SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

Managing Microservices For Open Systems

Zhongyi LU[1], Declan T. DELANEY[2] & David LILLIS[1*]

[1]*School of Computer Science, University College Dublin, Dublin 4, D04 V1W8, Ireland*
[2]*School of Electrical and Electronic Engineering, University College Dublin, Dublin, D04 V1W8, Ireland*

Abstract The evolution of the digital era has increased the need for open microservice systems that are intelligent and allow arbitrary microservice behaviour. This requires systems to have a strong ability to manage microservices. A common microservice management strategy is allocating microservices into domains. Traditional domain-management strategies, which group microservices by predefined criteria, fail in these open environments as they require prior system knowledge. To address this, we propose a novel domain-management framework that autonomously constructs "Similarity Domains" based on microservice functionality. The framework computes the functional similarity between microservices and builds domains that store similar microservices together. Our framework first employs Natural Language Processing (NLP) techniques to generate semantic embeddings from OpenAPI documentation, effectively capturing functional semantics. These embeddings are then clustered to form domains of functionally similar microservices without any prior knowledge, enabling resilient service discovery and seamless integration of newcomers. Novel evaluation methods were designed for the proposed framework. Evaluations conducted on real-world OpenAPI documents, as well as through simulations of an open microservice system, demonstrate that the proposed approach performs effectively compared to existing domain-management strategies, facilitating appropriate service discovery, domain formation, and integration of newcomer microservices.

Keywords Microservices, Open Systems, Microservice Management, NLP, Intelligent Systems.

Citation Managing Microservices For Open Systems. Sci China Inf Sci, for review

1 Introduction

As the digital era evolves, intelligent systems are increasingly deployed across diverse sectors and must support autonomous and unbounded scaling. Such "open" systems [1,2], operate in continuously changing and unpredictable environments without fixed boundaries.

Open systems do not focus on specific tasks and can react to environments that are changing constantly, unlimitedly, and unanticipatedly [3]. With the growth of modern technology being used in daily lives, the world can be seen as an expanding open system. Digital devices, software, and systems are part of this open and highly dynamic system. Unlike closed systems where the boundaries are controlled, open systems may contain entities with various responsibilities and behaviours. Traditional monolithic systems have long been considered unfit for open systems, as they lack resilience, scalability, maintainability, etc. Meanwhile, Microservice architectures (MSA), being an extended notion of Service-Oriented Programming (SOA), are showing great strength in scalability and flexibility. The core concept of MSA is to divide a large system into many small, lightweight microservices and provide services by working cohesively via messaging (usually through RESTful API) [4]. The adoption of MSA has been a long-established and major trend in the software engineering industry. Microservices have no fixed boundaries and support a wide range of use cases. In such environments, microservices can join or leave freely, regardless of their ownership or origin. Therefore, some companies and researchers have formed the view that MSA can be the solution to building open systems, such as online business systems, transportation systems, and IoT-based applications [5,6]. Although this kind of design enables scalability, it also creates challenges in the effective management and discovery of microservices [2].

Beyond discovery, handling missing microservices is another challenge. Microservices may be missing intentionally if the owner shuts them down or unintentionally if there are network problems, system crashes, or other errors. If one microservice is missing, the management system should be able to quickly discover a potential substitute for the missing microservice. This requires the system to know the functionality of each microservice and be aware of all microservices that provide the same service without needing to know the business logic, as the system has no boundaries.

* Corresponding author (email: david.lillis@ucd.ie)

These challenges highlight the importance of designing a lightweight and adaptable microservice management framework for open systems that can be easily integrated across different contexts and support dynamism, which is needed for intelligent systems. In many microservices architectures, microservices are organised into domains. Domains are organisational units that form systems [7] and allow practitioners to group microservices based on some type of shared characteristic and improve system understanding [8]. The use of domains has been shown to increase scalability, ease maintenance and updates, improve communications, and decentralise data management, which are beneficial to open systems [7, 8].

The definition of domains can be narrow or broad, but typically are organised independently [9]. In practice, Domains may be based on aspects like bounded context [10], business logic [7], or trust levels [11]. Both bounded context and business logic models are based on the business areas that the system covers. They are the most adopted domain management approach in the industry [12]. However, they are designed for closed systems, where the business scope and the services that systems provide can be determined before system construction. Trust-level-based domains require knowledge of the trustworthiness of microservices within the system to allocate them to their matching domain, which makes allocating new microservices with no established trust difficult. Moreover, since the trust-level-based method classifies microservices only by their trust levels, the functionalities of microservices within each domain are mixed, which brings more challenges to microservice retrieval. All these reasons indicate that existing domain management approaches are not able to cope with the open-world settings and the highly dynamic environment of open microservices systems. A domain management approach that can fully support open systems is needed to support the coming wave of the digital era.

This paper proposes a microservice management framework for open systems that groups microservices into what we term "Similarity Domains", based on the semantic descriptions of their functionality. The semantic descriptions can be obtained from API documentation, which is required to be open to others. The proposed framework comprises: i) a microservice embedding method that uses Natural Language Processing (NLP) techniques to extract useful information from the entirety of OpenAPI documentation[1)] of microservices and vectorises the entire OpenAPI documentation with pre-trained word embeddings; and ii) a microservice management approach that clusters microservice embeddings to form Similarity Domains, enabling the system to efficiently locate services that match a given semantic request or offer a suitable alternative to a missing microservice. A novel simulation-based evaluation using OpenAPI-based microservices is conducted. Our framework is demonstrated to be effective in this environment, specifically for newcomer discovery, service management, and handling missing microservices.

The main contributions of this paper are summarised as follows:

- We propose a **semantic microservice embedding method** that represents the functionality of a microservice by jointly encoding the full OpenAPI documentation using NLP techniques and pre-trained word embeddings. Unlike prior studies that measure similarity only among endpoints within a single service, our method enables *cross-service semantic similarity computation* between independent microservices.
- We design a **Similarity Domain–based management framework** for open microservice systems that clusters functionally related services according to their semantic embeddings. This formulation removes the need for predefined business boundaries or trust assumptions and, to the best of our knowledge, is the first domain management approach specifically tailored for *open-world microservice environments*.
- We construct a **new OpenAPI microservice dataset and two evaluation benchmarks** that support (i) semantic similarity measurement and (ii) domain management tasks such as service discovery, newcomer integration, and missing-service substitution, enabling systematic and reproducible evaluation of open microservice management methods.
- Through a **simulation-based empirical study**, we demonstrate that the proposed framework effectively improves service discovery and substitution performance in dynamic open-system scenarios.

The paper is structured as follows: Section 2 reviews related work; Section 3 presents the proposed method; Section 4 reports the evaluation; and Section 5 concludes the paper.

1) https://www.openapis.org/

2 Related Work

2.1 Open Microservice Systems

Microservice architecture (MSA) is an evolution of SOA [13]. Applications are composed of fine-grained, single-responsibility microservices [14], which are operationally independent [13]. Building on MSA, this paper adopts Lu et al.'s [15] definition of Open MSA: systems where heterogeneous, autonomous microservices can join or leave freely, regardless of ownership or origin. Such open systems require enhanced management capabilities, including resilience to missing services, rapid onboarding of newcomers, and fast retrieval [2]. However, existing definitions primarily characterise the problem space rather than providing concrete mechanisms for scalable service discovery and organisation in highly dynamic environments.

2.2 Managing Microservices

While research on closed-system microservice management, ranging from etcd-based self-management [16] and template-driven architectures [17] to Cloud Foundry-based containerisation [18] and reinforcement learning agents like RESP [19], offers insights for open systems, these methods face significant limitations. They often rely on rigid, pre-defined rules that fail to adapt to diverse failure modes or require extensive, environment-specific retraining for learning-based models, making them difficult to scale across complex, dynamic microservice architectures.

