# API Usage Recommendation Via Multi-View Heterogeneous Graph Representation Learning

Yujia Chen, Cuiyun Gao, *Member, IEEE*, Xiaoxue Ren, Yun Peng, Xin Xia, *Member, IEEE*, and Michael R. Lyu, *Fellow, IEEE*

***Abstract*—Developers often need to decide which APIs to use for the functions being implemented. With the ever-growing number of APIs and libraries, it becomes increasingly difficult for developers to find appropriate APIs, indicating the necessity of automatic API usage recommendation. Previous studies adopt statistical models or collaborative filtering methods to mine the implicit API usage patterns for recommendation. However, they rely on the occurrence frequencies of APIs for mining usage patterns, thus prone to fail for the low-frequency APIs. Besides, prior studies generally regard the API call interaction graph as homogeneous graph, ignoring the rich information (e.g., edge types) in the structure graph. In this work, we propose a novel method named *MEGA* for improving the recommendation accuracy especially for the low-frequency APIs. Specifically, besides *call interaction graph*, MEGA considers another two new heterogeneous graphs: *global API co-occurrence graph* enriched with the API frequency information and *hierarchical structure graph* enriched with the project component information. With the three multi-view heterogeneous graphs, MEGA can capture the API usage patterns more accurately. Experiments on three Java benchmark datasets demonstrate that MEGA significantly outperforms the baseline models by at least 19% with respect to the Success Rate@1 metric. Especially, for the low-frequency APIs, MEGA also increases the baselines by at least 55% regarding the Success Rate@1 score.**

***Index Terms*—API recommendation, graph representation learning, multi-view heterogeneous graphs.**

## I. Introduction

In the daily software development process, developers often use the application programming interface (API) provided by some libraries to reduce development time when implementing a function. For instance, the API $BufferedInputStream.read()$ provides an efficient way to read data from an input stream and store the data in a buffer array. However, it is difficult for developers to be familiar with all APIs, because APIs are extensive in quantity and rapidly evolving [1], [2]. In the past two decades, the number of Java Development Kit (JDK) APIs has increased more than 20 times (from 211 classes in the first version of 1996 to 4,403 classes in 2022) [3] [4]. Therefore, when selecting APIs, developers often refer to official technical documentation, raise questions on sites (e.g., Stack Overflow), or query on search engines (e.g., Google), etc. Obviously, the whole process relies on developers' experience, and could be time-consuming since useful information is usually buried in massive contents [5], [6].

Regarding the issues above, previous studies [7], [8] propose to automatically recommend a list of API candidates according to previously-written code, which is demonstrated to be beneficial for improving the API searching process and facilitating software development. For example, MAPO [9] and UP-Miner [10] are based on mining frequent patterns clusters from collected projects to obtain common API usage patterns. PAM [11] uses probabilistic modelling technique in API call sequence to mine usage patterns. FOCUS [12] uses a context-aware collaborative-filtering [13] technique to recommend APIs, relying on the similarity between methods and projects. GAPI [14] applies graph neural networks [15] based collaborative filtering to exploit the relationship between methods and APIs. In this scenario, the API recommendation task can be described as "Which APIs should this piece of code invoke, given that it has already invoked some APIs?," similar to [12], [14]. However, these techniques focus on recommending commonly-used APIs, and tend to fail to mine the usage patterns of the low-frequency APIs. According to our analysis in Section II, the low-frequency APIs occupy a significant proportion of all APIs. According to the statistics, the rarely-appeared APIs account for 76% of the whole APIs in the $SH_L$ dataset which contains java projects retrieved from GitHub and is released by FOCUS [12]. Nevertheless, the recommendation success rate of rare APIs (7.9%) is much lower than that of common APIs (54.2%). Thus, *how to effectively learn low-frequency APIs usage patterns is a great yet under-explored challenge* [16]. Besides, the existing techniques highly rely on the homogeneous interaction information between APIs and methods, ignoring the rich contextual information in source code

Manuscript received 3 August 2022; revised 14 February 2023; accepted 20 February 2023. Date of publication 3 March 2023; date of current version 16 May 2023. This work was supported in part by the National Key R&D Program of China under Grant 2022YFB3103900, in part by the National Natural Science Foundation of China under Grant 62002084, in part by the Natural Science Foundation of Guangdong Province under Grant 2023A1515011959, in part by Shenzhen Basic Research under Grant JCYJ20220531095214031, in part by Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies under Grant 2022B1212010005, and in part by the Major Key Project of PCL under Grants PCL2022A03, PCL2021A02, and PCL2021A09. Recommended for acceptance by T. Menzies. *(Corresponding author: Cuiyun Gao.)*

Yujia Chen and Cuiyun Gao are with the Harbin Institute of Technology, Shenzhen, Guangdong Province 518055, China (e-mail: yujiachen@stu.hit.edu.cn; gcyydxf@gmail.com).

Xiaoxue Ren, Yun Peng, and Michael R. Lyu are with the The Chinese University of Hong Kong, Hong Kong (e-mail: xiaoxueren@cse.cuhk.edu.hk; ypeng@cse.cuhk.edu.hk; lyu@cse.cuhk.edu.hk).

Xin Xia is with the Software Engineering Application Technology Lab, Huawei 518129, China (e-mail: xin.xia@acm.org).

Digital Object Identifier 10.1109/TSE.2023.3252259

(e.g., co-occurring APIs and hierarchical structure in projects and packages). In fact, APIs under the same package are more likely to be called together (e.g., *file.open()* and *file.close()*) are under the package *java.io*), which is important external information for API recommendation. Therefore, *how to involve contextual information in API recommendation is another challenge*.

In this work, we propose MEGA, a novel API usage recommendation method with **M**ulti-view h**E**terogeneous **G**raph represent**A**tion learning. In a software project, the *method declaration* is the smallest functional unit, which consists of a name, a list of parameters, a return type, and a body. Following the prior studies [12], [14], we define the name of the method declaration as "method" and the API call in the code body as "API". Thus, "methods" refer to the name of method declarations in a library's client code, and an API may be a third-party library method or method defined in the same project. Different from the prior studies, MEGA employs heterogeneous graphs, which are constructed from multiple views, i.e., method-API interaction from local view, API-API co-occurrence from global view, and project structure from external view. Specifically, MEGA builds upon three heterogeneous graphs, i.e., the common *call interaction graph*, and two new graphs, i.e., *global API co-occurrence graph* and *hierarchical structure graph*. The *call interaction graph* establishes the relations between methods and corresponding called APIs, and is commonly adopted by previous approaches [11], [12], [14], [17], [18]. Models based on only such graph cannot well capture the representations of the APIs with rare called frequencies. To improve the API representations, the *global API co-occurrence graph* is introduced to build the relations between APIs with the co-occurrence frequencies incorporated. To enrich the representations of APIs and methods with contextual structure information, MEGA also involves the called information by projects and packages, composing the *hierarchical structure graph*. A graph representation model is then proposed to learn the matching scores between methods and APIs based on the multi-view graphs. To integrate the multi-view knowledge, a frequency-aware attentive network and a structure-aware attentive network are proposed to encode the co-occurrence information and hierarchical structure, respectively. In the early stage of development, developers may be more interested in knowing which APIs have been called by the methods with similar functionality for achieving the current functional requirements [12]. For example, when a developer implements a client method for opening the server and receiving data from the client, our approach could recommend some relevant network programming APIs *ServerSocket()*, *ServerSocket.accept()*, *Socket.getInputStream()* and *Socket.read()*. With the recommended results, the developer could know which APIs should this piece of code invoke. In addition, we conduct a user study to investigate whether it could help developers with their implementation tasks. The results show that 69% of the participants think that the API lists recommended by MEGA are relevant and correct for current programming.

We evaluate the effectiveness of MEGA on three Java benchmark datasets consisting of 610 Java projects from GitHub and 868 JAR archives from the Maven Central Repository. In addition, we also simulate the real development scenario [12] where a developer has already called some APIs in a method. Then MEGA recommends APIs based on the called APIs by client methods, and calculates the evaluation metrics. The experimental results show that MEGA outperforms the baseline approaches (PAM [11], FOCUS [12] and GAPI [14]) by at least 19% with respect to the Success Rate@1 metric. For the low-frequency APIs, MEGA also achieves an increased rate at more than 55% compared to the baselines. The source code is publicly accessible at https://github.com/hitCoderr/APIRec_MEGA.git for researchers to validate and conduct further research.

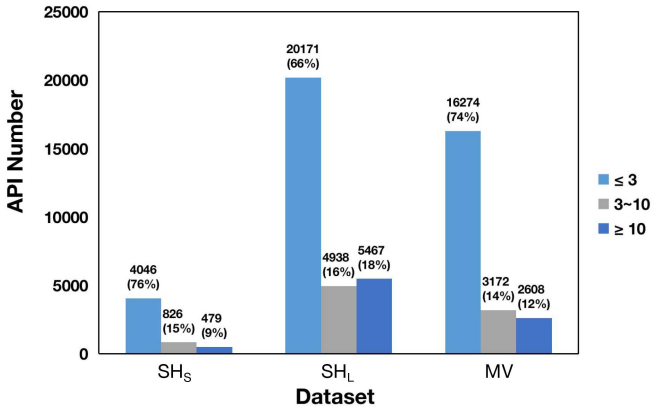



Fig. 1. The distribution of APIs called frequency in $\mathbf{SH_S}$, $\mathbf{SH_L}$ and $\mathbf{MV}$ datasets.

In summary, our main contributions in this paper are as follows:

- To the best of our knowledge, we are the first work to construct multi-view heterogeneous graphs for more accurate API usage recommendation.
- We propose a novel API recommendation approach named MEGA, which designs a graph representation model with a frequency-aware attentive network and a structure-aware attentive network to generate enhanced representations of methods and APIs.
- We perform experiments on three benchmark datasets, and the results demonstrate that MEGA outperforms the state-of-the-art API usage recommendation approaches, even for the low-frequency APIs.

*Outline.* The rest of paper is organized as follows: Section II introduces details of our motivation. Section III presents the overall workflow of MEGA and architecture of the graph representation model in MEGA. Sections IV and V are the settings and results of evaluation, respectively. Section VII analyzes some implications and threats to validity. Section VIII succinctly describes related works. In the end, in Section IX, we conclude the whole work.

## II. Motivation

Fig. 1 shows the distribution of APIs with different occurrence frequencies in three benchmark datasets [12], i.e., $SH_S$, $SH_L$ and $MV$, with detailed statistics of the datasets shown in Table II. Obviously, APIs with lower occurrence frequencies (i.e.,

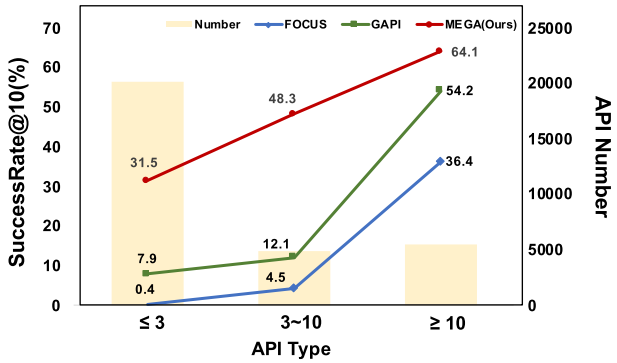



Fig. 2. API recommendation performance of FOCUS [12], GAPI [14] and our proposed MEGA (corresponding to the APIs with different frequencies on the $\mathbf{SH_L}$ dataset).

