A Data-centric Method for Microservice Identification: From Data Models Analysis to Source Code Assignment

Yamina Romani[a], Okba Tibermacine[b], Chouki Tibermacine[c]

[a]*Computer science department, University of Biskra, Algeria*
[b]*National School of Artificial Intelligence, Algiers, Algeria*
[c]*IRISA, University of Southern Brittany, Vannes, France*

Abstract

Context: Microservice-based architecture has emerged as a robust architectural style for developing software systems as loosely coupled modules with several high-quality characteristics. Given characteristics like maintainability, independent deployability, and scalability, many practitioners embrace this architectural style while developing new applications or migrating existing monolithic ones.

Objective: In this paper, we are interested in the migration process which is an exhausting task that requires splitting the monolithic software system into a set of highly internally-cohesive and loosely externally-coupled services.

Method: Our proposed process takes a data model-centric perspective. We argue that microservice identification cannot rely on source code analysis only, because the source code of the monolith has not been written originally with a microservice-based architecture style in mind. Later in the migration process, this code has to be modified to match the new architecture. The identification of microservice candidates is thereby divided into two stages. In the first stage, topic modeling is performed to decompose the monolithic system's database model into a set of sub-models. Each derived sub-model refers to a potential microservice's own independent database. In the second stage, the source code of the monolith is analyzed and distributed to the identified sub-models based on semantic similarity and structural relationships among source code elements.

Email addresses: yamina.romani@univ-biskra.dz (Yamina Romani), okba.tibermacine@ensia.edu.dz (Okba Tibermacine), chouki.tibermacine@univ-ubs.fr (Chouki Tibermacine)

Results: We applied our process to two open-source applications with an available microservice version. Our quantitative evaluation showed good results. Additionally, we compared our process with existing methods from the literature in terms of modularity and cohesion by measuring four state-of-the-art metrics on three common open-source projects. Results showed that our process effectively identifies a modular and cohesive set of microservices.

Conclusion: In this paper, we demonstrate that considering data models as the preliminary drivers for microservices identification provides an effective approach for defining microservice boundaries.

Keywords: Microservices, database-per-service pattern, monolithic to microservice migration, software architecture, topic identification, clustering, source code assignment.

---

1. Introduction

Microservice architectures have emerged in the last decade as one of the mostly "elegant" architecture styles for developing enterprise applications. Whether in new projects or in existing still-profitable projects, many professionals leverage this style when designing the architecture of their software systems. Their goal is to benefit from several quality characteristics, like maintainability, independent deployability and scalability [1]. Developers can thereby benefit from these "ilities" either when designing new projects or when modernizing (by migrating) still-profitable monolithic systems. However, migrating a monolithic system to microservices presents significant challenges, especially in the identification and decomposition of the monolith software into a set of highly internally-cohesive and loosely externally-coupled services that align with the domain-driven design principle. This task is non-trivial due to the intricate dependencies and tightly coupled code-base parts often found within the monolithic system. In the literature, many existing approaches [2] heavily rely on source code analysis to identify potential microservice boundaries. These approaches have limitations because the source code of a monolith system is not written with a microservice architecture in mind. Consequently, many parts of this code do not reflect clear boundaries between domain-related and independently-deployed modules (microservices). The code may include many dependencies that crosscut the different business domain parts of the software system. In a microservice application, boundaries between microservices are clearly de-

fined through "contractual" APIs. These are even designed so that they can be used remotely (at least from another process, not necessarily from another physical/virtual machine). Regarding this aspect, microservices are highly decoupled and independent. So, when migrating to microservices, the monolith's source code has to be, at some places (at microservice boundaries), heavily changed. This is why, we argue in our work that performing an analysis of the monolith's source code alone cannot provide pertinent information on potential microservice candidates in the migrated application. Our take to solve this search problem is to rely first on data models (database schemas, for instance).

This paper proposes a two-stage process for microservice identification that takes a data model-centric perspective alongside with source code analysis. Our approach leverages the inherent structure of the system data model and its relationship with the code to guide the "decomposition" process.

Our proposed process tackles the challenge of migrating monoliths to microservices in two key stages:

1. **Microservice candidate identification,** where we use topic modeling [3] to decompose the monolithic systems database schema into a set of thematically cohesive sub-models. Each sub-model represents a potential microservice candidate with its own independent database, respecting the database per service pattern [1]. This data-driven approach helps to identify functional areas within the monolith system that can be effectively separated into individual microservices;
2. **Source code assignment,** where we focus on the analysis of the source code of the monolith, and its distribution across the identified potential microservices. This attribution is guided by semantic similarity and structural relationships between the code elements and the identified sub-models. This guarantees that the code responsible for functionalities/business logic is aligned with the appropriate sub-model and its corresponding microservice.

