Comparing the Similarity of OpenAPI-Based Microservices

Zhongyi Lu
School of Computer Science
University College Dublin
Dublin 4, Ireland
zhongyi.lu@ucdconnect.ie

Declan T. Delaney
School of Electrical and Electronic Engineering
University College Dublin
Dublin 4, Ireland
declan.delaney@ucd.ie

David Lillis
School of Computer Science
University College Dublin
Dublin 4, Ireland
david.lillis@ucd.ie

ABSTRACT

Microservices constitute the state of the art for implementing distributed systems and have been seen as a potential solution towards open systems. The characteristics of open systems require structured microservice management, including grouping microservices that are functionally similar. Microservices use RESTful APIs, often documented via OpenAPI specifications, to demonstrate their functionalities. Existing similarity metrics for microservice APIs have primarily focused on individual RESTful endpoints. However, understanding the full functionality of a microservice within an open system requires that the entirety of its OpenAPI documentation be considered. Thus, an approach that can compute a measure of similarity between entire microservice definitions in open environments is needed. In this paper, we propose an approach that can extract key information from the OpenAPI descriptions of microservices using Natural Language Processing (NLP) techniques, vectorise the extracted information using GLoVe embeddings, and cluster similar microservices using embedded API file vectors. Evaluations were conducted on real-world OpenAPI documents to demonstrate the effectiveness of the proposed approach.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; • **Information systems → Similarity measures**.

KEYWORDS

Microservice architectures, OpenAPI, RESTful API, NLP, clustering

ACM Reference Format:
Zhongyi Lu, Declan T. Delaney, and David Lillis. 2024. Comparing the Similarity of OpenAPI-Based Microservices. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605098.3635963

1 INTRODUCTION

The Microservices Architecture (MSA) is a distributed architectural style that divides monolithic applications into smaller, fine-grained service instances that act as single-responsibility units. It is extended from Service Oriented Architecture (SOA) [21]. Microservice instances communicate with lightweight mechanisms, often making use of open HTTP resource API descriptions [30]. Some researchers have thus focused on the use of MSAs to build open systems [7, 25]. Open systems do not focus on specific tasks and can react to environments that are changing constantly, unlimitedly, and unanticipatedly [3]. Singh and Huhns outline key characteristics of open environments, namely: i) system components should be autonomous; ii) system components can be heterogeneous; and iii) the environment can be dynamic [24, pp. 7–10].

Based on the definitions above, we define open microservices systems as systems where heterogeneous and autonomous microservice instances may join or leave the system casually without restriction as to ownership or origin.

The characteristics of open microservices systems require systems to have methods to effectively manage and discover microservices [14]. For example, when managing Web services (another form of SOA), a very efficient approach is to cluster services that are functionally similar to one another [13]. This logic can also be applied to the microservices domain. However, this type of approach has typically been applied within closed systems, in which the labels of the clusters are predefined. In open systems, however, clusters cannot be predefined and boundaries for microservices cannot be set. The similarity between microservices in this context can only be learned after they join the system. The functionalities and communication mechanisms of microservices are typically described in API files [10], and so a method that can compute a meaningful metric of similarity between API files is desirable.

For this research, we focus on files that are modelled with OpenAPI, which is a de facto standard for RESTful API documentation [6, 27]. Previous research, discussed in Section 2.2, has introduced techniques to compute similarities between RESTful operations within a single OpenAPI document. In contrast, we use the entirety of the OpenAPI file for each microservice. Comparing similarities between RESTful operations and entire OpenAPI files is different because: i) the latter contains all endpoints and other metadata relating to microservice responsibilities, whereas the former only contains details relating to a single endpoint; ii) the writing styles of the descriptive components can vary as different files can be written by different developers, whereas one OpenAPI file is typically written by one person or a group of people conforming to an agreed writing style; iii) the description of one endpoint only gives the information of one HTTP operation, it cannot provide the context of the responsibility of the microservice, which is often provided elsewhere in the API file; and iv) some microservices need multiple endpoints to implement a single responsibility, which means API endpoints cannot be treated in isolation. Thus, existing techniques are not suitable for computing a similarity between

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3635963

complete OpenAPI files. This motivates the creation of a novel approach to computing the similarity between entire OpenAPI-based microservice descriptions.

This paper introduces our approach to computing similarity between microservices based on their OpenAPI documentation. We use Natural Language Processing (NLP) techniques to extract useful information from OpenAPI files, vectorise the OpenAPI documentation with pre-trained word embeddings, and cluster similar microservices using their respective file vectors. Evaluation was conducted using a dataset that was collected from real-world OpenAPI documentation. The results show that the proposed approach has the ability to cluster similar microservices effectively.

Section 2 introduces the background of the research. Section 3 explains the proposed approach to embedding microservice definitions into a vector representation to compute similarity. Evaluation is described in Section 4. Finally, Section 5 concludes the work.

2 RELATED WORK

This section outlines the concept of OpenAPI and discusses previous work on computing similarity between API elements.