$\leq 3$) account for significant proportions (i.e., $> 65\%$) among all the APIs in each dataset. Although appearing less frequently, the large proportion of such APIs indicates developers' strong demands for specific functions, and accurately recommending the APIs is critical for facilitating their daily programming.

Fig. 3 illustrates an example of a client method which queries the low-frequency APIs. In this scenario, the developer is working on a method to get the name of a JAR package, but is not sure which APIs to use next. The "true API calls" in Fig. 3 list the APIs in ground truth, in which both "*JAXBContext.createUnmarshalle()*" and "*Unmarshaller.unmarshal(java.io.InputStream)*" are rarely appear in the datasets. Both popular models including FOCUS [12] and GAPI [14] learn the API representations ineffectively, and fail to recommend the APIs. Fig. 2 depicts the API recommendation performance of the two models corresponding to APIs with different frequencies on the $SH_L$ dataset with respect to the SuccessRate@10 score. We find that the APIs appearing rarely, e.g., $\leq 3$, present significantly poor performance than the APIs appearing frequently, e.g., $\geq 10$. The results show that the existing models are difficult to recommend the low-frequency APIs.

Besides, existing approaches [10], [12], [14], [18], [19] generally regard the API call interaction graph as homogeneous graph, ignoring the rich heterogeneous information (e.g., edge types) in the graph. For example, the state-of-the-art models, FOCUS [12] and GAPI [14] are based on collaborative filtering for measuring the similarities between all methods to recommend APIs. The learning process in the models tends to rely on the commonly-used APIs in methods, resulting in ineffective API recommendation. As the example shown in Fig. 3, the API recommended by FOCUS for improving the speed and efficiency of byte stream operations comes from the *BufferedReader* class, which is a very general yet function-irrelevant operation for the current client method.

*Our Approach.* To address the above limitations of the existing models, we try to exploit the rich heterogeneous information in source code from multiple views, including method-API interaction from local view, API-API co-occurrence from global view, and project structure from external view, respectively. Specifically, we build three heterogeneous graphs from each view, i.e., call interaction graph, global API co-occurrence graph and

```
Client Method:
private String getJarName(String path, JAXBElement<Model> fhElement){
    Model model = fhElement.getValue();
    Modules ms = model.getModules();
    String originalName = null;
    if (ms == null) originalName = model.getArtifactId();
    else originalName = ms.getModule();
    JAXBContext jaxbContext = JAXBContext.newInstance("pom._4_0");
    // which APIs are next?
}
```

True API Calls :
- ✓ JAXBContext.createUnmarshaller()
- ✓ FileInputStream.FileInputStream(java.lang.String)
- ✓ Unmarshaller.unmarshal(java.io.InputStream)

API Calls Returned by FOCUS :
- BufferedReader.BufferedReader()
- BufferedReader.close()
- BufferedReader.readLine()

API Calls Returned by GAPI :
- StreamResult.StreamResult()
- Date.getTime()
- Class.forName()

API Calls Returned by OURS :
- JAXBContext.createUnmarshaller()
- Model.getVersion()
- Unmarshaller.unmarshal(java.io.InputStream)

Fig. 3. An Example of API usage recommendation. *(The true API calls and TOP-3 APIs recommended by FOCUS [12], GAPI [14] and our proposed MEGA, respectively.).*

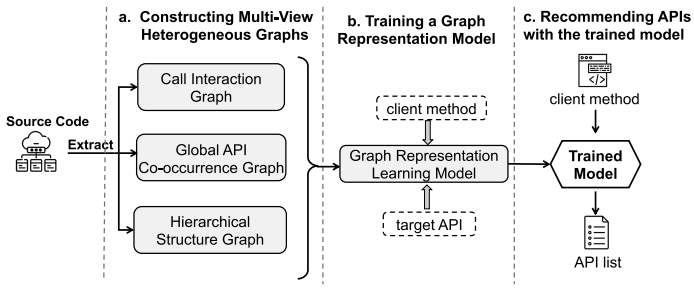



Fig. 4. The overall workflow of MEGA.

hierarchical structure graph. Moreover, two new attentive networks are designed for encoding frequency-based co-occurrence information and structure-based hierarchical information during learning the representations of APIs and methods. As the example shown in Fig. 3, MEGA captures the co-occurring pattern with the API "*JAXBContext.NewInstance*($\ddot{P}OM^{\cdot\cdot}$)" and their similar structural information (i.e., under the same class), so it successfully recommends the rare API "*JAXBContext.CreateUnmarshaller()*".

## III. Methodology

### A. Workflow of MEGA

Fig. 4 illustrates the MEGA's workflow which includes three main stages, i.e., *constructing multi-view heterogeneous graphs*, *training a graph representation model* and *recommending APIs with the trained model*. In the first stage, we construct three heterogeneous graphs, i.e., *call interaction graph*, *global API*

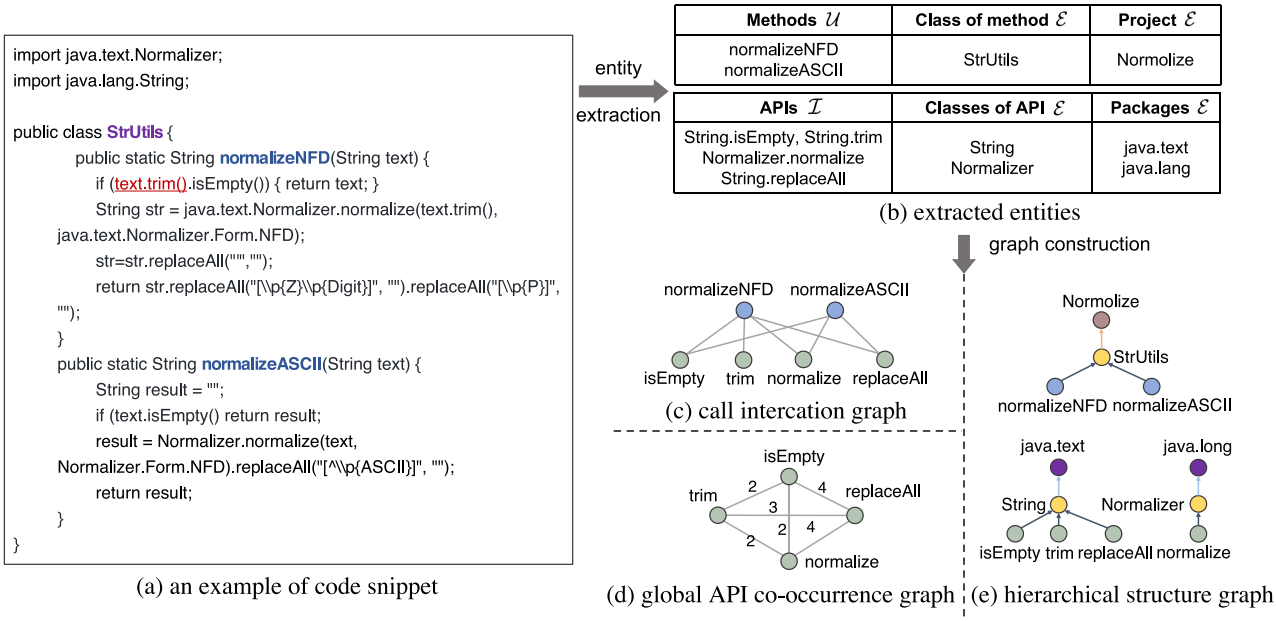



Fig. 5. An example of element extraction and graph construction from a code snippet.

TABLE I
SYMBOLS AND CORRESPONDING DESCRIPTIONS

| Notation | Description |
|---|---|
| $\mathcal{U}$,$\mathcal{I}$ | The set of methods, APIs |
| $\mathcal{E}$,$\mathcal{R}$ | The set of entities, relations |
| $\mathcal{G}_I$ | The call interaction graph |
| $\mathcal{G}_C$ | The global API co-occurrence graph |
| $\mathcal{G}_H$ | The hierarchical structure Graph |
| $(i, f, j)$ | The co-occurrence triple |
| $(h, r, t)$ | The structure triple |
| $u, i$ | The client method, the target API |
| $L$ | The hop number of information encoding |
| $\mathcal{E}_u^l$ | The $l$-hop entity set of $u$ |
| $\mathcal{S}_u^l$ | The $l$-hop triple set of $u$ |
| $e_u^{(o)}, e_i^{(o)}$ | The local-view representation of the method $u$,$i$ |
| $e_u^{(c)}, e_i^{(c)}$ | The global-view representation of $u$,$i$ |
| $e_u^{(h)}, e_i^{(h)}$ | The external-view representation of $u$,$i$ |
| $e_u, e_i$ | The fianl representation of $u$,$i$ |

*co-occurrence graph*, and *hierarchical structure graph*. The nodes of these graphs include APIs, methods, classes, projects, and packages. We extract the relations between nodes from the source code. Then, in the second stage, a graph representation model is proposed to encode the three graphs and integrate the graph representations for recommendation. In the last stage, we employ the trained model to return a ranked list of API usage recommendation according to the code snippet of the current client method. The details of our approach are explained in the following parts. For facilitating readers' understanding of the proposed approach, we list the key notations in Table I.

### B. Constructing Multi-View Heterogeneous Graphs

In this section, we present the graph construction process of the multi-view graphs. We parse the source code in the dataset, identify methods, APIs, classes, projects and packages, and extract different relationships between these code entities. Fig. 5(a) illustrates a toy example of a project with one class and some methods/APIs. After parsing, we identify the code entities from the project *Normolize* – methods (*normalizeNFD* and *normalizeASCII*), APIs (*String.isEmpty*, *String.trim*, *String.replaceAll* and *Normalizer.normalize*), classes of methods/APIs (*StrUtils*, *String* and *Normalizer*) and packages (*java.text* and *java.lang*), as shown in Fig. 5(b).

Meanwhile, we also record the method-API interactions (e.g., *normalizeNFD* calls *String.isEmpty*, *String.trim*, etc), the API-API co-occurrence information (e.g., *String.isEmpty* and *Normalizer.normalize* are used together twice in the project), and the hierarchical structure (e.g., *normalizeNFD* is declared in *StrUtils* and *StrUtils* is one class of *Normolize*). With these code entities as graph vertices and the relationships between entities as graph edges, the three graphs can be constructed, as plotted in Fig. 5(c), (d), and (e).

We analyze all the projects in the datasets and collect all called APIs in the code repositories to form the API set $\mathcal{I}$, following previous work [12], [14]. We also obtain the method set $\mathcal{U}$. Besides, all the classes, projects, and packages are denoted as node sets $\mathcal{E}$, and all relationships are represented as an edge set $\mathcal{R}$. We present the details of the graph construction as follows.

*1) Call Interaction Graph* $\mathcal{G}_\mathbf{I}$. It represents the call relations between methods and APIs, denoted as a bipartite graph $\mathcal{G}_I = \{(u, y_{ui}, i)|u \in \mathcal{U}, i \in \mathcal{I}\}$, where $y_{ui} = 1$ indicates a method $u$ calls an API $i$. For example, *[myFile.createFile(), 1, java.io.File.exists()]* indicates that a method *myFile.createFile()* calls an API *java.io.File.exists()*. The call interaction graph reflects the basic relations between APIs and methods, and is commonly adopted by prior studies [11], [12], [14], [17], [18].

