


Article

RESTful API Analysis, Recommendation, and Client Code Retrieval

Shang-Pin Ma , Ming-Jen Hsu, Hsiao-Jung Chen and Chuan-Jie Lin

Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung 202301, Taiwan

* Correspondence: albert@ntou.edu.tw

Abstract: Numerous companies create innovative software systems using Web APIs (Application Programming Interfaces). API search engines and API directory services, such as ProgrammableWeb, Rapid API Hub, APIs.guru, and API Harmony, have been developed to facilitate the utilization of various APIs. Unfortunately, most API systems provide only superficial support, with no assistance in obtaining relevant APIs or examples of code usage. To better realize the “FAIR” (Findability, Accessibility, Interoperability, and Reusability) features for the usage of Web APIs, in this study, we developed an API inspection system (referred to as API Prober) to provide a new API directory service with multiple supplemental functionalities. To facilitate the findability and accessibility of APIs, API Prober transforms OAS (OpenAPI Specifications) into a graph structure and automatically annotates the semantic concepts using LDA (Latent Dirichlet Allocation) and WordNet. To enhance interoperability, API Prober also classifies APIs by clustering OAS documents and recommends alternative services to be substituted or merged with the target service. Finally, to support reusability, API Prober makes it possible to retrieve examples of API utilization code in Java by parsing source code in GitHub. The experimental results demonstrate the effectiveness of the API Prober in recommending relevant services and providing usage examples based on real-world client code. This research contributes to providing viable methods to appropriately analyze and cluster Web APIs, and recommend APIs and client code examples.

Keywords: OpenAPI Specification; Latent Dirichlet Allocation; cluster analysis; service recommendation; code example; GitHub



Citation: Ma, S.-P.; Hsu, M.-J.; Chen, H.-J.; Lin, C.-J. RESTful API Analysis, Recommendation, and Client Code Retrieval. *Electronics* **2023**, *12*, 1252. <https://doi.org/10.3390/electronics12051252>

Academic Editor: Ricardo Santos

Received: 10 January 2023

Revised: 24 February 2023

Accepted: 2 March 2023

Published: 5 March 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

An increasing number of companies, such as Google, Facebook, Microsoft, and Netflix, are promoting the creation of innovative software systems using APIs (Application Programming Interfaces). The most common approach to creating APIs is the representational state transfer (REST) architecture [1] and corresponding RESTful services [2]. Meanwhile, applications of electronics, such as consumer electronics, robotics, medical applications, and automobiles, have become a significant part of our lives. Most electronic applications also need to integrate Web APIs to build their software. Hence, effectively utilizing APIs is becoming an essential factor in building modern electronic applications. At present, more than 20,000 REST-style services have been published in the ProgrammableWeb API directory system [3]; however, dealing with the high complexity and sheer number of APIs can be overwhelming for most users. Several API directory services and API search engines, such as Rapid API Hub [4], APIs.guru [5], and API Harmony [6], have been established to deal with this issue; however, they provide only the basic API information, popular APIs, and basic categories, without sufficient information for relevant services or appropriate usage examples.

This paper presents a novel API inspection system based on OpenAPI specifications (OAS) [7], the most widely used API description language. The proposed scheme, referred to as API Prober, allows users to assess the characteristics of RESTful services, find relevant

services, and view representative examples of code. Since the “FAIR” principles (principles of Findability, Accessibility, Interoperability, and Reusability) are widely accepted for the management of scientific data [8], the API Prober is also designed to realize findability, accessibility, interoperability, and reusability for Web APIs (note that the terms “Web API”, “Web service”, “RESTful service”, and “REST-style service” have the same meaning in this paper). The operational concepts are shown in Figure 1. For findability and accessibility, API Prober structurally analyzes OAS documents for Web APIs, and semantically annotates the elements of each API (described in Section 3.1) to facilitate the retrieval and filtering of services that meet the user requirement, based on common Web service design practices [9]. For interoperability, it classifies APIs into multiple clusters based on the annotated information (described in Section 3.2) and recommends services based on specific input/output parameters (described in Section 3.3). For reusability, it finds Java examples of service client code on GitHub using an AST (abstract syntax tree) and following a set of matching rules (described in Section 3.4). In addition, the OAS documents in API Prober are collected from APIs.guru, an API directory system with a large number of REST-style services.

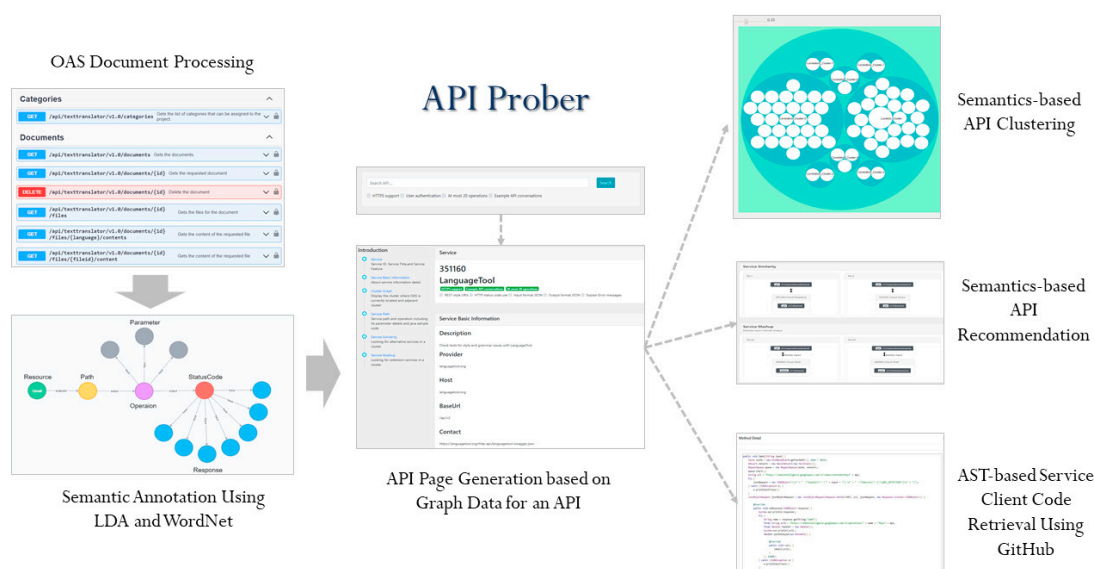


Figure 1. API Prober: operational concepts.

The organization of this paper is as follows. Section 2 provides background information on the existing API directory systems. Section 3 outlines the design of the API Prober and its core methods. The results of experiments conducted to assess the proposed approach are presented in Section 4. Section 5 offers a brief overview of the paper and indicates future research.

2. Background and Related Work

OpenAPI specifications (OAS, originally called Swagger) are guidelines that help users discover RESTful Web services and understand their capabilities. It has prompted the emergence of extended open-source projects aimed at automating the generation of back-end services, test programs, and front-end graphical interfaces. The API Prober is based on OAS.

Graph databases are a type of NoSQL database used for the management of vast sets of structured, semi-structured, or unstructured data. Neo4j [10] is an open-source graph database written in Java. It enables the development of graph-powered systems to leverage the rich interconnectivity of data using graphs. Because we need to collect and analyze many linkages between APIs and between the elements of an API, API Prober uses Neo4j to store data for the structure of OAS documents for Web APIs.