The two stages aim to achieve a more robust and domain-driven identification of potential microservices, along with a well-defined allocation of source code to these identified services.

To evaluate the performance of our method, we applied the proposed process to two open-source monolithic applications that also have available microservice versions. We used the $a2a$ and $c2c_{cvg}$ metrics to assess the results quantativley. Moreover, we compared our approach with existing compet-

itive methods. We measured four metrics, Structural Modularity Quality (SMQ) and Conceptual Modularity Quality (CMQ) to quantify the degree of modularization of the obtained microservices. Cohesion at Message Level (CHM) and Cohesion at Domain Level (CHD) to assess the cohesion of the obtained microservices. These evaluations were conducted using three common open-source projects. The results demonstrate that our process gives a good performance in identifying a cohesive set of microservices in terms of domain and message level.

The rest of the paper is organized as follows: Sect. 2 presents the proposed two-stage process for identifying potential microservices and allocating appropriate code to them. Sect.3 demonstrates the conducted experiments to evaluate the proposed process. Sect.4 presents the obtained results and the discussion. Threats to validity are presented in Sect. 4.1. Sect. 5 summarizes the related work from the literature. Conclusions and perspectives are discussed in Sect. 6.

2. Proposed approach

In this section, we present our proposed method for microservice identification. Figure 1 illustrates the overall progression to identify microservices from a given monolithic software system. The method proposes a two-phase process: the first phase is a "Data-Centric Process for Microservices Identification". It takes as input the database/data model of the monolithic software system and the number of expected clusters. The database/data model is analyzed through various steps. The first step is Document Construction, where we collect symbols from SQL tables/No-SQL documents, which are the names of tables and their attributes/columns, and then we pre-process them. At the end of this step, we construct a set of cleaned documents. The second step is Document Semantic Enrichment, where the set of symbols of each document is augmented with synonyms. The last step is Document Clustering, which begins with the vectorization of the enriched documents. We use a vectorization method to transform them into vectors. Next, a clustering algorithm utilizes these vectors to construct the database clusters. The obtained clusters represent groups of symbols that refer to topics. Each topic is considered as a label for a potential microservice.