*2) Global API Co-occurrence Graph* $\mathcal{G}_\mathbf{C}$. It records the co-occurrence relations between APIs, e.g., the two API *file.open()*

**Algorithm 1:** Global API Co-Occurrence Graph Construction($S$, $\varepsilon$).

**input :** An API sequence set $S$ and an integer $\varepsilon$
**output:** A global API co-occurrence graph
1 Let $V$ and $E$ represent the set of vertices and edges in the co-occurrence graph respectively;
2 $V \leftarrow \emptyset$, $E \leftarrow \emptyset$;
3 Let $\omega$ represent a weight function : $E \rightarrow \mathbb{R}^{\geq 0}$, where each edge $e$ in $E$ has a weight $\omega(e)$;
4 **foreach** $A$ ***in*** $S$ **do**
5  append all API nodes in $A$ into $V$ ;
6  **foreach** $i_m$ ***in*** $A$ **do**
7  $E \leftarrow E \cup \{(i_m, i_n) | i_n \in A \wedge n \in \{m, m+\varepsilon\}\}$ ;
8 **foreach** $e$ ***in*** $E$ **do**
9  $\omega(e) \leftarrow \omega(e) + 1$ ;
10 $\mathcal{G}_C \leftarrow$ build co-occurrence graph with $V$, $E$ and $\omega$ ;
11 **return** $\mathcal{G}_C$;

**Algorithm 2:** Encoding Process of $\mathcal{G}_C$ and $\mathcal{G}_H$.

**input :** a constructed graph $\mathcal{G}$, a number of max-hop $L$, *AttentiveNetwork*, an entity set $\mathcal{E}$
**output:** a list of representation $K_e$
1 let $N(v, \mathcal{G})$ be the set of $v$'s neighbors in $\mathcal{G}$;
2 let $\mathcal{E}$ be the first-hop entity set $\mathcal{E}^{(0)}$;
3 $K_e \leftarrow \emptyset$;
4 **for** $l$=0, $1 \cdots$ , $L$ **do**
5  $S^{(l)} \leftarrow \{(i, e(i,j), j) | i \in \mathcal{E}^{(l)} \wedge j \in N(i, \mathcal{G})\}$;
6  $e^{(l)} \leftarrow$ *AttentiveNetwork*($S^{(l)}$);
7  $\mathcal{E}^{(l+1)} \leftarrow \{ j \mid i \in \mathcal{E}^{(l)} \wedge j \in N(i, \mathcal{G})\}$;
8  append $e^{(l)}$ into $K_e$;
9 **return** $K_e$;

and *file.close()* are connected since they ever appeared together in some methods. Algorithm 1 shows the pseudo-code for global API co-occurrence graph construction. The graph is built based on a set of API sequences $S$ and an integer $\varepsilon$. Specifically, we first initialize the set of vertices $V$=$\emptyset$ and the set of edges $E$=$\emptyset$ in the co-occurrence graph (line 2). For each sequence $A$ in the API sequence set $S$ (line 4), we add all the APIs in the sequence $A$ to the vertices sets $V$ (line 5). Then, for each API $i_m$ in sequence $A$ (line 6), any API $i_n$ within the distance of $\epsilon$ has a co-occurrence relationship with API $i_m$. To represent the co-occurrence relationship between $i_m$ and $i_n$, we add an edge $(i_m, i_n)$ into the edge set $E$ (line 7). Next, for each edge $e$ in $E$, we update the $\omega(e)$ by counting the occurrence frequencies (lines 8-9). Finally, we build the co-occurrence graph $\mathcal{G}_C$ based on $V$, $E$ and $\omega$, and return the co-occurrence graph $\mathcal{G}_C$ (lines 10-11). $\mathcal{G}_C$ is denoted as $\{(i, f, j) | i, j \in \mathcal{I}, f \in \mathcal{T}\}$, where each triplet describes that API $i$ and API $j$ are invocated together $f$ times.

For example, *[file.open(), 10, file.close()]* indicates that *file.open()* and *file.close()* appear together 10 times. The global API co-occurrence graph contains frequency-enriched API relationships, which is beneficial for enriching APIs with by their relevant APIs in this graph.

*3) Hierarchical Structure Graph* $\mathcal{G}_\mathbf{H}$. The hierarchical information, e.g., the belonging projects/packages, implies the functionality of APIs and methods, thereby helpful for API recommendation.

We consider both project-level and package-level information, i.e., the projects where methods are declared and packages that APIs belong to, for constructing the *hierarchical structure graph*. We construct the graph as a directed graph, denoted as $\mathcal{G}_H = \{(h, r, t) | h, t \in \mathcal{E}, r \in \mathcal{R}\}$, in which each triplet $(h, r, t)$ represents there is a relation $r$ from head entity $h$ to tail entity $t$, $\mathcal{E}$ is the set of all entities, including *API*, *method*, *class*, *project*, and *package*, and $\mathcal{R}$ is the set of relations including *belong-to-class*, *belong-to-project* and *belong-to-package*. As the example depicted in Fig. 5(e), projects and packages are organized as a tree structure. The hierarchical structure graph contains the attribute information of the APIs. These classes and packages are considered as the side information to enrich the representations of APIs.

### C. Training Graph Representation Model and Recommendation

This section introduces how MEGA trains a graph representation model based on the constructed multi-view graphs, and utilizes the trained model to make API recommendation, corresponding to the second stage and third stage in Fig. 4, respectively.

Fig. 6 illustrates the whole process of training and recommendation, including three graph encoding modules, i.e., *call interaction encoding*, *co-occurrence information encoding* and *hierarchical structure encoding*, as well as one fusion and prediction module. Given a client method, a target API, and the three heterogeneous graphs as input, the graph representation model aims to predict the probability of the target API invocated by the client method. In the first module, an *Embedding Network* is employed to encode basic interaction information into local-view representations of the client method and the target API, as shown in Fig. 6 (1). Then, in the second module, as illustrated in Fig. 6 (2), a *Frequency-aware Attentive Network* is designed to encode frequency-based co-occurrence information into representations of the client method and the target API from global view. Next, in the third module, a *Structure-aware Attentive Network* is designed to encode structure-based hierarchical information into representations of the client method and the target API from external view, as shown in Fig. 6 (3). Finally, in the last module, the local-view, global-view and external-view representations are concatenated as the final representations of the client method and the target API.

*1) Call Interaction Encoding:* Call interaction reflects basic information of the client method $u$ and the target API $i$, respectively. We represent the client method $u$ by its called APIs and represent the target API $i$ by its related methods, integrating the call interaction information into their local-view representations. Specifically, for each client method $u$, the called API set is denoted as $\mathcal{E}_u = \{i \mid i \in \{i \mid (u, y_{ui}, i) \in \mathcal{G}_\mathcal{I} \; and \; y_{ui} = 1\}\}$. We then obtain the client method $u$ representation according to its called API set: $e_u^{(o)} = \frac{\sum_{i \in \mathcal{E}_u} e_i}{|\mathcal{E}_u|}$, where $e_i$ is the embedding

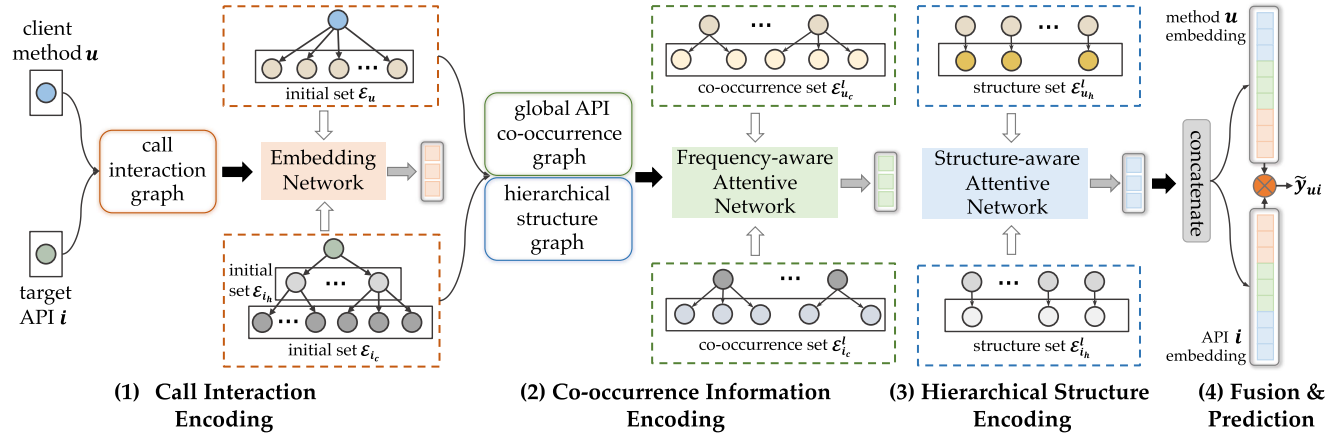
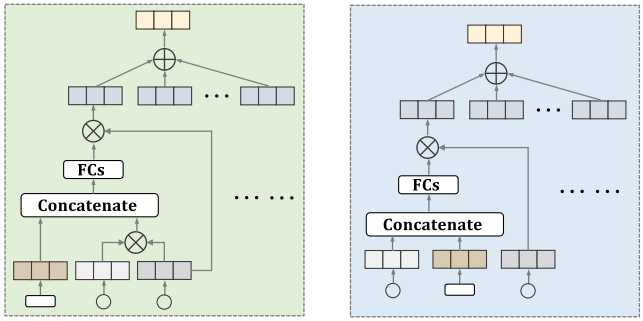



Fig. 6. Training and recommending process of the graph representation learning model in MEGA.




(a) Frequency-aware Attentive Network. (b) Structure-aware Attentive Network.

Fig. 7. Illustration of two attentive networks in the graph representation model.

of API $i$ and $|\mathcal{E}_u|$ is the set size. Similarly, we obtain the target API $i$ representation $e_i^{(o)}$.

*2) Co-Occurrence Information Encoding:* For the client method $u$, co-occurred APIs of its called APIs are potential to be called next. For the target API $i$, co-occurred APIs reflect its usage patterns (which APIs are used frequently with the target API). Thus, in this module, we represent the client method $u$ and the target API $i$ by their co-occurred APIs, integrating the co-occurrence information into the representations. According to the definition of the global API co-occurrence graph, the weights between a pair of API nodes present the co-occurrence frequency, implying the extent of their relevance. To encode the global-view information of the client method $u$ and the target API $i$, we design a *frequency-aware attentive network*, as shown in Fig. 7(a). Algorithm 2 shows the pseudo-code for the encoding. For the client method $u$, the APIs co-occurred with its called APIs reveal the method's potential call need. Thus, we utilize the API set $\mathcal{E}_u$ obtained from Section III-C1 as the initial set $\mathcal{E}_u^{(0)}$ for the first-hop propagation on $\mathcal{G}_C$ (line 2). After initialization, we conduct information encoding to generate co-occurrence representation in each hop (line 4-8).

*Information Encoding (line 4-8).* For each triple in global API co-occurrence graph, i.e., $(i, f, j) \in \mathcal{G}_C$, we define the $l$-hop triple set based on the entity set $\mathcal{E}_u^{(l)}$ as: $\mathcal{S}_u^l = \{(i, f, j) | i \in \mathcal{E}_u^l\}$ ($l$ begins with 0). Following previous works [20], [21], we sample a fixed-size triple set instead of using a full-size set to reduce the computation overhead.