2.1 OpenAPI

The OpenAPI Specification (OAS) [17], formerly known as the Swagger Specification, defines a standard, language-independent interface to HTTP APIs that allows both humans and machines to understand the functionalities of microservices [16]. OAS has become the market leader and is supported by many leading industry vendors such as Google, Microsoft, AWS, etc [12, 27].

2.2 API Similarity

Previous research has introduced different measures to compute the functional similarity between API endpoints of software applications and Web services using semantics.

One common approach is to utilise word embeddings, where the semantic meanings of words are represented in vector form. Several options are available with varying characteristics [28]. Word embeddings can be used to convert API endpoints into vectors, so that the similarity between the vectors can be used to represent the similarity between API endpoints. Xu et al. [29] trained their own embeddings with word2vec, a continuous vector representation of words. Jiang et al. [11] introduced a method to compute the similarity between API files using a pairwise semantic comparison of their properties, which is also computed using word2vec embeddings. Gao et al. [9] convert the introduction section of the API file to a word vector in which words are embedded into the feature vector.

Some methods firstly filter unimportant information that does not represent API endpoints before computing the semantic similarity between the filtered information. The semantic similarity is representative of the functional similarity. Al-Debagy et al. [1] extracted the name and parameters of each operation. The operation names and parameters were converted into word representations using fastText: a library for text representation and classification. Semantically similar operations were clustered together by applying the Affinity Propagation algorithm to the word representations. In the aforementioned works, semantic similarity between API files is typically calculated as the cosine similarity between the relevant embeddings.

Instead of using word vectors as the basis of clustering API endpoints, some methods use semantic similarity itself as the basis of clustering similar microservices. Sun et al. [26] introduced a method to find endpoints within an API file that are similar to a particular target endpoint. The method first computes the semantic similarity between the words extracted from all URI endpoints and responses and the words extracted from the target endpoint. It then builds an API similarity graph where APIs are the vertices, and the weighted edges between adjacent nodes represent the overall similarities. The edges with low weights will be removed, which decomposes a graph into several sub-graphs, whereby each sub-graph is a cluster. Baresi et al. [4] proposed an automated process to cluster similar HTTP operations by analysing the input specification with regard to a reference vocabulary. The system uses OpenAPI specifications as input, and extracts the operation names as the basis to compute the similarity between operations. Then, they used DISCO (DIStributionally related words using CO-occurrences), a pre-computed database of collocations and distributionally similar words, to find the most matching schema on Schema.org. API endpoints with similar schemas are included in the same cluster.

3 CLUSTERING OF OPENAPI MICROSERVICES

We propose an method to cluster similar microservices using their OpenAPI documentation, as illustrated in Figure 1. The approach comprises three major steps: preprocessing, vectorisation, and clustering. Each block illustrates one step, which is described in the sections that follow. The approach uses the OpenAPI descriptions of microservices as input. Each file is processed and vectorised. The vectors are then clustered to reflect the similarity between microservices.

3.1 Preprocessing

The first major step of the proposed approach is preprocessing. The goal of this step is to extract keywords and phrases from the OpenAPI file of each microservice in the system, which can then be used to embed the API file in the next step.

For this research, we focus on documents that conform to OAS version 3.0 and higher[1]. Figure 2 shows the example structure of an OpenAPI 3.0 document. According to OAS 3.0, the `openapi`, `info`, and `paths` fields are required[2], which means that all OpenAPI documents must contain these fields. The `openapi` field only contains the OAS version that the file conforms to. Therefore, the functional information is extracted from two fields: `info` and `paths`.

The `info` field consists of one Info Object[3], which contains seven fixed fields: *title*, *summary*, *description*, *termsOfService*, *contact*, *license*, and *version*. Among these fields, *title*, the title of the API, is required. The *summary* field contains a short summary of the API, with a more detailed description contained in the *description*

[1] OpenAPI Specification 3.0 was released in 2017 [18], and 3.1 was released in 2021 [17]
[2] The `paths` field is no longer required as at v3.1.0, but it is very often present in practice.
[3] Examples of the OpenAPI objects discussed in this paper can be found in the respective specifications.

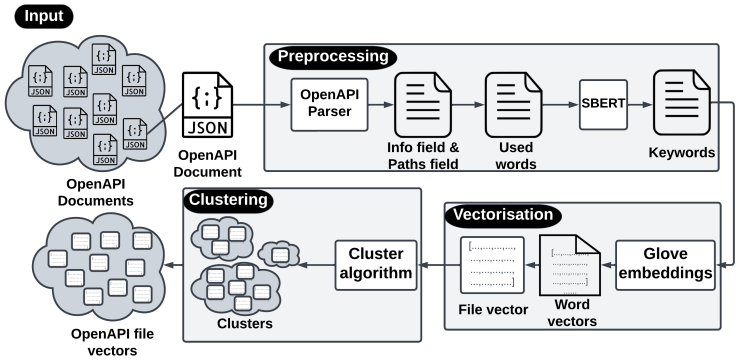

Figure 1: The proposed approach.

