generate accurate output predictions for previously unseen inputs, when provided with a task-related prompt. Prompt-based approaches have been proposed for various tasks including machine translation [42], text evaluation [53], and program synthesis [43]. The prompt format can be designed manually [25] or learned automatically [30]. In this work, we apply prompt-based LLMs in the task of API migration.

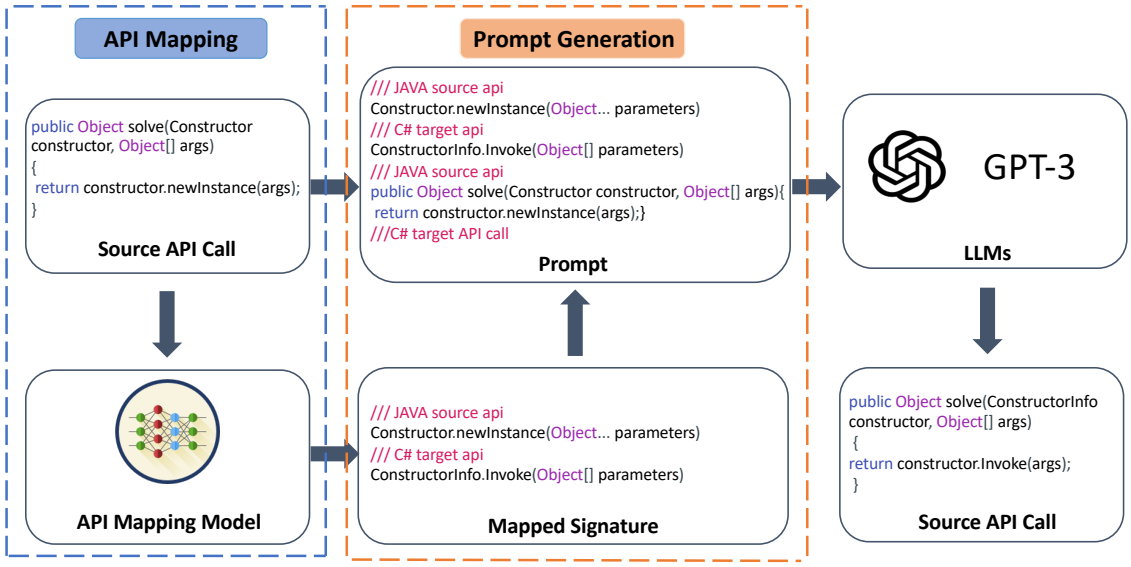


Figure 1: Overview of the proposed approach HAPIM.

3 APPROACH

In this section, we present the proposed API migration approach HAPIM. We start with the overall framework, and then describe the key steps.

3.1 Overview

The overview of our approach is shown in Figure 1, which consists of two key steps:

- 1) *API mapping* that identifies the target API name corresponding to the source API name. The key idea behind this is to accurately learn the API embeddings using code samples from both domains, and then identify the target API name through similarity computation.
- 2) *prompt generation* that includes the target API name as part of the prompt to guide LLMs to complete the API migration task. In this step, we aim to take advantage of the power of LLMs and we design a prompt that can lead the LLMs to accurately obtain the target API code given the source API code.

Note that the proposed approach HAPIM is fully *automatic*. That is, HAPIM returns not only the API names from the target domain, but also automatically fills in the API parameters given the API code from the source domain. HAPIM is also fully *unsupervised* as it does not require any labelling of API mappings or migration pairs. Instead, it learns the API mappings by capturing the API semantics with their usages (which can be collected from open-source platforms such as GitHub) and declarations, and then directly queries the existing LLMs with the learned API mappings.

In the next subsections, we describe the two steps in detail.

3.2 API Mapping Model

Figure 2 illustrates the pipeline of our API mapping method. We first parse the collected code from both source and target domains into Abstract Syntax Trees (ASTs) and perform static analysis to derive API usage sequences. Then the API usage sequences from both source and target domains are fed into a Transformer [47] based learning model to train joint API embeddings. Here, the shared embeddings of the subwords and control tokens in API usage sequences help to align the API embeddings from two domains into the same vector space. Finally, given a query API of the source domain, we calculate the cosine similarities of related APIs in the target domain and return the results in descending order.

3.2.1 API Usage Sequence Construction. There are two desiderata when constructing the API usage sequences: 1) *conciseness* meaning that API usages should be sufficiently concise and capture the common features of different languages, so that two similar APIs also share similar API usages; 2) *expressiveness* meaning that two different API usages should be easily distinguishable from each other.

Figure 3 shows an example of extracting API usage sequences. To achieve conciseness, we parse each code file into a simplified AST. Compared to the corresponding complete AST, a simplified AST contains only tokens related to API and control structures. It removes AST tokens that are not directly related to the APIs of interest (e.g., `ImportDeclaration`, `FieldDeclaration`, and `VariableDeclaration`) and possible language-dependent features such as the `type` of the returned object and the parameters.