Based on co-occurred API triple sets, i.e., $(i, f, j) \in S_u^l$, we learn $l$-hop co-occurrence representation of the client method $u$ by its related APIs. Since APIs with different co-occurrence frequencies have different contributions on representing the client method $u$, we design a coefficient $\pi(i, f, j)$ to control the attentive weight:

$$e_u^{(l)} = \sum_{(i,f,j)\in\mathcal{S}_u^l} \pi(i, f, j)\, e_j, \tag{1}$$

where coefficient $\pi(i, f, j)$ is attentively calculated based on the API similarity and their co-occurrence frequency:

$$\pi(i, f, j) = \frac{\exp(\text{mlp}((e_i \odot e_j)||e_f))}{\sum_{(i',f',j')\in\mathcal{S}_u^l} \exp(\text{mlp}((e_{i'} \odot e_{j'})||e_{f'}))}, \tag{2}$$

where the notation $\odot$ denotes the element-wise multiplication operation, and $||$ denotes the concatenation operation. $e_i$ and $e_j$ are the embeddings of API $i$ and its co-occurred API $j$, respectively. $e_f$ is the embedding of frequency $f$. mlp($\hat{\text{A}}\cdot$) is a three-layer MLP with Relu [22] as the nonlinear activation function. The attention mechanism for encoding the $l$-hop co-occurrence representation (i.e., (1) and (2)) explicitly introduces co-occurred frequency $f$ into calculating the influence of co-occurred API $i$ on the representation of API $j$.

After performing $L$-hop information encoding, where $L$ is the max hop number, we obtain the global-view representations $e_u^{(c)}$ of the client method $u$ by appending the representations from all hops: $e_u^{(c)} = \{e_u^{(0)}, e_u^{(1)}, \ldots, e_u^{(L)}\}$. Similarly, we obtain the global-view representation $e_i^{(c)}$ of the target API $i$. The global-view representations $e_u^{(c)}$ and $e_i^{(c)}$ captures frequency-enriched co-occurrence information for enhancing the semantic representations of the client method and the target API, respectively.

*3) Hierarchical Structure Encoding:* External hierarchical attribute information such as classes can also contribute to enriching the representations. In this module, we represent the

client method $u$ and the target API $i$ by structure entities, integrating the hierarchical structure information into their representations. According to the definition of *hierarchical structure graph*, as described in Section III-B, different head entities and relations endow tail entities with different semantics. To obtain the representations of the client method $u$ and the target API $i$ from external view, we design a *structure-aware attentive network*, as shown in Fig. 7(b).

The encoding process for the hierarchical structural graph is similar to the encoding process of the API co-occurrence information, as illustrated in Section III-C2, except for the design of the attentive network. Specifically, for the $l$-hop triple set $\mathcal{S}_u^l = \{(h,r,t)|h \in \mathcal{E}_u^l\}$ in *hierarchical structure graph*, we learn $l$-hop structure representation for client method $u$ by:

$$e_u^{(l)} = \sum_{(h,r)\in\mathcal{S}_u^l} \pi(h,r)\, e_t, \tag{3}$$

where coefficient $\pi(h,r)$ is attentively calculated as:

$$\pi(h,r) = \frac{\exp(\text{mlp}(e_h||e_r))}{\sum_{(h',r',t')\in\mathcal{S}_u^l} \exp(\text{mlp}(e_{h'}||e_{r'}))}, \tag{4}$$

where $e_h$, $e_t$ are the embeddings of head entity $h$ and tail entity $t$, respectively. $e_r$ is the embedding of relation $r$. The structure-aware attention mechanism (i.e., (3) and (4)) explicitly endows the relevance calculation of tail entity $t$ with the relation $r$. Based on the structure encoding, we finally obtain the external-view representations $e_u^{(h)}$ and $e_i^{(h)}$ for the client method $u$ and the target API $i$, respectively.

*4) Fusion and Prediction:* In this module, the three representations with different semantic information are concatenated to form the final representation of the client method $u$ and the target API $i$, i.e., $e_u = e_u^{(o)}||e_u^{(c)}||e_u^{(h)}$ and $e_i = e_i^{(o)}||e_i^{(c)}||e_i^{(h)}$. Then, we can calculate the call probability between the client method $u$ and each target API $i$ using their learned representations. In this paper, we use the inner product as the similarity function: $\hat{y}_{ui} = e_u^\top e_i$. Finally, we rank all the candidates by descending order and return the top-k APIs to the developer.

## IV. Experimental Setup

In this section, we conduct extensive experiments to evaluate the proposed approach with the aim of answering the following research questions:

- *RQ1:* How does MEGA perform compared with the state-of-the-art API usage recommendation approaches?
- *RQ2:* What is the impact of the three encoding components (i.e., *Call Interaction Encoding, Co-occurrence Information Encoding* and *Hierarchical Structure Encoding*) in the graph representation model on the performance of MEGA?
- *RQ3:* How does MEGA perform on low-frequency APIs?
- *RQ4:* How do different hyper-parameter settings affect MEGA's performance?

TABLE II
Statistics of the Three Datasets: $\mathbf{SH_S}$, $\mathbf{SH_L}$ and $\mathbf{MV}$. The Call-Avg Means the Average Calls Per Method

| | $SH_S$ | $SH_L$ | $MV$ |
|---|---|---|---|
| # Projects | 200 | 610 | 868 |
| # Packages | 253 | 714 | 340 |
| # Classes | 4,285 | 91,060 | 23,207 |
| # Methods | 4,530 | 191,532 | 32,987 |
| # APIs | 5,351 | 30,576 | 22,054 |
| # Calls | 27,312 | 1,027,644 | 343,010 |
| # Call-Avg | 6 | 5 | 10 |

### A. Dataset Description

To evaluate the effectiveness of MEGA, we utilize three publicly available benchmark datasets: $SH_S$, $SH_L$, and $MV$:

- $\mathbf{SH_L}$ contains 610 java projects, filtered from 5,147 randomly downloaded java projects retrieved from GitHub via the Software Heritage archive [23].
- $\mathbf{SH_S}$ is comprised of 200 java projects with small file sizes extracted from $SH_L$. It is designed to evaluate some time-consuming baselines such as PAM [11].
- $\mathbf{MV}$ consists of 868 JAR archives collected from the Maven Central repository. There are 3,600 JAR archives in the original dataset, and 1,600 JAR archives remain after being deduplicated by the previous work [12], [14]. While through our manual inspection, we find that the cleaned dataset still contains highly similar projects. For example, some projects have snapshot versions during the development process and a release version at the end, such as *commons-1.0.2.RELEASE.jar* and *commons-1.0.2.BUILD-SNAPSHOT.jar*. Besides, some projects may have their renamed versions, such as *eclipse.equinox.common-3.6.200.jar* and *common-3.6.200.jar*. In these cases, the two projects are nearly identical. Too many similar projects in a dataset may introduce bias in evaluation [12]. Therefore, we decided further clean this dataset by removing duplicated project versions, i.e., the projects with snapshot versions or renamed versions. We finally obtain 868 JAR archives from 3,600 JAR archives for the $MV$ dataset.

From the source code in datasets, we extract the method declarations and corresponding API calls, and hierarchical structure of the projects and packages containing methods/APIs. We summarize the detailed statistics of the three datasets in Table II.

### B. Baselines

When selecting baselines, we consider their inputs, whether their code is released and the datasets they used. First, our approach recommends APIs according to the previously-called APIs in the method thus it is a code-based API recommendation approach. Therefore, some API recommendation approaches based on query are not compared, such as DeepAPI [24], GeAPI [19], BRAID [25], etc. To fairly compare our approach with baselines, we only choose the baselines with open-sourced code, so we exclude approaches such as Codekernel [18]. Besides, our approach requires nothing but the source code of

software projects, so we do not include the baselines requiring extra information (e.g., code changes) such as APIREC [26]. Finally, PAM [11], FOCUS [12] and GAPI [14] are selected as baselines. For all the baselines including PAM, FOCUS and GAPI, we reuse their original implementations instead of re-implementing from scratch.

- *PAM* [11] is a statistical approach to mine API usage patterns, which mainly adopts a probabilistic model to mine API usage patterns for every target project collected from the GitHub Java corpus. When conducting API recommendation, it searches mined API usage patterns based on some APIs called in the evaluated method and returns an API list.
- *FOCUS* [12] leverages collaborative filtering technique to recommend APIs via a context-based rating matrix. It uses the same API calls of the evaluated method as input, calculates the similarity to other methods and projects, and then scores all APIs in similar projects. Finally, it outputs a ranked list of APIs with scores in descending order.
- *GAPI* [14] is a state-of-the-art graph-based collaborative filtering technique. It employs a graph neural network to exploit the high-order connectivity on an integrated graph of client methods and target APIs from source code repositories. In the recommendation stage, it inputs some APIs of the evaluated method into the trained model, calculates the similarity between client methods and all APIs, and outputs a ranked list of APIs with similarity scores in descending order.

### C. Evaluation Metrics

Following previous approaches [12], [14] on API usage recommendation, we adopt $successRate@K$, $Precision@K$ and $Recall@K$ to evaluate the quality of top-K API usage recommendation. Given a top-K ranked recommendation list $REC_k(m)$ for a test method $m \in \mathcal{M}$ and the ground-truth set $GT(m)$, we adopt $MATCH_k(m) = REC_k(m) \cap GT(m)$ to present the correctly predicted API set. The $SuccessRate@K(SR@K)$, $Precision@K(P@K)$, and $Recall@K(R@K)$ are defined as follows:

- $SuccessRate@K$ is the proportion of at least one successful match among the top-K APIs.

$$SR@K = \frac{count_{m \in \mathcal{M}}(MATCH_k(m) > 0)}{|\mathcal{M}|} \quad (5)$$

- $Precision@K$ is the proportion of correctly predicted APIs amongst the top-K APIs.

$$P@K = \frac{|MATCH_k(m)|}{k} \quad (6)$$

- $Recall@K$ is the proportion of correctly predicted APIs amongst the ground-truth APIs.

$$R@K = \frac{|MATCH_k(m)|}{|GT(m)|} \quad (7)$$

### D. Implementation Details

Following the evaluation methodology of previous works [12], [14], we consider the developer is working at a project $p$, and the methods of $p$ are split as training methods and testing methods, which are added into training set and testing set, respectively. After splitting all projects, we obtain 190, 600 and 838 methods for the testing sets of $SH_S$, $SH_L$ and $MV$ (removing some methods with fewer than 4 API calls), respectively. For each method $m$ in the test set, we keep the first four API calls as visible context and the rest invocations are taken as the ground truth $GT(m)$.

We implement MEGA in PyTorch. The embedding size is set to 64 for the model in MEGA. We employ the binary cross-entropy loss as the loss function. To initialize the model parameters, we utilize the default Xavier initializer [27]. Also, we choose Adam optimizer [28] to train our model, with a learning rate of 0.002, a coefficient of $L2$ normalization of $10^{-5}$, a batch size of 1024 and an epoch number equal to 40 fixed for all datasets.

Following previous work [29], we set the maximum distance of adjacent APIs $\epsilon$ as 3 in constructing $\mathcal{G}_C$. Considering that the edge attribute in $\mathcal{G}_C$ is a continuous variable, we adopt the equidistant bucket discretization method, and regard the bucket number as the edge type. The optimal number of buckets $|\mathcal{T}|$ in discretization, the max hop number $L$ and the size of triple set $|\mathcal{S}_u^l|$ in each hop $l$ on three datasets are determined based on the experimental performance. The best settings of the hyper-parameters for all the baseline approaches are defined following the original papers. All approaches are trained on NVIDIA Tesla V100 GPU.