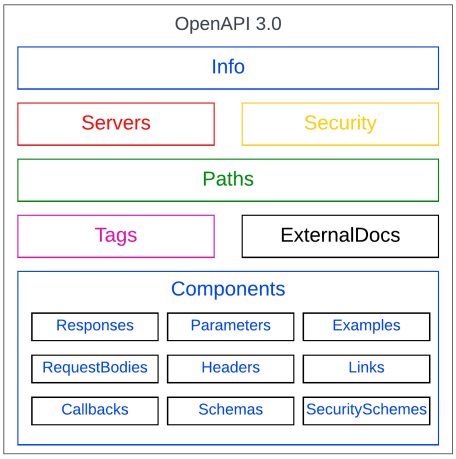

Figure 2: Structure of an OpenAPI document [27].

field. Although *summary* and *description* are optional, they are commonly used, and are helpful in understanding the functionality of microservices. Thus, we also extract these from the API document.

The `paths` field contains one Paths Object that includes a set of Paths Item objects, each of which describes the operations available on a an endpoint. A Paths Item object contains thirteen fields. We use the URL of the endpoint, *summary*, *description*, and HTTP request methods to learn about its functionality.

There are eight HTTP request methods in total. However, in this approach, we only select four methods: POST, GET, PUT, and DELETE, as they map to the basic REST design principles of Creating, Reading, Updating, and Deleting (CRUD) operations, respectively. OpenAPI uses Operation Objects to define HTTP operations. We use *summary*, *description*, and *operationId* to learn about each operation.

We use the Swagger Parser[4] to parse API files and extract the required information.

The next stage is to extract words that are used in the file. After extracting the relevant information from the aforementioned fields, we remove non-descriptive contents such as URLs, markdown labels, symbols, etc. We also split camel case, snake case, and pascal case into individual words. After this step, we check all the words and see if they are included in the GloVe embeddings. For the words that are not included (including compound words that are written all in lowercase, which is a common feature of path names and is more challenging to split), we use the OpenAI API to give prompts to GPT-3.5[5]. The language model returns predicted completions in an attempt to split these lowercase compound words into individual words. In this paper, the language model that we use is "`text-davinci-003`", which can understand and generate natural language. After the words are split, we check whether the suggested words are in the GloVe embeddings. If a word cannot be split or cannot be found in the GloVe embeddings after being split, it is skipped.

We then identify a subset of all the words that were extracted from each file that can be used as keywords to describe the semantics of the functions of the microservice. In this approach, we use Sentence-BERT (SBERT) [22], a modification of the BERT network [5]. SBERT can handle tasks like large-scale semantic similarity comparison, clustering, etc., which makes it suitable for keyword extraction.

The extraction consists of three steps: finding candidate keyword phrases, using SBERT to get the embeddings of the candidate phrases and the collection of all the words extracted from the OpenAPI file, and finally, finding the N most similar candidate phrases using cosine similarity. Figure 3 illustrates the process.

CountVectorizer from the `scikit-learn` library [19], which can convert a collection of texts to a matrix of token counts, is used to find candidate keyword phrases from the extracted words. All candidate trigram phrases are obtained after this step. We chose three as the phrase length, as our findings show that longer phrases often contain duplicate keywords, which is unnecessary.

After obtaining the candidate phrases, we use the concatenations of each group of candidate phrases and the entirety of all the extracted words as input to transform them into a single embedding using the SBERT Sentence Transformer[6]. This is to capture the semantic meaning of the entire API definition, with a view to

[4]https://github.com/swagger-api/swagger-parser

[5]https://platform.openai.com/docs/models/gpt-3

[6]https://github.com/UKPLab/sentence-transformers

later identifying a set of keywords that best capture this meaning. The model we use is '`distilbert-base-nli-mean-tokens`', which maps texts to a 768-dimensional dense vector space. This allows us to embed each candidate phrase and the entirety of the extracted words. The embeddings can later be used to compute the similarity between the candidates and the overall extracted words.

Finally, we compute the cosine similarities between the candidates and the single embedding of all extracted words. The candidates with the N highest (i.e., $N = 5$) cosine similarity score will be the keyphrases used to represent the API file. The words in the keyphrases will become the keywords.

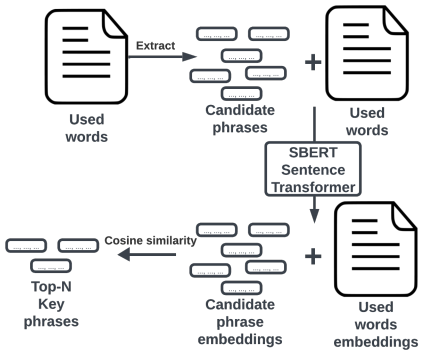

Figure 3: Keywords extraction process.

3.2 File Vectorisation

After obtaining a set of keywords from an OpenAPI file, we vectorise them so that they can be used to represent the microservice.

Initially, we find the vector of each extracted word using GloVe embeddings [20]. GloVe provides pre-trained word vectors that are based on different text corpora. For this research, we use the Common Crawl word vectors. It contains 840 billion tokens and 2.2 million cased vocabulary. Each word vector has 300 dimensions.