Domain-centric management defines microservice boundaries based on application areas, such as bounded contexts [7], business logic [10], or multi-tree architectures [20], while others utilise trust-based classifications evaluated by third-party brokers [11]. However, these strategies are primarily static, often failing to adapt to fluid runtime dependencies or requiring extensive domain expertise that is difficult to scale.

Recent work integrates Large Language Models (LLMs) into microservice management, with agents like ServiceOdyssey [21] employing curriculum learning and AdaptiFlow [22] using event-driven reasoning to orchestrate scaling and fault recovery. However, despite their advanced reasoning, these frameworks often lack the formal safety guarantees and deterministic reliability required for mission-critical production environments due to potential LLM hallucinations.

2.3 API Similarity

Previous work has explored computing functional similarity between API endpoints using semantic representations. A common approach represents words as vectors, with vector similarity reflecting API similarity. Xu et al. [23] trained embeddings via word2vec, Jiang et al. [24] computed API file similarity through pairwise semantic comparisons, and Gao et al. [25] embedded words from API introductions into feature vectors.

Some methods filter irrelevant information before computation. Al-Debagy et al. [26] extracted operation names and parameters, converted them to fastText embeddings, and clustered similar operations using Affinity Propagation, with cosine similarity measuring API similarity.

Other approaches cluster microservices by semantic similarity. Sun et al. [27] use weighted API similarity graphs, pruning low-weight edges. Baresi et al. [28] cluster HTTP operations via name-vocabulary comparison and DISCO schema matching. Nita [29] leverages ontological knowledge and graph-based similarity for API endpoint clustering.

3 The proposed framework

3.1 An Embedding Method For OpenAPI-Based Microservices

The embedding method (Figure 1) comprises two major steps: preprocessing and vectorisation.

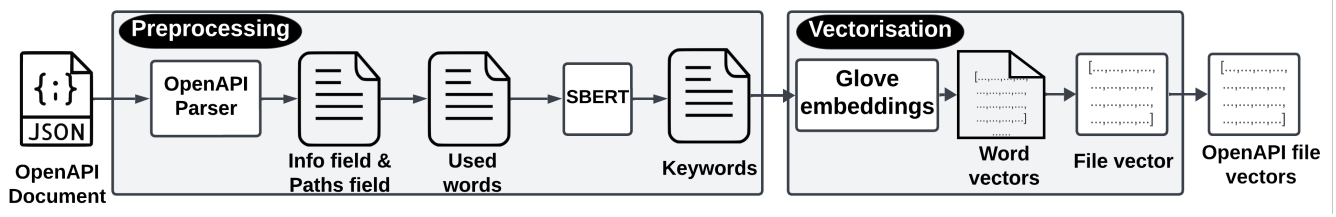

Figure 1 Embedding an OpenAPI document.

3.1.1 *Preprocessing*

Key Information Extraction The OpenAPI Specification (OAS) [30] is the standard for API documentation [31]. This study focuses on OAS version 3.0 [32], which remains widely adopted, while selectively extracting useful fields from later versions. OpenAPI definitions in JSON are parsed using the Swagger Parser[2)].

An OAS 3.0 document has eight fixed fields. The functional information is extracted from two fields: `info` and `paths`. The `info` field provides metadata. We use fields `title` and `description` for descriptive information. Another field included in OAS 3.1, `summary`, is also selected. The `paths` field describes endpoint operations, including `$ref` (URL), *summary*, *description*, and HTTP methods. We focus on POST, GET, PUT, and DELETE (CRUD) operations, extracting each operation's *summary*, *description*, and *operationId* to capture its functionality.

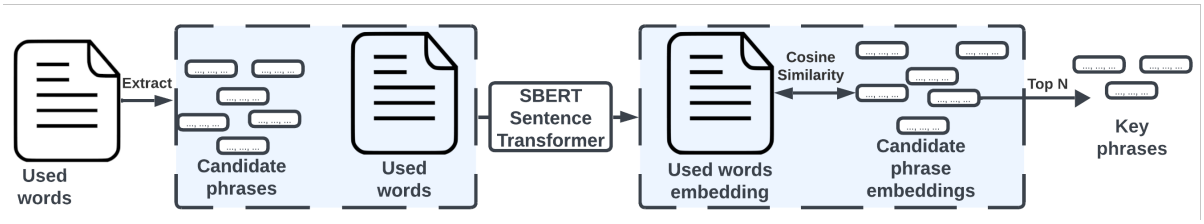

Figure 2 Keyword extraction process.

Keywords Extraction To capture microservice semantics, the source file undergoes a cleaning process where URLs, symbols, and formatting are removed, and multi-word identifiers (camel, snake, or pascal case) are split. These extracted terms are then mapped to vector representations using a pre-trained embedding vocabulary. While our specific implementation utilises GloVe [33] embeddings (300-dimensional vectors trained on the Common Crawl dataset) for their balance of performance and reproducibility, this framework is embedding-agnostic and can be integrated with any word embedding model. To ensure comprehensive keyword coverage, we handle unrecognisable words, especially lowercase compound words, with the OpenAI API (GPT-4-Turbo [3)]) to split these compound words into their constituent parts. These words are then re-mapped to the embedding space; remaining unrecognised terms are excluded from further analysis.

Keyword extraction proceeds in three main steps. First, candidate keyword phrases are identified from the extracted words using CountVectorizer [34], producing trigram phrases. Trigrams were chosen because longer phrases often add redundancy without improving semantic coverage, while shorter phrases (unigrams and bigrams) may fail to capture sufficient context; Section 4.2.2 shows trigrams achieve higher V-measure scores. Next, both candidate phrases and all extracted words are embedded using Sentence-BERT (SBERT) [35, 36] with the '`distilbert-base-nli-mean-tokens`' model to obtain 768-dimensional vectors, allowing accurate semantic comparison between phrases and the full API context. Finally, cosine similarity is computed between each candidate phrase and the sum of all extracted word embeddings, and the top N phrases ($N = 5$) are selected as representative keywords. The individual words within these phrases are defined as the primary keywords, capturing the core semantics of the API documentation (Figure 2).

3.1.2 *Vectorisation*

First, each extracted keyword is mapped to a dense semantic vector using a pre-trained embedding model.

Next, to create a single, meaningful representation for each API file, a file embedding is computed by averaging the word vectors of its extracted keywords, applying weights to enhance semantic meaning. This weighted average uses the Smooth Inverse Frequency (SIF) method [37], which assigns a weight to each word w based on its estimated frequency $p(w)$, defined as $a/(a + p(w))$, where a is a constant (set to 0.001). The word frequency corpus, $p(w)$, used in this research to calculate these weights is derived from the Wikimedia Database backup dumps[4)], which contain over 2 million unique English terms, ensuring accurate weighting for the final file embedding.

The calculation of an API file vector V_{file} is computed as shown in Equation 1, where N denotes the number of keywords in the file, $\mathbf{V}_{\text{word},i}$ represents the GloVe embedding of the i-th keyword, and $\mathbf{w}_{\text{word},i}$ is the weight of the i-th keyword.

2) https://github.com/swagger-api/swagger-parser
3) https://platform.openai.com/docs/models/gpt-4-turbo
4) https://dumps.wikimedia.org/backup-index.html

$$\mathrm{V}_{\mathrm{file}} = \frac{\sum_{i=0}^{N} V_{\mathrm{word},i} \times w_{\mathrm{word},i}}{N}. \tag{1}$$

Our model-agnostic embedding pipeline supports any encoder that maps OpenAPI docs to dense vectors. While we utilise GloVe and SBERT for efficiency, these can be replaced by LLM-based embeddings without altering the core clustering or domain management processes. Consequently, future advances in embedding technology will enhance the framework without requiring structural changes.