Java Parser [11] is a lightweight tool for analyzing, converting, and generating java source code. It allows users to create an abstract syntax tree (AST) for the source code of interest to facilitate analysis of the internal architecture at each code level. API Prober retrieves ASTs to find code fragments representative of each service, which are then returned to the user.

Cluster analysis [12,13] is a common approach to grouping objects to facilitate statistical analysis. Clustering is used in various fields, such as information retrieval, machine learning, image analysis, and biometric analysis. Note that search engines also use clustering to classify documents to streamline the retrieval of documents. There are several clustering methods proposed to be applied in the field of Web services. Rahman et al. [14] designed a two-level topic model for clustering Mashup services and recommending APIs based on matrix factorization. Fletcher [15] proposed a method that considers user-personalized preferences to make personalized web API recommendations by regularizing matrix factorization. Zou et al. [16] devised a service clustering framework by integrating a deep neural network and considering service composability relationships. All the above methods effectively cluster and recommend services; however, they require data of mashups or service invocations, which are difficult to acquire in typical situations. Hence, we plan to design a clustering scheme for Web APIs without the data of mashups or service utilizations, to facilitate the identification of relevant services further.

There are a lot of API directory services to allow users to search for and understand Web services. APIs.io [17] is an experimental API website using APIs.json to describe API operations and find APIs on the internet via query results. APIs.guru [3] lists APIs based on OAS with basic and detailed information. Mashape [18] collects and displays APIs using a market-oriented approach similar to that seen on Google Play and the App store. Mashape also provides a fixed code example for each API and allows online testing. Mashape has been integrated with RapidAPI [4] to provide advanced functionality, such as API category, popular APIs, and API recommendations based on QoS (quality of service). ProgrammableWeb [3] lists more than 20,000 RESTful APIs with basic API information and links to external web pages with more detailed information. Client code examples are presented in code templates or related articles. Note that new or non-mainstream APIs seldom include the resources users require, and ProgrammableWeb requires that users manually indicate the type of service they seek. API Harmony [6] provides detailed API content, analyzes the features of RESTful services, and retrieves code examples (on GitHub) and articles (on Stack Overflow) for Web APIs.

We compared and analyzed the above-mentioned related service matching approaches along with four dimensions: REST features, service clustering, client code discovery, and service recommendation. The results are shown in Table 1 (\times indicates “not supported”, \circ expresses “fully supported”, and \blacktriangle means “partially supported”). Unlike other systems, API Prober can extract essential features related to REST and common service design practices, find examples of client code, cluster APIs according to type, and recommend relevant services to facilitate the implementation of Web APIs.

Table 1. Comparison of similar systems.

	REST Features	Service Clustering	Client Code Discovery	Service Recommendation
APIs.io	\blacktriangle	\times	\times	\times
APIs.guru	\times	\times	\times	\times
Mashape/Rapid API	\times	\blacktriangle	\blacktriangle	\circ (QoS-based)
ProgrammableWeb	\times	\times	\blacktriangle	\times
API Harmony	\blacktriangle	\times	\blacktriangle	\times
API Prober	\circ	\circ	\circ	\circ (interface-based)

3. Approach Descriptions

As shown in Figure 2, the proposed API Prober system's architecture is divided into Runtime and Analyzer. The Analyzer block deals with the collection and analysis of data. A Service Feature Analyzer module first analyzes the features of OAS documents retrieved from APIs.guru. As recommended in [9], the proposed scheme identifies nine service features common to RESTful service design to assist in identifying well-designed services and figuring out how to use them. Descriptions of the services and the analyzed features are then converted into a set of nodes and a set of relationships in the graph database. The OAS Clusterer module performs clustering based on OAS. The Service Recommender module collects services that are relevant to the target service. The API Usage Explorer module crawls GitHub to find code examples as a reference for the assembly of new applications or services. The API Prober Runtime block deals with the front-end user interface (UI). Upon accessing the service content page, the Service Web Controller module translates the user's requests into instructions for the Service Manager. These commands encompass a variety of tasks, such as service searches, service filtering via annotated tags, the presentation of service information and clusters, service recommendations, and the provision of sample client code for services.

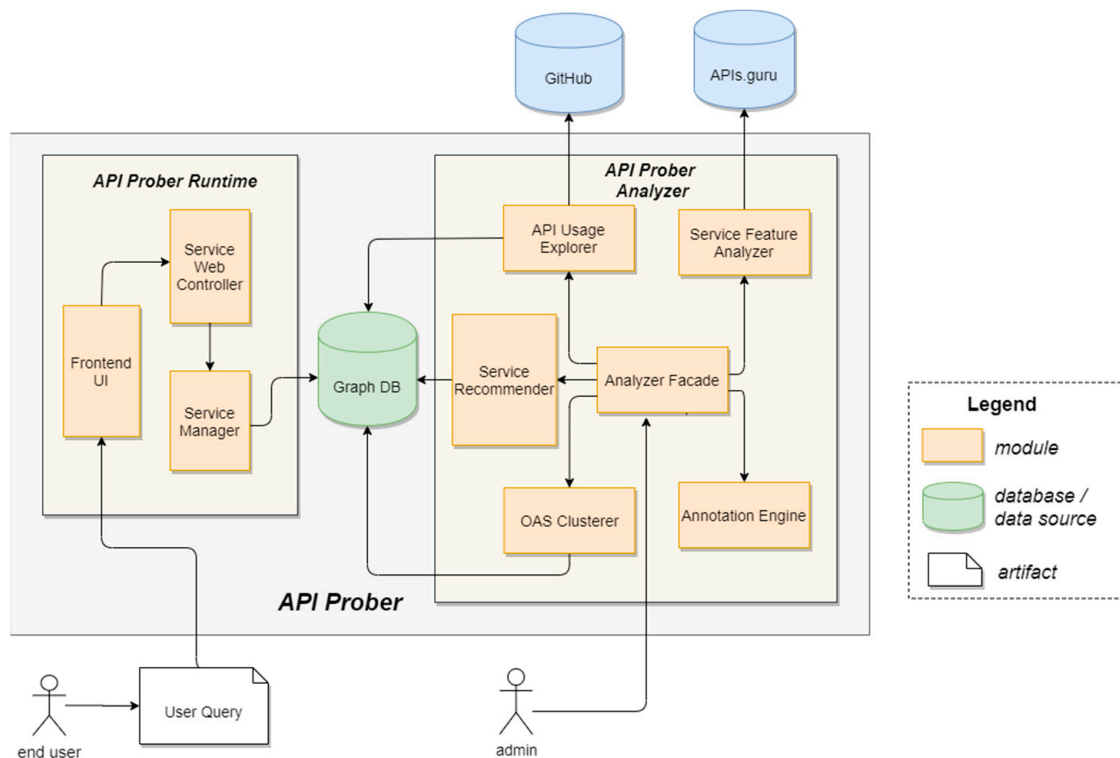


Figure 2. API Prober: system architecture.