We then compute the embedding of each API file using the word vectors. The file embedding is calculated using a weighted average. In this approach, we use smooth inverse frequency (SIF) [2] to compute the weight of each word embedding. In SIF, the weight of a word w is given as $a/(a + p(w))$. a is a constant (i.e., 0.001) and $p(w)$ is the estimated word frequency of word w. In SIF, $p(w)$ can be estimated from different corpora. The word frequency corpus that we use is the word frequency list gathered from all the articles in the Wikipedia Database backup dumps[7]. It contains more than 2 million unique English terms. The calculation of an API file vector V_{file} is given in Equation 1, where N represents the number of keywords, V_{word_i} represents the GloVe embedding of the i-th keyword, and w_{word_i} represents the weight of the i-th keyword.

$$V_{file} = \frac{\sum_{i=0}^{N} V_{word_i} \times w_{word_i}}{N} \quad (1)$$

3.3 Clustering

After obtaining vectors of all the OpenAPI files, we cluster microservices that are functionally similar.

Density-based algorithms are one of the most common techniques for clustering documents [15]. In our approach, we use the Mean Shift algorithm [8]. Unlike alternatives such as k-means, it does not require a specific number of clusters to be specified before clustering. Another reason to choose Mean Shift is that it will not consider any nodes to be noise, which is important for this approach as it is essential that all microservices be included. This finds neighbourhoods based on density functions, whereby the average data point is the centre of the cluster. We use the `MeanShift` implementation from the *sklearn.cluster* package [19] to realise clustering.

4 EVALUATION

To evaluate the approach, we use a real-world OpenAPI dataset[8] that we have collected from APIs.guru[9], GitHub[10], and SwaggerHub[11]. As the proposed approach targets OpenAPI files relating specifically to microservices, and the definition of microservices suggests that each of them should only hold one responsibility, we filtered out OpenAPI files that relate to more than one responsibility: usually those that are working on more than one entity. This process was conducted manually, as automated identification of microservice definitions holding multiple responsibilities is outside the scope of the present work. This includes OpenAPI files that describe entire products (e.g., Google Maps, Microsoft Azure, Gmail, etc.) and OpenAPI files that describe services with more than one functionality (e.g., a service that maintains both a client list and a provider list). The dataset contains 428 OpenAPI files that follow OAS v3.0 or later.

To facilitate the evaluation, we label each file with three categories of labels: the "action" label, the "entity" labels, and the "area" label, described as follows:

- The *action* label is a verb chosen by the assessor to describe the functionality. This is based on the descriptive information and the HTTP endpoints within the file, which means that it is not necessarily the HTTP request method. Each API file is assigned an action label. Examples of action labels in the dataset include "get" (to get information about an entity), "subscribe" (e.g. to an email newsletter) and "manage" (e.g. create/edit/delete/get contacts in an address book).
- The *entity* labels are the entities that the action operates upon. Each API file may be assigned multiple entity labels.
- The *area* label is the industrial area that the microservice is related to. Each API file is assigned an area label.

As an example, consider an OpenAPI file that describes a microservice that gets the bus schedule. Its action label would be "get", its entity labels would be "bus" and "schedule", and its area label would be "transport". Table 1 illustrates how some of the OpenAPI files of microservices are labelled.

In total, the dataset contains 50 area labels. Table 2 lists the distribution of the area labels of all the files in the dataset.

Two evaluations were conducted: a single-area cluster evaluation and a cross-area cluster evaluation. The single-area cluster

[7] https://dumps.wikimedia.org/backup-index.html

[8] Available at https://github.com/zhongyilucy/OpenAPIMicroserviceFiles.git
[9] https://apis.guru/
[10] https://github.com
[11] https://swagger.io/tools/swaggerhub/

File	Action	Entity	Area
Train Station arrival.json	Get	Train, Arrival	Transport
Flight Offers Price.json	Get	Flight, Offer	Transport
Email Subscription API.json	Subscribe	Letter	Marketing
Authentiq API.json	Authenticate	Account	Security
ContactApp.json	Manage	Contact	Contact

Table 1: The labels of some OpenAPI files.

Area	Count	Area	Count
Account	5	Marketing	4
Address	2	Membership	3
Alert	4	Movie	6
API	3	Music	10
Application	2	Name	3
Article	6	News	4
Audio	2	Notification	9
Banking	24	Payment	11
Blog	3	PDF	13
Book	11	Phone number	8
Business	58	Postcode	2
Claims	2	Recipe	3
Code	2	Referral	3
Communication	9	School	2
Contact	4	Science	2
Data	7	Search	4
Document	6	Security	21
Facility	4	System	10
Finance	8	Time	3
Hotel	11	Transport	42
HR	17	Travel	5
Image	9	User	28
Language	9	Vehicle	4
Local Info	10	Weather	5
Logistics	2	Web	2

Table 2: Ground truth labels in the dataset.

evaluation aims to evaluate the suitability of our approach in systems that only focus on one specific industrial area. Each test is conducted on API files that have the same area label. The evaluation is to examine the extent to which our approach can successfully cluster microservices that relate to similar entities. Since each API file can have more than one entity label, this evaluation is akin to multi-label classification.