3.2 Microservice Management Using Similarity Domains

Based on OpenAPI descriptions, this work introduces "Similarity Domains", which are groups of microservices with similar functionality. This concept facilitates management in open systems by narrowing the search space for service discovery and simplifying the replacement of missing services, while retaining the benefits of traditional domains.

As illustrated in Figure 3, the system includes one Similarity Evaluator (Section 3.2.2), one Domain Manager (Section 3.2.3), and multiple Similarity Domain (Section 3.2.1). When a new microservice is uploaded, the Similarity Evaluator identifies its appropriate Similarity Domain. The Domain Manager then assigns the microservice to that domain and handles all subsequent domain management tasks. Both the Evaluator and Manager are implemented as microservices for easy integration into existing systems.

Microservices are dynamically grouped into Similarity Domains via clustering. In open systems where services frequently join and leave, these domains can experience topic drift, making periodic reclustering necessary. A key challenge is balancing the cost of frequent reclustering against the assignment errors caused by infrequent updates, especially as the system's size and stability evolve.

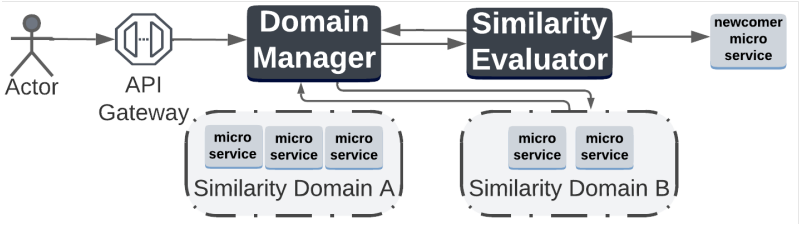

Figure 3 Illustration of clustering newcomer microservices into Similarity Domains.

3.2.1 *Similarity Domains*

Microservices within each domain should be connected, focused, and functionally similar to improve retrieval efficiency. As shown in Figure 3, microservices are clustered by functional similarity into Similarity Domains, formed by the Similarity Evaluator and managed by the Domain Manager.

Each domain is represented by a centroid, which is the average of its microservice vectors; and a list of keywords, which is the top N frequent words from microservice descriptions. The centroid helps match new microservices without reclustering, while keywords match client requests.

The primary role of Similarity Domains is to support the discovery of functionally similar microservices. The prototype uses SQLite in-memory databases[5)] to keep all data in RAM, with periodic snapshots for persistence. Although persistent storage may introduce additional latency, the system's core functionality is unaffected. Each domain stores server URLs, keywords, and domain IDs in a table that is synchronised with the Domain Manager to keep management information current. Using in-memory storage simplifies implementation, accelerates prototyping and avoids overhead. The architecture remains modular and database-agnostic, allowing seamless migration to industrial databases with minimal modification. In practice, the database should be selected according to the actual situation.

To keep each Similarity Domain's database up to date, the system uses Prometheus [38]'s Alert Manager. The alert `PrometheusTargetMissing` marks unresponsive microservices as missing, and its information is then removed from the corresponding domain database.

5) https://www.sqlite.org/

3.2.2 *Similarity Evaluator*

The Similarity Evaluator embeds newcomers and assigns them to matching Similarity Domains, following four main steps: embedding, domain matching, allocation decision, and updating the nearest domain.

In a new system (Figure 4), the Domain Manager creates an initial domain for the newcomer. In a running system (Figure 5), the evaluator first checks if reclustering is needed. If so, all microservices are reclustered, and the updates are sent to the Domain Manager. Otherwise, the newcomer is assigned to the nearest domain based on centroid distance, updating the domain's centroid and keywords.

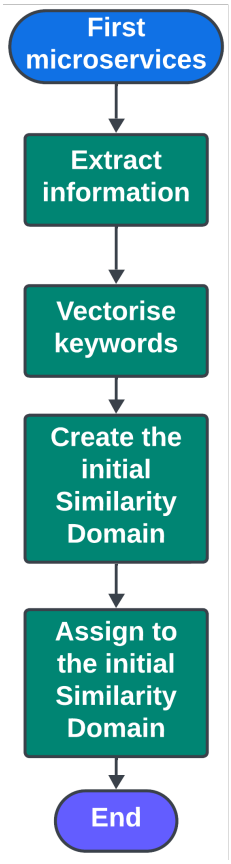

Figure 4 Process to start a new system with its first microservice.

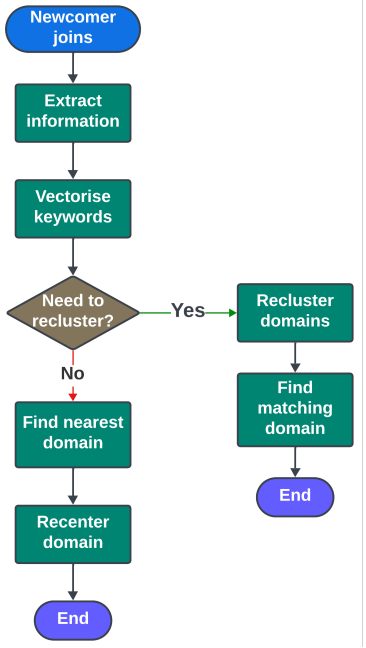

Figure 5 Process to accommodate a newcomer microservice.

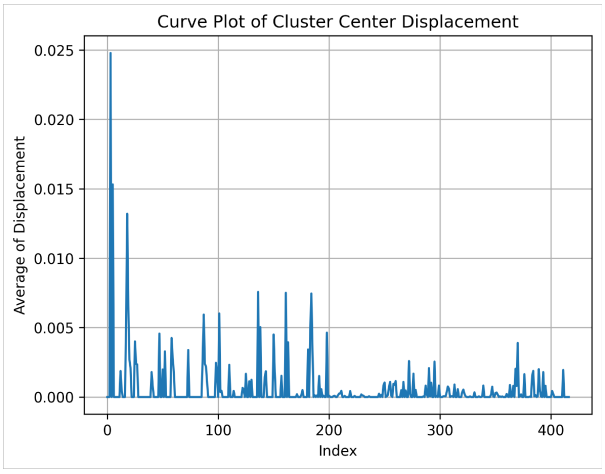

Figure 6 The displacement of the centre of clusters

Embedding: Newcomer microservices are embedded using the method introduced in Section 3.1.

Decision on Reclustering: Reclustering for every new microservice is impractical. Therefore, it occurs after every n additions, with n adjusted based on system size. Smaller systems recluster more often, whereas larger systems require less. If reclustering is unnecessary, the newcomer is assigned to the nearest domain; otherwise, a full reclustering is performed.

To determine the frequency, we experimented with real-world OpenAPI microservices (Section 4). After each addition, all embeddings were reclustered, and the Euclidean distance between the new and nearest old centroids was measured. As shown in Figure 6, average displacement was initially high due to unstable clusters but decreased as the system grew, with occasional spikes. Reclustering is therefore periodic: more frequent in early stages and less so as the system stabilises. The Similarity Evaluator sets a dynamic n that grows with system size M (Equation 2), which is empirically tuned based on the average displacement. These breakpoints were obtained by inspecting the average displacement trends in Figure 6 to identify stable regimes, and were validated through small local perturbations to ensure robustness.

$$n = \begin{cases} 1 & \text{if } M < 30 \\ 10 & \text{if } 30 <= M < 100 \\ 25 & \text{if } 100 <= M < 200 \\ 100 & \text{if } 200 <= M. \end{cases} \tag{2}$$

Clustering: Density-based clustering is common for document clustering [39], with methods like HDBSCAN, DBSCAN, and OPTICS. Mean Shift [40] is selected based on the conduction of initial trials to confirm the structure of the algorithms. Mean Shift does not require a predefined number of clusters and includes all points in clusters, ensuring all microservices are assigned. It identifies clusters by finding dense neighbourhoods, with the mean representing the cluster centre. However, other clustering algorithms are also compatible, as the datasets may vary. Implementation uses `MeanShift` from *sklearn.cluster* [34].