## V. Results

### A. Effectiveness of MEGA Compared With Baselines (RQ1)

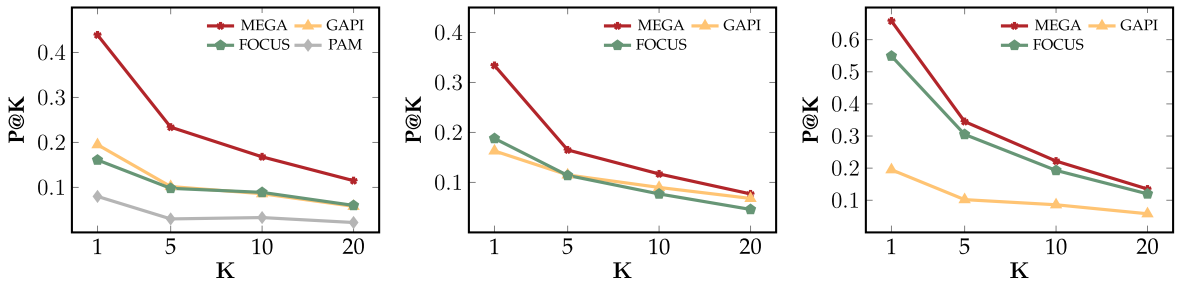
Table III presents overall results of all baselines along with MEGA in terms of $SR@K$ metric, and the comparison curves of $P@K$ and $R@K$ on three datasets (with $K = 1, 5, 10, 20$) are shown in Fig. 8 and Fig. 9, respectively. Intuitively, MEGA consistently achieves the best performance on all datasets. Note that, we only test PAM on $SH_S$ due to its long execution time and scaling poorly in a large dataset. Detailed observations are as follows:

*Comparison of $SR@K$ on a Single Dataset.* Without loss of generality, we take the $SH_S$ dataset as an example to illustrate the comparison here, and similar trends can also be observed on other datasets. In the $SH_S$ dataset, the results show that MEGA greatly outperforms GAPI and FOCUS in terms of various K settings of SR@K, with an improvement of 125.1% in SR@1, 85.12% in SR@5 and 67.76% in SR@10. This suggests that the APIs recommended by MEGA match better with the developer's context. Besides, from Table III we can see that SR@K of MEGA increases to 0.794 when K increases to 10. This means that in most cases, MEGA can identify the correct API in the Top-10 results, while other baseline models can only identify about 47% of correct APIs in the Top-10 results.

*Comparison of $SR@1$ on Multiple Datasets.* To evaluate the performance of MEGA among multiple datasets compared with baseline models, we choose the $SR@1$ metric as it considers both whether a correct API can be included and whether a correct API can get a higher rank. Overall, in terms of $SR@1$, MEGA

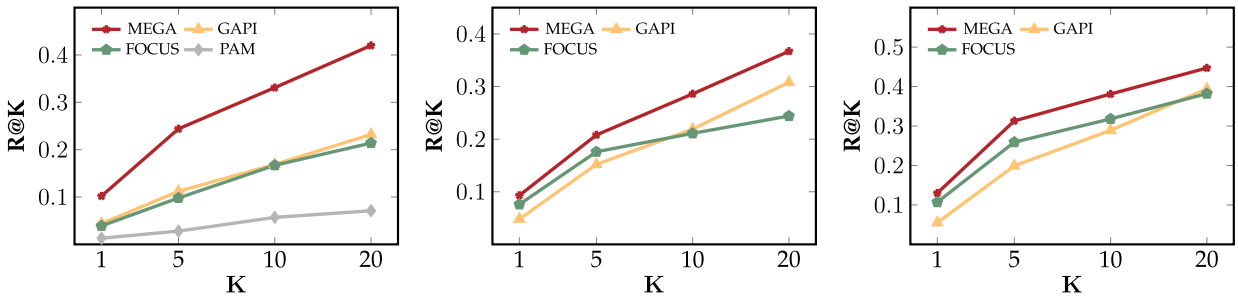
TABLE III
THE PERFORMANCE COMPARISON OF $SR@K$ BETWEEN MEGA AND THE BASELINES ON THREE DATASETS

| Method | $SH_S$ | | | | $SH_L$ | | | | $MV$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *SR@1* | *SR@5* | *SR@10* | *SR@20* | *SR@1* | *SR@5* | *SR@10* | *SR@20* | *SR@1* | *SR@5* | *SR@10* | *SR@20* |
| PAM | 0.080 | 0.150 | 0.275 | 0.335 | - | - | - | - | - | - | - | - |
| FOCUS | 0.161 | 0.256 | 0.328 | 0.422 | 0.188 | 0.292 | 0.349 | 0.388 | 0.549 | 0.709 | 0.769 | 0.819 |
| GAPI | 0.195 | 0.363 | 0.479 | 0.600 | 0.163 | 0.402 | 0.532 | 0.670 | 0.260 | 0.569 | 0.714 | 0.837 |
| MEGA | **0.439** | **0.672** | **0.794** | **0.836** | **0.334** | **0.544** | **0.641** | **0.731** | **0.658** | **0.810** | **0.840** | **0.875** |




(a) $SH_S$ (b) $SH_L$ (c) $MV_S$

Fig. 8. The performance comparison of $P@K$ between MEGA and the baselines on three datasets.




(a) $SH_S$ (b) $SH_L$ (c) $MV_S$

Fig. 9. The performance comparison of $R@K$ between MEGA and the baselines on three datasets.

improves 125.1% and 77.65% compared to the best baseline GAPI on $SH_S$ and $SH_L$, and 19.85% compared to the best baseline FOCUS on $MV$. This demonstrates that MEGA is more effective on multiple datasets than baselines. We use Wilcoxon signed-rank test [30] to verify whether the performance gain is significant, and Cliff'd Delta (or $d$) [31] to measure the effect size. The significance test result ($p$-value$<$0.01) and large effect size on the metrics ($d$=0.479) of MEGA and the best baseline GAPI indicate that the proposed model significantly improves the recommendation performance. We notice that GAPI underperforms FOCUS on $MV$. One possible explanation is that the average number of API calls per method of $MV$ is the largest. The abundant direct call interactions between methods and APIs are enough for FOCUS to identify similar methods and recommend relevant APIs, while GAPI uses more complicated connectivity which introduces more noise into the representation of methods and APIs instead, leading to a negative effect.

Figs. 8 and 9 give detailed precision and recall curves on different datasets by varying k from 1 to 20. As can be seen, MEGA shows the best performance among all the approaches. Specifically, MEGA increases the performance of the baseline by an average of 54.98% and 49.36% over the three datasets with respect to precision and recall, respectively. There is usually a trade-off between precision and recall. As the number of recommended APIs (i.e., k) increases, the recall increases as more correct APIs are found, while the precision decreases as more irrelevant APIs are involved. For example, MEGA achieves the top-1 precision of 0.439 and recall of 0.102 and the top-5 precision of 0.234 and recall of 0.244 (as shown in Figs. 8 and 9).

The above experimental results show that pattern-based methods, such as PAM, relying on mining frequent subsequences generally perform worse than learning-based methods such as GAPI and MEGA. This indicates the significance of exploring high-order connections and making full use of external information.

> *Answer to RQ1:* MEGA is quite effective on API usage recommendation, outperforming the state-of-the-art approaches in terms of Success Rate, Precision and Recall, respectively. Besides, MEGA achieves consistent performance on all datasets.

TABLE IV
THE PERFORMANCE COMPARISON OF $SR@K$ BETWEEN MEGA AND ITS VARIANTS ON THREE DATASETS

| Method | $SH_S$ | | | | $SH_L$ | | | | $MV$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *SR@1* | *SR@5* | *SR@10* | *SR@20* | *SR@1* | *SR@5* | *SR@10* | *SR@20* | *SR@1* | *SR@5* | *SR@10* | *SR@20* |
| MEGA w/o H&C | 0.333 | 0.566 | 0.688 | 0.751 | 0.142 | 0.291 | 0.413 | 0.558 | 0.521 | 0.694 | 0.756 | 0.802 |
| MEGA w/o HS | 0.349 | 0.614 | 0.704 | 0.773 | 0.167 | 0.316 | 0.429 | 0.578 | 0.533 | 0.704 | 0.784 | 0.842 |
| MEGA w/o CO | 0.423 | 0.651 | 0.757 | 0.815 | 0.311 | 0.521 | 0.629 | 0.728 | 0.614 | 0.746 | 0.780 | 0.816 |
| MEGA | **0.439** | **0.672** | **0.794** | **0.836** | **0.334** | **0.544** | **0.641** | **0.731** | **0.658** | **0.810** | **0.840** | **0.875** |

### B. Ablation Study (RQ2)

To investigate the effectiveness of each component of the graph representation model in MEGA, we perform ablation studies by considering the following three variants.

- $MEGA_{\mathrm{w/oHS}}$: This variant removes the hierarchical structure encoding module from the model to study the effect of external information derived from the project and package.
- $MEGA_{\mathrm{w/oCO}}$: This variant deletes the co-occurrence information encoding module from the model to investigate the impact of global information obtained between APIs.
- $MEGA_{\mathrm{w/oH\&C}}$: This variant only preserves the call interaction encoding in the model to gain the primary representations of the method and the API, without any supplementary information.

The experimental results are shown in Table IV. We find that the performance of MEGA drops in three variants compared with the complete model, which demonstrates the effectiveness of the hierarchical structure encoding module and co-occurrence information encoding module.

$\mathrm{MEGA}_{\mathrm{w/o\ H\&C}}$ performs worst since the variant only utilizes historical call information. Moreover, we notice that the performance degradation is most significant on $SH_L$. For example, $SR@1$ decreases from 0.311 to 0.142. Note that, in $SH_L$, the average number of call interactions is 5, which is the smallest among all datasets. This demonstrates the prominent advantage of appending co-occurrence information and structure information to the final representation when the interactions are insufficient.

In addition, the performance of $\mathrm{MEGA}_{\mathrm{w/o\ HS}}$ is better than $\mathrm{MEGA}_{\mathrm{w/o\ CO}}$, meaning that hierarchical structure information is more critical than co-occurrence information. One possible reason is that the external project/package structure can provide more contextual information instead of just internal call relation, which is more beneficial for capturing the semantic match between methods and APIs.

*Answer to RQ2:* Encoding both API co-occurrence information and project/package hierarchical structure information into the final representations of methods and APIs is beneficial to MEGA's performance improvements. Especially, project/package hierarchical structure information is more critical, benefiting from it contains contextual information of methods and APIs.

TABLE V
THE PERFORMANCE COMPARISON OVER LOW-FREQUENCY APIS ON THREE DATASETS

| | *SR@1* | *SR@10* | *P@1* | *P@10* | *R@1* | *R@10* |
|---|---|---|---|---|---|---|
| | | | $SH_S$ | | | |
| FOCUS | 0.017 | 0.156 | 0.017 | 0.016 | 0.009 | 0.043 |
| GAPI | 0.020 | 0.081 | 0.020 | 0.013 | 0.007 | 0.034 |
| MEGA | **0.081** | **0.263** | **0.081** | **0.046** | **0.029** | **0.176** |
| | | | $SH_L$ | | | |
| FOCUS | 0.003 | 0.004 | 0.003 | 0.004 | 0.002 | 0.015 |
| GAPI | 0.040 | 0.079 | 0.040 | 0.012 | 0.020 | 0.047 |
| MEGA | **0.081** | **0.315** | **0.081** | **0.040** | **0.057** | **0.236** |
| | | | $MV$ | | | |
| FOCUS | 0.002 | 0.003 | 0.003 | 0.002 | 0.001 | 0.007 |
| GAPI | 0.009 | 0.018 | 0.009 | 0.003 | 0.006 | 0.016 |
| MEGA | **0.014** | **0.041** | **0.014** | **0.008** | **0.006** | **0.029** |

### C. Performance of MEGA on Low-Frequency APIs(RQ3)

As stated in Section II, we design MEGA to alleviate the problem that current approaches on low-frequency APIs. We define APIs that are called by methods fewer than or equal to three times as low-frequency APIs, as introduced in Section II. According to the definition, there are 4,046, 20,171 and 16,274 low-frequency APIs in $SH_S$, $SH_L$ and $MV$, respectively. To evaluate the recommendation effectiveness on low-frequency APIs, we validate MEGA and baselines on methods which contain low-frequency APIs in the ground truth. In this experiment, the metrics $SR@K$, $P@K$, and $R@K$ are calculated based on the correctly predicted low-frequency APIs. To further evaluate the performance of MEGA on low-frequency APIs, we design a metric $HitRate@K$ ($HR@K$), which is the proportion of correctly predicted low-frequency APIs amongst all predicted low-frequency APIs. A higher $HitRate@K$ indicates that the model can recommend more correct low-frequency APIs.