To preserve expressiveness, we extract API usage sequences from the simplified AST using the structure-based traversal (SBT) algorithm [24]. Consequently, the extracted sequences contain some additional structure signs (e.g., `cond-end` and `if-end`) as shown in Figure 3. Straightforward traversals (e.g., depth-first traversal) could result in the same sequence for two different simplified ASTs. In contrast, SBT ensures the one-to-one correspondence between simplified ASTs and traversal sequences, and thus the simplified ASTs can be unambiguously restored from the API usage sequences.

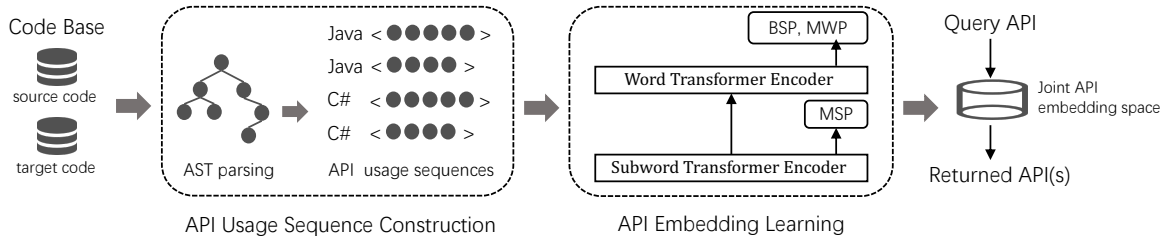


Figure 2: Overview of our API mapping model.

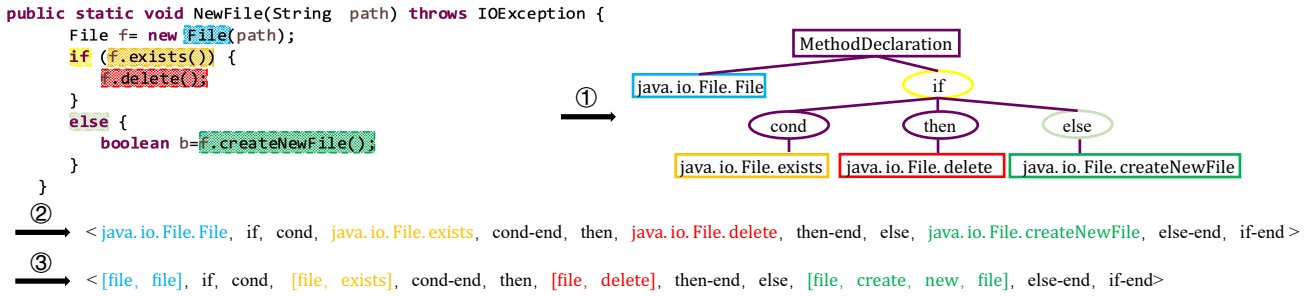


Figure 3: An example of API usage sequence construction. ①: parse code into a simplified AST; ②: traverse the simplified AST to form the API usage sequence; ③: delete the package name of APIs and split the rest into subwords.

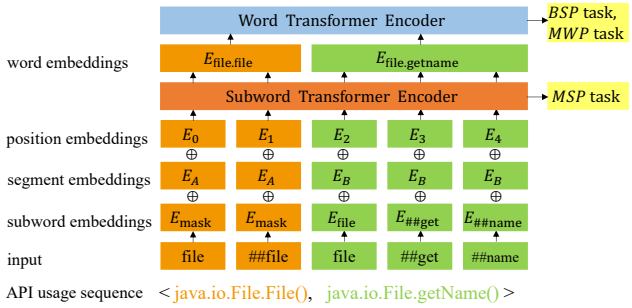


Figure 4: The learning model of our API mapping method.

Finally, to better connect the source domain and target domain with API names, we split each API name into subwords according to the camel-case naming convention. This helps the alignment based on word embeddings. In order to further reduce the sparseness of the vocabulary, we extract stems for the subwords as well. We also delete the package names of APIs as they tend to be diverse and misleading. We keep the class name and method name because both of them are closely related to the functionality of the API.

3.2.2 API Embedding Learning. Figure 4 shows our learning model, which contains two encoders and three training tasks. The two encoders are subword encoder and word encoder, respectively. To capture the long-distance dependencies between API elements, we adopt a BERT-like architecture [17] with multi-layer bidirectional Transformer encoders [48] for both subwords and words.

Subword Encoder. Subword encoder takes subword-level API usage sequences as input and outputs subword-level representations. The representations of multiple subwords belonging to a word are averaged to obtain the initial representation of the word.