Finding Matching Domains After Clustering: After clustering, the Domain Manager is updated with the newcomer's assigned domain, which may differ from the initial match. Dense domains remain stable, while splits can occur in sparser regions.

Finding the Nearest Domain and Recentring: Each Similarity Domain has a centroid, which is the mean of its microservice embeddings. A newcomer is matched to the domain with the nearest centroid (Euclidean distance). After insertion, the centroid is updated, and the domain's ID, centroid, and keywords are synchronised with the Domain Manager and the Similarity Evaluator's database.

3.2.3 *Domain Manager*

The Domain Manager is a microservice that manages all Domains and handles requests that actors send to the API Gateway for specific functionalities. Its first role is domain management. When newcomers join, it updates domains: if reclustering occurs, affected domain databases are replaced. Their top-X keywords, domain IDs, and centroids are recorded. If no reclustering is needed, only the matching domain is updated with the newcomer's info. Its second role is request handling. Actor requests, sent as JSON, are tokenised and cleaned using NLTK [41], then vectorised with spaCy's '`en_core_web_md` ' into 300-dimensional vectors. Domain vectors are averaged from keywords, and cosine similarity identifies the most relevant domain. A microservice from that domain is returned by randomly choosing if multiple options exist, allowing the actor to invoke it directly.

4 Evaluation

We adopt a two-phase evaluation: Phase one tests the embedding method on real-world OpenAPI data, simulating closed and open systems. Phase two evaluates the microservice domain management approach in a simulated environment, measuring robustness and scalability against baselines.

4.1 Data Collection

We collected a real-world OpenAPI dataset[6] from APIs.guru[7], GitHub[8], and SwaggerHub[9]. To ensure microservices' single-responsibility focus, files describing multiple responsibilities—such as full products (e.g., Google Maps) or multifunction services (e.g., managing multiple lists)—were manually removed. The final dataset contains 424 OpenAPI files conforming to OAS v3.0 or later, which helps approximate different service scenarios. While runtime load is not explicitly modeled, our focus is on API structure and behaviour.

Each file is annotated with three labels for evaluation:

- **Action Label**: A verb describing the core functionality of the microservice, derived from descriptive info and HTTP endpoints; one per API file.
- **Entity Labels**: The entities affected by the action; multiple per API file.
- **Area Label**: The application domain of the microservice; one per API file.

Table 1 shows example labels for selected OpenAPI files.

The dataset includes 50 unique area labels. Table 2 presents their distribution, showing coverage of most microservice application areas.

6) https://github.com/zhongyilucy/OpenAPIMicroserviceFiles.git
7) https://apis.guru/
8) https://github.com
9) https://swagger.io/tools/swaggerhub/

Table 1 The labels of selected OpenAPI files.

File	Action	Entity	Area
Train Station arrival.json	Get	Train, Arrival	Transport
Flight Offers Price.json	Get	Flight, Offer	Transport
Branded Fares Upsell.json	Get	Flight, Offer	Transport
API Documentation for Train Schedule.json	Get	Train, Schedule	Transport
Airports API v2.json	Find	Airport	Transport
Flight Offers Search.json	Search	Flight, Offer	Transport

Table 2 Ground truth labels in the dataset.

Area	Count	Area	Count	Area	Count	Area	Count	Area	Count
Account	5	Business	56	HR	17	News	4	Search	4
Address	2	Claims	2	Image	9	Notification	9	Security	21
Alert	4	Code	2	Language	9	Payment	11	System	10
API	3	Communication	9	Local Info	10	PDF	13	Time	3
Application	2	Contact	4	Logistics	2	Phone number	8	Transport	41
Article	6	Data	7	Marketing	4	Postcode	2	Travel	5
Audio	2	Document	6	Membership	3	Recipe	3	User	27
Banking	24	Facility	4	Movie	6	Referral	3	Vehicle	4
Blog	3	Finance	8	Music	11	School	2	Weather	5
Book	11	Hotel	11	Name	3	Science	2	Web	2

4.2 Embedding Method Evaluation

We introduce a novel evaluation method for embedding entire OpenAPI documents to capture microservice functionality and identify similar services, differing from prior work that compares single endpoints. Differences include: i) An OpenAPI document is formed by multiple components, including one or more endpoints and other metadata, whereas the former only includes the information of a single endpoint; ii) under the setting of an open environment, different OpenAPI documents can be written by different developers in various writing styles and programming manners, whereas one OpenAPI file is typically written by one developer or a developing group following a consensual writing style; iii) The context of a microservice application cannot be fully described by the information provided by one endpoint, which is often described in other components inside the API file; and iv) it is common that multiple endpoints are needed to accomplish the single responsibility of one microservice, which means API endpoints cannot be treated as isolated individuals. As the method operates at a different level, no baseline models are used; the focus is on intrinsic analysis of its ability to cluster functionally similar microservices.

Effectiveness for Similarity Domains is evaluated via clustering: if embeddings capture functionality, algorithms like MeanShift should group similar microservices. The dataset from Section 4.1 is used.

Two evaluations are conducted: single-area and cross-area clustering. Single-area groups API files within the same industrial area, resembling multi-label classification, since files can have multiple entity labels. Cross-area test systems spanning multiple, possibly unrelated areas, a single-label classification, as each file has one area label. This assesses the method in both domain-specific and open-system scenarios.

4.2.1 *Single-Area Cluster Evaluation*

The single-area cluster evaluation simulates systems focused on one industrial area. API files are grouped by area label and then clustered to group microservices operating on similar entities while separating those with different entities. Each file's entity labels indicate the entities its microservice handles.

Evaluation relies on API file entity labels per area. Accuracy (0–1) measures how well embeddings cluster files with the same entities while separating different ones. For area A, the score is the mean of the macro F1 score $\mathrm{MF1}_A$ and the duplication score dup_A(Equation 3).

$$\mathrm{Accuracy}_A = \frac{\mathrm{MF1}_A + \mathrm{dup}_A}{2}, \tag{3}$$

The macro F1 score(0-1) for area A, $\mathrm{MF1}_A$, is calculated as Equation 4. Each cluster i has an F1 score $\mathrm{F1}_{\mathrm{Cluster},i}$,

reflecting how well API files with similar entity labels are grouped and the overall clustering accuracy within the area.

$$\mathrm{MF1}_A = \sum_{i=0}^{N} \frac{\mathrm{F1}_{\mathrm{Cluster},i}}{N}. \tag{4}$$

A cluster's F1 score is computed by assigning it labels based on its API files' entity labels. For area A, the cluster's labels Labels_A include any entity present in $\geqslant$50% of its files; if none meet this, the most common label(s) are used. Multiple labels are allowed if they meet the threshold or tie, ensuring accurate F1 calculation.

Each API file's entity labels are compared with its cluster labels. The cluster F1 score is calculated using standard machine learning definitions: an entity label is True Positive (TP) if it appears in the cluster labels, False Positive (FP) if it does not, and False Negative (FN) if a cluster label is missing from the API file's entity labels. These cluster F1 scores are then used to compute the area's macro F1 score.

In Table 1, all API files form one cluster, and no label appears in over half the files. The cluster labels are "Flight" and "Offer", the most frequent labels. Table 3 shows the TP, FP, and FN counts, used to compute the cluster F1 score, which contributes to the macro F1 of "Transport".

