Automated Clustering of Microservices Using Natural Language Processing and Clustering Algorithms

Santiago di Sabato
ISISTAN/UNICEN
Tandil, Buenos Aires, Argentina
santiagodisabatto@gmail.com

Guillermo Rodríguez
ISISTAN/UNICEN
Tandil, Buenos Aires, Argentina
exarodriguez@gmail.com

Claudia A. Marcos
ISISTAN/UNICEN
Tandil, Buenos Aires, Argentina
cmarcos@exa.unicen.edu.ar

Santiago Vidal
ISISTAN/UNICEN
Tandil, Buenos Aires, Argentina
svidal@exa.unicen.edu.ar

José Renan A. Pereira
VIRTUS/UFCG
Federal University of Campina Grande (UFCG), Paraiba, Brazil
jose.pereira@embedded.ufcg.edu.br

Danyllo Albuquerque
VIRTUS/UFCG
Federal University of Campina Grande (UFCG), Paraiba, Brazil
danyllo.albuquerque@virtus.ufcg.edu.br

Mirko Perkusich
VIRTUS/UFCG
Federal University of Campina Grande (UFCG), Paraiba, Brazil
mirko@virtus.ufcg.edu.br

ABSTRACT

Microservices have become a leading architectural style for creating scalable systems. Designing these architectures involves breaking down large APIs into well-defined service boundaries. While many existing clustering techniques rely on having access to the complete source code, API specifications (such as OpenAPI) are often more readily available in practice. This is particularly true when code is distributed across different teams, linked to legacy systems, or expensive to analyze. However, API specifications are often poorly documented, which hampers the effectiveness of clustering. Large Language Models (LLMs) present a promising solution by inferring semantic relationships even from sparse or incomplete descriptions. This paper introduces *ODAM* (OpenAPI Documentation Analysis and Modeling), a Python-based pipeline that *(i)* fills missing endpoint descriptions with ChatGPT, *(ii)* embeds all texts, *(iii)* performs an intermediate topic induction, and *(iv)* feeds the resulting vectors to either K-Means or hierarchical clustering. We investigated the ODAM's feasibility using a real-world API (Twilio, 45 microservices), comparing six pipeline variants—three intermediate strategies crossed with two description settings—against an expert-defined reference, using four evaluation criteria: success rate, failure rate, Silhouette score, and execution time. The results showed the LLM-augmented pipeline achieved a *64%* success rate—34% higher than the best non-LLM baseline—and uncovered five coherent business domains at k=5, where Silhouette peaked at *0.55*. Statistical pipelines ran in under 100 seconds, while the LLM-enhanced version took 1.6 hours, but dropped to under one minute with cached descriptions. Final clustering quality remained stable across K-Means and hierarchical algorithms, with 0.05 variation in Silhouette scores. Overall, injecting LLM-generated semantics into sparsely documented APIs materially improves microservice clustering accuracy and exposes high-level capabilities; practitioners can trade runtime for precision via a simple cache toggle.

KEYWORDS

Software Architecture, LLM, Software Modernization, Microservices, Architectural Decomposition

1 Introduction

Modern software systems increasingly adopt microservice architectures to achieve scalability, modularity, and ease of maintenance [1]. Nevertheless, many organizations still operate monolithic systems or legacy APIs that have evolved organically over time, leading to unclear service boundaries and undocumented functionalities. This complexity hinders efforts to understand, maintain, and modernize software portfolios [20]. In practice, the code underlying these systems may be fragmented across multiple repositories, maintained by different teams, tied to outdated build processes, or subject to access restrictions—making direct source code analysis costly or impractical. In such contexts, the API specification (e.g., OpenAPI) often serves as the most accessible, standardized artifact from which to reason about system capabilities.

Clustering microservices into coherent groups is a key step toward simplifying these systems, yet it remains a challenging task [24]. Manual decomposition is time-consuming and error-prone [27]. Existing automated methods often assume full access to source code or richly documented OpenAPI endpoints—conditions rarely met even within a single organization [18, 22]. Real-world scenarios include post-merger system integration, governance of large API portfolios across distributed teams, and preliminary audits of legacy systems where code exists but is prohibitively expensive to analyze in depth. These situations motivate the need for tools capable of extracting, enriching, and clustering service descriptions from limited or incomplete system information.

In this paper, we explore whether Large Language Models (LLMs), particularly ChatGPT, can bridge documentation gaps and improve the automated functional grouping of microservices [13, 14]. To that end, we introduce ODAM (OpenAPI Documentation Analysis and Modeling), a Python-based pipeline that performs: (i) automatic completion of missing descriptions via ChatGPT, (ii) linguistic preprocessing using SpaCy, (iii) semantic vectorization and topic induction, and (iv) clustering using traditional algorithms such as K-Means or hierarchical clustering.

We evaluated ODAM on the Twilio REST API—a public and heterogeneous system encompassing 45 microservices—using expert-curated groupings as the reference standard. The experiments tested various configurations of intermediate strategies and clustering algorithms [17], with and without LLM-based completion. Our findings demonstrate that LLM-augmented pipelines significantly outperform traditional approaches, achieving up to 64% correct pairings and uncovering high-level functional domains with minimal manual intervention.

The contributions of this paper are threefold: (i) an end-to-end, reproducible pipeline that integrates LLMs with NLP and clustering techniques for service grouping; (ii) an empirical evaluation comparing six pipeline variants, highlighting accuracy–cost trade-offs; and (iii) practical guidelines for selecting clustering strategies based on service characteristics and architectural goals.

Overall, ODAM offers a scalable and effective solution for microservice clustering in scenarios where source code is unavailable or documentation is sparse. It enables practitioners to recover functional groupings and gain architectural insights while also contributing to ongoing research on LLM-driven software engineering automation.

The remainder of this paper is structured as follows. Section 2 reviews foundational concepts and related work. Section 3 introduces our microservice-identification approach, and Section 4 details the validation methodology. Section 5 presents the experimental results and expert evaluation, while Section 6 discusses implications for research and practice. Section 7 addresses threats to validity. Finally, Section 8 concludes the paper and highlights directions for future research.

2 Fundamentals

In this section, we first outline the core principles of Microservices. Then, we review traditional and LLM-based approaches, highlighting how our study uniquely addresses functional grouping from textual requirements.

Microservice foundations Modern web applications must ingest large data volumes and support rich user interactions while remaining flexible and highly scalable [16]. Early systems met these needs with a *monolithic* style, placing all functionality in a single deployable unit [9]. Although monoliths simplify testing and initial deployment, their tightly coupled code bases hamper scalability, reliability, and technology evolution [1]. To overcome such limits, the community embraced the *microservice architecture*: a system is decomposed into many small, autonomous services, each owning a well-defined business capability [20, 25]. A typical deployment might assign user authentication to one service and e-mail handling to another, each documented through OpenAPI specifications that expose endpoints, parameters, and usage contracts [11].

While microservices solve several monolithic pain points, they introduce new challenges—chiefly the operational overhead of deploying, synchronizing, and monitoring dozens (or hundreds) of independent services [15]. As a system grows, architects must decide *how* to partition these services into clusters that share infrastructure—optimising resource use, reducing inter-service latency for frequent collaborators, and trimming maintenance costs relative to "one host per service" deployments [7].