Table V shows $SR@K$, $P@K$ and $R@K$ ($K = \{1, 10\}$) for all approaches on three datasets. To sum up, that MEGA greatly outperforms other approaches in all metrics. In detail, $SR@10$ is improved by 298.7%, $P@10$ is improved by 233.3%, and $R@10$ is improved by 402.1% compared to the latest approach GAPI on $SH_L$. Looking into the performance of baselines, GAPI obtains better performance than FOCUS, indicating the effectiveness of incorporating complicated connectivity information in enriching the representation of APIs with fewer direct interactions. To further evaluate the performance of MEGA on low-frequency APIs, we adopt a metric $HitRate$ ($HR@K$), which measures

TABLE VI
THE PERFORMANCE COMPARISON OF HR@K OVER LOW-FREQUENCY APIS ON THREE DATASETS

| | $SH_S$ | | $SH_L$ | | $MV$ | |
|---|---|---|---|---|---|---|
| | HR@1 | HR@5 | HR@1 | HR@5 | HR@1 | HR@5 |
| FOCUS | 0.157 | 0.496 | 0.103 | 0.432 | 0.300 | 0.400 |
| GAPI | 0.112 | 0.351 | 0.171 | 0.515 | 0.243 | 0.350 |
| MEGA | **0.247** | **0.623** | **0.279** | **0.754** | **0.467** | **0.605** |

TABLE VII
TIME COST FOR MODEL TRAINING AND PREDICTION OF MEGA AND THE BASELINE APPROACH ON $MV$

| Approach | Training Time | Prediction Time |
|---|---|---|
| FOCUS | 1 min | 31 ms / client method |
| GAPI | 21.6 h | 0.111 ms / client method |
| MEGA | 1.2 h | 0.368 ms / client method |

the hit rate when the model has recommended low-frequency APIs. Following Table VI, MEGA significantly outperform the baselines with an average increase of 41.09% in terms of HR@5. These results demonstrate that MEGA can capture more useful usage patterns of low-frequency APIs and recommend accurate APIs. Although MEGA presents a significant improvement on the recommendation performance, the results of low-frequency APIs are still quite limited, which may be attributed to the functional particularity of the APIs and needs more future research.

*Answer to RQ3:* MEGA consistently outperforms the state-of-the-art baselines for recommending low-frequency APIs on the three benchmark datasets. Despite the superior performance of MEGA, low-frequency recommendation is still challenging and needs more future research.

### D. Parameter Sensitivity Study (RQ4)

We conduct experiments to analyze the impact of following hyper-parameters with different settings on MEGA's performance.

*Impact of max hop Number L.* We vary the number of hops in propagating to observe the performance change of MEGA. Fig. 11(a) depicts the results in terms of $SR@10$. We observe that MEGA achieves the best results with one hop on three datasets, and the performance gradually decreases with the hop number increases.

One possible explanation is that, in the graph, short-distance nodes have a strong correlation with the original node, while the relevance decays as the distance increases. Consequently, the positive impact of short-distance propagation is greater, while long-distance propagation brings more noise than useful signals.

*Impact of Bucket Number $|\mathcal{T}|$.* To study the impact of bucket numbers, we conduct different experiments by setting different bucket numbers. The experimental results in terms of $SR@10$ are presented in Fig. 11(b), which shows that for $SH_S$, $SH_L$, and $MV$, the best performance is achieved when the number of buckets is 15, 10, and 15, respectively.

One possible reason for this phenomenon is that when the number of buckets is too small, i.e., few relation types, the graph contains less information, which compromises the trained model's expressiveness. While a large number of buckets, i.e., many relation types, makes the information in the graph too rich, leading to over-fitting the model.

*Impact of Triple set Size $|\mathcal{S}_u^l|$ in Each hop $l$.* We change the number of neighbors selected by the client method and the target API in each hop to explore the effects of triple set size on MEGA's performance. The results of $SR@10$ on the $SH_S$, $SH_L$ and $MV$ are demonstrated in Fig. 10(a), (b), and (c), respectively.

Jointly analyzing the three sub-figures, when the size increases, the results get better first and then worse. This means that when the size is moderately large, the benefits of more information included improves the performance. However, when the size is extremely large, the noise introduced outweighs the useful information introduced and thus it can hurt the performance. Overall, 16 or 32 is the suitable size of triple set in each hop for both the method and the API on three datasets.

### E. Timing Efficiency

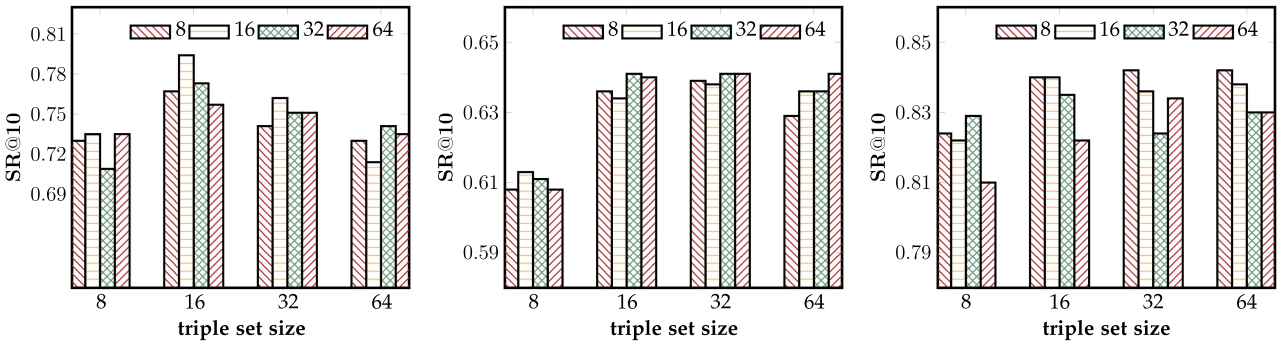
We conduct a timing efficiency experiment of MEGA on the largest $MV$ dataset. In this experiment, we record how long it takes to train the model and how long it takes to make predictions. The detailed time cost of three approaches is shown in Table VII.

*Training Time.* As reported by GAPI's authors [14], their approach takes 21.6 hours of model training, since their approach is based on a graph convolutional network, which is computationally expensive during training. The training time cost of FOCUS [12] is 1 minute since it only needs to construct a 3D context-based rating matrix. As shown in Table VII, MEGA takes 1.2 hours to train, which is relatively slow, and almost the whole time cost is due to training two attentive networks. The training phase is offline, which indicates that MEGA can scale well on larger datasets.

*Prediction Time.* Since MEGA needs to calculate the call probabilities of all candidate APIs for every client method in the prediction phase, the major computation cost is dependent on the number of candidate APIs and the client methods in the test set. For the prediction time, FOCUS is the slowest (31 milliseconds) to provide recommendation, while GAPI is the fastest (0.111 milliseconds). MEGA (0.368 milliseconds) is slower than GAPI. Since FOCUS has been used in the Eclipse IDE by developers, MEGA could also be suitable in a real development environment.

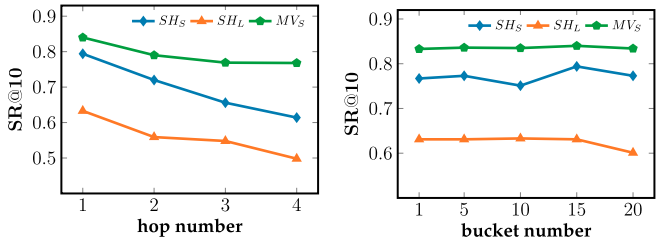
## VI. USER STUDY

In this section, we conduct a user study to further validate the effectiveness of the proposed MEGA from a developer perspective. The user study is conducted through online questionnaire.




(a) $SH_S$ (b) $SH_L$ (c) $MV_S$

Fig. 10. The results of $SR@10$ on three datasets along with different sizes of triple set.




(a) The effect of hop numbers (b) The effect of bucket numbers

Fig. 11. The parameter sensitivity study of hop and bucket numbers.

### A. Study Design

*Questionnaire.* The online questionnaire includes 25 code snippets randomly selected from $SH_S$ testing dataset. In the questionnaire, each question comprises a code snippet, three API lists recommended by FOCUS, GAPI and MEGA, respectively. The quality of the recommended API lists is evaluated from two aspects, including "relevancy" and "preference". Specifically, the metric "relevancy" measures the extent of function relevance between the code snippet and API list; And the metric "preference" implies the extent of helpfulness of API lists recommend for developers. The "relevancy" is scored based on 1-5 Likert scale (1 for *complete irrelevancy* and 5 for *full relevancy*). The "preference" is evaluated on 1-3 rank (1 for the *most preferred* and 3 for the *least preferred*).

*Participants.* We invite 6 developers with an average Java development experience of 4.3 years. The statistic indicates that these participants are familiar with Java development. Each participant needs to read 25 code snippets and evaluates the quality of the API lists recommended by FOCUS, GAPI and MEGA. They are not aware of which API list is recommended by which model. Each participant is paid 22 USD upon completing the questionnaire.

### B. Result Analysis

We finally received 150 sets of scores totally from the user study. Each set contains scores regarding "relevancy" and "preference," for the recommended API lists of FOCUS, GAPI and MEGA, respectively. Fig. 12 depicts the results of user study. For the metric "relevancy," the higher scores the better; while for the metric "preference," the lower scores the better. As can be seen, the API lists recommended by MEGA received the best scores among all three API lists in terms of "relevancy" and "preference," respectively.

*Relevancy.* We compute the distribution and proportion of average relevancy scores of the API lists recommended by FOCUS, GAPI and MEGA, shown in Fig. 12(a) and (c), respectively. As illustrated in Fig. 12(a), the relevancy scores for API lists recommended by MEGA are concentrated in the higher area of boxplot, while the scores of FOCUS and GAPI are rather close, both in the lower area. Combined with Fig. 12(c), we can observe that nearly 80% participants gave the API lists recommended by FOCUS and GAPI "neutral/irrelevancy" ratings (lower than 4 points), and the number of "relevancy" rating (4/5 points) for the API lists recommended by MEGA is 2.5 times than those for the API lists of FOCUS and GAPI. These results imply that the API lists recommended by MEGA tend to be more relevant to the current code snippet.