Table 3 *True Positives*, *False Positives*, and *False Negatives* of example API files

File	TP	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
Flight Offers Search.json	2	0	0

After calculating an area's macro F1, its duplication score dup_A (0–1) is computed. A score of 0 means all clusters share the same labels; 1 means all labels are distinct. For area A with M API files in N clusters, each cluster i has $\mathrm{Label}_{\mathrm{total}}$ (all labels) and $\mathrm{Label}_{\mathrm{unique}}$ (labels exclusive to it). The duplication score is the weighted average of $\frac{|\mathrm{Label}_{\mathrm{unique}}|}{|\mathrm{Label}_{\mathrm{total}}|}$, weighted by cluster size. A higher score reflects greater label uniqueness and less functional overlap.

$$\mathrm{dup}_A = \sum_{i=0}^{N} \frac{|\mathrm{Cluster}_i|}{M} \times \frac{|\mathrm{Label}_{\mathrm{unique},i}|}{|\mathrm{Label}_{\mathrm{total},i}|}. \tag{5}$$

Single-area evaluations are done only for areas with more than eight API files (the dataset's average size). Table 4 shows macro F1, duplication, and accuracy scores.

Table 4 Precision, Recall, and F1-Scores within single areas.

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

Macro F1 scores indicate the approach effectively groups most API files with the same entity labels, with only three areas below 0.8. Duplication scores are less consistent: roughly one-third of areas show significant label overlap (scores under 0.2), often where many files share a common label (e.g., User, Image) or only a few files have unique labels (e.g., Notification, Book). For instance, in the User area, all microservices operate on the same entity, so clusters overlap, and the duplication score is 0. Areas with more diverse entity labels achieve higher duplication

scores, reflecting better separation. Overall, embeddings perform best when microservices in a single area operate on different entities.

The single-area evaluation shows the embedding method can distinguish API files with different entity labels. However, for highly similar microservices, it sometimes overfits, causing files with the same entity labels to be inconsistently clustered.

4.2.2 *Cross-Area Cluster Evaluation*

The cross-area evaluation tests whether API files with the same area label are correctly clustered. Each microservice has only one area label, making this a single-label classification task.

V-measure [42] is used for cross-area cluster evaluation. It is the harmonic mean of homogeneity (how much each cluster contains microservices from the same area) and completeness (how well microservices of the same area are grouped together). V-measure ranges from 0 to 1, with higher values indicating more accurate clustering.

Table 5 The Homogeneity (H), Completeness (C), and V-Measure (V) scores of different combinations.

	Combination	H	C	V
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

To evaluate cross-area clustering, three scenarios of increasing difficulty were designed based on human-judged similarity: (i) areas with no shared characteristics, (ii)clearly similar areas, and (iii) areas not directly similar but potentially co-occurring. The evaluation includes five unrelated, five similar, and four partially related area groups. Examples in Table 5 include: "Notification" and "Alert" (similar in purpose), "Music" and "Movie" (both entertainment-related), "Book" and "Business" (not similar but can co-exist in retail scenarios), and "PDF," "Phone Number," and "Weather" (entirely unrelated). These scenarios allow assessment of the method across varied levels of similarity and complexity.

Each API file's area label is used as its class. As shown in Table 5, the approach achieves an average homogeneity of 0.81 and an average completeness of 0.713, indicating strong cluster purity and class cohesion across different area combinations. The V-measure, as the harmonic mean of these two metrics, summarises the embedding method's overall effectiveness in capturing microservice functionality.

For dissimilar area combinations, the average V-measure is around 0.8, demonstrating that microservices are well separated even when areas share little similarity. For similar areas, homogeneity exceeds completeness, showing that clusters are internally consistent. Though closely related areas (e.g., "Notification" and "Alert") may not fully consolidate into a single cluster, still, V-measure remains above 0.71. For areas that are not directly similar but can co-occur, clustering performance varies with combination diversity: distinct combinations (e.g., "Travel+Vehicle+Hotel") are well clustered, whereas less diverse or overlapping sets (e.g., "Marketing+Contact+Data") show reduced separation.

The cross-area evaluation shows that the embedding method effectively separates microservices with different area labels. For closely related areas, however, microservices with the same label may be split across clusters, highlighting a limitation in clustering precision for overlapping or similar areas.

Overall, the embedding method effectively differentiates and clusters similar OpenAPI files in both closed and open systems. Distinguishing highly similar microservices remains challenging, possibly due to the clustering algorithm's limitations, parameter choices (e.g., cluster bandwidth) leading to overfitting, or the presence of repetitive,

highly similar API files in the dataset.

Table 6 V-Measure (V) scores of different combinations under N-gram settings.

	Combination	unigram	bigram	trigram
Not Similar	Transport+Banking(+User)	0.000	0.000	0.778
	Blog+Recipe+News+Contact+Science	0.128	0.667	0.918
	Code+Communication+Facility	0.417	0.492	0.686
	PDF+Phone_Number+Weather	0.477	0.611	0.941
	Time+Music+Membership+Marketing	0.128	0.499	0.768
Similar	Alert+Notification	0.277	0.501	0.472
	Movies+Music	0.829	1.000	0.809
	Contact+Communication	0.201	0.074	0.520
	PDF+Image	0.817	1.000	1.000
	Book+Article(s)	0.152	0.129	0.768
Not overtly similar	Travel+Vehicle+Hotel	0.343	0.697	0.941
	Contact+Data+Marketing	0.448	0.146	0.465
	Book+Business	0.000	0.644	0.695
	Music+Membership+User+Payment	0.000	0.000	0.727

Selection of N-gram We compared unigram, bigram, and trigram settings using V-measure (Table 6). Overall, trigrams achieve the highest V-measure across categories, indicating they capture more specific semantic patterns while avoiding redundancy, and are therefore chosen for keyword extraction.

4.3 Microservice Domain Management Approach Evaluation

Having shown in Section 4.2 that our embedding method effectively clusters similar microservices, we now evaluate its impact on microservice domain management.

To address the lack of benchmarks for open systems, we propose a novel evaluation method simulating dynamic microservice environments. The evaluation uses the same 424 OpenAPI files across 50 industrial areas from the embedding study, representing synthetic microservices that accept inputs and return predefined responses. This allows assessment of domain management performance without executing actual business logic. All microservices run on localhost with unique ports, and an API Gateway handles actor interactions. Experiments were conducted on a MacBook Pro with an Apple M1 Pro (10 cores, 32GB RAM) running macOS Sequoia 15.0.1.

4.3.1 *Evaluation Scenarios*

This model aims to be resilient to missing microservices and to be capable of retrieving a suitable microservice as quickly as possible. To test these two features, we designed two scenarios:

Increasing microservices: The system operates without missing microservices. We incrementally add one group of 12 microservices at a time in no particular order to simulate continuous microservice uploads. The group size (i.e., 12) is determined by dividing the total number of microservices by the number of increments that result in significant centroid displacement. After each batch, a predefined request (i.e., "Account Lookup") is submitted to measure retrieval speed. This setup allows the clustering algorithm to stabilise while tracking performance across different system sizes. Predefined requests are used only for standardization and do not affect the evaluation results.

Missing microservices: After populating the system, 13 microservices matching the "Account Lookup" request are removed one by one to test resilience and substitution capability. After each removal, a request is submitted to measure the system's response time in identifying and retrieving a suitable alternative, assuming at least one replacement microservice exists.

Performance is measured by request response time (from the API Gateway receiving a request to the Domain Manager returning a recommendation), CPU, memory, and system load. Prometheus monitors these metrics in both scenarios to evaluate the model's efficiency.

4.3.2 *Comparison With Prior Works*

The proposed model is benchmarked against Bounded Context and Trust-Level-Based approaches, both of which rely on predefined boundaries that limit their utility in open systems. To ensure a fair comparison, all models were implemented with identical system-level configurations. While the proposed model utilises an in-memory SQLite database, the baselines were equivalently configured using hash-based dictionaries in memory to eliminate I/O-related bias. By removing disk-based lookups from all processes, the resulting data on latency, CPU usage, and system load reflects differences in retrieval logic and algorithmic complexity rather than storage overhead.