Related work. Several research efforts offer solutions to the problem of clustering or otherwise organizing microservices based on functional or structural similarity. Traditional NLP-centric approaches remain influential: Al-Debagy et al.[5] proposed decomposing monoliths via semantic similarity combined with hierarchical clustering over OpenAPI descriptions, exploiting fastText and Word2Vec embeddings to infer the number of clusters automatically. Baresi, Garriga, and De Renzis [8] also rely on semantic analysis of OpenAPI specifications but emphasise carefully defined interface boundaries to achieve an appropriate granularity of candidate services. Subsequently, Al-Debagy et al.[4] introduced quality metrics such as the Service Granularity Metric (SGM) and the Lack of Cohesion Metric (LCOM) for assessing microservice designs produced by API analysis. Rocha Araujo et al. [12] moved away from monolith-to-microservice decomposition and instead applied topic-modelling (LDA, LSI, NMF) to cluster API descriptions directly, laying the groundwork for grouping functionally related services without access to source code.

More recently, LLM-driven approaches have begun to augment or replace classical NLP pipelines. Adams et al.[2] showed that ChatGPT-4 can scaffold a complete SockShop-style microservice system, automatically producing Spring-Boot services and controllers, though human validation is still required to fix outdated dependencies and handle context-window limits. Stojanović and Lazarević [23] demonstrated that GPT-3.5 can propose plausible service boundaries directly from textual requirements, making it useful in early architectural design while still demanding expert review to avoid oversimplifications. Quevedo et al.[19] went a step further by injecting static-analysis artefacts—such as call graphs—into ChatGPT prompts; the enriched context markedly improved answers to architecture-level queries about existing microservice systems. Alsayed et al.[6] introduced MicroRec, a dual-encoder LLM framework that leverages Stack Overflow posts, Dockerfiles, and README files to recommend relevant microservices, achieving up to 14× higher precision than traditional search. Chauhan et al.[10] presented an API-first generator in which an LLM iteratively refines server-side code derived from an OpenAPI definition, closing the loop with log analysis to accelerate prototyping. Complementing these automation-focused studies, Ahmad et al.[3] explored human–bot collaborative architecting, outlining roles and hand-off patterns that let architects steer ChatGPT while the model handles repetitive design tasks—underscoring that effective microservice design will likely emerge from synergistic workflows rather than full autonomy.

Taken together, these LLM-centric studies push the state of the art beyond classical clustering by: (i) automating the generation and iterative refinement of microservice code, (ii) enriching architectural analyses with artifacts drawn from the running system (e.g., call graphs, logs), and (iii) supporting search, recommendation and discovery in large service ecosystems. At the same time, they expose fresh challenges—dependency drift, context-window limits, and the persistent need for expert oversight—that argue for hybrid pipelines combining traditional similarity metrics with LLM-powered semantic reasoning. Our work positions itself within this trajectory: we strive to deliver service clusters that capture both functional intent and structural cohesion, while keeping manual intervention to a minimum.

3 Proposal of the Microservice Identification Process

This study explores the feasibility of leveraging LLMs to derive functional groupings from textual software requirements. To support this investigation, we introduce ODAM, a Python-based pipeline that analyzes and organizes the textual content of microservice OpenAPI specifications. The goal is to cluster services based on functional similarity, enabling structured understanding and comparison of service roles. ODAM enables the identification of semantic relationships between services, facilitating their management, maintenance, and evolution within a microservices-based architecture.

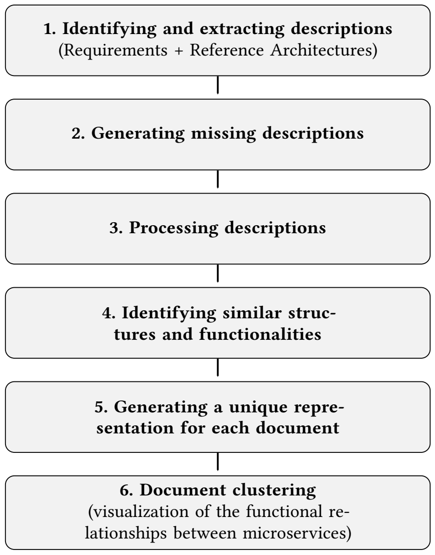

1. Identifying and extracting descriptions
(Requirements + Reference Architectures)
2. Generating missing descriptions
3. Processing descriptions
4. Identifying similar structures and functionalities
5. Generating a unique representation for each document
6. Document clustering
(visualization of the functional relationships between microservices)

Figure 1: Methodological Workflow.

ODAM proceeds through six consecutive stages that match those in Figure 1. It begins by identifying and extracting every textual description found in the OpenAPI specifications, requirements, and reference architectures (Step 1). When a description is missing, the tool automatically generates a concise replacement with ChatGPT, using endpoint metadata (resource name, HTTP verb, and parameters) as context (Step 2). All collected texts then undergo lightweight NLP pre-processing—tokenisation, stop-word removal, and lemmatisation—to yield a clean lexical representation (Step 3). Next, the processed descriptions are embedded and grouped to uncover recurrent functional themes, thereby revealing similar structures and behaviors (Step 4). Each microservice is subsequently encoded as a single vector that aggregates its membership across those themes, providing a compact numerical portrait of its functionality (Step 5). In the final step, these vectors are fed to either K-Means or hierarchical clustering to form the definitive microservice groups (Step 6). The following subsections detail every step.

Step 1: Identifying and Extracting Descriptions

The first stage of our approach consists of identifying and extracting the textual descriptions associated with the methods defined in each OpenAPI specification file. This procedure involves systematically traversing all routes declared in the specifications, collecting the available descriptions at each endpoint, regardless of the microservice to which they originally belong. All extracted descriptions are stored in a common set, which facilitates their subsequent treatment from a unified perspective.

Each endpoint in the OpenAPI specification contains metadata that describes its functionality, with the description section being a key source of semantic information. This stage is critical, as the collected descriptions constitute the textual basis from which vectorization and, later, clustering algorithms will be applied. Descriptions were chosen as the primary input for the process due to their high informational value. These descriptions contain structural information about the service and references to its behavior and functional purpose.

This semantic richness enables a more precise representation of each microservice's characteristics, facilitating the identification of functional similarities in subsequent stages. Figure 2 illustrates this procedure, showing how descriptions extracted from different specification files are consolidated into a single set, which will be used as the basis for the semantic analysis and clustering phases.

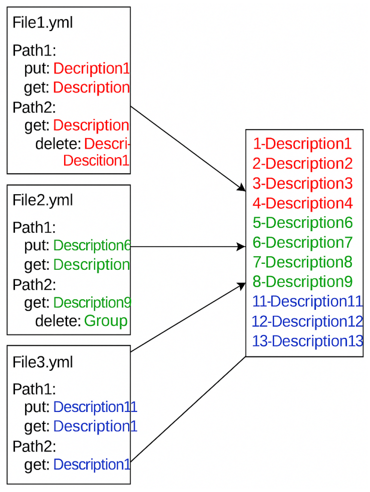


Figure 2: Extraction of descriptions from OpenAPI documents.