3.1. Transforming OAS into Nodes

Based on our previous research results [19,20], we divided the OAS into six parts to ease the processing of service analyzing, clustering, recommendation, and client code retrieval. The following nodes are listed in the Neo4J graph database, including resource nodes, path nodes, operation nodes, parameter nodes, status codes, and response nodes. We then established relationships among the nodes; for example, resource nodes link to one or more path nodes representing multiple service paths, and path nodes link to one or more operation nodes indicating the operations associated with that service path. The information is retrieved in accordance with the methods described in [20], including (1) performing tokenization and stemming, (2) extracting topic words from OAS documents using the LDA (Latent Dirichlet Allocation), and (3) conducting term expansion based on WordNet

to add expanded terms whose similarity (based on the Edge Counting Method [21]) to an original term is larger than 0.9. Finally, the original words analyzed in Step (2) and the expanded words analyzed in Step (3) are assigned to their corresponding nodes. All the above information is stored in the graph database and used by the functionality of API Prober.

Based on the stored nodes for an API, API Prober can integrate all related information into an API page (an example is shown in Figure 3) to allow users to check the features of an API. We believe the page can facilitate the accessibility of Web APIs.

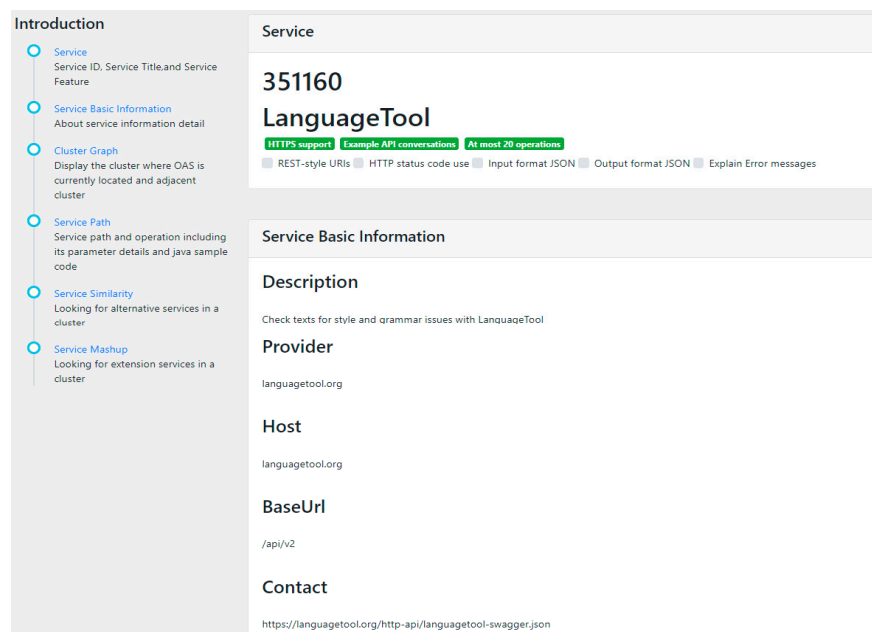


Figure 3. API page in API Prober: an example of a “Language Tool” API.

In addition, as mentioned, the REST resource-based design style has become the de facto way for most Web services [9,22]. REST uses a resource identifier to determine the specific resources involved in a service interaction. In API Prober, we sought to identify multiple RESTful service features and practices commonly used in Web service design. This makes it more likely that the users will be able to retrieve services with the desired features and helps to avoid the inappropriate use of services. Thus, we identified the nine REST Web service designs commonly used by developers to help users detect and filter services: HTTPS support, User authentication, Maximum of 20 operations, REST-style URIs, HTTP status code use, Explain error messages, Example API conversations, Output JSON (JavaScript Object Notation), and Input format JSON. Please refer to [20] to check the details.

3.2. Web API Clustering

As mentioned in [23], grouping the services effectively improve web service discovery. Obtaining usable service recommendations requires that services undergo pre-classification to facilitate filtering out unrelated services and ease of browsing. There are three main tracks for service clustering: vector space representation methods, model-based clustering, and extraction of semantically relevant words [23]. We devised a hybrid way that integrates the above three methods to divide OAS documents into multiple clusters. OAS clustering in API Prober involves the following tasks: (1) calculating similarity scores to compare OAS documents; (2) determining parameters for clustering; and (3) evaluating clustering performance to facilitate optimization.

3.2.1. Document Concept Score (DCS)

The Document Concept Score (DCS) is used to calculate the degree of similarity between the target OAS file and a candidate OAS file, based on the resource node and operation node of the target OAS. DCS calculation is based on the vector space model (VSM). API Prober converts the original text pertaining to resources and operations into V_{ow} , and the set of WordNet-extended words into V_{ww} , and aggregates the above sets as $Resource = \{V_{ow}, V_{ww}\}$ and $Operation = \{V_{ow}, V_{ww}\}$. Note that only the original text is considered for the target OAS, whereas both the original and the extended text are considered for the candidate OAS. The VSM's term count model is employed to compute the resource and operation scores, as follows:

$$sim^{RS}(T, C) = sim(V_{ow}^T, V_{ow}^C) + W_{WN} * sim(V_{ow}^T, V_{ww}^C) \quad (1)$$

where T and C , respectively, refer to the target and candidate OAS files; $sim^{RS}(T, C)$ indicates the VSM score for resources; and W_{WN} indicates the weight of WordNet. The settings for W_{WN} are discussed later. Note that $sim^{OP}(T, C)$ is also calculated using Equation (1).

The VSM scores for resources and operations are then used to obtain the final DCS, as follows (Equation (2)):

$$DCS = W_{RS} * sim^{RS}(T, C) + W_{OP} * sim^{OP}(T, C) \quad (2)$$

where W_{RS} is the weight of $sim^{RS}(T, C)$ in DCS and W_{OP} is the weight of $sim^{OP}(T, C)$ in DCS. W_{RS} and W_{OP} range from 0 to 1, and their sum is 1.

Notably, the primary query mechanism of API prober is also realized based on the DCS score. The query string can be treated as the resource's name for a target API without specifying other details. Matched APIs are retrieved and sorted according to the calculated scores between the target API and all candidate APIs. The query functionality could better realize the findability for Web APIs.

3.2.2. Parameter Settings for Clustering

We sought to determine the optimal combination of the four parameters with an overriding effect on the clustering results:

- The number of LDA topics: API Prober uses the words of the retrieved LDA topics for the annotation of OAS documents;
- Weights of resources and operations for DCS: API Prober uses the resource and operations portions of OAS files to calculate the DCS score (Equation (2)). In general, the resource part describes the overall purpose of the service, and the operations part specifies the functionality of a service endpoint;
- Weight of *WordNetScore*: Using WordNet, API Prober generates additional terms similar to the topic words produced by LDA. Word expansion can significantly increase the probability of matching services and enhance matching precision. In API Prober, W_{WN} in Equation (1) ranges from 0 to 1;
- Clustering method applied: Finally, it is necessary to select a clustering method from among complete linkage, average linkage, centroid linkage, and weighted linkage. Note that single linkage [24] is not included due to the fact that it produces unreasonably large clusters rather than size-balanced clusters.

Clustering results are used to assess the clustering performance of various parameter combinations. The evaluation methods and results are discussed in the following subsection.

3.2.3. Evaluation Methods and Parameter Settings for Service Clustering

Clustering evaluation usually calculates the similarities between clusters or the similarities within clusters. In this research, we adopted the Adjusted Rand Index (ARI) [25], a widely used external evaluation method, to assess the degree of similarity between

two clustering results. A higher RI value (ranging from 0 to 1) indicates a high degree of similarity between the expected and obtained clustering results.