The Bounded Context model is widely used in industry and targets closed systems. Following the design principles from [7] and Conway's Law [43], microservices are organised into 50 domains corresponding to the area labels in the OpenAPI dataset. The architecture features a frontend for keyword-based request processing (same as Section 3.2.3) and a backend that utilises an in-memory dictionary for domain matching. Since bounded contexts define domains rather than specific functions, the system finalises selection by matching request embeddings to the most relevant microservice within the identified domain.

The Trust Level model is based on Azarmi et al. [11] and also targets closed systems. Trust levels of microservices are randomly assigned as "certified", "trusted", or "untrusted". A trust broker, implemented as an in-memory dictionary storing domain IDs and trust levels, manages these domains. This trust broker is also used in trust management, which is not in the scope of this paper. Clients send requests to the API Gateway specifying the desired functionality and trust level. The system first selects a candidate domain matching the trust level, then identifies the microservice with the closest functional match, using the same method as the Bounded Context model.

For completeness, we also considered trivial non-semantic strategies such as random or round-robin service selection. These approaches ignore service functionality and effectively reduce to random matching among all available microservices. In an open system with N services and only k functionally valid substitutes, the expected success probability of random selection is k/N. In our dataset ($N = 424$, $k \approx 13$ for the "Account Lookup" functionality), this corresponds to an expected success rate of approximately 3%, implying that tens of trials would be required on average before locating a valid service. Such behaviour would incur substantially higher latency than all evaluated domain-based approaches and therefore represents a trivial lower bound rather than a meaningful competitor. Consequently, we focus our empirical comparison on stronger and practically relevant management strategies.

The proposed approach, bounded-context approach, and trust-level approach are all tested with the same fixed request for an "Account Lookup" microservice. The proposed approach is expected to have the shortest retrieval time and CPU usage in both test scenarios due to its lower complexity. It is expected to have a higher memory load as it is heavily reliant on in-memory databases. The other two approaches, having similar complexity, are expected to yield comparable results.

4.3.3 *Results*

Expansion of Microservices As described in Section 4.3.1, we measure system speed using average request processing time. For each batch of 12 microservices, we execute the same request 1,000 times, reaching full population after 35 rounds. We compare our proposed Similarity Domains (Sim.) approach against two baselines: bounded-context (B.C.) and trust-level (Trust) models. We anticipate the Sim. model will yield the shortest processing times, lower CPU usage, and lower system load, albeit with higher memory consumption. Results are detailed in Table 7.

The proposed Similarity Domain model consistently outperformed the baseline approaches in processing time. It initially handled requests in 0.004 seconds, compared with over 0.14 seconds for both baseline models. As the system scaled, this gap widened: by the final round, the Similarity model required only 0.038 seconds, whereas the bounded-context and trust-level models reached approximately 4.27 and 4.30 seconds, respectively. Despite a 34-fold increase in system size, the Similarity model's processing time increased by only a factor of 6, compared with 29 and 30-fold increases for the baseline approaches. This significant efficiency gain stems from its design, which groups functionally similar microservices and minimises retrieval loops, resulting in substantially faster response times under load.

In terms of CPU usage, the Similarity model averaged 10.621%, markedly lower than the bounded-context (17.723%) and trust-level (18.859%) models. Lower CPU usage indicates more efficient computation, reduced energy consumption [44], and increased stability. The lower standard deviation observed in the Similarity model confirms that its CPU usage is more predictable and consistent.

The Similarity model exhibited the highest average memory usage due to its caching mechanism, which enables rapid access to domain information. Nonetheless, memory consumption remained within acceptable limits for

Table 7 Combined performance metrics of different models across microservice groups. **Sim.** represents the proposed approach.

Group	Processing Time (s)			CPU Usage (%)			Memory Usage (MB)			System Load (Processes)		
	Sim.	B.C.	Trust	**Sim.**	B.C.	Trust	**Sim.**	B.C.	Trust	**Sim.**	B.C.	Trust
Group1	0.004	0.144	0.140	10.15	11.35	10.50	43.211	43.789	34.367	6.797	3.898	2.907
Group2	0.006	0.302	0.292	13.00	15.45	8.15	43.305	43.797	37.477	4.760	3.677	2.601
Group3	0.006	0.355	0.351	14.75	13.85	15.30	43.383	43.797	37.477	4.543	3.647	3.020
Group4	0.007	0.451	0.426	11.45	19.50	23.25	43.391	43.797	37.477	4.249	4.235	5.412
Group5	0.009	0.513	0.507	17.05	22.80	17.00	43.414	43.805	37.461	4.011	4.392	3.992
Group6	0.010	0.570	0.570	9.40	13.10	15.85	43.453	41.547	37.469	3.457	4.104	4.198
Group7	0.011	0.623	0.633	9.45	19.45	17.50	43.492	41.500	37.477	3.834	4.589	4.503
Group8	0.012	0.939	0.943	13.20	18.40	18.05	43.555	41.492	37.531	3.677	4.926	4.705
Group9	0.013	1.106	1.118	9.05	18.90	18.50	43.547	41.492	37.531	3.540	4.956	4.175
Group10	0.015	1.208	1.221	13.25	19.85	18.15	43.539	41.492	37.563	3.995	5.101	4.389
Group11	0.018	1.409	1.421	8.55	17.15	15.35	43.547	41.492	37.563	3.296	5.377	4.649
Group12	0.019	1.467	1.470	8.20	20.90	14.85	43.555	41.492	37.602	3.626	5.172	4.255
Group13	0.019	1.538	1.549	9.80	23.30	6.10	43.563	41.500	37.617	3.842	4.505	2.066
Group14	0.022	1.850	1.957	10.90	22.05	18.50	43.563	41.516	37.734	2.806	5.002	4.028
Group15	0.023	2.326	2.347	10.05	19.95	22.60	43.578	41.539	37.719	3.071	4.371	4.464
Group16	0.022	2.388	2.220	9.60	26.70	25.25	43.602	40.211	37.719	3.296	5.299	5.254
Group17	0.021	2.098	2.120	10.70	17.15	22.75	43.602	40.227	37.750	3.510	3.503	4.154
Group18	0.022	2.229	2.262	9.25	19.05	7.40	43.633	40.242	37.750	3.384	3.787	2.856
Group19	0.024	2.329	2.338	9.00	23.80	9.40	43.664	40.641	37.781	3.922	4.664	2.990
Group20	0.026	2.391	2.434	11.60	17.85	13.00	43.680	40.063	37.789	3.073	4.196	3.849
Group21	0.026	2.517	2.570	9.20	22.40	23.55	43.703	40.094	37.727	3.228	4.069	3.353
Group22	0.029	2.632	2.690	9.40	16.80	21.00	43.672	37.633	37.500	3.078	5.128	3.827
Group23	0.032	2.990	3.022	9.80	16.70	20.80	43.672	37.742	35.375	3.301	4.165	4.839
Group24	0.032	3.127	3.167	10.55	14.40	23.25	43.680	36.758	34.945	3.284	3.640	5.308
Group25	0.030	2.987	3.019	10.20	14.20	23.60	43.688	38.500	34.828	3.497	3.016	4.832
Group26	0.031	3.046	3.104	10.60	13.40	16.60	43.688	38.492	36.969	3.000	3.091	5.553
Group27	0.032	3.118	3.152	9.80	21.30	15.65	43.695	38.531	37.078	3.379	3.691	3.827
Group28	0.033	3.196	3.246	11.15	13.75	15.00	43.695	38.539	37.133	3.562	3.367	3.554
Group29	0.034	3.285	3.370	9.10	13.15	22.70	43.695	38.539	37.102	4.025	3.038	5.498
Group30	0.035	3.360	3.425	9.40	11.55	22.75	43.711	38.633	37.086	3.153	2.390	4.611
Group31	0.035	3.421	3.485	10.40	11.45	21.80	43.727	38.672	35.453	5.023	3.538	5.809
Group32	0.036	4.618	5.164	13.30	16.85	14.25	43.727	38.656	36.859	4.233	2.639	2.983
Group33	0.037	4.033	4.203	9.85	17.20	11.10	43.734	38.094	36.961	3.846	3.523	3.372
Group34	0.037	4.773	4.791	9.10	19.95	5.75	43.742	38.234	36.992	4.186	3.423	2.349
Group35	0.038	4.266	4.301	11.50	16.65	14.80	43.758	36.188	36.977	3.681	4.057	4.671
Avg	0.023	2.217	2.258	10.621	17.723	18.859	43.596	40.250	37.081	3.748	4.062	4.082
Std Dev	0.011	1.294	1.347	1.891	3.855	5.475	0.133	2.115	0.919	0.735	0.779	0.969

contemporary systems. By contrast, fluctuations observed in the bounded-context model's memory usage may be attributed to background processes active during the experiment.