Specifically, we denote a word-level API usage sequence as $A = [a_1, a_2, \ldots, a_N]$, and the corresponding subword-level sequence as $S = [s_{1,1}, \ldots, s_{1,l}, s_{2,1}, \ldots, s_{2,l}, \ldots, s_{N,1}, \ldots, s_{N,l}]$, where N is the maximum length of API usage sequence and l is the maximum length of subwords in one word.[2] Essentially, a_i is an API or a control token and $s_i = [s_{i,1}, s_{i,2}, \ldots, s_{i,l}]$ includes the subwords of a_i. To keep the order and dependency of subwords, we use position embeddings and segment embeddings as well. Here, the segment embedding indicates which word-level token the current subword-level token belongs to. For instance, the segment id of $s_{2,1}$ is 2, indicating that $s_{2,1}$ belongs to a_2.[3] Then the sum of the three embeddings (subword embedding, position embedding, and segment embedding) are input into the subword Transformer encoder. The output of the subword encoder is a sequence of vectors, $X = [x_{1,1}, \ldots, x_{1,l}, x_{2,1}, \ldots, x_{2,l}, \ldots, x_{N,1}, \ldots, x_{N,l}]$, where $x_{i,j} \in \mathbb{R}^{d_h}$ indicates the latent state vector of subword $s_{i,j}$ and d_h is its dimension size.

Word Encoder. Next, the subword vectors are averaged to form word-level embeddings as follows,

$$y_i = \sum_{j=1}^{l} x_{i,j}/l. \quad (1)$$

[2]Padding and truncation are done to ensure length.
[3]To distinguish from the position id, we use alphabetical order (i.e., A, B, C, etc.) as segment id in Figure 4.

The word-level embedding sequence $Y = [y_1, y_2, \ldots, y_N]$ are then fed into another multi-layer bidirectional Transformer encoder to obtain word-level vectors $Z = [z_1, z_2, \ldots, z_N]$, where $z_i \in \mathbb{R}^{d_h}$ corresponds to token a_i in the input API usage sequence.

Pre-training Task. The output vectors X and Z can be used in the pre-training tasks to learn the API embeddings. In this work, we consider the following three pre-training tasks: (1) *Bigram Shift Prediction*: change the position of two tokens in the API usage sequence, and predict whether the position of each token is correct (with vectors Z); (2) *Masked Subword Prediction*: mask some API subwords and predict the masked subwords (with vectors X); (3) *Masked Word Prediction*: mask some APIs and predict the masked APIs (with vectors Z).

Bigram Shift Prediction. This task is used to test whether an encoder is sensitive to legal API orders. Specifically, with certain probability, we exchange two successive tokens in an API usage sequence as follows. If the API usage sequence contains a control token (e.g., if) and this token is followed by an API token, we exchange the positions of these two tokens. The reasons for such setups are as follows. First, since an API usage sequence is not usually a total order, exchanging two successive APIs may still result in legal API sequences (e.g., exchanging two successive APIs that create IO streams to read/write different files), thus making this bigram shift task meaningless. In contrast, the order of a control structure and its followed API usually matters (e.g., decide if a file path is null before opening it). Following [3], we set the exchange probability to 15%. We have also set up a label `isShift` $\in \{0, 1\}$ to indicate whether an API sequence has been changed. The objective of bigram shift prediction is to maximize the following term,

$$\sum_{i=1}^{M} \log p(isShift_i|\theta_s, \theta_w, \theta_1), \tag{2}$$

where we use a logistic classifier to predict `isShift`, θ_s is the parameter of the subword encoder layer, θ_w is the parameter of the word encoder layer, θ_1 is the parameter of the logistic classifier, and M is the number of API usage sequences. Note that θ_w depends on θ_s.

Masked Subword Prediction. Recall that we split each API name into subwords. This task is designed to predict the randomly selected subwords. Specifically, we first select 20% subwords per sequence for prediction. Here, when a subword is selected, we select all the subwords in the API it belongs to. Then, similar to the masked language model task in BERT [17], we replace the selected subwords with `[masked]` 80% of the time, keep the selected subwords unchanged 10% of the time, and use random subwords to replace the selected subwords 10% of the time. Finally, we predict each selected subword by using its latent state vector and fully connected layers. The objective function is as follows,

$$\sum_{i=1}^{M} \sum_{s \in S_i} \log p(s|\theta_s, \theta_2), \tag{3}$$

where θ_2 is the parameter for masked subword prediction, θ_s is the parameter of the subword encoder layer, and S_i contains all the selected subwords in the i-th API usage sequence.

Masked Word Prediction. This task is designed for the model to remember the context between APIs. Since we only preserve the subword embeddings and do not have additional word embedding input as shown in Figure 4, we directly mask the APIs whose subwords are masked in the previous training task. Similarly, we predict the masked APIs with latent state vectors and fully connected layers, and the maximization objective function is as follows,

$$\sum_{i=1}^{M} \sum_{w \in W_i} \log p(w|\theta_s, \theta_w, \theta_3), \tag{4}$$

where θ_3 is the parameter for masked word prediction, θ_s is the parameter of the subword encoder layer, θ_w is the parameter of the word encoder layer, and W_i contains all the masked words in the i-th API usage sequence.