Step 2: Generating Missing Descriptions

In many cases, OpenAPI documents lack descriptions for the defined methods, either due to developer oversight or the use of automatic tools that do not include this section [8]. This gap represents a significant limitation, as descriptions are a key source of semantic information. To address this issue, a module for automatic description generation using ChatGPT is incorporated, a natural language model capable of generating coherent text based on the provided context.

ChatGPT usage is enabled by including a personal authentication key in the configuration file. Through a predefined prompt, the model is requested to generate a brief and representative description of a specific endpoint's functionality. Here is an example of a predefined prompt:

Predefined prompt

```
Given the following endpoint from an OpenAPI file,
generate a description that briefly explains the endpoint's
functionality. The description should be generated as a single
sentence.
```

Then, ODAM generates this description:

Generated Description

```
The /pet endpoint with the PUT method allows modifying an
existing pet in the database, using information from the
request in JSON or XML format, and requires authentication to
access its write and read operations.
```

All generated descriptions are stored in a "GeneratedDescriptions.txt" file, where each path and its corresponding generated description are specified.

Step 3: Processing Descriptions

To optimize the semantic analysis of descriptions, they undergo a linguistic pre-processing procedure using the SpaCy library, a high-performance tool for natural language processing in Python. The objective is to extract the most representative terms from each description to facilitate subsequent clustering. The processing consists of four stages:

- Removal of stop-words.
- Calculation of word frequency.
- Selection of the most relevant terms.
- Restructuring of the description based on these terms.

For example, an initial description such as the below example is transformed through this process into a compact representation, including terms such as fax, endpoint, allow, user, access, manipulate, send, receive, highlighting both structural and functional aspects of the service.

Initial Description

```
This endpoint allows users to access and manipulate fax
resources in order to send, receive, and manage faxes using
the Twilio API.
```

Step 4: Identifying Similar Structures and Functionalities

Once processed, the descriptions are grouped based on their semantic similarity in a process called Identifying Similar Structures and Functionalities. This intermediate clustering does not consider the source of the descriptions, treating them as a single set. The goal is to identify common functional patterns that allow characterizing microservices in more abstract and homogeneous terms. Three approaches were implemented for this step:

- Traditional algorithms such as K-Means and hierarchical clustering.
- Clustering based on semantic similarity between descriptions.
- An approach based on ChatGPT, leveraging its contextual comprehension capabilities.

The choice of method is configured via the method parameter in the corresponding configuration file, where each execution uses a single method. The clustering output is represented in JSON format, where each key represents a group, and the values are the numeric identifiers of the descriptions belonging to each group. In the implementation, when using ChatGPT, group names should be representative of the elements being clustered. Figure 3 illustrates how descriptions 1 and 3 are associated with group 1, description 2 with group 3,

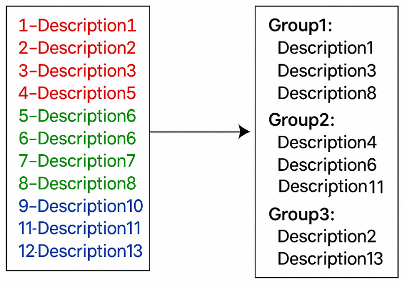


Figure 3: Mapping descriptions to their corresponding group.

Method 1: Clustering Using ChatGPT Only. This method explores the use of ChatGPT as the sole tool to cluster processed descriptions based on their semantic similarity. It starts with a prompt requesting the grouping of numbered items into representative groups, requiring JSON format and descriptive names for each group:

Prompt

```
Given a list of numerically identified topics, group them
according to their semantic relationship into well-defined
and representative groups. A topic cannot belong to more than
one group; no ungrouped topics are allowed, and no topic may
appear in multiple groups. The response will be in JSON format,
where each attribute will be the group name and the value a
numeric list of the corresponding topics. The group names must
be descriptive and represent the theme of the topics in their
group. Also, avoid creating groups with only one topic.
```

Given the token limit of the ChatGPT API, when the permitted threshold is exceeded, the task is split into multiple queries, including the previously generated groups in each. However, several limitations were detected: incomplete or incorrect group names, multiple inclusions of descriptions in different groups, omission of some descriptions, and incorrect response formats. Although it is possible to reiterate the query to correct errors, this significantly increases execution time and token-associated costs. Consequently, manual assistance was chosen to review and correct the output generated by the model.

Method 2: Clustering Using ChatGPT and SpaCy. This approach seeks to reduce queries to the model by using ChatGPT solely for generating new groups while SpaCy verifies if a description can be included in an existing group. This is done through the similarity function [1], which returns a value between 0 and 1 to measure semantic similarity. If the value exceeds a configurable threshold, the description is assigned to the corresponding group.

For example, if a description has a similarity of 0.85 with an existing group and the threshold is 0.7, it is considered part of that

[1]https://spacy.io/usage/spacy-101#vectors-similarity

group. Descriptions not exceeding the threshold are processed by ChatGPT to generate new groups.

To optimize the process and limit errors, the number of allowed tokens per query is configured via the chunks attribute. It is recommended not to exceed 4000 tokens (approximately 10,000 input characters) due to model limitations and the need for space for the response. Additionally, a mechanism is implemented to prevent the same description from appearing in multiple groups, retaining only its first occurrence.

This method leverages the semantic generation capabilities of ChatGPT and minimizes errors and resource consumption while employing NLP tools like SpaCy for automatic validations.

Method 3: Using Clustering Algorithms. As an unsupervised alternative, classic clustering algorithms were implemented to group vectorized descriptions. The TfidfVectorizer from the Scikit-Learn library [2] was used, transforming each description into a vector based on the frequency and importance of words in the corpus. This vector reflects the relevance of each term, allowing clustering techniques to be applied.

It is important to note that high dimensionality and word dispersion can introduce noise and outliers in the data, affecting the quality of the generated groups.

Two main algorithms were tested:

- *K-Means:* efficient for large data volumes, requires prior definition of the number of clusters. It is sensitive to initialization and may converge to suboptimal solutions if centroids are not properly configured.
- *Hierarchical Clustering:* builds a hierarchy of groups represented by a dendrogram. It does not require predefining the number of clusters, making it useful for exploring internal structures. Its main limitation is the high computational cost when working with large datasets.

Step 5: Generating a Unique Representation for Each Document

Each description is assigned to a group and represented as a binary vector, where a "1" indicates the group to which it belongs. For example:

- Group 1: (1,0,0).
- Group 2: (0,1,0).
- Group 3: (0,0,1).

In this way, each OpenAPI document can be described as a vector formed by the summation of the vectors it contains. For example, Figure 4 shows how document 1 is vectorized, where the group is first replaced by the corresponding vector and then the summation is performed.

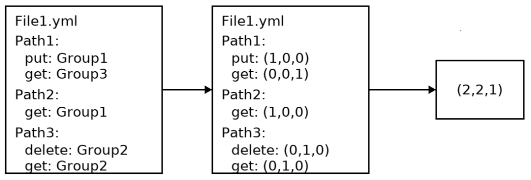


Figure 4: Vector generation for a microservice based on its functionality and structure.