The cross-area cluster evaluation evaluates the approach in various scenarios where systems focus on more than one area that are not necessarily related to one another. The intention of this evaluation is to simulate the characteristics of an open environment. Each test is conducted on API files with different area labels. The evaluation is to examine the extent to which our approach can successfully cluster microservices according to their industry area. Since each API file can only have one area label, this evaluation is akin to a single-label classification. These two evaluations can not only evaluate the proposed approach in the context of open environments, but also in systems that are more domain-specific, which is common in practice.

4.1 Single-Area Cluster Evaluation

The single-area cluster evaluation was conducted within individual areas to simulate the scenario where a system only focuses on one kind of industrial area. The API files in the corpus are initially divided according to their area labels. API files with similar area labels will be considered to belong to the same area. Then, clustering is applied within each area, with the aim that microservices that operate on similar entities will be clustered together, and those operating on dissimilar entities will be in different clusters. The entity labels of each API file are used to represent the entities operated upon by each microservice.

The entity labels of API files within individual areas are the basis for the evaluation. We use a value between 0 and 1 (i.e., an accuracy score) to show how well the proposed approach can ensure API files inside the same cluster have the same entity labels, while ensuring API files with dissimilar entity labels are stored in different clusters. The calculation of the accuracy score is given by Equation 2. The accuracy score of an area A is the arithmetic mean of two other scores: the macro F1 score $MF1_A$ and the duplication score dup_A.

$$Accuracy_A = \frac{MF1_A + dup_A}{2} \tag{2}$$

The macro F1 score of each area is the arithmetic mean of the F1 score of each cluster. Equation 3 explains how the macro F1 score of an area A, $MF1_A$, is calculated. A is clustered into N clusters using the Mean Shift algorithm. The F1 score of a cluster i is $F1_{Cluster_i}$. The macro F1 score reflects the ability of the proposed approach to cluster OpenAPI files with the same entity labels together.

$$MF1_A = \sum_{i=0}^{N} \frac{F1_{Cluster_i}}{N} \tag{3}$$

To calculate an F1 score for an individual cluster, it is necessary to associate at least one label with each cluster, based on the microservices contained therein. The set of cluster label(s) associated with area A, denoted as $Labels_A$, is given by Equation 4. Any label that is assigned to no fewer than half of the API file instances will be a cluster label. If no such label exists, the most common entity labels will be the cluster labels. A cluster may have more than one label when more than one entity label is assigned to $\geq 50\%$ of labels, or there are multiple most common entity labels with equal frequency.

$$\begin{aligned} Labels_A = \{&labels\ associated\ with\ \geq 50\%\ of\ instances\} \\ &\cup \{labels\ with\ the\ highest\ label\ frequency\} \end{aligned} \tag{4}$$

For evaluation purposes, the entity label(s) of each API file in a cluster are compared with the cluster label(s). The F1 score of each cluster within an area can then be calculated according to its standard machine learning formula, which is used to calculate the macro F1 score of the area. The calculation is based on the definitions of *True Positives*, *False Positives*, and *False Negatives* given in Table 3.

True Positive	An entity label of the API file instance that is contained within the cluster label(s).
False Positive	An entity label of the API file instance that is not contained within the cluster label(s.)
False Negative	A cluster label that is not contained within the entity labels of the API file instance.

Table 3: Definitions of *True Positive*s, *False Positive*s, and *False Negative*s.

Given the example in Table 2, if all seven API files listed in the table are put into one cluster, there is no label that is associated with more than half of the API file instances. Therefore, the cluster labels are "Flight" and "Offer" as they are the entity labels with the equal highest label frequency. The count of *True Positive*s, *False Positive*s, and *False Negative*s for each API file is listed in Table 4. The F1 score of this cluster can then be calculated, which contributes to the calculation of the macro F1 score of area "Transport".

File	TF	FP	FN
Train Station arrival.json	0	2	2
Flight Offers Price.jso	2	0	0
Branded Fares Upsell.json	2	0	0
API Documentation for Train Schedule.json	0	2	2
Airports API v2.json	0	1	2
Airport & City Search.json	0	1	2
Flight Offers Search.json	2	0	0

Table 4: *True Positive*s, *False Positive*s, and *False Negative*s of exampled API files

Area	Macro F1	Duplication	Accuracy
Banking	0.899	0.750	0.825
Book	0.823	0.136	0.480
Business	0.911	0.662	0.786
Communication	0.875	1.000	0.938
Hotel	0.960	0.273	0.616
HR	0.778	0.733	0.756
Image	0.971	0.000	0.485
Language	0.635	0.556	0.595
Local info	0.844	0.550	0.697
Music	0.944	0.636	0.790
Notification	1.000	0.111	0.556
Payment	0.887	0.455	0.671
PDF	0.912	0.038	0.475
Security	0.885	0.111	0.498
System	0.833	0.675	0.754
Transport	0.624	0.452	0.538
User	0.993	0.000	0.497

Table 5: Precision, Recall, and F1-Scores within single areas.