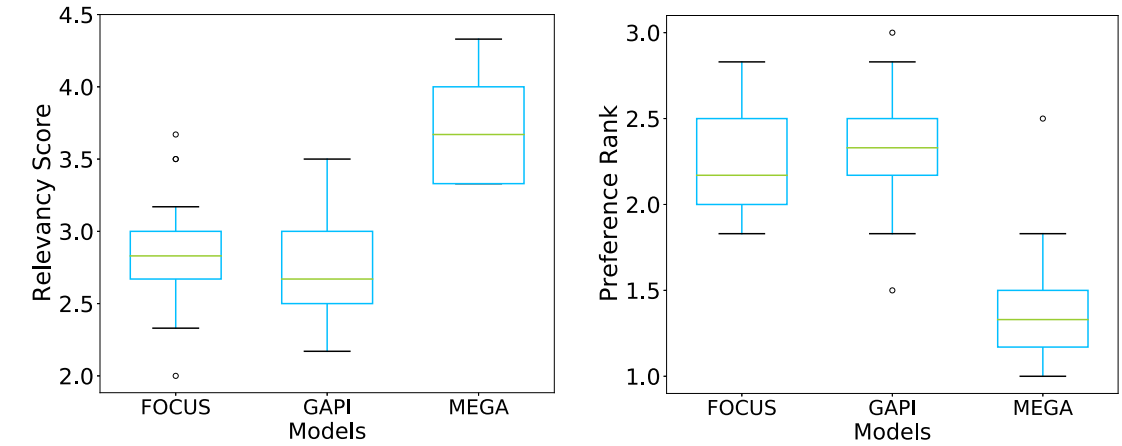
*Preference.* Figs. 12(b) and (d) illustrate the distribution and proportion of average preference ranks, respectively. We discover that the majority of participants rank the API lists recommended by MEGA as the most favored (69%). Also, the API lists recommended by MEGA present a much lower preference rank than other API lists on average, i.e., 1.43 versus 2.32/2.24 (as shown in Fig. 12(b)). This indicates that the participants think the API list recommended by MEGA is more favorable.

The user study further validates the effectiveness of the proposed MEGA for API recommendation."

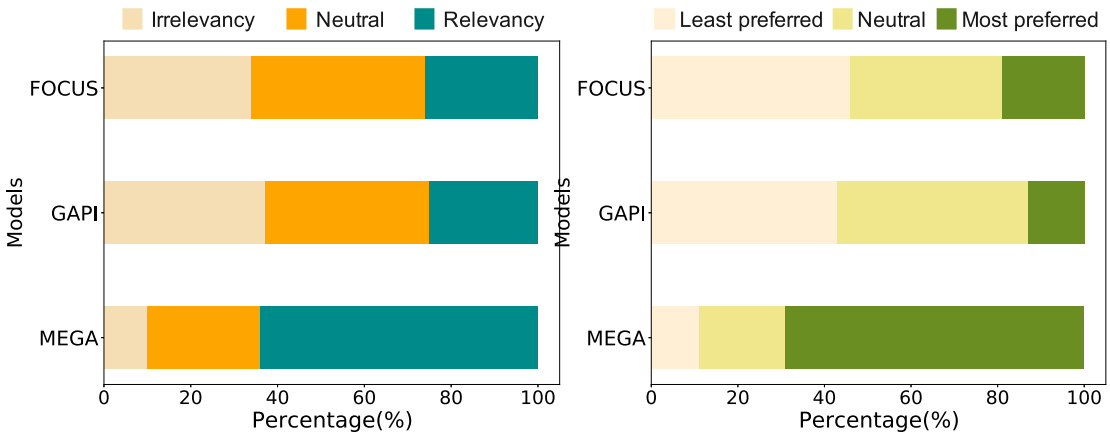
## VII. Discussion

### A. Why Does MEGA Work or Fail?

To explore how the three graphs help the recommendation process, we select a few samples from testing data of $SH_S$ for a case study. Fig. 13 depicts two successful API recommendation cases to demonstrate why MEGA works. Due to the space limitation, we only show the called APIs of client methods, instead of their complete code snippets. The first client method "*getJarName()*" is employed as an example in Section II. As illustrated in the call interaction subgraph, the interaction path "*getJarName()* $\rightarrow$ *getValue()* $\rightarrow$ *getProjectVersion()* $\rightarrow$ *createUnmarshaller()/unmarshal()*" indicates that




(a) Distribution of relevancy scores. (b) Distribution of preference ranks.




(c) Proportion of relevancy scores. (d) Proportion of preference ranks.

Fig. 12. Results of user study.

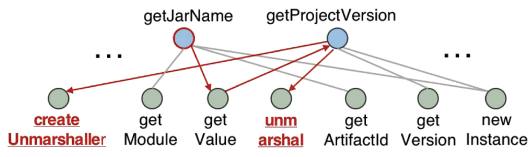
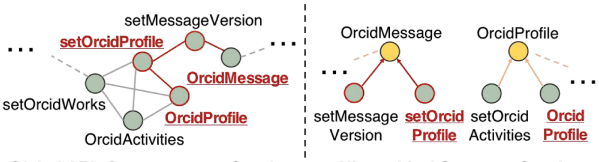
| Client Method | Subgraph | Model | |
|---|---|---|---|
| **SettingUtil/getJarName():**<br>Modules.getModule()<br>JAXBContext.newInstance()<br>Model.getArtifactId()<br>JAXBElement.getValue() |  **Call Interaction Graph** | Ground Truth | JAXBContext.createUnmarshaller()<br>FileInputStream.FileInputStream()<br>Unmarshaller.unmarshal() |
| | | FOCUS | BufferedReader.BufferedReader()<br>BufferedReader.close()<br>BufferedReader.readLine() |
| | | GAPI | StreamResult.StreamResult()<br>Date.getTime()<br>Class.forName() |
| | | **MEGA** | **JAXBContext.createUnmarshaller()**<br>Model.getVersion()<br>**Unmarshaller.unmarshal()** |
| **UidObtainer/getOrcidWorks():**<br>**OrcidMessage.setMessageVersion()**<br>OrcidProfile.setOrcidActivities()<br>OrcidActivities.setOrcidWorks()<br>OrcidActivities.OrcidActivities() |  **Global API Co-occurrence Graph** **Hierarchical Structure Graph** | Ground Truth | OrcidProfile.OrcidProfile()<br>OrcidMessage.setOrcidProfile()<br>OrcidMessage.OrcidMessage() |
| | | FOCUS | PersonDAO.getPersonByUnId()<br>PersonDAOImpl.PersonDAOImpl()<br>Person.getGivenName() |
| | | GAPI | Model.addAttribute()<br>MessageEncoder.encodePassword()<br>String.split() |
| | | **MEGA** | **OrcidProfile.OrcidProfile()**<br>**OrcidMessage.setOrcidProfile()**<br>**OrcidMessage.OrcidMessage()** |

Fig. 13. Case study for successful recommendation. The left column presents the client method. The middle column presents relevant subgraphs of the client method. The right column denotes the ground-truth APIs and the top-3 recommended results of different models.

*createUnmarshaller()/unmarshal()* may be needed after the invocation of *getValue()*. MEGA captures this connection between APIs, and successfully recommends relevant APIs.

For the second client method *getOrcidWorks()*, we can discover that both *setOrcidProfile()* and *OrcidMessage()* are called together with *setMessageVersion()* from the global API co-occurrence subgraph, which may be potential APIs. Besides, the project structure also carries useful information. As shown in the hierarchical structure subgraph, both *setMessageVersion()* and *setOrcidProfile()* belong to the same class *OrcidMessage()* in the project, so *getOrcidWorks()* can also be related to *setOrcidProfile()* by the structure path of "*getOrcidWorks()* $\rightarrow$ *setMessageVersion()* $\rightarrow$ *OrcidMessage()* $\rightarrow$ *setOrcidProfile()*". From the perspective of co-occurrence and structure relationships, MEGA finally recommends *OrcidProfile.OrcidProfile()*, *OrcidMessage.setOrcidProfile()* and *OrcidMessage.OrcidMessage()*.Through the analysis of the two successful cases, the three graphs reflect the project content from different perspectives. By leveraging the abundant relationships in multi-view graphs, MEGA can discover potential APIs for client methods.

We also randomly analyze some bad cases. For example, for a client method *onShortLink()* which has called APIs *HttpServletRequest.getHeader()*, *PrintStream.println()*, *Browser.getName()* and *UserAgent.getBrowser()*, MEGA fails to recommend related APIs. The reason could be that *Browser.getName()* and *UserAgent.getBrowser()* are only called in this method. Our model is incapable of exploring useful usage patterns for these APIs from other methods and makes an inaccurate recommendation based on the common APIs *HttpServletRequest.getHeader()* and *PrintStream.println()*. In the future, we will attempt to incorporate more external knowledge (e.g., API documentation) or conduct data augmentation (e.g., generating contrastive samples) to alleviate the problem of low-frequency APIs, improving the recommendation performance.

### B. Usefulness of MEGA

We believe that MEGA is useful and the performance is not marginal. (1) MEGA greatly outperforms the best baseline methods in terms of the overall performance. For example, MEGA greatly outperforms the best baseline with an average improvement of 44.90% in SuccessRate@5, 61.99% in Precision@5 and 52.27% in Recall@5 over the three datasets. (2) MEGA is helpful in recommending low-frequency APIs. Specifically, MEGA presents a significant improvement on the recommendation performance (e.g., 298.7% in SuccessRate@10, 233.3% in Precision@10, and 402.1% in Recall@10), with marginal increase of time cost (e.g., the prediction time of per client method increases around 0.25 ms). The finding that the results of low-frequency APIs are not as high as those for other APIs can inspire future relevant research. We also provide some research directions such as incorporating more external knowledge (e.g., API documentation) or conducting data augmentation (e.g., generating contrastive samples) to alleviate the problem of low-frequency APIs. (3) The user study further verifies the effectiveness of MEGA. For example, 69% of the participants think that the API lists recommended by MEGA are relevant and correct for current programming. Among the 25 client methods, 10 of them have low-frequency APIs in the ground-truth sets, and the recommendation results are preferred by 76% of the participants.

### C. Implications

In this section, we discuss the implications that would be helpful for software researchers and software developers.

*Software Researchers.* In Section V, we achieve that the heterogeneous information in source code is greatly beneficial for improving the recommendation performance of APIs including the low-frequency APIs. However, we also find that the results of low-frequency APIs are still quite limited, presenting a large gap with those of common APIs. The limited results may be attributed to the functional particularity of the low-frequency APIs, and could impact the practical usage of current API recommendation tools. Thus, we suggest researchers working on API recommendation to focus more on the recommendation of low-frequency APIs by combining external knowledge such as API documentation or exploring data augmentation techniques.

*Software Developers.* We retrieve the low-frequency APIs of three benchmark datasets in the java API documentation [4]. The result shows that 3,027 of the 4,046 low-frequency APIs in $SH_S$, 18,481 of the 20,171 low-frequency APIs in $SH_L$ and 13,130 of the 16,247 low-frequency APIs in $MV$ do not have corresponding API documentation. Thus, we achieve that low-frequency APIs are usually not associated with the API documentation. API documentation which contains usage samples and instructions is helpful for learning the representations of APIs [32], [33]. Thus, we encourage developers to write some descriptions or usage examples for facilitating the API recommendation task.

### D. Threats to Validity

*Internal Validity.* In this paper, following [20], [21] we sample a fixed-size of neighbors on graphs instead of using a full size triple sets for the trade off of computation overhead. This may slightly influence the performance of MEGA. To alleviate the impact of this threat, we conduct each experiment five times and obtain average performance as shown in Section V. Furthermore, our experiments on parameter sensitivity also demonstrates that different sizes of triple set influence the performance of MEGA slightly.