[2]https://scikit-learn.org/

This final vector indicates that the document has 2 descriptions from group 1, 2 from group 2, and 1 from group 3. To enhance this representation, the semantic relationship between groups is considered, particularly useful in cases where a description could belong to more than one group (border areas). Using SpaCy and its similarity method, relationships between group names are calculated. For example, if group 1 has a similarity of 0.8 with group 2 and 0.3 with group 3, its vector is adjusted from (1,0,0) to (1, 0.8, 0.3), generating vectors richer in semantic information and improving clustering. This improvement is applied only when ChatGPT is used for initial clustering, as it ensures representative and coherent group names.

Step 6: Document clustering

Once the documents are vectorized, clustering algorithms can be directly applied to group them by similarity. Unlike previous methods (e.g., ChatGPT or semantic similarity) that work with text, this stage requires vector inputs. The tool allows configuring the type of clustering algorithm to use: K-Means or hierarchical clustering, enabling comparison between both approaches to determine which generates more accurate clusters.

K-Means. K-Means is a widely used clustering algorithm due to its simplicity and efficiency. Its operation is based on dividing a dataset into a fixed number of clusters, which must be defined in advance. Each cluster is represented by a centroid, and the algorithm assigns each data point to the cluster with the nearest centroid. It then recalculates the centroids and repeats this process until the assignments no longer change significantly. One of the main advantages of K-Means is its low computational cost, making it ideal for working with large datasets. However, it has some important limitations. The need to specify the number of clusters beforehand can be problematic if there is no prior domain knowledge.

Additionally, as it works solely with quantitative variables, it is not suitable for categorical data. Another aspect to consider is that its results can vary between runs due to randomness in centroid initialization, which can lead to local rather than global optimal solutions. K-Means is also sensitive to outliers, as these can significantly alter the centroid positions, affecting clustering quality. To minimize these effects, it is recommended to normalize the data and remove outliers before applying the algorithm.

In summary, K-Means is a powerful tool for segmenting data into groups, with advantages such as ease of implementation and efficiency in handling large datasets. However, it presents challenges, including the need to define the number of clusters in advance, its limitation to quantitative variables, the lack of convergence to consistent solutions, and its susceptibility to outliers.

Hierarchical Clustering. Hierarchical clustering proposes a different approach to grouping data. Instead of requiring a fixed number of clusters, it builds a tree-shaped hierarchy where data are progressively merged or split based on their similarity. This approach can be agglomerative (starting with each data point as an independent cluster and then merging them) or divisive (starting with all data in one group and then splitting them).

The main advantage of hierarchical clustering is that it does not require pre-specification of the number of clusters, allowing the user to explore different dendrogram cuts (the visual representation of the tree) depending on the desired granularity. It is also more robust to noise or outliers, as it considers the global structure of the dataset at each process stage. Additionally, it allows the use of various distance metrics, providing greater flexibility to adapt to different data characteristics.

However, its main drawback is the high computational cost, especially when working with large datasets, which can hinder its application in large-scale scenarios. Moreover, as the number of elements increases, interpreting the dendrogram can become complex, making it difficult to identify meaningful groups.

In summary, hierarchical clustering is a valuable technique in data analysis due to its ability not to require predefining the number of clusters, its resistance to noisy data, and its flexibility in adjusting distance metrics. However, it has disadvantages in terms of computational costs, scalability, and the difficulty of interpreting dendrograms in large datasets.

The choice between K-Means and hierarchical clustering will depend on the specific analysis needs and data characteristics. Both methods have strengths and limitations. The decision between them will depend on the dataset size, available resources, and analysis objectives.

Analyzing Clusters. After applying clustering algorithms, it is crucial to analyze the obtained results to assess how effectively the groups have been formed. For this purpose, metrics that measure cluster quality are used. This study employs the Silhouette measure, a widely recognized metric that combines concepts of cohesion and separation between data points [21]. Cohesion refers to how close points within the same cluster are to each other, while separation evaluates the distance between formed clusters. Ideally, points within a cluster should be as close as possible (high cohesion) and as far as possible from points in other clusters (high separation).

The Silhouette coefficient is calculated for each point considering both metrics and is expressed by the Equation 1:

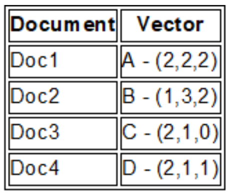
$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}, \tag{1}$$

where

- $a(i)$ (*Cohesion*) is the average dissimilarity (e.g., Euclidean distance) between point i and all other points in *its own* cluster;
- $b(i)$ (*Separation*) is the *smallest* average dissimilarity between point i and the points in any *other* cluster (i.e., the nearest neighboring cluster).

This value can range from -1 to 1. A value close to 1 indicates that the point is well assigned to its cluster and that there is good separation from other clusters. A value close to 0 suggests that the point is on the boundary between two groups, while a negative value implies poor assignment. Averaging the Silhouette coefficients of all points yields a global metric indicating the overall clustering quality. This metric is key to validating whether the constructed vector representation of the documents is precise enough to enable coherent microservice segmentation. If low or negative values are obtained, it will be necessary to review earlier stages, such as description clustering or vectorization processes, to identify possible improvements.

Graphical Representation of Points in Space. In addition to numerical analysis, it is also useful to have a graphical representation of the documents in vector space to visually understand how the points are distributed and how they might cluster. Figure 5 provides an example of how each document can be represented as a three-dimensional vector. These vectors correspond to points in a three-dimensional space. As illustrated in Figure 6, observing the arrangement of points A (Doc1) and B (Doc2), and C (Doc3) and D (Doc4), one can infer that A and B form a cluster due to their proximity, as do C and D. This visualization facilitates cluster interpretation when the number of documents is small.



Document	Vector
Doc1	A - (2,2,2)
Doc2	B - (1,3,2)
Doc3	C - (2,1,0)
Doc4	D - (2,1,1)

Figure 5: Representation of documents as vectors.

However, in practice, when working with large datasets and high-dimensional vectors, it is not always possible to visualize points directly. Therefore, the use of automatic clustering algorithms becomes essential to discover patterns without supervision or prior knowledge of data relationships.

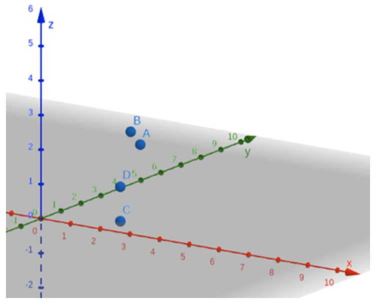


Figure 6: Representation of 3-dimension points.

4 Proposal Validation

This section specifies *how* we validated ODAM. It covers the Research Questions (RQs), experimental factors, subject system, procedure, response variables, and data-analysis methods.

Objectives. The overarching goal is to determine whether ODAM can recover functionally meaningful clusters of microservices from OpenAPI specifications. We distilled this goal into six RQs:

- RQ1 Which intermediate technique (*ChatGPT*, *SpaCy-semantic*, or *unsupervised vector clustering*) is most effective at uncovering semantic relationships among endpoint descriptions?
- RQ2 Does auto-completing missing descriptions with ChatGPT improve the final clustering quality?
- RQ3 What is the optimal number of clusters (k) for the final grouping step?
- RQ4 How different are the final clusterings produced by the competing pipelines?
- RQ5 Which pipeline offers the shortest execution time?
- RQ6 Between *K-Means* and *hierarchical clustering*, which delivers the best final grouping when fed with ODAM's document vectors?