To determine appropriate parameters, we assessed four clustering methods in terms of ARI scores: complete linkage, average linkage, weighted linkage, and centroid linkage. Average linkage yielded the highest ARI results and was thus selected for parameter optimization: Clustering Threshold, 0.8; LDA Topic, 4; Resource Weight, 0.6; Operation Weight, 0.4; and WordNet Weight, 0.9. Please check [20] for the evaluation details.

3.3. API Recommendations Based on Identified Clusters

API Prober can also recommend APIs for a target service based on matching input and output parameters. Note that the recommendations are based on the results of service clustering, i.e., recommended services belong to the cluster to which the target service belongs. API recommendations in API Prober can be divided into recommendations for substitutable services and mergeable services:

- Substitutable services have input and output parameters similar to those of the target service. Substitutable services can be used as an alternative in situations where the target service fails to satisfy user requirements pertaining to quality or functionality. For example, a user could build a flight reservation system integrating multiple ticket ordering services from various airlines;
- Mergeable services provide “horizontal” service compositions integrating multiple services with similar inputs or outputs. Input-oriented mergeable services can be used to integrate output data from different services, based on a given input. For example, a user could build a housing recommendation system integrating information related to the proximity of restaurants, convenience stores, and schools. Output-oriented mergeable services can help users to collect multiple data items for multiple queries. For example, a user could build a music discovery system that searches for songs based on singer names, song names, composers, lyricists, and movie names.

In the following, we outline the three scoring methods used to generate service recommendations:

1. Calculation of Parameter Concept Scores (PCS):

PCS indicates the degree of similarity between two parameters, PC_t (matching target) and PC_s (matching source), based on VSM. The formal representation of the vectors of PC_t and PC_s are $PC_t = \{ V_{toc} \}$, which includes only the vector of the original concepts, and $PC_s = \{ V_{soc}, V_{swc} \}$, which includes vectors of the original concepts and expanded WordNet concepts. The calculation is performed as follows (Equation (3)),

$$PCS(PC_t, PC_s) = \frac{V_{toc} \cdot V_{soc} + V_{toc} \cdot V_{swc}}{\| V_{oc1} \| \cdot \| V_{oc2} \|} \quad (3)$$

2. Calculation of Hungarian Mapping Scores (HMS):

HMS indicate the degree of interface compatibility between two services, based on the concept of input–output covering (IOC) proposed in our previous work [26]:

- $HMS_{input}(TS, CS)$: API Prober calculates the degree to which the input parameters of the target service (TS) match the input parameters of a candidate service (CS);
- $HMS_{output}(TS, CS)$: API Prober calculates the degree to which the output parameters of a CS match the expected output parameters of the TS.

For the input, API Prober calculates PCS from each parameter of the TS to each input parameter of the CS. For the output, API Prober calculates PCS from each parameter of the CS to each output parameter of the TS.

Subsequently, API Prober utilizes the Hungarian algorithm to determine the optimal pairing by employing PCS to rate matching pairs. For inputs, API Prober divides the total rating of the best combination by the number of service input parameters to obtain

$HMS_{input}(TS, CS)$. For outputs, API Prober divides the total rating of the best combination by the number of output parameters to obtain $HMS_{output}(TS, CS)$.

3. Service Recommendation:

HMS and SRS (Equation (4)) are used to sort through candidate services and discard those deemed inappropriate. We calculated three types of scores for the recommendation of input-oriented mergeable services, output-oriented mergeable services, and substitutable services based on $HMS_{input}(TS, CS)$, $HMS_{output}(TS, CS)$, and $SRS(TS, CS)$.

$$SRS(TS, CS) = \frac{HMS_{input}(TS, CS) + HMS_{output}(TS, CS)}{2} \quad (4)$$

Figure 4 illustrates the process used to determine whether a candidate service is a substitutable or mergeable service. Briefly, services with input parameters similar to those of the target service are recommended as input-oriented mergeable services and sorted based on the HMS_{input} scores. Services with output parameters similar to those of the target service are recommended as output-oriented mergeable services and sorted based on the HMS_{output} scores. Services with input and output parameters similar to those of the target service are recommended as substitutable services and sorted based on the SRS scores. Threshold $t1$ is used to filter out non-mergeable services for input, and $t2$ is used to filter out non-mergeable services for output. In general, the service recommendation scheme in Web API can effectively facilitate the interoperability of Web APIs by suggesting possible replacements and composition for Web APIs.

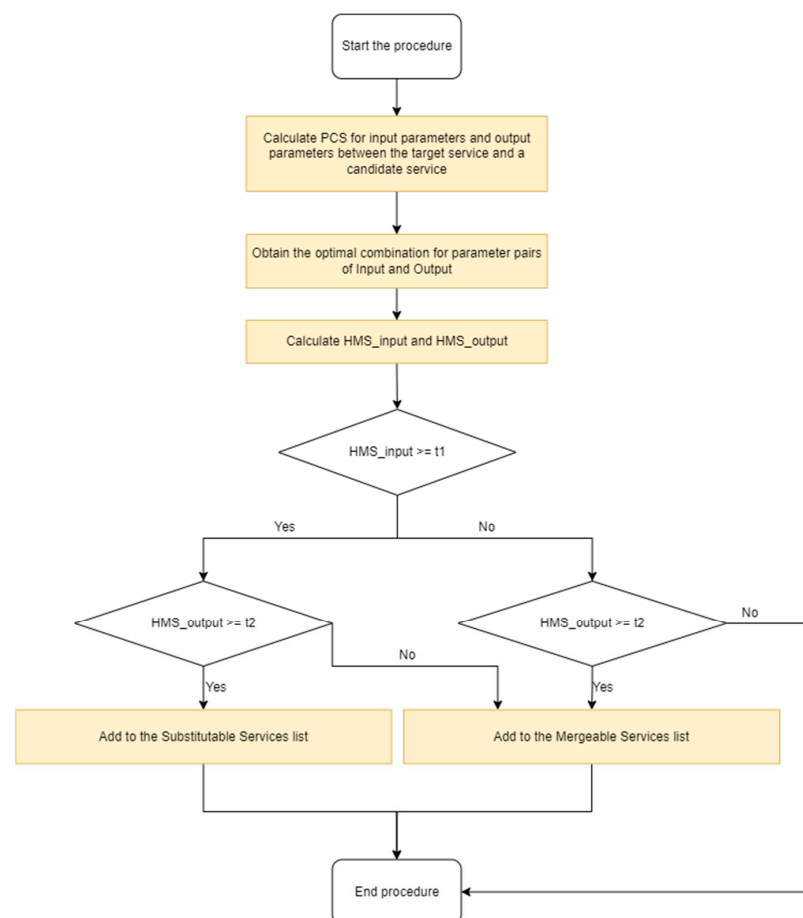


Figure 4. The process used to differentiate substitutable services from mergeable services.

3.4. Discovering Service Client Code Examples

REST API client developers may face productivity problems when lacking usage examples [27]. To address this issue, API Prober supports the discovery of API code examples to assist in the implementation of APIs. GitHub provides hosting services for software source code, which allows individuals and organizations to create and access code. It is currently the world's largest open-source code community. AUE (API Usage Explorer) in API Prober is responsible for finding relevant code through GitHub for a given RESTful service. According to [28], there are three ways to collect code from GitHub repositories: GitHub Archive, GHTorrent, and the GitHub Search API. The first two provide only archived data; therefore, we selected the GitHub Search API to collect code in this research.