Overall, the training objective is as follows,

$$\begin{aligned} L(\theta_s, \theta_w, \theta_1, \theta_2, \theta_3) = &\sum_{i=1}^{M} \log p(isShift_i|\theta_s, \theta_w, \theta_1) \\ &+ \sum_{i=1}^{M} \sum_{s \in S_i} \log p(s|\theta_s, \theta_2) \\ &+ \sum_{i=1}^{M} \sum_{w \in W_i} \log p(w|\theta_s, \theta_w, \theta_3). \end{aligned} \tag{5}$$

3.2.3 API Mapping via Similarity Computation. When given a query API at the inference stage, we identify the corresponding target API as follows. First, we split the names of both source API API_s and candidate target API API_t into subwords and obtain their learned embeddings from the learned model above. Then, we compute the similarity between these two APIs as,

$$sim(API_s, API_t) = \cos(\sum_{i=1}^{O_1} E_{sub_i}, \sum_{j=1}^{O_2} E_{sub_j}), \tag{6}$$

where O_1 (O_2) is the number of subwords in the API_s (API_t), and E_{sub_i} is the embedding of the i-th subword. The similarity sim is calculated by cosine similarity, as it is observed to perform better than many other metrics such as the Jaccard coefficient and Euclidean distance [22].

Next, since there could be multiple APIs with the same API name but different parameters, we further revise the similarity computation result by taking the parameters into consideration. Specifically, we compute the similarity between parameter types using BERT embeddings [16] similar to Eq. 6 and average it with the $sim(API_s, API_t)$ to obtain the final similarity result. APIs in the target domain are ranked in descending order based on the similarity results.

3.3 Prompt Design

To assist LLMs in recalling its pre-training knowledge, prompts are developed as an input format or template for downstream tasks. In our API migration task, we design the prompt to contain both the natural language descriptions of the migration task, and the mapped APIs obtained from our first step.

The template of our prompt as well as a concrete example is shown in Figure 5 and Figure 6. Our prompt template begins with natural language descriptions of both the mapped source API signature and the target API signature, followed by a mapped pair of

```
[API signature Info]
///source api
<source API signature>

///target api
<target API signature>

[description of the API migration tasks]
///source code
<source API code>

///target code
```

Figure 5: Prompt template for API migration.

```
We provide a pair of mapped API signatures between the
      source Java junit library and the target Java
     testng library below:
///source api
junit.framework.Assert.assertEquals(String message, double
     expected, double actual, double delta)

///target api
org.testng.Assert.assertEquals(double actual, double expected,
     double delta, String message)

Try to migrate the source code below from the source
     Java junit library to the target Java testng library
      using the mapped signature we provide above
///source code
double esp = 17.2;
double comp =  devolvido.comparticipaCom();
assertEquals("Doesn't Match", esp, comp, 0);

///target code
```

Figure 6: Example prompt for cross-library API migration.

```
double esp = 17.2;
double comp = devolvido.comparticipaCom();
assertEquals(comp, esp, 0, "Doesn't Match");
```

Figure 7: The generated result of Figure 6.

instance signatures. We then provide LLMs with further descriptions of our API migration task, such as source API code, source library/language, and target library/language. In the example of Figure 6, we first provide the source API signature belonging to the *j*unit library and the target API signature belonging to the *t*estng library, along with the corresponding pair of API signatures in a structured way. This structural correspondence, as opposed to a question-and-answer format, enables LLMs to provide more precise responses. Then we describe the API migration task to LLMs and encourage them to utilize the mapped API signatures. The source code and structured prompt information are presented in the prompt in the same format as the mapped API signatures. The generated target API code is shown in Figure 7.

4 EXPERIMENT

In this section, we present the experimental results. Our experiments are designed to answer the following two research questions:

Table 1: Java library pairs used in mining API mappings.

Source library	Target library
easymock	mockito
junit	testng
slf4j	log4j
json	gson
httpclient	commons-httpclient

- **RQ1:** How effective is the overall framework HAPIM for API migration, in both cross-library and cross-language cases?
- **RQ2:** How effective is our API mapping method, compared with the existing API mapping methods?

4.1 Experimental Setup

4.1.1 Datasets. We curated two datasets in our experiments. The first dataset is for cross-library API migration. We collect five pairs of Java libraries whose APIs can be mapped (see Table 1). To collect the usages of these 10 libraries, we search on GitHub using the library names as keywords. We then use all the returned 10,842 Java projects and extract 63,340 API usage sequences from them as training data. For the test set, we use the API migrations/mappings provided by [5] and identify 51 of them that belong to the 10 collected Java libraries.

The second dataset is for cross-language API mapping. For the training data, we use 14,807 Java projects provided by Allamanis and Sutton [4], and collect the latest 16,383 C# projects on GitHub with at least two stars. We then extract 145,475 API usage sequences. For the test set, we use the API migrations/mappings defined in Java2CSharp,[4] and identify 72 of them that are covered by the collected projects.

4.1.2 Baselines. We first compare our API mapping model with the following five baselines. The former three are specially designed for API mapping, and the latter two are pre-trained code embedding models that can be directly used in API mapping. We choose these baselines as when conducting the API migration task, they all share the same problem setup with our method, i.e., using the source code only as input without requiring extra knowledge such as documents.