The second leading phase is "Source Code Assignment". It receives the

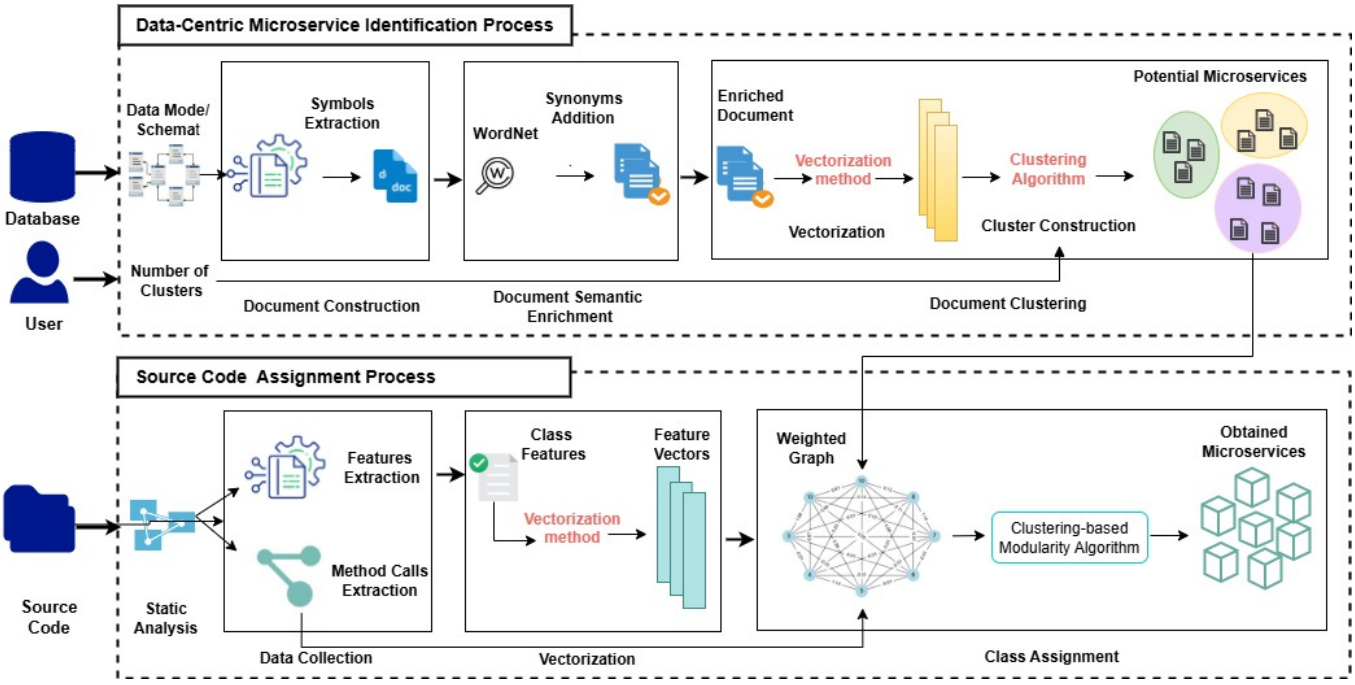


Figure 1: Proposed approach

source code as input and targets to assign the different programs[1] to their appropriate potential microservices obtained via the first leading phase. It consists of three main steps. The first step is Data Collection, where we extract Features and Method calls from the source code using static analysis. The second step is Vectorization, which is conducted in the same way as in the first phase. Thereby, the extracted features are converted into vectors. The last step is the Class Assignment, where we assign classes to their relevant clusters using a clustering-based modularity algorithm, which receives a weighted graph. The nodes of the graph refer to the monolith's classes and its edges refer to the relationships between them. The edge weights are calculated using semantic, structural, and data relationships. In the following subsections, we present in detail these phases.

[1]We consider in our work classes, because in the current implementation of the method we analyze object-oriented class-based programs.

2.1. Data-Centric Process for Microservices Identification

At this phase, the database/data model of a given monolithic software system is processed using the following steps:

2.1.1. Document Construction

In this step, we construct a set of documents using the database schema model of the monolithic software system. Each document comprises a set of symbols extracted from a given relational table (or No-SQL document) in the data model. A symbol could represent a table's name, a table's attribute, or a relationship between two tables. Some symbols have a different representation in their documents according to their importance. A given symbol can be replicated many times in the document, which gives it more weight and provides it a greater significance in (later) clustering. The number of repetitions is fixed empirically, where we tested our process with different values on small projects. According to the obtained results, we modeled the symbols as follows:

- The table's name is regarded as an important symbol in both the software database and the business domain. Thus, we replicate it **3** times in the document to provide it more weight.

- A relationship between tables indicates a functional or structural connection between them. This connection is represented by foreign keys/references which act also as significant symbols.

  Two types of foreign keys are identified after analyzing some database schema models:

  1. **Singleton foreign key**. It links a source table to only one target table, creating a strong relationship between the two documents. As a representation, we replicate the foreign key and its original/source table's name **4** times in the target document. Equally, we replicate the target table's name in the source document. An example that illustrates the singleton foreign key is presented in figure 2, where the "types" table links only the "pets" table with a foreign key.
  2. **Non-singleton foreign key**. It links the source table with multiple target tables, producing a less strong relationship between the two documents compared with the **singleton foreign key**.

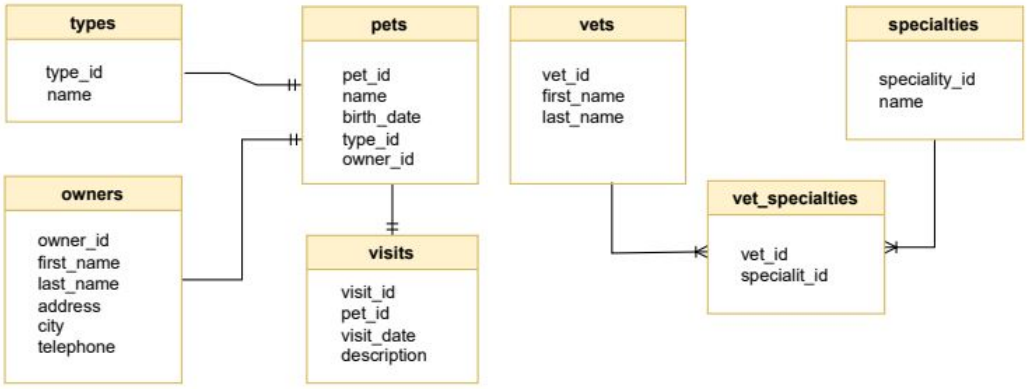


Figure 2: Entity-relationship Diagram of Petclinic

> Therefore, we replicate this foreign key and its source table's name **3** times in the target documents, as well as the names of the target tables in the source document. We have an example of this type of foreign keys in figure 3: the "users" table links three tables, the "posts", "visits" and "stored_files" with a non-singleton foreign key.