At the start of the extraction process, AUE obtains the scheme, host, and basePath from a resource node as well as the path name from a path node within the graph database. These elements are then merged to produce a target service path (e.g., <https://api.github.com/emojis>, accessed on 10 January 2023), as illustrated in Table 2. AUE then uses the combined service path as a query parameter and calls the GitHub Search API to locate potentially usable service codes.

Table 2. Example of service path combination.

Node	Information
Resource	\$.info.scheme: https
	\$.info.host: api.github.com
	\$.info.basePath: /
Path	\$.paths.{path_name}:/emojis

The extraction of code is a five-step process. The first 100 search results returned by the GitHub Search API are initially retrieved. AUE uses GitHub scores to arrange the search results in descending order. Next, the target service path is tokenized to facilitate matching with the retrieved code files. Subsequently, text_matches (i.e., matching fragments of each retrieved code file) are tokenized. The service path is then compared to text_matches to identify if all of the tokens in the service path are included in the fragment, thereby filtering out unsuitable results and preventing the retrieval of erroneous examples. In the final step, qualified results are gathered and saved in the Neo4J database.

The examples collected by AUE are then re-processed using the proposed Java Method Parser (JMP) module. This involves extracting the code fragments that use the target service path.

In terms of filtering, the JMP first performs structural analysis of the code files collected using AUE by inspecting the abstract syntax tree (AST) generated by the Java Parser. JMP retrieves seven parts of the AST:

- (1) Import: libraries used in the Java file, indicating the relevant functions that may be used in the Java file;
- (2) Class: class of the Java file;
- (3) Class/instance variable: variables declared in the Java file at the class level;
- (4) Inner class: one or more internal classes are contained in Java files, and these are retrieved via a recursive search;
- (5) Method name: name of a method declared in the class;
- (6) Method body: internal representation of the method body;
- (7) Statement: a statement line in the method, which is the smallest unit in Java Parser.

JMP uses the seven parts to compare code and the target service path. Here, it is crucial to understand the service path's design and the characteristics of the Java file to ensure precision in the search for code fragments. We developed a novel approach to service path matching, which includes four sub-methods to improve search precision:

1. The service path may have a valid superset. When designing an API, the handling of resources by the service provider tends to vary according to service operations, which commonly results in service paths with a superset/subset design. For example, we may have the following similar but different operations: *http://mysearch.com/search*, *http://mysearch.com/search/group*. To reduce the probability of false searches when searching for code fragments, JMP determines whether the retrieved path is part of a longer path (i.e., a subset of the retrieved path) and keeps only the longer path;
2. APIs based on the REST style may contain resources that can be manipulated. A service based on the REST style may contain resource-based operations such as *http://mysearch.com/search/group/{groupid}*;
3. The fact that there is no way to identify the arguments used in a resource-style operation makes it difficult to match codes. In this case, we use a substitution symbol (such as <<token>>) instead. JMP analyzes the URI according to its syntax (shown in Figure 5) and uses an optional query component (question mark) as a separator during tokenization. It keeps only the first token due to the fact that the second token may contain the query string or the content to be transmitted. For example, for the service path *http://mysearch.com/search?q=wiki*, JMP keeps only *http://mysearch.com/search* for subsequent matching;

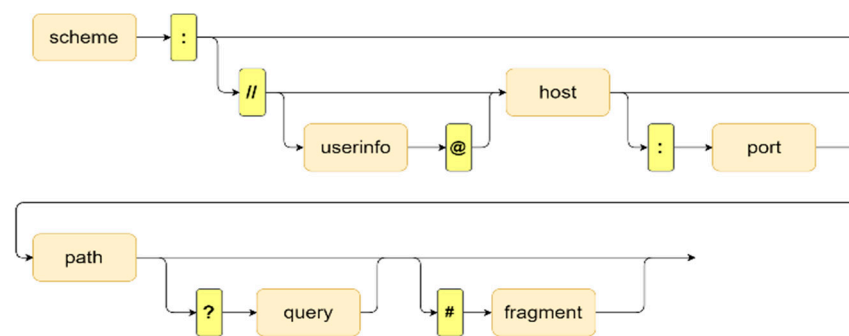


Figure 5. URL syntax diagram for extraction of code examples.

4. Java language features: JMP uses semicolons and commas as terminal symbols when matching service paths. Note that the semicolon indicates that the statement ends, whereas the comma indicates the insertion of an argument in Java.

JMP uses the above service path matching method to extract appropriate client codes, as follows.

1. Performing service path matching for different cases: there are two common situations involving using Web APIs in Java.
 - Used after a class variable or an instance variable is declared: JMP searches for class/instance variables using AST and compares them using the service path matching method to determine whether the service path is included in the variable. If the use of the service path is confirmed, then the variable's name is recorded, and the methods that use the variable are also retrieved and saved;
 - Declared directly in the method and used in the method: JMP searches for all methods using AST and compares them using service path matching to determine whether the service path is used in the method. If the use of the service path is confirmed, then the method is saved;
 - Used in the return statement: JMP also uses the service path matching method to determine whether the service path is contained in the return statement. If the use of the service path is confirmed, then the method with the return statement is also saved.

2. Scores are assigned to extracted codes using the following scoring rules. (1) If the example does not pass the previous step, then it is assigned a score of s1, (2) the scores of s2 to s5 in Table 3 are based on the following guidelines:
 - s2: Service paths are used only in the return statement and are not used in the method. This kind of example is not very helpful for users;
 - s3: service paths are used in instance/class variables or methods;
 - s4 and s5: From the Maven website, we collected widely used HTTP libraries commonly used to invoke Web APIs. If one or more libraries are used in the source code, then it is likely that it is a service client invoking Web APIs. Using the Import information in AST, JMP determines whether a code example imports the HTTP library to determine whether it is a possible service code example. The HTTP libraries supported by API Prober include Apache Commons HttpClient, Apache Httpclient, Apache Httpcore, Google Volley, Loopj Android Http, Mashape Unirest, okhttp3, and RestTemplate.

Table 3. Score evaluation mechanism.

Score	Description
s1	Original results of GitHub search API
s2	Service path used in a return statement
s3	Service paths used in class/instance variables or methods
s4	Service path used in return statement and HTTP library applied
s5	Service paths used in class variables or class methods and HTTP library applied

Finally, we establish the relationship between the top three examples and the Path node and store them in the Neo4J database. Notably, the support of retrieving service client codes can considerably facilitate the reusability of Web APIs.

4. Experimental Evaluations

This section outlines the experiments used to evaluate the efficacy of the proposed schemes for service clustering, service recommendation, and discovering examples of service code. The codebase of API Prober is available at <https://github.com/a11057002/api-prober2-front> (accessed on 10 January 2023) and <https://github.com/a11057002/api-prober2-back> (accessed on 10 January 2023).