*External Validity.* We evaluate MEGA under Java datasets, while MEGA may show different performance on datasets in other programming languages. To reduce the impact from different programming languages, when designing three multi-view graphs, we try to exclude the language-specific information and only maintain the structure information such as call relationships and definition relationships. We believe our design can be easily adapted to most programming languages.

## VIII. Related Work

In this section, we review existing work about API usage recommendation. The work on API usage recommendation

can be divided into two categories: pattern-based methods and learning-based methods. Pattern-based methods utilize traditional statistical methods to capture usage patterns from API co-occurrences. Learning-based methods leverage deep learning models to automatically learn the potential usage patterns from a large code corpus and then use them to recommend patterns.

*Pattern-Based Methods.* Zhong et al. propose MAPO [9] to cluster and mine API usage patterns from open source repositories, and then recommends the relevant usage patterns to developers. Wang et al. improve MAPO and build UP-Miner [10] by utilizing a new algorithm based on $SeqSim$ to cluster the API sequences. Nguyen et al. propose APIREC [26], which uses fine-grained code changes and the corresponding changing contexts to recommend APIs. Fowkes et al. propose PAM [11] to tackle the problem that the recommended API lists are large and hard to understand. PAM mines API usage patterns through an almost parameter-free probabilistic algorithm and uses them to recommend APIs. Liu et al. propose RecRank [34] to improve the top-1 accuracy based on API usage paths. Nguyen et al. propose FOCUS [12], which mines open-source repositories and analyzes API usages in similar projects to recommend APIs and API usage patterns based on context-aware collaborative-filtering techniques. Some graph-based techniques, such as [35], GRAPAC [36], and MuDetect/AUGs [37] parse each code snippet individually to build API-Uage graphs (AUGs). For APIs, these graphs contain only the call interactions between them and methods. However, MEGA also constructs API co-occurrence relationships in different methods and package/project structure relationships that comprehensively represent the contexts surrounding APIs.

*Learning-Based Methods.* Nguyen et al. propose a graph-based language model GraLan [17] to recommend API usages. Gu et al. propose DeepAPI [24]. They reformulate API recommendation task as a query-API translation problem and use an RNN Encoder-Decoder model to recommend API sequences. Ling et al. propose GeAPI [19]. GeAPI automatically constructs API graphs based on source code and leverages graph embedding techniques for API representation. Gu et al. propose Codekernel [18] by representing code as object usage graphs and clustering them to recommend API usage examples. Zhou et al. build a tool named BRAID [25] to leverage learning-to-rank and active learning techniques to boost recommendation performance. Previous learning-based methods fail to recommend usage patterns for low-frequency APIs due to the data-driven feature, MEGA encodes API frequency with global API co-occurrence graph to alleviate this problem.

## IX. Conclusion

In this paper, we propose a novel approach MEGA for automatic API usage recommendation. MEGA employs heterogeneous graphs, which are constructed from multiple views, i.e., method-API interaction from local view, API-API co-occurrence from global view, and project structure from external view. A graph representation model with a frequency-aware attentive network and a structure-aware attentive network is then proposed to learn the matching scores between methods and APIs based on the multi-view graphs. Experiment demonstrates MEGA's effectiveness both on overall API usage recommendation and low-frequency API usage recommendation. For future work, in addition to the information extracted from projects, some information from API official documentation or Q&A sites also contributes to mining API usage patterns. Therefore, we plan to design some new modules that encode more information from different sources.

## Acknowledgments

We would like to thank all the anonymous reviewers for their insightful comments.

## References

[1] Hou and X. Yao, "Exploring the intent behind API evolution: A case study," in *Proc. IEEE 18th Work. Conf. Reverse Eng.*, Limerick, Ireland, M.D. PinzgerPoshyvanyk and J. Buckley, Eds., 2011, pp. 131–140.
[2] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus, "Characterizing the usage, evolution and impact of Java annotations in practice," *IEEE Trans. Softw. Eng.*, vol. 47, no. 5, pp. 969–986, May 2021.
[3] I. Gvero, "Core Java volume I: Fundamentals, 9th edition by Cay S Horstmann and Gary Cornell," *ACM SIGSOFT Softw. Eng. Notes*, vol. 38, no. 3, p. 33, 2013.
[4] Oracle, "JDK 18 documentation," 2022. [Online]. Available: https://docs.oracle.com/en/java/javase/18/docs/api/index.html
[5] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Softw.*, vol. 26, no. 6, pp. 27–34, Nov./Dec. 2009.
[6] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *Proc. IEEE 28th Int. Conf. Softw. Maintenance*, Trento, Italy, Sep. 23–28, 2012, pp. 25–34.
[7] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Dubrovnik, Croatia, I. Crnkovic and A. Bertolino, Eds., Sep. 3–7, 2007, pp. 25–34.
[8] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *Proc. IEEE 34th Int. Conf. Softw. Eng.*, Zurich, Switzerland, M. Glinz, G. C. Murphy, and M. Pezzè, Eds., Jun. 2–9, 2012, pp. 782–792.
[9] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proc. Object-Oriented Program. 23rd Eur. Conf.*, ser. Lecture Notes in Computer Science, vol. 5653, S. Drossopoulou, Eds., Springer, Genoa, Italy, Jul. 6–10, 2009, pp. 318–343.
[10] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proc. IEEE 10th Work. Conf. Mining Softw. Repositories*, San Francisco, CA, USA, T. Zimmermann, M. D. Penta, and S. Kim, Eds., May 18–19, 2013, pp. 319–328.
[11] M. Fowkes and C. Sutton, "Parameter-free probabilistic API mining across github," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Seattle, WA, USA, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds., Nov. 13–18, 2016, pp. 254–265.
[12] P. T. Nguyen, J. D. Rocco, D. D. Ruscio, L. Ochoa, T. Degueule, and M. D. Penta, "FOCUS: A recommender system for mining API function calls and usage patterns," in *Proc. IEEE 41st Int. Conf. Softw. Eng.*, Montreal, QC, Canada, J. M. Atlee, T. Bultan, and J. Whittle, Eds., May 25–31, 2019, pp. 1050–1060.
[13] B. M. Sarwar, G. Karypis, J. A. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proc. 10th Int. World Wide Web Conf.*, Hong Kong, China, V. Y. Shen, N. Saito, M. R. Lyu, and M. E. Zurko, Eds., May 1–5, 2001, pp. 285–295.
[14] C. Ling, Y. Zou, and B. Xie, "Graph neural network based collaborative filtering for API usage recommendation," in *Proc. IEEE 28th Int. Conf. Softw. Anal. Evol. Reengineering*, Honolulu, HI, USA, Mar. 9–12, 2021, pp. 36–47.
[15] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2009.
[16] Y. Peng et al., "Revisiting, benchmarking and exploring API recommendation: How far are we?," *IEEE Trans. Softw. Eng.*, pp. 1–21, 2022.

[17] T. Nguyen and T.N. Nguyen, "Graph-based statistical language model for code," in *Proc. IEEE 37th Int. Conf. Softw. Eng.*, vol. 1, Florence, Italy, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds., May 16–24, 2015, pp. 858–868.

[18] X. Gu, H. Zhang, and S. Kim, "Codekernel: A graph kernel based approach to the selection of API usage examples," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, San Diego, CA, USA, Nov. 11–15, 2019, pp. 590–601.

[19] C. Ling, Y. Zou, Z. Lin, and B. Xie, "Graph embedding based API graph search and recommendation," *J. Comput. Sci. Technol.*, vol. 34, no. 5, pp. 993–1006, 2019.

[20] F. Zhang et al., "RippleNet: Propagating user preferences on the knowledge graph for recommender systems," in *Proc. 27th ACM Int. Conf. Inf. Knowl. Manage.*, Torino, Italy, A. Cuzzocrea, J. Allan, N. W. Paton, D. Srivastava, R. Agrawal, A. Z. Broder, M. J. Zaki, K. S. Candan, A. Labrinidis, A. Schuster, and H. Wang, Eds., Oct. 22–26, 2018, pp. 417–426.

[21] Z. Wang, G. Lin, H. Tan, Q. Chen, and X. Liu, "CKAN: Collaborative knowledge-aware attentive network for recommender systems," in *Proc. 43rd Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, China, J. Huang, Y. Chang, X.J. Cheng, V. Kamps, J. Murdock Wen, and Y. Liu, Eds., Jul. 25–30, 2020, pp. 219–228.

[22] A. F. Agarap, "Deep learning using rectified linear units (RELU)," 2018, *arXiv:1803.08375*.

[23] R. D. Cosmo and S. Zacchiroli, "Software heritage: Why and how to preserve software source code," in *Proc. 14th Int. Conf. Digit. Preservation*, Kyoto, Japan, S.S. HaraSugimoto and M. Goto, Eds., Sep. 25–29, 2017.

[24] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Seattle, WA, USA, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds., Nov. 13–18, 2016, pp. 631–642.

[25] Y. Zhou, H. Jin, X. Yang, T. Chen, K. Narasimhan, and H. C. Gall, "BRAID: An API recommender supporting implicit user feedback," in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Athens, Greece, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds., Aug. 23–28, 2021, pp. 1510–1514.

[26] A. T. Nguyen et al., "API code recommendation using statistical learning from fine-grained changes," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Seattle, WA, USA, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds., Nov. 13–18, 2016, pp. 511–522.

[27] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. 13th Int. Conf. Artif. Intell. Statist.*, ser. JMLR Proceedings, vol. 9, Chia Laguna Resort, Sardinia, Italy, Y.W. Teh and D.M. Titterington, Eds., May 13–15, 2010, pp. 249–256.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations*, San Diego, CA, USA, Y. Bengio and Y. LeCun, Eds., May 7–9, 2015.

[29] Z. Wang, W. Wei, G. Cong, X.X. LiMao, and M. Qiu, "Global context enhanced graph neural networks for session-based recommendation," in *Proc. 43rd Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, Virtual Event, China, J.Y. Huang, X. ChangCheng, J. Kamps, V. Murdock, J. Wen, and Y. Liu, Eds., 2020, pp. 169–178.

[30] S. E. Ahmed, "Effect sizes for research: A broad application approach," *Technometrics*, vol. 48, no. 4, p. 573, 2006.

[31] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics*. Berlin, Germany: Springer, 1992, pp. 196–202.

[32] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of API methods from feature requests," in *Proc. IEEE/ACM 28th Int. Conf. Automated Softw. Eng.*, Silicon Valley, CA, USA, E. Denney, T. Bultan, and A. Zeller, Eds., Nov. 11–15, 2013, pp. 290–300.

[33] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-API knowledge gap," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, Montpellier, France, M. Huchard, C. Kästner, and G. Fraser, Eds., Sep. 3–7, 2018, pp. 293–304.

[34] X. Liu, L. Huang, and V. Ng, "Effective API recommendation without historical software repositories," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, Montpellier, France, M. Huchard, C. Kästner, and G. Fraser, Eds., Sep. 3–7, 2018, pp. 282–292.

[35] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Amsterdam, The Netherlands, H. van Vliet and V. Issarny, Eds., Aug. 24–28, 2009, pp. 383–392.

[36] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "GraPacc: A graph-based pattern-oriented, context-sensitive code completion tool," in *Proc. IEEE 34th Int. Conf. Softw. Eng.*, Zurich, Switzerland, M. Glinz, G. C. Murphy, and M. Pezzè, Eds., Jun. 2–9, 2012, pp. 1407–1410.

[37] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static API-misuse detection," in *Proc. IEEE 16th Int. Conf. Mining Softw. Repositories*, Montreal, Canada, M. D. Storey, B. Adams, and S. Haiduc, Eds., May 26–27, 2019, pp. 265–275.