The *RQ1* targets the very first decision point in ODAM: if conceptually related endpoints are not recognized early, downstream steps cannot recover a coherent architecture. *RQ2* addresses the chronic sparsity of industrial OpenAPI files and tests whether LLM augmentation truly adds value. *RQ3* is concerned with architectural *granularity*: too few clusters blur distinct services, whereas too many fragment cohesive domains. *RQ4* gauges robustness; high variance between pipelines would expose configuration sensitivity. *RQ5* introduces a pragmatic angle—reverse engineering must scale to large codebases and CI/CD cycles. Finally, *RQ6* contrasts the two most common clustering back-ends, giving practitioners guidance on the trade-off between K-Means' speed and Hierarchical clustering's determinism.

Subject System. We used the *Twilio REST API*[3] as the sole case-study system. Its public OpenAPI bundle describes *45 microservices* that cover SMS, voice, fax, video, e-mail, and verification features. This system was chosen because (i) it is publicly available under a permissive license, ensuring full replicability; (ii) its 45 microservices span heterogeneous domains—SMS, voice, video, e-mail, verification and IoT—thereby stressing ODAM with diverse vocabularies; (iii) roughly one-third of its endpoints lack human-written descriptions, providing a realistic test case for RQ2; and (iv) Twilio's rich documentation and wide user base allowed two independent software architects—each with over 10 years of experience in service-oriented and API-based system design, and no prior involvement with Twilio—to produce an expert-defined clustering used as the *gold standard* for evaluation. Both experts had previously worked on microservice decomposition projects in large-scale distributed systems. They independently grouped the 45 Twilio services into functional domains based solely on the available OpenAPI specifications, producing labeled cluster assignments that were then consolidated through consensus.

Experimental Factors. To explore how ODAM behaves under different configurations, we systematically manipulated four independent variables—each selected to reflect a meaningful architectural or algorithmic decision point:

- F_{int}: *Intermediate clustering technique*, with three levels: , *ChatGPT*, ; *SpaCy-semantic*, ; *TF-IDF+clustering*,. This factor governs how functional themes are derived from endpoint descriptions. It reflects the balance between traditional NLP (TF-IDF), lightweight semantic matching (SpaCy), and LLM-based semantic abstraction (ChatGPT). Comparing them tests whether deep semantic inference outperforms shallow text similarity.
- F_{desc}: *Description completion* (yes vs. no). Many OpenAPI files are partially documented. This factor tests whether auto-generating missing endpoint descriptions with ChatGPT significantly improves the quality of downstream clustering. It simulates realistic documentation scenarios, especially in brownfield systems.
- F_{fin}: *Final clustering algorithm* (k-Means vs. hierarchical). These are two widely adopted clustering methods with distinct characteristics—K-Means is scalable and fast, while hierarchical clustering yields interpretable dendrograms and requires no prior k. The goal is to assess their impact on ODAM's grouping quality when applied to functional vectors.
- F_k: *Target number of clusters* $5, 10, 15, 20$. This factor allows us to test different levels of architectural granularity. Fewer clusters may expose broader business domains, while more clusters may capture finer-grained services. It directly addresses RQ3 on optimal cluster count.

Together, this $3 \times 2 \times 2 \times 4$ full-factorial design yields 48 distinct experimental treatments. Each combination simulates a plausible ODAM usage scenario, enabling a robust, comparative evaluation across configurations that vary in semantic depth, documentation completeness, algorithmic strategy, and clustering resolution.

Response Variables. To assess ODAM's clustering quality and practical viability, we selected four response variables that capture both semantic accuracy and operational performance:

- *Success Rate* (*SR*): the proportion of elements grouped together by ODAM that are also grouped together by the expert. This metric reflects how well the automated clustering aligns with human architectural understanding. It directly measures functional coherence from a domain-specific perspective, making it essential for evaluating the utility of the output.
- *Failure Rate* (*FR*): the proportion of elements grouped by ODAM that should not be grouped according to the expert baseline. While *SR* captures correctness, *FR* captures overgeneralization or noise. Together, they provide a balanced view of precision and error.
- *Silhouette Score* (*SS*): a widely used internal metric for clustering validation, it quantifies how tightly grouped an element is within its assigned cluster (*cohesion*) versus how far it is from other clusters (*separation*). Values near 1 indicate well-defined clusters, and scores above 0.5 are generally considered acceptable. This metric is algorithm-independent and offers insight into the structural quality of the vector representations.
- *ExecTime*: the total wall-clock time (in seconds) from the beginning of the pipeline to the generation of the final JSON output. This metric reflects ODAM's feasibility for real-world use, especially in large-scale or time-constrained environments such as CI/CD pipelines.

These variables were chosen because they jointly address both *effectiveness* (semantic alignment and cluster cohesion) and *efficiency* (runtime overhead). *SR* and *FR* rely on expert judgment and domain knowledge, capturing external validation. *SS* and *ExecTime*, in contrast, reflect internal validation and performance trade-offs, enabling a comprehensive evaluation of the ODAM in different usage scenarios.

Experimental Procedure. In what follows, we describe the main steps performed in experimental activities.

1. *Dataset preparation* – download Twilio's OpenAPI files and normalize them to a single directory.
2. *Description completion* – if F_{desc} = yes, invoke ChatGPT on every endpoint whose `description` field is empty.
3. *Pre-processing* – tokenise, lemmatise and remove stop-words with SpaCy; persist cleaned texts.
4. *Intermediate clustering* – apply the technique selected by F_{int} to group individual descriptions into functional topics; store the mapping *description* $\longrightarrow$ *topic*.
5. *Vectorization* – turn each microservice into a binary or weighted membership vector over the discovered topics.
6. *Final clustering* – cluster those vectors using the algorithm and k dictated by F_{fin} and F_{k}; export the JSON result.
7. *Measurement* – compute *SR*, *FR*, *SS* and ExecTime; log all raw data for replication.

[3]https://www.twilio.com/docs/usage/api

Each treatment was executed three times to mitigate randomness in both ChatGPT completions and algorithm initialization. Results were averaged per treatment.

Data Analysis. Our analysis is strictly descriptive, mirroring the data that were collected. For every treatment, we ran three replicates and reported the *arithmetic mean* of each response variable.

- For *RQ1–4* we juxtapose *Success Rate* (*SR*), *Failure Rate* (*FR*) and *Silhouette Score* (*SS*) across treatments. No null–hypothesis testing is performed; We simply highlight the largest deltas and interpret them in the context of the RQs.
- *RQ5* is answered by comparing mean wall-clock *ExecTime*. Because variance is low (coefficient of variation < 5%), visual inspection of the values is sufficient.
- For *RQ6*, we hold k constant and contrast the means obtained with K-Means versus hierarchical clustering. The discussion focuses on practical trade-offs (speed vs. stability of the centroids) rather than on statistical significance.

Artifacts Availability. All OpenAPI descriptions, entries, ODAM source-code files, JSON files, among other artifacts used to compute the study's results, are available online in our supplementary material [4] to ensure transparency and facilitate replication.