4.1. Analysis for Service Clustering

4.1.1. Experimental Setup

As mentioned in Section 3.2.2, based on the internal experiment by applying the objective indicator, Adjusted Rand Index (ARI), the experimental results showed that average linkage yielded the highest ARI and was thus selected for parameter optimization (clustering threshold, 0.8; LDA topic, 4; resource weight, 0.6; and WordNet Weight, 0.9). Based on the identified settings, we conducted the clustering for 1439 Web APIs including 24,819 endpoints on APIs.guru. To evaluate the clustering results, we raised the first research question (RQ1): *Does the proposed API clustering approach yield reasonable API clusters?*

Accordingly, the relevance and cohesion in each large cluster (with more than ten services) were evaluated and voted on by three evaluators (they are young software engineers familiar with API integration). In other words, if two or more evaluators regarded an API cluster as appropriate (from the perspective of API usage), then the cluster was regarded as valid.

4.1.2. Experimental Results

Figure 6 presents the number of services in all of the clusters. There was a total of 441 service clusters, which included 378 small clusters with less than 5 services, 46 medium clusters with 5 to 9 services, and 17 large clusters with 10 to 90 services.

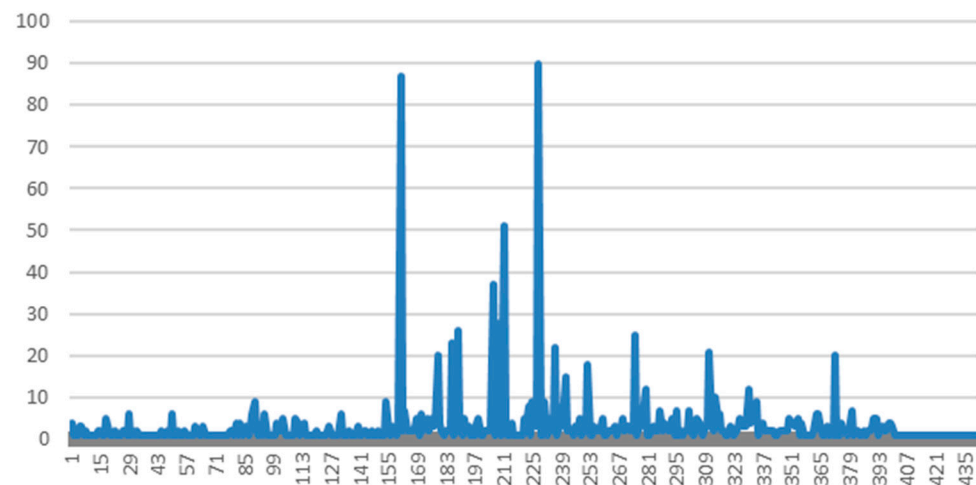


Figure 6. Number of services in all service clusters.

The evaluation results and details for the top-five clusters are shown in Table 4. Overall, the results show that the produced clusters were semantically reasonable and the APIs in the same cluster were semantically similar or relevant. In the clustering results, for example, the topic identified in the largest cluster (with 90 services) was application performance management, based on the fact that the associated services are meant to provide insight into applications, infrastructure, and big data. For RQ1, we could conclude that the proposed clustering approach can yield reasonable API clusters without relying on any linkage records for services (e.g., mashups or service invocations).

Table 4. Details for top-5 clusters.

Topic (Named Manually)	Number of Services	Representative Services	Evaluation Results
Application performance management	90	<ul style="list-style-type: none"> ApplicationInsightsManagementClient HDInsightManagementClient InfrastructureInsightsManagementClient 	Valid
Database management	87	<ul style="list-style-type: none"> Azure SQL Database SqlManagementClient MariaDBManagementClient 	Valid
Security	51	<ul style="list-style-type: none"> Security Center AttestationManagementClient Cloud Identity 	Valid
Network management	37	<ul style="list-style-type: none"> NetworkManagementClient NetworkAdminManagementClient Service Networking 	Valid
Authorization management	28	<ul style="list-style-type: none"> AuthorizationManagementClient PolicyClient Cloud User Accounts 	Valid

Although the clusters identified in this study could make it easier for users to find similar or relevant services, there are two limitations:

- API Prober cannot directly produce an appropriate name for each cluster. The cluster names could be given manually to improve the readability of the cluster graph;
- When new Web APIs are added, the proposed clustering process needs to be re-conducted; it needs a considerable processing time.

4.2. Evaluation of Recommended Mergeable and Substitutable Services

4.2.1. Experimental Setup

To evaluate the efficacy of the proposed service recommendation method, we raised the second research question (RQ2): *Does the proposed API recommendation approach locate appropriate services?*

Accordingly, we collected the recommended services and verified if the recommendation was suitable. In this experiment, the Top-K Precision indicator is commonly used to evaluate the accuracy of data search systems. We, therefore, used Top-K Precision to evaluate the efficacy of the proposed service recommendation system. The calculation method of Top-K Precision $SRP^k(TS)$ was as follows (Equation (5)):

$$SRP^k(TS) = \frac{|RS_{relevant}^k|}{k} \quad (5)$$

where $RS_{relevant}^k$ indicates the number of satisfactory services among the Top-K recommended services. Note that the designation “satisfactory” was based on voting by the three evaluators (as in the previous experiment). Similarly, if two or more evaluators regarded a recommended service as useful (from the perspective of application development), then the service was deemed satisfactory. The evaluation criteria were as follows:

- Substitutable services: both input and output parameters of the recommended service were sufficiently similar to those of the target service;
- Input-oriented mergeable services: input parameters of the recommended service were sufficiently similar to those of the target service;
- Output-oriented mergeable services: output parameters of the recommended service were sufficiently similar to those of the target service.

Note that we analyzed only the Top-1, Top-2, and Top-3 Precisions due to the difficulties in discovering substitutable and mergeable services among public APIs. In many cases, no services can be recommended or only a few are recommended for a small proportion of target services. The three OAS files selected for each type of recommendation were evaluated in terms of average SRP for the selected target services.

In addition, to the best of our knowledge, no existing service recommendation scheme considers the similarity and relevancy of input and output of Web APIs; thus, there are no comparable targets with the proposed system. Accordingly, we focused exclusively on the SRP indicator in assessing the services recommended by API Prober.

4.2.2. Experimental Results

The experiment results are shown in Figure 7. It indicates that API Prober was able to find suitable services for all three types of recommendations. The fact that all SPR indicators exceeded 0.77 means that at least two of the three recommended services were deemed useful. Note that all initially recommended services were deemed satisfactory for building service applications. For RQ2, we could conclude that the proposed recommendation approach can locate useful services for substitution or composition.