- SAR [8, 9]. SAR is the state-of-the-art API mapping method that takes source code only as input. Inspired by [28], it aligns two vector spaces with GAN. In this work, we adopt the unsupervised version for a fair comparison.
- DocInfer [33]. DocInfer uses the related documents to compute API similarities for API mapping. In this work, we adapt it by using subword sequences in API names as the documents.
- HybridEmb [11]. HybridEmb is a recent API mapping method that takes both source code and API names/documents as input. In this work, we adapt it by keeping only the source code and API names.
- CodeBERT [18]. CodeBERT is built upon a Transformer architecture and is trained using a large set of source code

[4]Java2CSharp, http://sourceforge.net/projects/j2cstranslator/.

and documentation. We tokenize API names and obtain the embeddings for each subword to form the final API embeddings.
- CuBERT [26]. CuBERT is another pre-trained code embedding model. Different from CodeBERT, it mixes natural-language tokens with source-code tokens. We take the same step with CodeBERT to obtain API embeddings.

For the complete API migration task, we compare the state-of-the-art methods, i.e., MigrationMapper [5] in cross-library API migration and MuST-CoST [57] in cross-language API migration, respectively.[5] For both cross-library and cross-language tasks, GPT-3 [7] is also employed as a baseline method.

- MigrationMapper [5]: This is a cross-library API migration method that uses code submission records. We revise this method by using the code segments with similar functionalities from the source and target libraries implemented in the change code, as part of the prompts for LLMs to generate API migrations.
- MuST-CoST [57]: MuST-CoST is ross-language code migration method. It trains a encoder-decoder structure to directly translate the source API code to the target one.
- GPT-3 [7] is a strong pre-trained LLM. We only provide the source API code and a natural language description of the desired target language or library as input, and do not provide our API mapping results in the prompt.

4.1.3 Evaluation Metrics. We use *Rec@K* and *Mean Reciprocal Rank (MRR)* to measure the performance of different API mapping methods. *BLEU* and *CodeBLEU* are employed as evaluation metrics for API migration tasks.

- *Rec@K*. It means the proportion of returned results when the correct API mapping is within the top-K list. The value of K is set as 1, 5, 10, and 20.
- *Mean Reciprocal Rank (MRR).* Let *pos* be the ranking position of the ground-truth API in the returned list, and reciprocal rank is defined as $\frac{1}{pos}$. MRR is defined as the average reciprocal rank over all query APIs.
- *BLEU* [38]. BLEU score is a commonly used metric for machine translation tasks. We use BLEU-4 here to examine the n-gram matching between the generated target API code and the ground-truth.
- *CodeBLEU* [45]. CodeBLEU is proposed for measuring the similarity between code-related tasks. It measures the n-gram match, weighted n-gram match, syntax match, and semantic match between the generated API code and the ground-truth. We follow the original weight settings for CodeBLEU in [45].

4.1.4 Implementations. We use `fAST` [52] to parse both Java and C# code files into ASTs. For the extracted API usage sequences, they contain 15 APIs and 34 APIs on average for the cross-library and cross-language datasets. After deleting package names, splitting subwords, lowercasing, and stemming, the maximum subword length of an API is 8 and 5, respectively. We then set the maximum length of subword-level API usage sequence in our model to 120 and 170, respectively. Both subword Transformer encoder and word Transformer encoder contain 12 attention heads and 12 hidden layers. For the latent state size, we search it within {96, 192, 384, 768} and we fix is as 768 by default. We use the Adam optimizer, and perform gradient pre-normalization as used in BERT [17]. We set the batch size to 16, and the learning rate to 0.000001. The code is implemented using Tensorflow v1.13.1, and the experiments are run on a server with 2 Tesla T4 GPUs. We use GPT-3 API from OpenAI and select "gpt-3.5-turbo" as our LLMs.[6] The temperature of LLMs is set to 0 for lower randomness.

4.2 Experimental Results

Table 2: API migration results. The proposed HAPIM generally outperforms the competitors especially in the CodeBLEU metric which is more suitable for code-related tasks.

Methods	Cross-Language Results		Cross-Library Results	
Method	BLEU	CodeBLEU	BLEU	CodeBLEU
MuST-CoST	**0.941**	0.779	-	-
MigrationMapper	-	-	0.755	0.826
GPT-3 (gpt-3.5-turbo)	0.720	0.646	0.553	0.693
HAPIM	0.920	**0.928**	**0.820**	**0.883**

4.2.1 API Migration Results. The API migrations results are shown in Table 2. It can be first observed that HAPIM generally outperforms the competitor methods especially in terms of CodeBLEU scores, which are specifically designed for evaluating code generation tasks. HAPIM achieves the highest CodeBLEU score of 0.928 for the cross-language task and 0.883 for the cross-library task, achieving 19.1% and 6.9% relative improvements compared with the best competitors, respectively. These results indicate that HAPIM produces target API code that are more similar to the ground-truth than the competitors.