5 Validation Results

Our experimental matrix combines *three* intermediate strategies (ChatGPT topic induction, unsupervised K-Means/Hierarchical, and SpaCy semantic matching) with *two* description-completion settings (on / off) and *two* final clusters (K-Means, Hierarchical). Each setting is replicated four times, totaling $3 \times 2 \times 2 \times 4 = 48$ pipeline runs. Accuracy is first judged at the expert's reference granularity ($k = 15$); later, k is varied to probe cohesion (RQ3).

Because the expert solution partitions Twilio into 15 clusters, we kept $k = 15$ while comparing pipelines head-to-head on *success*, *failure*, and *execution time*. Three "macro–scenarios" emerge:

- *Scenario 1* – ChatGPT groups descriptions semantically.
- *Scenario 2* – Unsupervised statistical grouping (K-Means/Hierarchical) without LLM semantics.
- *Scenario 3* – SpaCy cosine similarity–based grouping.

Within each scenario, we evaluate the impact of (i) enabling description completion and (ii) choosing K-Means or Hierarchical as the final grouper. The outcomes are summarized in Table 1, which reports the mean *SR*, *FR*, and wall-clock time for every configuration.

Regarding the *SS*, it was used to evaluate the experiment in two aspects. Firstly, it analyzed the intermediate clustering methods, hierarchy, and K-Means. Secondly, after defining the best intermediate clustering method, it was used to analyze different "k" values to determine the optimal number of final clusters. The best *SS* was obtained with k = 5 (0.55), indicating that, from a vector perspective, the set of microservices could be organized into fewer well-defined groups. This response to RQ3 suggests that the optimal number of clusters may differ from that established by the expert, depending on the chosen criterion (semantic vs. structural).

RQ1 – Which intermediate technique is most effective?. Table 1 leaves little doubt that the ChatGPT-driven pipeline (Scenario 1) is the clear front-runner: when completion is enabled, it attains an average success rate of 0.64 while keeping failure at 0.40—a 34% relative improvement in correct pairings over the best purely statistical alternative (Scenario 2, 0.48). Two mechanisms explain the gap. First, the LLM synthesizes short, information-dense sentences that inject the same domain nouns ("verify", "token", "voice call") into endpoints that originally shared no lexical overlap; this densification gives TF-IDF and cosine similarity something tangible to work with. Second, the grouping prompt forces ChatGPT to assign high-level labels ("Identity verification", "Programmable voice") that act as ready-made semantic centroids. Once such centroids exist, whether the final grouper is K-Means or hierarchical matters little (Δ SR ≤ 0.01), confirming that representation quality dominates the downstream algorithm.

RQ2 – Does completing missing descriptions help?. Figure 7 shows that completion is always beneficial, but its payoff is pipeline-dependent. In Scenario 1, filling the 31% of Twilio endpoints that lacked text is transformative, pushing SR from 0.48 to 0.64 and cutting FR by one-third. In Scenarios 2 and 3, the lift is more modest (+3–5 pp), yet still valuable: short, consistent sentences reduce vocabulary sparsity and keep cosine distances from collapsing to zero. In other words, completion turns opaque API stubs into usable signals; the more semantic the downstream stage, the larger the dividend.

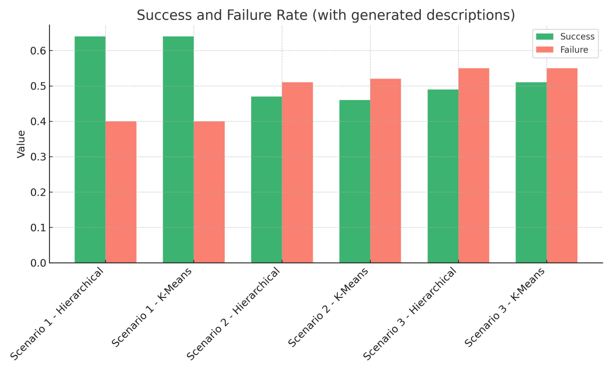


Figure 7: Success and fail rates.

RQ3 – What cluster count (k) maximizes cohesion and separation?. After locking in the best pipeline from RQ1, we varied k and recomputed Silhouette (Table 2). The curve crests at $k = 5$ (SS = 0.55), then plateaus. Manual inspection shows these five groups map almost one-to-one to Twilio's business domains—Messaging, Voice, Video, Email, and Trust & Safety—suggesting that ODAM surfaces a high-level capability view that even Twilio's documentation leaves implicit. Values of k beyond 10 add little cohesion; values below 5 begin to merge orthogonal capabilities (e.g., SMS and Voice). Thus, ODAM gives architects an evidence-based upper and lower bound for meaningful granularity.

RQ4 – How sensitive are the results to pipeline configuration?. Changing the final cluster, toggling completion, or swapping the statistical intermediate method moves the final Silhouette by at most 0.05 (Tables 2 and 3). Even when ChatGPT is removed entirely, the ranking of intermediate strategies remains stable. This low variance indicates that ODAM's two-level vectorization (topics $\rightarrow$ document vectors) absorbs noise and provides a configuration-robust output, crucial when the tool must be executed in automated workflows that favor default settings.

[4] https://doi.org/10.6084/m9.figshare.29495540

Table 1: Analysis of the scenarios according to success rate (*SR*) and failure rate (*FR*) metrics (k = 15).

k = 15			With Generated Descriptions			Without Generated Descriptions		
Scenario	Intermediate Clustering	Final Grouping	Success	Failure	ExecTime (sec)	Success	Failure	ExecTime (sec)
1	ChatGPT	Hierarchical	0.64	0.40	5586.16	0.48	0.60	1813.85
		K-Means	0.64	0.40	5585.24	0.47	0.60	1813.89
2	Hierarchical	Hierarchical	0.47	0.51	39.40	0.46	0.57	23.67
		K-Means	0.46	0.52	38.78	0.40	0.59	23.09
2	K-Means	Hierarchical	0.48	0.54	38.62	0.48	0.55	23.00
		K-Means	0.44	0.58	37.99	0.47	0.59	22.93
3	Semantic	Hierarchical	0.49	0.55	98.62	0.59	0.59	60.73
		K-Means	0.51	0.55	98.56	0.59	0.62	60.66

Table 2: Silhouette score (*SS*) analysis for different k values using ChatGPT as intermediate clustering method

k	Final Method	Final Score
5	k-means	0.55
	hierarchical	0.53
10	k-means	0.49
	hierarchical	0.49
15	k-means	0.49
	hierarchical	0.50
20	k-means	0.50
	hierarchical	0.50

RQ5 – Which pipeline is fastest?. Execution-time columns in Table 1 reveal a stark, two-order-of-magnitude split. Statistical and SpaCy pipelines finish in 40–100 s, compatible with CI/CD feedback loops. ChatGPT pipelines, however, require about 5.600 s (1.6 h) because every missing description triggers an API call. At current OpenAI pricing, the full run costs roughly $0.45—economically trivial but operationally slow. A pragmatic workflow is therefore to cache completed descriptions on disk and re-run the slow pipeline only when the OpenAPI spec changes; day-to-day builds can rely on the cached text and enjoy sub-minute runtimes.