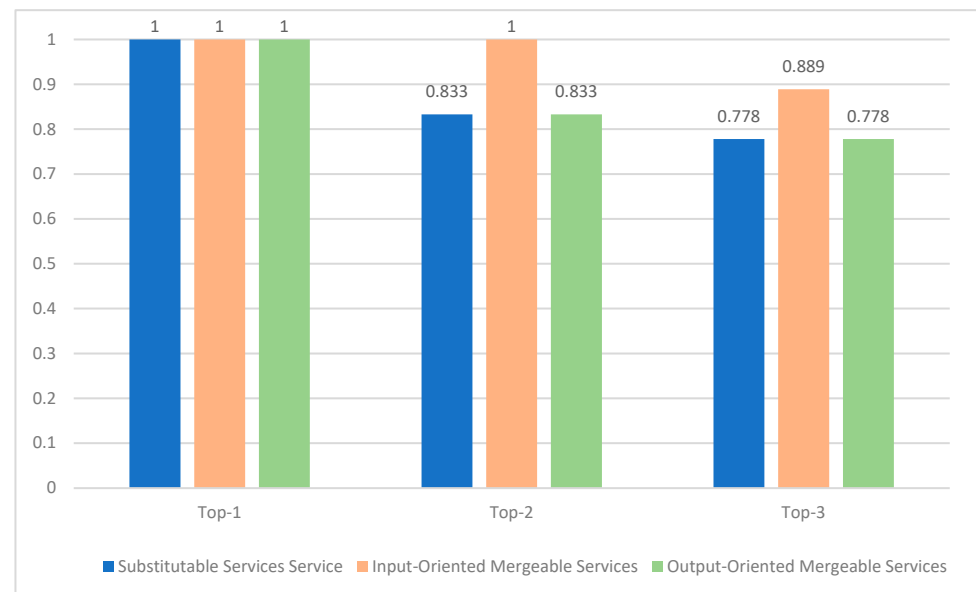


Figure 7. Top-K SRP for three types of service recommendation.

Although the API prober could effectively recommend services, there are still erroneous recommendations uncovered. The problem is due to the descriptions of some of the APIs being too short and vague to extract terms capable of representing the meaning of the input/output parameters. For example, the term “resource” was too general to express input/output parameters. If no other terms were extracted, subsequent recommendations were compromised. In short, the quality of OAS for APIs can profoundly impact the precision of the proposed service recommendation scheme.

4.3. Evaluations of Discovery of API Client Code Examples

4.3.1. Experimental Setup

Finally, to evaluate the efficacy of the proposed method in discovering API client codes, we raised the third research question (RQ3): *Does the proposed retrieval approach for service client code discover the correct code?*

For the evaluation indicator, we adjusted the Top-K method to fit the characteristics of the discovered Java code. This was necessary to account for the fact that multiple methods in the same Java file may use a given service, thereby precluding classification as entirely correct or incorrect. In this research, Code search precision (CSP) was calculated as follows (Equation (6)):

$$CSP^k(E_i) = \frac{|CSP_{relevant}^k|}{|CSP^k|} \quad (6)$$

where k represents the Top-K Java files; E_i represents a service path of the OAS file; CSP^k represents all of the Java methods found in the first k Java files; and $CSP_{relevant}^k$ refers to Java methods using the target service path. Note that if a retrieved code is “relevant” to an API this was also based on voting by the three evaluators (as in the previous two experiments).

We randomly selected 100 service paths, calculated their CSP values, and then calculated the overall accuracy, example precision (EP), as follows (Equation (7)):

$$EP^k(n) = \frac{\sum_{i=1}^n CSP^k(E_i)}{n} \quad (7)$$

where EP refers to example precision, and n indicates the total number of OAS service paths.

In addition, to establish a benchmark to be compared, we also evaluated the search results on the first page of the GitHub Search API (that is, the results without subsequent analysis) and calculated EP for the same 100 service paths that serve as the search keywords.

4.3.2. Experimental Results

The evaluation results are shown in Figure 8. The average precision of the Top-1~Top-3 for API Prober was observed to exceed 0.87. In addition, API Prober outperforms the original GitHub Search API obviously. In general, API Prober proved highly effective in finding suitable code examples invoking Web APIs. For RQ3, we could conclude that the proposed code retrieval approach could precisely discover the correct Java code in most situations.

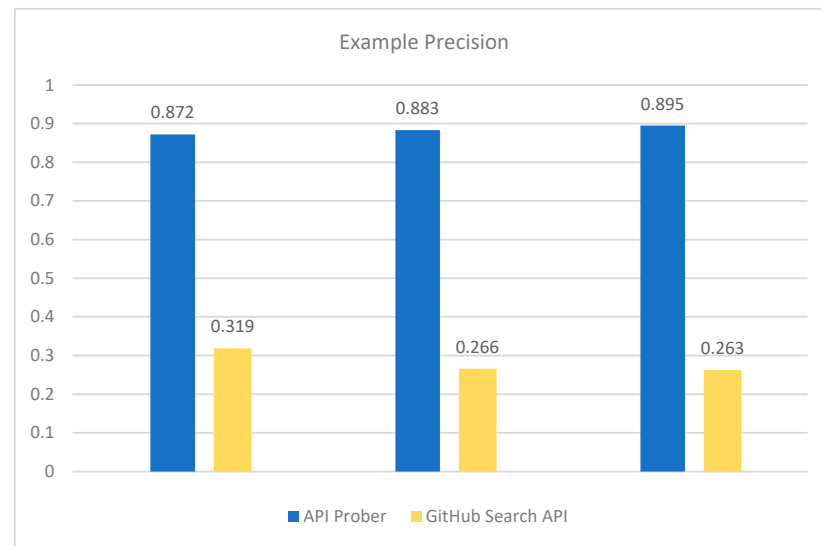


Figure 8. EP Top-3 evaluation results.

Although the experiment results were relatively good, there were a number of cases in which API Prober was unable to extract suitable fragments, as discussed in the following and will be improved in future work:

1. Service path was not used: In some Java files, the service path is used as a string for a specific output or description, rather than for service invocation. A real-world example is presented in Table 5;
2. Variable names of excessive simplicity: Some class variables do not have specialized and meaningful names, such that the Java Method Parser misidentified the class variable name. In the real-world example in Table 6, the service path was declared in the class variable (url), such that API Prober misidentified a code example;
3. Special use behavior: In some Java coding, the service path is split into different pieces to operate different service paths under the same OAS service. In the real-world example in Table 7, API Prober found the service path in the class variable and its method; however, the user added other resource operations to the class variable, resulting in an error of code discovery.

Table 5. Example 1: service path is not used.

Score	3
Service path	https://www.googleapis.com/gmail/v1/users/{userId}/messages/send (accessed on 10 January 2023)
Error fragment	endpoint.setDescription("Give the API method endpoint to send email" + " (E.g: -for gmail: https://www.googleapis.com/gmail/v1/users/[userId]/messages/send)");

Table 6. Example 2: variable of excessive simplicity.

Score	5
Service path	https://www.googleapis.com/blogger/v3/blogs/byurl (accessed on 10 January 2023)
Class variable	static String url = "https://www.googleapis.com/blogger/v3/blogs/byurl?url=http://strandedhhj.blogspot.com/";
Error fragment	String blogURL = mResponseObject.getString("url");

Table 7. Example 3: special use behavior.

Score	5
Service path	https://api.ebay.com/sell/fulfillment/v1/order (accessed on 10 January 2023)
Class variable	public static final String getOrderUrl = "https://api.ebay.com/sell/fulfillment/v1/order";
Error fragment	public static void getOrder(String accessToken, String orderId) { String value = httpClient.doGet(getOrderUrl+"/"+orderId", headers, null); System.out.println(value); }

5. Conclusions

To realize the “FAIR” (Findability, Accessibility, Interoperability, and Reusability) features for the usage of Web APIs, this paper introduces a novel system, API Prober, for the analysis of OAS (OpenAPI Specification) documents, the extraction of service code examples, the clustering of results to clarify the use of target RESTful services, and the discovery of additional relevant services. Overall, the experimental results are relatively stable and reach the research objectives: producing appropriate clusters for APIs, recommending suitable APIs, and retrieving correct examples of service client code for most situations.