After obtaining the macro F1 score of the area, we then calculate its duplication score. The duplication score is also a value between 0 and 1, where 0 means the cluster labels of all the clusters within the area are the same and 1 means the cluster labels of all the clusters inside the area are entirely distinct. Equation 5 explains how the duplication score of an area A is calculated, denoted as dup_A. Assume A contains M API files, which are distributed into N clusters. For each cluster i, there is a $Label_{total}$, which is the set of all labels for this cluster, and a $Label_{unique}$, which contains the labels that are unique to this cluster (i.e., those that have not been assigned to any other clusters in the area). The duplication score of A is the weighted average of the size of $Label_{unique}$ divided by the size of $Label_{total}$ of each cluster. The weight of each cluster is its size divided by M.

$$dup_A = \sum_{i=0}^{N} \frac{|Cluster_i|}{M} \times \frac{|Label_{unique_i}|}{|Label_{total_i}|} \tag{5}$$

To ensure that the areas being tested have enough API files for the clustering algorithm to learn, we only conduct single-area evaluations on areas that contain more than 8 API files (the average size of the areas). Table 5 lists the macro F1, duplication, and accuracy scores of the tested areas. It is worth noting that in practice, it may be impossible to achieve a 1.0 score for either macro F1 or duplication score due to the mixing of labels.

The macro F1 score indicates that the approach can make sure that the majority of the API files within one cluster have the same entity labels. Only three areas achieve a macro-F1 below 0.8. The results in terms of duplication are somewhat less consistent. Approximately one third of areas show considerable overlap between cluster labels, with a duplication score below 0.2. This illustrates the difficulty in putting all the API files with the same entity labels into the same cluster and keeping cluster labels unique. This kind of problem is especially prominent when most of the API files within one area have the same entity label (e.g., the User area, the Image area, the Hotel area, and the PDF area), or only very few API files do not contain the same entity labels that the rest of the API files have (i.e., the Notification area, the Book area).

Taking the User area as an example, all the microservices in that area operate on a "user" entity. Therefore, even though a large majority of instances with that entity label are clustered into a large cluster, all other clusters that are created also have "user" as an entity label and thus the duplication score is 0. This is exacerbated in this specific instance because there are many other entities in that area with the potential to result in additional cluster labels. Areas with a wider variety of entity labels tend to exhibit a higher duplication score, reflecting a strong ability to successfully cluster these entities separately. Based on this observation, it is worth noting that when applied to systems that only focus on one area, the proposed approach will work best when microservices operate on different entities within that area.

The single-area cluster evaluation demonstrates that the proposed approach can differentiate API files with different entity labels in the same area. However, for highly similar microservices, it sometimes overfits and cannot ensure that all microservices with the same entity labels can be put into the same cluster.

4.2 Cross-Area Cluster Evaluation

The aim of our approach is to design a similarity metric for OpenAPI-based microservices in open environments. However, since it is

	Combination	Homogeneity	Completeness	V-Measure
Not similar	Transport+User+Banking	0.887	0.692	0.778
	Blog+Recipe+News+Contact+Science	1.000	0.849	0.918
	Code+Communication+Facility	0.643	0.739	0.686
	PDF+Phone Number+Weather	1.000	0.888	0.941
	Time+Music+Membership+Marketing	0.673	0.804	0.768
Similar	Notification+Alert	0.762	0.341	0.472
	Music+Movie	1.000	0.679	0.809
	Contact+Communication	0.720	0.407	0.520
	PDF+Image	1.000	1.000	1.000
	Book+Article	0.768	0.768	0.768
Not overtly similar	Travel+Vehicle+Hotel	1.000	0.889	0.941
	Marketing+Contact+Data	0.425	0.514	0.465
	Book+Business	0.645	0.754	0.695
	Music+Membership+User+Payment	0.816	0.655	0.727

Table 6: The Homogeneity, Completeness, and V-Measure scores of different combinations.

unfeasible to create a fully open environment in practice, we use API files with different area labels to simulate open environments where microservices can relate to a variety of industrial areas. The goal of cross-area cluster evaluation is to test whether the approach can group API files with the same area label into the same cluster.

We use V-measure [23] to evaluate the clustering result. V-measure is an external entropy-based cluster evaluation measure. It is the harmonic mean of the homogeneity and completeness scores. Homogeneity indicates the extent to which the data points inside one cluster are from the same class. Completeness indicates the extent to which the data points from the same class are put into the same cluster. V-measure is a value between 0 and 1. The higher the value, the more homogeneous and complete the clustering result is.