Second, in terms of the regular BLEU scores, HAPIM also achieves a score of 0.920 for cross-language task and 0.820 for cross-library task. MuST-CoST achieves the higher BLEU score for the cross-language task. This can be attributed to the fact that the translation nature of MuST-CoST is more suitable for more general measure of text similarity. However, it performs much worse on the CodeBLEU metric which is designed specifically for evaluating code generation tasks, capturing the nuances and complexities of code syntax and structure.

Third, the GPT-3 baseline performs relatively poor, and our HAPIM outperforms it by 43.7% and 27.4% w.r.t. CodeBLEU in the two tasks, respectively. In addition, MigrationMapper is better than GPT-3 but worse than HAPIM. These results highlight the usefulness of combining LLMs with small models that provide more information about API migration tasks. Additionally, providing API mapping into the prompt (HAPIM) also works better than providing similar code snippets (MigrationMapper).

[5] MusT-CoST and MigrationMapper are task-specific in our experiments since MuST-CoST is not able to perform cross-library tasks and MigrationMapper only collects code snippets for cross-library tasks.

[6] As the submission time, Codex has been deprecated by OpenAI, although we observe even better results using Codex.

Table 3: Effectiveness comparison results of API mapping. The proposed API mapping model outperforms the baselines.

Methods	Cross-library results					Cross-language results				
	Rec@1	Rec@5	Rec@10	Rec@20	MRR	Rec@1	Rec@5	Rec@10	Rec@20	MRR
SAR	37.0%	57.4%	68.5%	76.4%	0.452	28.8%	53.4%	72.6%	80.8%	0.394
DocInfer	54.9%	60.8%	64.7%	78.4%	0.581	38.8%	65.3%	77.8%	80.6%	0.504
HybridEmb	56.9%	68.6%	74.5%	78.4%	0.634	44.4%	61.1%	76.4%	83.3%	0.528
CodeBERT	41.2%	62.7%	68.6%	72.5%	0.493	-	-	-	-	-
CuBERT	56.9%	66.7%	74.5%	80.4%	0.629	-	-	-	-	-
Ours	**62.7%**	**72.5%**	**84.3%**	**90.2%**	**0.681**	**51.4%**	**79.2%**	**83.3%**	**94.4%**	**0.631**

Table 4: The effect of each pre-training task. Our method uses all the three tasks (i.e., BSP + MSP + MWP). The results show that all the three training tasks are useful in terms of improving the accuracy of API mapping.

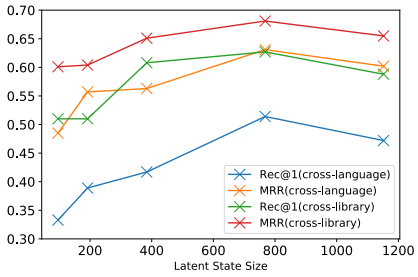
Methods	Cross-library results					Cross-language results				
	Rec@1	Rec@5	Rec@10	Rec@20	MRR	Rec@1	Rec@5	Rec@10	Rec@20	MRR
BSP + MSP	62.7%	70.6%	82.4%	86.3%	0.671	50.0%	72.2%	81.9%	91.7%	0.611
BSP + MWP	62.7%	68.6%	78.4%	86.3%	0.670	51.4%	70.8%	84.7%	91.7%	0.603
MSP + MWP	60.8%	70.6%	78.4%	90.2%	0.661	45.8%	73.6%	84.7%	91.7%	0.591
Ours	62.7%	72.5%	84.3%	90.2%	0.681	51.4%	79.2%	83.3%	94.4%	0.631

4.2.2 API Mapping Results. We first compare the effectiveness of different API mapping methods, and the results are shown in Table 3. The results of our mapping models are the average of five runs, and the standard deviations are all smaller than 0.004. We do not show the results of CodeBERT and CuBERT in the cross-language case as these two models do not provide pre-training results for C#.

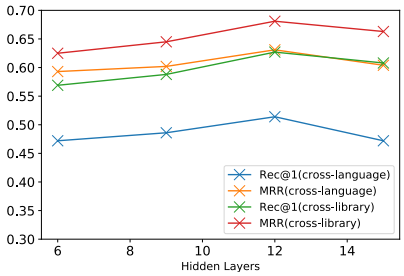
The result shows that our API mapping method performs relatively well: the top-1 accuracy is above 50% and the top-20 accuracy is above 90% for both cross-library and cross-language API mappings. This means that over half of correct API mappings are in the first place of the returned results and over 90% are covered in the first 20 results.

Second, the proposed API mapping approach outperforms all the compared baselines. Compared with the best competitor HybridEmb, it improves it by 3.9% - 11.8% in the cross-library case, and by 6.9% - 18.1% in the cross-language case. HybridEmb separately embeds usages and names for APIs, computes their similarities, and then combines the results with pre-defined weights. In contrast, we propose a joint learning model which simultaneously encodes API usages and API names into the same vector space. This result indicates the power of our learning model. Our mapping approach is much better than SAR (using only API usages) and DocInfer (using only API names), which means that both API usages and API names are important for API mapping. It is also better than the pre-trained models CodeBERT and CuBERT. Although these pre-trained models are built upon large code base, they produce general code embeddings that are not specially designed for API embedding learning. In contrast, our model is trained using API usage sequences as input.