The Similarity model also exhibited the lowest system load, indicating lower demand on computational resources and less power to operate effectively. It is worth noting that variations in system load may partly result from concurrent background processes.

Overall, the proposed model improves microservice retrieval and management more effectively than existing models, delivering better performance with lower resource consumption.

Reduction of Microservices Results of microservice reduction are presented in Table 8.

The Similarity Domain model outperformed the others in recovery time, averaging 0.036 seconds—less than 1%

Table 8 Impact of removing microservices on system performance metrics. **Sim.** represents the proposed approach.

	Recovery Time (s)			CPU Usage (%)			Memory Usage (MB)			System Load (Process)		
Removal	**Sim.**	B.C.	Trust	**Sim.**	B.C.	Trust	**Sim.**	B.C.	Trust	**Sim.**	B.C.	Trust
1	0.037	5.063	5.758	13.100	19.150	13.350	41.250	33.758	38.180	2.705	4.970	3.524
2	0.037	4.205	6.978	11.850	18.950	25.750	39.094	35.281	37.297	4.684	4.966	5.567
3	0.036	4.252	4.292	8.950	19.350	16.250	39.102	35.234	34.898	3.691	3.680	4.419
4	0.036	5.435	4.274	8.800	32.150	6.550	39.117	35.492	35.875	3.383	7.090	2.659
5	0.036	4.185	4.236	9.900	29.200	14.200	39.133	34.617	35.352	3.346	4.666	4.037
6	0.036	4.171	4.285	10.600	20.450	23.100	39.156	36.039	34.867	3.729	4.628	6.457
7	0.038	4.249	5.413	10.700	29.550	13.650	39.156	39.766	32.648	3.318	4.863	3.951
8	0.036	5.325	4.194	10.650	27.450	16.800	39.164	39.203	33.750	3.941	5.792	4.243
9	0.036	6.700	6.868	9.650	19.000	8.200	39.164	37.234	31.938	3.457	3.964	2.656
10	0.036	4.533	5.051	9.350	33.800	14.500	39.172	37.164	31.805	3.322	7.969	3.490
11	0.038	4.122	4.196	9.650	23.200	22.650	39.180	36.727	32.016	3.222	5.175	4.654
12	0.036	4.133	5.192	11.050	31.050	14.850	39.180	37.281	33.758	2.923	6.040	3.700
13	0.036	4.080	7.337	9.900	23.800	22.300	39.180	37.328	36.203	2.931	6.668	5.116
Avg	0.036	4.650	5.236	10.320	25.160	16.320	39.311	36.548	34.507	3.435	5.421	4.190
Std Dev	0.001	0.805	1.112	1.217	5.961	6.052	0.547	1.760	2.089	0.541	1.289	1.134

of the bounded-context (4.65s) and trust-level (5.236s) models. The trust-level model was slightly slower and less stable due to coarser domains, which shorten microservice selection but lengthen domain matching. Some outliers exhibited processing times longer than 5 seconds: Removals 1, 4, 8, and 9 for the bounded-context model and Removals 1, 2, 7, 9, 10, 12, and 13 for the trust-level model. These outliers may be due to unusually short or long retrieval loops after microservice removal or fluctuations in background system activity during testing, as the results represent averages over multiple runs.

Despite these outliers, the Similarity Domain model consistently outperforms the others. The bounded-context and trust-level models require two steps: locating the domain and then the substitute. The Similarity Domain model pre-groups similar microservices, enabling faster recovery.

The performance of the proposed model, bounded-context model, and trust-level model during the removal experiment mirrors the trends seen in the microservice expansion test. The Similarity Domain model consistently used less than half the CPU of the other models, with lower variability, highlighting its efficiency and predictability. Memory usage was higher due to in-memory caching, but remained within acceptable limits for all models. System load was consistently lowest for the Similarity Domain model, indicating it places less strain on the system than the alternatives.

In conclusion, the proposed model outperforms baseline approaches in recovering from missing microservices and consistently improves CPU usage, memory usage, and system load, demonstrating its effectiveness and suitability for managing dynamic microservices in open systems.

4.3.4 *Impact of Latency*

Experiments were conducted on a localhost deployment to isolate the computational cost of retrieval. In real open systems, additional network delays between the API Gateway, Domain Manager, and microservices will increase absolute response times for all approaches. However, this overhead is largely orthogonal to the retrieval strategy and acts as a common baseline. Methods requiring multiple sequential lookups or filtering stages would incur extra network hops, whereas the proposed approach minimises routing steps. Consequently, although absolute latencies may rise in distributed deployments, the relative performance advantages are expected to remain consistent.

4.3.5 *Scalability Analysis*

Our evaluation uses 424 OpenAPI specifications. Although moderate in size, the framework is designed to generalise to substantially larger systems. The pipeline comprises three stages: embedding generation, clustering, and runtime retrieval. Embedding generation scales linearly with the number of services ($O(N)$). Clustering is performed offline or incrementally and does not affect request latency. At runtime, discovery is restricted to the candidate similarity

domain rather than the full service set, reducing lookup from $O(N)$ to approximately $O(k)$, where $k \ll N$. Empirical results confirm this behaviour: increasing the number of services by 34× raises average processing time by only 6× while memory usage remains nearly constant, indicating near-linear scalability and supporting deployments with thousands of microservices.

5 Conclusion and Future Work

Open intelligent systems require infrastructures that support continuous scaling, dynamic participation, and autonomous evolution. However, existing microservice management approaches largely assume closed-world boundaries or pre-established trust, limiting their suitability for open environments. This paper presents a similarity-based management framework that organises services according to functional similarity derived from OpenAPI specifications, enabling scalable discovery and substitution without fixed contexts or prior trust relationships.

A two-phase evaluation demonstrates both effectiveness and efficiency. Clustering experiments on 424 OpenAPI specifications show that the embedding approach reliably groups functionally similar services, while system-level simulations indicate lower latency, faster recovery, and reduced CPU usage compared with bounded-context and trust-based baselines. Embedding generation scales linearly ($O(N)$), and runtime discovery is restricted to local similarity domains ($O(k)$, $k \ll N$), yielding near-linear performance as the number of services increases.

The current prototype is evaluated mainly in simulation, where services return predefined responses. Although this isolates discovery behaviour, it does not capture real-world invocation issues such as parameter or protocol compatibility. Future work will therefore focus on distributed deployment, incremental clustering, fault tolerance, and semantic interface matching to validate the framework with large-scale real-world microservices and ensure correct end-to-end service interaction.