RQ6 – K-Means or hierarchical for the final step?. Finally, answering RQ6, both Table 2 and Table 3 show that the final scores do not represent a significant difference in the results obtained. Therefore, it can be stated that while either method is suitable for the final clustering, the needs and computational requirements of the developer must be considered. On one hand, K-Means has lower computational costs, but its results may vary between runs due to centroid initialization. On the other hand, hierarchical clustering can produce identical results in different runs at the cost of higher computational requirements.

Table 3: Silhouette score analysis in intermediate clustering methods

Intermediate Method	Final Method	Intermediate Score	Final Score
k-means	k-means	0.02	0.67
k-means	hierarchical	0.02	0.67
hierarchical	k-means	0.01	0.65
hierarchical	hierarchical	0.01	0.67

6 Main Implications

The findings of this study carry implications across three dimensions: software engineering research, industrial practice, and architectural tooling. These implications are grounded in the empirical evidence obtained from our evaluation of ODAM.

From a research perspective, the results confirm that LLMs may improve the quality of architectural clustering when compared to traditional NLP pipelines. Specifically, RQ1 and RQ2 demonstrated that ChatGPT-augmented descriptions led to a 34% improvement in grouping accuracy over pipelines relying solely on TF-IDF or semantic similarity. This finding reinforces the growing understanding that LLMs are not merely text generators but effective instruments for semantic abstraction and knowledge structuring. Additionally, RQ6 showed that once high-quality intermediate representations are in place, the choice of clustering algorithm (K-Means vs. Hierarchical) has minimal impact on the final outcome. This suggests that future work should focus on enhancing the expressiveness and robustness of vector representations—through better embedding strategies or hybrid techniques—rather than only optimizing clustering parameters. Furthermore, the reliance on a manually curated expert clustering highlights a recurring challenge in this line of research: the lack of standardized, multi-annotator benchmarks for evaluating microservice decompositions. This limitation suggests an avenue for the community to invest in shared datasets and evaluation frameworks for architectural recovery.

In terms of practical implications, the study offers actionable insights for engineering teams working on legacy systems or undocumented APIs. The evidence from RQ2 suggests that LLMs are particularly effective at compensating for incomplete documentation, which is common in brownfield contexts. Moreover, RQ5 introduced an important trade-off: while ChatGPT-based pipelines provide superior clustering accuracy, they incur higher execution times. Fortunately, the overhead can be mitigated by caching the generated descriptions, allowing organizations to benefit from LLM-enhanced analysis without compromising runtime efficiency. The results from RQ3 also suggest that there is no single "correct" level of decomposition. For instance, a cluster count of $k = 5$ yielded the highest Silhouette score, even though the expert reference used $k = 15$. This indicates that ODAM can support different levels of architectural granularity depending on the business or operational goals—ranging from coarse domain-level partitioning to fine-grained service analysis.

Finally, **regarding tooling**, our findings highlight that ODAM can serve as a clustering engine and enabler of architectural visibility and governance. By injecting consistent terminology and high-level group labels, ChatGPT facilitates the interpretation of clusters, bridging the gap between informal documentation and structured design artifacts. This aligns with insights from RQ1 and RQ4, which show that pipelines integrating semantic enrichment produce more interpretable and stable clusters. In addition, the sub-minute execution time achievable via caching (RQ5) makes ODAM suitable for integration into CI/CD workflows or developer platforms such as Backstage or Visual Studio Code. In such contexts, ODAM could support tasks like service refactoring, architectural drift detection, or automated validation of domain boundaries.

7 Threats to Validity

Despite the promising results, several validity threats may influence the generalizability and replicability of our findings. These are organized under four commonly adopted dimensions in empirical software engineering: construct validity, internal validity, external validity, and reliability [26].

Internal Validity. The internal consistency of the results may be affected by the inherent variability of LLM-generated content. Although standardized prompts were used to mitigate variance, ChatGPT's outputs are sensitive to prompt phrasing, context window constraints, and token sampling randomness. Moreover, decisions such as group name assignment and error correction in clustering outputs required occasional human intervention. These steps, while minimal, introduce a risk of unconscious bias. In addition, as all experiments were conducted on a single dataset (Twilio), there is a potential risk of overfitting pipeline configurations (e.g., thresholds or prompt styles) to that particular dataset, limiting their generalizability.

Construct Validity. We chose the Silhouette score and expert-defined service groupings as our primary evaluation criteria. While these are widely used in clustering assessments, they do not fully capture domain semantics, business logic alignment, or performance-driven constraints inherent in microservice design. Additionally, the representation model assumes that textual similarity correlates with functional cohesion, which may not hold in all architectures (e.g., when two services have distinct roles but share similar verbs like "get", "create", or "send"). Furthermore, the expert-curated reference architecture used as a gold standard is itself subjective and may reflect individual bias or domain-specific assumptions. Multiple annotators and inter-rater agreement measures could provide a more robust ground truth in future studies.

External Validity. Our empirical evaluation is limited to a single domain-specific case (Twilio API), which—though heterogeneous—is still a communication-centric platform. Systems in finance, healthcare, IoT, or manufacturing might exhibit different documentation styles, domain vocabularies, and service partitioning philosophies. Therefore, further studies on diversified API ecosystems are needed to validate ODAM's effectiveness across different verticals and document quality conditions.

Reliability. While most stages of the ODAM pipeline are automated and reproducible, some configurations—especially ChatGPT-based groupings—rely on API calls whose responses may drift over time as the model evolves. This raises concerns about reproducibility unless ChatGPT versions, prompts, and outputs are cached and archived. Additionally, some decisions—like threshold tuning for SpaCy similarity—were made empirically rather than through cross-validation or statistical optimization, leaving room for future refinement. Finally, the pipeline depends on third-party services (e.g., OpenAI API), whose availability, pricing, and underlying model behavior may change over time, potentially affecting ODAM's long-term sustainability and repeatability.

8 Final Remarks

This study set out to determine whether text-based service descriptions—augmented by LLMs—can drive an automated, function-oriented clustering process for microservices. To that end, we introduced ODAM (OpenAPI Documentation Analysis and Modeling), a four-stage pipeline that: (i) fills in missing endpoint descriptions using ChatGPT, (ii) distills salient terms with SpaCy, (iii) encodes the resulting texts as vectors, and (iv) applies three alternative grouping strategies—ChatGPT-only, ChatGPT + SpaCy validation, and classical unsupervised algorithms—to generate coherent clusters of services.

The RQ1 showed that the ChatGPT-only strategy delivers the most semantically aligned intermediate clusters. RQ2 confirmed that LLM-generated descriptions boost clustering quality across all methods. In RQ3, we observed that a smaller number of final clusters (k = 5) maximizes the silhouette coefficient, whereas the expert baseline used k = 15. RQ4 revealed no statistically significant difference between K-Means and hierarchical clustering when fed high-quality vectors. RQ5 highlighted a clear trade-off: ChatGPT offers the best precision but incurs the highest execution time. Finally, RQ6 demonstrated that either K-Means or hierarchical clustering can serve as the final step, provided the earlier description grouping is robust.

Our work contributes (i) ODAM, a reproducible end-to-end tool that unifies LLM augmentation with traditional NLP for service clustering; (ii) an empirical comparison of six pipeline variants, offering practitioners guidance on accuracy-versus-cost trade-offs; and (iii) the first benchmark, to our knowledge, that contrasts human expert groupings with LLM-enhanced clustering of real-world OpenAPI microservices.