Third, the cross-library case has higher mapping accuracy than the cross-language case. This result is consistent with the intuition that APIs from the same programming language may share more subwords or naming conventions.



(a) The effect of latent state size



(b) The effect of hidden layer number

Figure 8: The parameter sensitivity study.

4.2.3 Performance Gain Analysis and Parameter Sensitivity. Since our API mapping method has several design choices and parameters, we conduct performance gain analysis and parameter sensitivity

experiments here. We first perform an ablation study to measure the effect of each training task, and the results are shown in Table 4. There are three training tasks in our mapping models, i.e., bigram shift prediction (BSP), masked subword prediction (MSP), and masked word prediction (MWP). In the table, 'BSP + MSP' means to keep the first two tasks and delete the MWP task. We can observe from the table that when we remove each of the three tasks, the API mapping accuracy generally decreases. This means that all three training tasks complement each other in terms of improving the API mapping accuracy.

We next study the sensitivity of the hyper-parameters in our API mapping model. Specifically, we vary the latent state size and the number of hidden layers in the encoders, and the results are shown in Figure 8. When tuning one certain hyper-parameter, we keep the others fixed. As we can see in the figure, as either the latent state size or the hidden layer number increases, the API mapping accuracy increases in the beginning and then starts to decrease when these two parameters exceed a certain threshold. This is probably due to overfitting. In this work, we fix the two hyper-parameters to 768 and 12, respectively.

5 THREAT TO VALIDITY

The uncertainty of LLMs is the first threat to validity. For example, the GPT-3 often generates different results even with the same input and same parameter setting due to the sampling strategy used in the generation process. We refer to this phenomenon as the uncertainty of LLMs. This feature might make it difficult for others to fully replicate our results. We will make the results generated by GPT-3 publicly available. In addition, we did not choose other LLMs as they do not provide out-of-the-box APIs and running LLMs locally requires a lot of computational resources.

The second threat comes from evaluation metrics. Although we have conducted experimental evaluations to demonstrate the effectiveness of our approach, the scope of our evaluation may be limited in terms of our evaluation metric BLEU which is the commonly used metric for machine translation. It may not fully capture the quality of the migrated API code. To mitigate this issue, we use CodeBLEU which is able to capture both the text-similarity and other code-related properties (*e.g.*, syntax and data flow). In addition, our experiments rely on automatic metrics to evaluate the effectiveness of our approach. These metrics may not fully capture real-world usability and effectiveness from the user's perspective.

Another threat comes from the datasets used in our experiment. Only one dataset may be biased towards certain types of code or APIs, which may not be representative of the broader API migration landscape. This could affect the validity of our results and limit the applicability of our approach to other datasets. To address this issue, we separately collected cross-library and cross-language datasets and carefully preprocess them, *e.g.*, by de-duplication, to ensure the diversity of APIs in each dataset.

6 CONCLUSION

In this paper, we address the problem of automated API migration, which is crucial for code migration between different libraries or programming languages. We propose a hybrid approach called HAPIM, which combines small API mapping models and large language models (LLMs). Our approach leverages the strengths of LLMs in generating code and the small API mapping models in identifying corresponding APIs across different libraries and programming languages. Specifically, we introduce an API embedding learning approach for unsupervised API mapping and employ the inferred mappings as part of the prompts to guide LLMs to generate the target API code corresponding to the source API code. We evaluate our approach on both cross-library and cross-language API migration datasets and show that HAPIM significantly outperforms other existing approaches. For example, our approach achieves a performance improvement of 43.7% in cross-library API migration and 27.4% in cross-language API migration compared to directly querying LLMs.

Overall, the proposed approach provides a fully automated solution for API migration, eliminating the need for developers to have extensive knowledge about source and target APIs. Future directions include conducting experiments on more datasets and with more LLMs. We also plan to evaluate the effectiveness of our API mapping model in more scenarios, and explore other possibilities of using LLMs in API migration as well as other related tasks.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Grants #62025202), the Fundamental Research Funds for the Central Universities (#020214380102), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Yuan Yao is the corresponding author.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. *CoRR* abs/2103.06333 (2021). arXiv:2103.06333 https://arxiv.org/abs/2103.06333

[2] Toufique Ahmed and Premkumar T. Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *ASE*. ACM, 177:1–177:5. https://doi.org/10.1145/3551349.3559555

[3] Çağla Aksoy, Alper Ahmetoğlu, and Tunga Güngör. 2020. Hierarchical Multitask Learning Approach for BERT. *arXiv preprint arXiv:2011.04451* (2020).

[4] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *MSR*. IEEE, 207–216.

[5] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *ICPC*. 347–357.