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

References

1 Macrae C. Managing risk and resilience in autonomous and intelligent systems: Exploring safety in the development, deployment, and use of artificial intelligence in healthcare. Risk Analysis, 2025, 45: 910–927
2 Lu Z, Delaney D T, Lillis D. A Survey on Microservices Trust Models for Open Systems. IEEE Access, 2023, 11: 28840–28855
3 Baresi L, Di Nitto E, Ghezzi C. Toward open-world software: Issues and challenges. Computer, 2006, 39: 36–43
4 Alshuqayran N, Ali N, Evans R. A systematic mapping study in microservice architecture. In: Proceedings of 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), 2016. 44-51
5 Fritzsch J, Bogner J, Wagner S, et al. Microservices Migration in Industry: Intentions, Strategies, and Challenges. In: Proceedings of 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019. 481-490
6 Sun L, Li Y, Memon R A. An Open IoT Framework Based on Microservices Architecture. China Communications, 2017, 14: 154–162
7 Steinegger R H, Giessler P, Hippchen B, et al. Overview of a domain-driven design approach to build microservice-based applications. In: Proceedings of The Third International Conference on Advances and Trends in Software Engineering (SOFTENG 2017), 2017
8 Zhong C, Li S, Huang H, et al. Domain-driven design for microservices: An evidence-based investigation. IEEE Transactions on Software Engineering, 2024, 50: 1425–1449
9 Hassan S, Ali N, Bahsoon R. Microservice ambients: An architectural meta-modelling approach for microservice granularity. In: Proceedings of 2017 IEEE International Conference on Software Architecture (ICSA), 2017. 1-10
10 Rademacher F, Sorgalla J, Sachweh S. Challenges of domain-driven microservice design: A model-driven perspective. IEEE Software, 2018, 35: 36–43
11 Azarmi M, Bhargava B, Angin P, et al. An End-to-End Security Auditing Approach for Service Oriented Architectures. In: Proceedings of 2012 IEEE 31st Symposium on Reliable Distributed Systems, 2012. 279-284
12 Gandhi J, Medicherla R K, Naik R. Domain aligned microservices decomposition. In: Proceedings of Proceedings of the 18th Innovations in Software Engineering Conference, New York, NY, USA: Association for Computing Machinery, 2025
13 Neri D, Soldani J, Zimmermann O, et al. Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Software-Intensive Cyber-Physical Systems, 2020, 35: 3–15
14 Dragoni N, Lanese I, Larsen S T, et al. Microservices: How to make your application scale. In: Proceedings of Petrenko A K, Voronkov A, editors, Perspectives of System Informatics, Cham: Springer International Publishing, 2018. 95–104
15 Lu Z, Delaney D T, Lillis D. Comparing the Similarity of OpenAPI-Based Microservices. In: Proceedings of Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, New York, NY, USA: Association for Computing Machinery, 2024. 1201–1208
16 Toffetti G, Brunner S, Blöchlinger M, et al. An architecture for self-managing microservices. In: Proceedings of Proceedings of the 1st International Workshop on Automated Incident Management in Cloud, New York, NY, USA: Association for Computing Machinery, 2015. 19–24
17 Wizenty P, Sorgalla J, Rademacher F, et al. MAGMA: build management-based generation of microservice infrastructures. In: Proceedings of Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, New York, NY, USA: Association for Computing Machinery, 2017. 61–65
18 Andry F, Ridolfo R, Huffman J. Migrating Healthcare Applications to the Cloud through Containerization and Service Brokering. In: Proceedings of Proceedings of the International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 5, Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, 2015. 164–171
19 Zou Y, Qi N, Deng Y, et al. Autonomous resource management in microservice systems via reinforcement learning. In: Proceedings of 2025 8th International Conference on Computer Information Science and Application Technology (CISAT), 2025. 991-995
20 Le D M, Dang D H, Vo H D. Layered microservices architecture: A multitree-based domain-driven approach. Information and Software Technology, 2025, 183: 107720
21 Yu F, Yang F, Qin X, et al. Enabling autonomic microservice management through self-learning agents, 2025
22 Zemtsop Ndadji B A, Bliudze S, Quinton C. Adaptiflow: An extensible framework for event-driven autonomy in cloud microservices. Electronic Proceedings in Theoretical Computer Science, 2025, 438: 123–147

23 Xu Y, Wu Y, Gao H, et al. Collaborative APIs recommendation for Artificial Intelligence of Things with Information Fusion. Future Generation Computer Systems, 2021, 125: 471–479
24 Jiang B, Liu P, Wang Y, et al. HyOASAM: A Hybrid Open API Selection Approach for Mashup Development. Mathematical Problems in Engineering, 2020, 2020: 4984375
25 Gao H, Qin X, Barroso R J D, et al. Collaborative Learning-Based Industrial IoT API Recommendation for Software-Defined Devices: The Implicit Knowledge Discovery Perspective. IEEE Transactions on Emerging Topics in Computational Intelligence, 2022, 6: 66–76
26 Al-Debagy O, Martinek P. Extracting Microservices' Candidates from Monolithic Applications: Interface Analysis and Evaluation Metrics Approach. In: Proceedings of 2020 IEEE 15th International Conference of System of Systems Engineering (SoSE), 2020. 289–294
27 Sun X, Boranbaev S, Han S, et al. Expert system for automatic microservices identification using API similarity graph. Expert Systems, 2022, page e13158
28 Baresi L, Garriga M, De Renzis A. Microservices Identification Through Interface Analysis. In: Proceedings of De Paoli F, Schulte S, Broch Johnsen E, editors, Service-Oriented and Cloud Computing, Cham: Springer International Publishing, 2017. 19–33
29 Nita E J. Ontology-aware matching algorithms for automated api composition using graph-based semantic similarity in heterogeneous web services. International Journal of Information Technology Research and Development (IJITRD), 2025, 6: 1–7
30 OpenAPI Initiative. OpenAPI Specification. https://spec.openapis.org/oas/v3.1.0, 2023. Version 3.1.0, Accessed: September 14, 2023
31 Lazar K, Vetzler M, Kate K, et al. Generating OpenAPI specifications from online API documentation with large language models. In: Proceedings of Rehm G, Li Y, editors, Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), Vienna, Austria: Association for Computational Linguistics, 2025. 237–253
32 OpenAPI Initiative. OpenAPI Specification. https://spec.openapis.org/oas/v3.0.3, 2023. Version 3.0.3, Accessed: September 14, 2023
33 Pennington J, Socher R, Manning C D. GloVe: Global Vectors for Word Representation. In: Proceedings of Empirical Methods in Natural Language Processing (EMNLP), 2014. 1532–1543
34 Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 2011, 12: 2825–2830
35 Reimers N, Gurevych I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In: Proceedings of 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, 2019. 3982–3992
36 Devlin J, Chang M W, Lee K, et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Proceedings of 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, 2019. 4171–4186
37 Arora S, Liang Y, Ma T. A Simple but Tough-to-Beat Baseline for Sentence Embeddings. In: Proceedings of International Conference on Learning Representations, 2017
38 Rabenstein B, Volz J. Prometheus: A Next-Generation Monitoring System (Talk). Dublin: USENIX Association, 2015
39 M. Mohammed S, Jacksi K, R. M. Zeebaree S. 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, 2021, 22: 552–562
40 Fukunaga K, Hostetler L. The estimation of the gradient of a density function, with applications in pattern recognition. IEEE Transactions on information theory, 1975, 21: 32–40
41 Bird S, Klein E, Loper E. Natural language processing with Python: analyzing text with the natural language toolkit. O'Reilly Media, 2009
42 Rosenberg A, Hirschberg J. V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure. In: Proceedings of Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), Prague, Czech Republic: Association for Computational Linguistics, 2007. 410–420
43 Conway M E. How do committees invent? Datamation, 1968, 14: 28–31
44 Weiser M, Welch B, Demers A, et al. Scheduling for Reduced CPU Energy, pages 449–471. Springer US, Boston, MA, 1996