Looking for future work, we plan to broaden ODAM's reach beyond the current case study by validating it on finance, e-commerce, and IoT APIs, thereby testing its robustness across distinct business domains. We will also enrich the input signals—combining runtime traces, log semantics, and version-control histories with the existing documentation pipeline—to capture both evolutionary and functional relationships among services. To make these capabilities actionable, we aim to embed ODAM in developer tooling such as Backstage or VS Code, complete with interactive visualizations and "what-if" regrouping that supports live architectural decision-making. Finally, we intend to investigate ensemble strategies that blend multiple LLMs and classical algorithms, mitigating single-model bias and boosting clustering stability. Together, these directions move us closer to fully automated, text-driven support for designing, evolving, and governing microservice architectures.

ACKNOWLEDGMENTS

This work has been partially funded by the project "iSOP Base: Investigação e desenvolvimento de base arquitetural e tecnológica da Intelligent Sensing Operating Platform (iSOP)" supported by CENTRO DE COMPETÊNCIA EMBRAPII VIRTUS EM HARDWARE INTELIGENTE PARA INDÚSTRIA - VIRTUS-CC, with financial resources from the PPI HardwareBR of the MCTI grant number 055/2023, signed with EMBRAPII.

REFERENCES

[1] Yalemisew Abgaz, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley, and Paul Clarke. 2023. Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4213–4242.

[2] Lauren Adams, Francis Boyle, Patrick Boyle, Dario Amoroso d'Aragona, Tomas Cerny, and Davide Taibi. 2023. ChatGPT for Microservice Development: How Far Can We Go?. In *Proceedings of the International Conference on Microservices*. 1–12. https://conf-micro.services/2023/papers/08-adams-chatgpt.pdf

[3] Aakash Ahmad, Muhammad Waseem, Peng Liang, Mahdi Fahmideh, Mst Shamima Aktar, and Tommi Mikkonen. 2023. Towards human-bot collaborative software architecting with chatgpt. In *Proceedings of the 27th international conference on evaluation and assessment in software engineering*. 279–285.

[4] Omar Al-Debagy. 2021. *Microservices Identification Methods and Quality Metrics*. Ph. D. Dissertation. Budapest University of Technology and Economics (Hungary).

[5] Omar Al-Debagy and Peter Martinek. 2019. A new decomposition method for designing microservices. *Periodica Polytechnica Electrical Engineering and Computer Science* 63, 4 (2019), 274–281.

[6] Ahmed S. Alsayed, Hoa K. Dam, and Chau Nguyen. 2024. MicroRec: Leveraging Large Language Models for Microservice Recommendation. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR '24)*. 419–430. https://doi.org/10.1145/3643991.3644916

[7] Sathurshan Arulmohan, Marie-Jean Meurs, and Sébastien Mosser. 2023. Extracting domain models from textual requirements in the era of large language models. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 580–587.

[8] Luciano Baresi, Martin Garriga, and Alan De Renzis. 2017. Microservices identification through interface analysis. In *Service-Oriented and Cloud Computing: 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings 6*. Springer, 19–33.

[9] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. 2022. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE access* 10 (2022), 20357–20374.

[10] Saurabh Chauhan, Zeeshan Rasheed, Abdul M. Sami, Zheying Zhang, Jussi Rasku, Kai-Kristian Kemell, and Pekka Abrahamsson. 2025. LLM-Generated Microservice Implementations from RESTful API Definitions. In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE '25)*. https://doi.org/10.5220/0013391000003928

[11] Leonardo da Rocha Araujo, Guillermo Rodríguez, Santiago Vidal, Claudia Marcos, and Rodrigo Pereira dos Santos. 2022. Empirical Analysis on OpenAPI Topic Exploration and Discovery to Support the Developer Community. *Computing and Informatics* 40, 6 (Feb. 2022), 1345–1369. https://doi.org/10.31577/cai_2021_6_1345

[12] Leonardo Henrique Da Rocha Araujo, Guillermo Horacio Rodríguez, Santiago Agustín Vidal, Claudia Andrea Marcos, and Rodrigo Pereira Dos Santos. 2022. Empirical analysis on openapi topic exploration and discovery to support the developer community. (2022).

[13] Rudra Dhar, Karthik Vaidhyanathan, and Vasudeva Varma. 2024. Can llms generate architectural design decisions?-an exploratory empirical study. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE, 79–89.

[14] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[15] Sam Newman. 2021. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".

[16] Idris Oumoussa and Rajaa Saidi. 2024. Evolution of microservices identification in monolith decomposition: A systematic review. *IEEE Access* 12 (2024), 23389–23405.

[17] Alina Petukhova, João P Matos-Carvalho, and Nuno Fachada. 2025. Text clustering with large language model embeddings. *International Journal of Cognitive Computing in Engineering* 6 (2025), 100–108.

[18] Sebastian Pinto-Agüero and Rene Noel. 2025. Microservices Evolution Factors: a Multivocal Literature Review. *IEEE Access* (2025).

[19] Ernesto Quevedo, Amr S. Abdelfattah, Alejandro Rodriguez, Jorge Yero, and Tomas Cerny. 2024. Evaluating ChatGPT's Proficiency in Understanding and Answering Microservice Architecture Queries Using Source Code Insights. *SN Computer Science* 5, 4 (2024), 422. https://doi.org/10.1007/s42979-024-02664-0

[20] Ana Martínez Saucedo, Guillermo Rodríguez, Fabio Gomes Rocha, and Rodrigo Pereira dos Santos. 2025. Migration of monolithic systems to microservices: A systematic mapping study. *Information and Software Technology* 177 (2025), 107590.

[21] Meshal Shutaywi and Nezamoddin N Kachouie. 2021. Silhouette analysis for performance evaluation in machine learning with applications to clustering. *Entropy* 23, 6 (2021), 759.

[22] Mehmet Söylemez, Bedir Tekinerdogan, and Ayça Kolukısa Tarhan. 2022. Challenges and solution directions of microservice architectures: A systematic literature review. *Applied sciences* 12, 11 (2022), 5507.

[23] Tatjana Stojanovic and Saša D. Lazarević. 2023. The Application of ChatGPT for Identification of Microservices. *E-business technologies conference proceedings* 3, 1, 99–105. https://ebt.rs/journals/index.php/conf-proc/article/view/181

[24] Fredy H Vera-Rivera, Eduard Gilberto Puerto Cuadros, Boris Perez, Hernán Astudillo, and Carlos Gaona. 2023. SEMGROMI—a semantic grouping algorithm to identifying microservices using semantic similarity of user stories. *PeerJ Computer Science* 9 (2023), e1380.

[25] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. 2021. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering* 26, 4 (2021), 63.

[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[27] Shenglin Zhang, Sibo Xia, Wenzhao Fan, Binpeng Shi, Xiao Xiong, Zhenyu Zhong, Minghua Ma, Yongqian Sun, and Dan Pei. 2024. Failure diagnosis in microservice systems: A comprehensive survey and analysis. *ACM Transactions on Software Engineering and Methodology* (2024).