[6] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *ICLR*. OpenReview.net. https://openreview.net/forum?id=Bke4KsA5FX

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[8] Nghi Bui. 2019. Towards zero knowledge learning for cross language API mappings. In *ICSE-C*. 123–125.

[9] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: Learning Cross-Language API Mappings with Little Knowledge. In *FSE*. 796–806.

[10] Chunyang Chen. 2020. SimilarAPI: mining analogical APIs for library migration. In *ICSE-C*. IEEE, 37–40.

[11] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2021. Mining Likely Analogical APIs Across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering* 47, 03 (2021), 432–447.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph,

Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[13] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *PLDI*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 487–502. https://doi.org/10.1145/3385412.3385988

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[15] Barthelemy Dagenais and Martin P. Robillard. 2009. SemDiff: Analysis and recommendation support for API evolution. In *ICSE*. 599–602. https://doi.org/10.1109/ICSE.2009.5070565

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*. 1536–1547.

[19] Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. 2013. Inferring likely mappings between APIs. In *ICSE*. 82–91.

[20] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *FSE*. 631–642.

[21] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sung Hun Kim. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In *IJCAI*.

[22] I Hajeer. 2012. Comparison on the effectiveness of different statistical similarity measures. *International Journal of Computer Applications* 53, 8 (2012).

[23] J. Henkel and A. Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *ICSE*. 274–283. https://doi.org/10.1109/ICSE.2005.1553570

[24] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC*. 200–210.

[25] Zhengbao Jiang, Frank F. Xu, Jun Araki, and Graham Neubig. 2019. How Can We Know What Language Models Know? *CoRR* abs/1911.12543 (2019). arXiv:1911.12543 http://arxiv.org/abs/1911.12543

[26] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *ICML*. 5110–5121.

[27] Marie-Anne Lachaux, Baptiste Rozière, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. *CoRR* abs/2006.03511 (2020). arXiv:2006.03511 https://arxiv.org/abs/2006.03511

[28] Guillaume Lample, Alexis Conneau, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. 2018. Word translation without parallel data. In *ICLR*.

[29] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. 2015. SWIN: Towards Type-Safe Java Program Adaptation between APIs. In *PEPM* (Mumbai, India) *(PEPM '15)*. Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/2678015.2682534

[30] Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. *CoRR* abs/2101.00190 (2021). arXiv:2101.00190 https://arxiv.org/abs/2101.00190

[31] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Ré mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (dec 2022), 1092–1097. https://doi.org/10.1126%2Fscience.abq1158

[32] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *CoRR* abs/2107.13586 (2021). arXiv:2107.13586 https://arxiv.org/abs/2107.13586

[33] Yangyang Lu, Ge Li, Zelong Zhao, Linfeng Wen, and Zhi Jin. 2017. Learning to infer API mappings from API documents. In *KSEM*. 237–248.

[34] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *ASE*. 457–468.

[35] Thanh Nguyen, Ngoc Tran, Hung Phan, Trong Nguyen, Linh Truong, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2018. Complementing global and local contexts in representing API descriptions to improve API retrieval tasks. In *FSE*. 551–562.

[36] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *ICSE-C*. 756–758.

[37] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *ICSE*. 438–449.

[38] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *ACL*. 311–318.

[39] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. 2017. Statistical Migration of API Usages. In *ICSE-C*. 47–50. https://doi.org/10.1109/ICSE-C.2017.17

[40] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *APR*. 69–75.

[41] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1139–1149. https://doi.org/10.18653/v1/P17-1105

[42] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[43] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-Modal Program Inference: A Marriage of Pre-Trained Language Models and Component-Based Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 158 (oct 2021), 29 pages. https://doi.org/10.1145/3485535

[44] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485535

[45] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[46] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NuerIPS*. 5998–6008.

[49] Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. 2016. Transforming Programs between APIs with Many-to-Many Mappings. In *ECOOP (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.25

[50] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. [n. d.]. Emergent Abilities of Large Language Models. *TMLR* ([n. d.]).

[51] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. AURA: a hybrid approach to identify framework evolution. In *ICSE*, Vol. 1. 325–334. https://doi.org/10.1145/1806799.1806848

[52] Yijun Yu. 2019. Fast: Flattening abstract syntax trees for efficiency. In *ICSE-Companion*. IEEE, 278–279.

[53] Weizhe Yuan, Graham Neubig, and Pengfei Liu. 2021. BARTScore: Evaluating Generated Text as Text Generation. *CoRR* abs/2106.11520 (2021). arXiv:2106.11520 https://arxiv.org/abs/2106.11520

[54] Zejun Zhang, Minxue Pan, Tian Zhang, Xinyu Zhou, and Xuandong Li. 2020. Deep-Diving into Documentation to Develop Improved Java-to-Swift API Mapping. In *ICPC*. 106–116.

[55] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A Survey of Large Language Models. arXiv:2303.18223 [cs.CL]

[56] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *ICSE*. 195–204.

[57] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual code snippets training for program translation. In *AAAI*, Vol. 36. 11783–11790.