The limitations of API Prober are twofold: (1) API Prober is based on OAS documents, which mainly support the descriptions for REST-style APIs. Event-driven services are not supported by API Prober now; and (2) currently, API Prober can only discover service client code in Java. In general, besides the above limitations, API Prober can inspect and recommend REST-style APIs and retrieve Java code for utilizing a target API.

Our future plans include the following: (1) addressing the issues of incorrect code discovery; (2) extraction of examples in various programming languages; (3) extraction of security features in RESTful services (e.g., authentication, authorization, and data encryption); and (4) inviting software developers to use API Prober in their real-world projects to provide heuristic evaluations on API Prober’s core functionality.

Author Contributions: Conceptualization, S.-P.M. and C.-J.L.; methodology, S.-P.M. and M.-J.H.; software, M.-J.H.; validation, M.-J.H. and H.-J.C.; writing—original draft preparation, M.-J.H.; writing—review and editing, S.-P.M.; supervision, S.-P.M.; project administration, S.-P.M.; funding acquisition, S.-P.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Science and Technology Council (NSTC) in Taiwan, grant number 110-2221-E-019-039-MY3.

Data Availability Statement: The data are not publicly available due to the stored data contains privacy data such as access tokens or API keys.

Acknowledgments: Thanks for Hsueh-Cheng Lu’s effort to maintain the codebase of API Prober.

Conflicts of Interest: The authors declare no conflict of interest.

References

- Fielding, R.T.; Taylor, R.N. Principled design of the modern web architecture. *ACM Trans. Internet Technol.* **2002**, *2*, 115–150. [CrossRef]
- Gat, I.; Remencius, T.; Sillitti, A.; Succi, G.; Vlasenko, J. The API economy: Playing the devil’s advocate. *Cutter IT Journal* **2013**, *26*, 6–11.

3. ProgrammableWeb. Available online: <http://www.programmableweb.com> (accessed on 10 January 2023).
4. RapidAPI Hub. Available online: <https://rapidapi.com/hub> (accessed on 10 January 2023).
5. APIs.guru. Available online: <https://apis.guru/> (accessed on 10 January 2023).
6. Wittern, E.; Muthusamy, V.; Laredo, J.A.; Vukovic, M.; Slominski, A.; Rajagopalan, S.; Jamjoom, H.; Natarajan, A. API Harmony: Graph-based search and selection of APIs in the cloud. *IBM J. Res. Dev.* **2016**, *60*, 12:1–12:11. [CrossRef]
7. OpenAPI Specification (OAS). Available online: <https://swagger.io/docs/specification/> (accessed on 10 January 2023).
8. Wilkinson, M.D.; Dumontier, M.; Aalbersberg, I.J.; Appleton, G.; Axton, M.; Baak, A.; Blomberg, N.; Boiten, J.-W.; da Silva Santos, L.B.; Bourne, P.E.; et al. The FAIR Guiding Principles for scientific data management and stewardship. *Sci. Data* **2016**, *3*, 1–9. [CrossRef] [PubMed]
9. Neumann, A.; Laranjeiro, N.; Bernardino, J. An Analysis of Public REST Web Service APIs. *IEEE Trans. Serv. Comput.* **2018**, *14*, 957–970. [CrossRef]
10. Webber, J. A programmatic introduction to neo4j. In Proceedings of the the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity, Tucson, AZ, USA, 19–26 October 2012; pp. 217–218.
11. JavaParser. Available online: <http://javaparser.org/> (accessed on 10 January 2023).
12. Agrawal, R.; Phatak, M. A novel algorithm for automatic document clustering. In Proceedings of the 3rd IEEE International Advance Computing Conference (IACC), Ghaziabad, India, 22–23 February 2013; pp. 877–882.
13. Reddy, V.S.; Kinnicutt, P.; Lee, R. Text Document Clustering: The Application of Cluster Analysis to Textual Document. In Proceedings of the International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 15–17 December 2016; pp. 1174–1179.
14. Rahman, M.M.; Liu, X.; Cao, B. Web API Recommendation for Mashup Development Using Matrix Factorization on Integrated Content and Network-Based Service Clustering. In Proceedings of the IEEE International Conference on Services Computing (SCC), Honolulu, HI, USA, 25–30 June 2017; pp. 225–232.
15. Fletcher, K. Regularizing Matrix Factorization with Implicit User Preference Embeddings for Web API Recommendation. In Proceedings of the IEEE International Conference on Services Computing (SCC), Milan, Italy, 8–13 July 2019; pp. 1–8.
16. Zou, G.; Qin, Z.; He, Q.; Wang, P.; Zhang, B.; Gan, Y. DeepWSC: Clustering Web Services via Integrating Service Composability into Deep Semantic Features. *IEEE Trans. Serv. Comput.* **2022**, *15*, 1940–1953. [CrossRef]
17. APIs.io. Available online: <http://apis.io/> (accessed on 10 January 2023).
18. Mashape. Available online: <https://www.mashape.com/> (accessed on 10 January 2023).
19. Ma, S.-P.; Lin, H.-J.; Hsu, M.-J. Semantic Restful Service Composition Using Task Specification. *Int. J. Softw. Eng. Knowl. Eng.* **2020**, *30*, 835–857. [CrossRef]
20. Ma, S.-P.; Hsu, M.-J.; Chen, H.-J.; Su, Y.-S. *API Prober—A Tool for Analyzing Web API Features and Clustering Web APIs*; Springer International Publishing: Cham, Switzerland, 2020; pp. 81–96.
21. Li, Y.; Bandar, Z.A.; McLean, D. An approach for measuring semantic similarity between words using multiple information sources. *IEEE Trans. Knowl. Data Eng.* **2003**, *15*, 871–882. [CrossRef]
22. Haupt, F.; Leymann, F.; Scherer, A.; Vukojevic-Haupt, K. A Framework for the Structural Analysis of REST APIs. In Proceedings of the IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, 3–7 April 2017; pp. 55–58.
23. Agarwal, N.; Sikka, G.; Awasthi, L.K. Web service clustering approaches to enhance service discovery: A review. In *Recent Innovations in Computing*; Springer: Singapore, 2021.
24. Aggarwal, C.; Zhai, C. A Survey of Text Clustering Algorithms. In *Mining Text Data*; Springer: Berlin/Heidelberg, Germany, 2012.
25. Vinh, N.X.; Epps, J.; Bailey, J. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *J. Mach. Learn. Res.* **2010**, *11*, 2837–2854.
26. Ma, S.-P.; Chen, Y.-J.; Syu, Y.; Lin, H.-J.; Fanjiang, Y.-Y. Test-Oriented RESTful Service Discovery with Semantic Interface Compatibility. *IEEE Trans. Serv. Comput.* **2021**, *14*, 1571–1584. [CrossRef]
27. Sohan, S.M.; Maurer, F.; Anslow, C.; Robillard, M.P. A study of the effectiveness of usage examples in REST API documentation. In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Raleigh, NC, USA, 11–14 October 2017; pp. 53–61.
28. Cosentino, V.; Izquierdo, J.L.C.; Cabot, J. Findings from GitHub: Methods, datasets and limitations. In Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 14–15 May 2016; pp. 137–141.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.