To more completely assess the effect of cross-area clustering, we created three scenarios to simulate three levels of difficulty in terms of the clustering problem. This level of difficulty is represented by the degree to which areas share common characteristics (as determined by human judgement). To facilitate this, we group areas into three types of combinations: areas that share no similar characteristics; areas that share similar characteristics; and areas that are not overtly similar, but can be applied to the same scenario. 5 groups of areas were selected where none are similar to any others, and cannot be logically formed into a business logic together. 5 further groups of areas were selected that are either similar or share the same characteristics. 4 groups of areas were selected that are not overtly similar to each other, but that could be combined for the same use cases in limited circumstances. The selected combinations are shown in Table 6. To choose some examples from this table: "Notification" and "Alert" share similar characteristics as they are used to inform users/services; "Music" and "Movie" share similar characteristics as they are both for artistic entertainment; "Book" and "Business" are not overtly similar as they do not share similar characteristics, but they are somewhat related because they can be used to build a retail system for book stores; "PDF", "Phone Number" and "Weather" are not similar at all, not only because they do not share similar characteristics, but also because there is no clear microservice use case whereby these would logically be combined.

We use the area label of each API file as its class so that we can conduct the tests. Table 6 shows the homogeneity, completeness, and V-measure of all the combinations. The average homogeneity score is 0.81, which reflects a strong performance in terms of how well the API files within one cluster have the same area label. The average completeness score is 0.713, which reflects a similarly strong performance in terms of how well the approach can include all API files with the same area label into the same cluster.

For dissimilar areas, with average V-measure scores around 0.8, the proposed approach showed an ability to separate microservices from different areas, and put them into matching clusters.

For areas that are similar, the average homogeneity score is significantly higher than the average completeness score. This means that the approach can ensure that the instances inside each cluster are from the same class. However, in the context of very similar areas (e.g. "Notification" and "Alert"), the approach is less effective at ensuring that all of the API files from the same class are put into the same cluster, which is not a particularly surprising result given its difficulty. Nevertheless, despite this issue, the overall V-measure for similar areas remains above 0.71 on average.

For areas that are not overtly similar but can be formed to support certain use cases, the performance is strongly related to the areas inside the combination. For combinations whose areas are very different from any others inside the combination (e.g., "Travel+Vehicle+Hotel", "Music+Membership+User+Payment"), the approach can differentiate API files and put them into the correct clusters. However, if the combination is less diverse (e.g., "Marketing" and "Contact" both can have API files that manage email addresses for different purposes), the approach will be less effective.

The cross-area cluster evaluation demonstrated that the proposed approach can differentiate microservices with different area labels. However, when it comes to combinations with similar area labels, the approach will be less accurate, and sometimes it will put microservices with the same area label into several clusters.

In general, the proposed approach shows the ability to differentiate and cluster similar OpenAPI files correctly. However, highly similar microservice definitions remain difficult to distinguish in some instances. The cause of this situation could be: i) the clustering algorithm used in the proposed approach is less efficient in

multi/single-label classification; ii) the parameters that were passed to the clustering algorithm (e.g., the bandwidth of the clusters) led to overfitting; or iii) the API files that are included in the dataset are rather repetitive and similar, which amplifies this weakness.

5 CONCLUSION AND FUTURE WORK

This paper proposes a similarity metric for OpenAPI-based microservices that can cluster similar microservices together while grouping dissimilar microservices separately. The methodology comprises three steps: preprocessing, file vectorisation, and clustering. First, we obtain descriptive information from OpenAPI files and extract keywords to represent the files. Then, with these keywords, we use GloVe embeddings and SIF to generate file vectors. Finally, we use the Mean Shift algorithm to cluster the file vectors.

The proposed approach was evaluated on a real-world OpenAPI file dataset with more than 400 API files that were collected from different applications. The tests demonstrated that the proposed approach has the ability to differentiate and cluster similar OpenAPI files but occasionally struggles with systems that contain highly similar or repetitive microservices.

In the future, we will work on improving the performance of the approach by adjusting the clustering algorithm and expanding the dataset. We will also examine the application of this approach in the context of microservices management for open systems.

ACKNOWLEDGMENTS

This research is funded under the SFI Strategic Partnerships Programme (16/SPP/3296) and is co-funded by Origin Enterprises Plc.

REFERENCES

[1] Omar Al-Debagy and Péter Martinek. 2020. Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach. In *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*. 289–294. https://doi.org/10.1109/SoSE50414.2020.9130466

[2] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A Simple but Tough-to-Beat Baseline for Sentence Embeddings. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SyK00v5xx

[3] L. Baresi, E. Di Nitto, and C. Ghezzi. 2006. Toward open-world software: Issues and challenges. *Computer* 39, 10 (Oct 2006), 36–43. https://doi.org/10.1109/MC.2006.362

[4] Luciano Baresi, Martin Garriga, and Alan De Renzis. 2017. Microservices Identification Through Interface Analysis. In *Service-Oriented and Cloud Computing*, Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen (Eds.). Springer International Publishing, Cham, 19–33. https://doi.org/10.1007/978-3-319-67262-5_2

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[6] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. OpenAPItoUML: A Tool to Generate UML Models from OpenAPI Definitions. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 487–491. https://doi.org/10.1007/978-3-319-91662-0_41

[7] Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. 2019. Microservices Migration in Industry: Intentions, Strategies, and Challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 481–490. https://doi.org/10.1109/ICSME.2019.00081

[8] Keinosuke Fukunaga and Larry Hostetler. 1975. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on information theory* 21, 1 (1975), 32–40.

[9] Honghao Gao, Xi Qin, Ramón J. Durán Barroso, Walayat Hussain, Yueshen Xu, and Yuyu Yin. 2022. Collaborative Learning-Based Industrial IoT API Recommendation for Software-Defined Devices: The Implicit Knowledge Discovery Perspective. *IEEE Transactions on Emerging Topics in Computational Intelligence* 6, 1 (Feb 2022), 66–76. https://doi.org/10.1109/TETCI.2020.3023155

[10] Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. 2018. Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. 257–259. https://doi.org/10.1109/ICTC.2018.8539409

[11] Bo Jiang, Pengxiang Liu, Ye Wang, and Yezhi Chen. 2020. HyOASAM: A Hybrid Open API Selection Approach for Mashup Development. *Mathematical Problems in Engineering* 2020 (Apr 2020), 4984375. https://doi.org/10.1155/2020/4984375

[12] István Koren and Ralf Klamma. 2018. The Exploitation of OpenAPI Documentation for the Generation of Web Frontends. In *Companion Proceedings of the The Web Conference 2018* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 781–787. https://doi.org/10.1145/3184558.3188740

[13] Banage T. G. S. Kumara, Incheon Paik, Wuhui Chen, and Keun Ho Ryu. 2014. Web Service Clustering Using a Hybrid Term-Similarity Measure with Ontology Learning. *Int. J. Web Serv. Res.* 11, 2 (Apr 2014), 24–45. https://doi.org/10.4018/ijwsr.2014040102

[14] Zhongyi Lu, Declan T. Delaney, and David Lillis. 2023. A Survey on Microservices Trust Models for Open Systems. *IEEE Access* 11 (2023), 28840–28855. https://doi.org/10.1109/ACCESS.2023.3260147

[15] Shapol M. Mohammed, Karwan Jacksi, and Subhi R. M. Zeebaree. 2021. A state-of-the-art survey on semantic similarity for document clustering using GloVe and density-based algorithms. *Indonesian Journal of Electrical Engineering and Computer Science* 22, 1 (Apr 2021), 552–562. https://doi.org/10.11591/ijeecs.v22.i1.pp552-562

[16] Wardani Muhamad, Suhardi, and Yoanes Bandung. 2022. Transforming OpenAPI Specification 3.0 documents into RDF-based semantic web services. *Journal of Big Data* 9, 1 (Apr 2022), 55. https://doi.org/10.1186/s40537-022-00600-8

[17] OpenAPI Initiative. 2023. OpenAPI Specification. https://spec.openapis.org/oas/v3.1.0. Version 3.1.0, Accessed: September 14, 2023.

[18] OpenAPI Initiative. 2023. OpenAPI Specification. https://spec.openapis.org/oas/v3.0.3. Version 3.0.3, Accessed: September 14, 2023.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[20] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. https://doi.org/10.3115/v1/D14-1162

[21] Vinay Raj and S. Ravichandra. 2018. Microservices: A perfect SOA based solution for Enterprise Applications compared to Web Services. In *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*. 1531–1536. https://doi.org/10.1109/RTEICT42901.2018.9012140

[22] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, 3982–3992. https://doi.org/10.18653/v1/D19-1410

[23] Andrew Rosenberg and Julia Hirschberg. 2007. V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Association for Computational Linguistics, Prague, Czech Republic, 410–420. https://aclanthology.org/D07-1043

[24] Munindar P Singh and Michael N Huhns. 2005. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons.

[25] Long Sun, Yan Li, and Raheel Ahmed Memon. 2017. An Open IoT Framework Based on Microservices Architecture. *China Communications* 14, 2 (2017), 154–162. https://doi.org/10.1109/CC.2017.7868163

[26] Xiaoxiao Sun, Salamat Boranbaev, Shicong Han, Huanqiang Wang, and Dongjin Yu. 2022. Expert system for automatic microservices identification using API similarity graph. *Expert Systems* (2022), e13158. https://doi.org/10.1111/exsy.13158

[27] Aimilios Tzavaras, Nikolaos Mainas, and Euripides G. M. Petrakis. 2023. OpenAPI framework for the Web of Things. *Internet of Things* 21 (Apr 2023), 100675. https://doi.org/10.1016/j.iot.2022.100675

[28] Congcong Wang and David Lillis. 2020. A Comparative Study on Word Embeddings in Deep Learning for Text Classification. In *Proceedings of the 4th International Conference on Natural Language Processing and Information Retrieval (NLPIR 2020)*. Seoul, South Korea. https://doi.org/10.1145/3443279.3443304

[29] Yueshen Xu, Yinchen Wu, Honghao Gao, Shengli Song, Yuyu Yin, and Xichu Xiao. 2021. Collaborative APIs recommendation for Artificial Intelligence of Things with Information Fusion. *Future Generation Computer Systems* 125 (Dec 2021), 471–479. https://doi.org/10.1016/j.future.2021.07.004

[30] Olaf Zimmermann. 2017. Microservices Tenets. *Computer Science - Research and Development* 32, 3 (Jul 2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0