After this step, the document symbols that are represented by common abbreviations are mapped to their full words. For example: `qty` is mapped to `quantity`. Then, all documents are processed and cleaned using some NLP (*Natural Language Processing*) techniques, where we tokenize [4] each document symbol into distinct words, and apply lemmatization that converts each word into its lemma (root) using a morphological analysis [4].

2.1.2. Document Semantic Enrichment

As acknowledged, unsupervised/supervised machine learning techniques necessitate a large amount of data to yield good results, For that reason, we enhance our documents by enriching them with semantically related symbols. Specifically, we use the WordNet lexical database to generate synonyms for document symbols. For each symbol, we select the first ten synonyms and add them to the document. This enrichment process is crucial for improving the accuracy of our subsequent clustering phase, as it helps bridge the vocabulary gap often found in software artifacts.

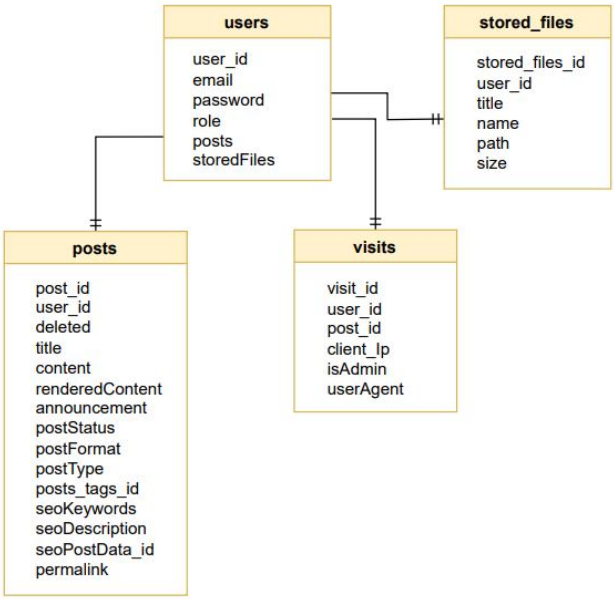


Figure 3: Part of the Entity-relationship Diagram of SpringBlog

2.1.3. Document Clustering

In this step, we group all the symbols in clusters, where each cluster contains the most similar symbols. To accomplish this step, it is essential to vectorize the enriched documents, specify the expected number of clusters, and then apply the clustering algorithm.

Document Vectorization. To transform each document symbol into a vector space model and provide a weight for each symbol, we vectorize our documents using **Tf-IDF** [5] that considers the value of this symbol within its document and across the entire document collection. In addition, we utilize **BERT** (Bidirectional Encoder Representations from Transformers) [6] that considers the context of symbols within documents. We selected these techniques because they produced better results with many testing examples than other alternative methods. Additionally, they have shown good results in many state-of-the-art studies [7][8][9][10][11].

After the calculation of **TF-IDF** and **BERT** vectors, we concatenate them to improve our clustering results. Finally, we obtain a set of vectors that represent the weights of the symbols in each document.

Number of clusters identification. The number of clusters is an important parameter in clustering problems. There are some techniques used

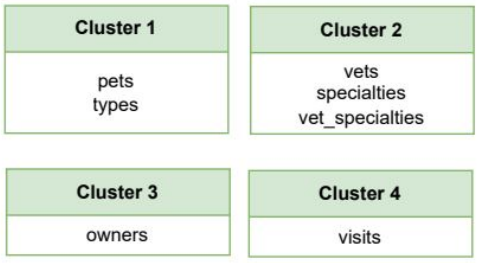


Figure 4: Database Clustering Results of Petclinic

to determine the optimal number of clusters that best fit the input data. One commonly used method is the "Elbow" method [12], which we used in our previous work [13]. This method calculates the cost function generated using various values of the number of clusters, assessing the squared distance between the documents and their assigned cluster centroids. Usually, we select the number of clusters at the point where the curve of the sum of squared distance begins to form an elbow.

However, in some cases, particularly when dealing with a small dataset, the "Elbow" method fails to identify the optimal number of clusters, as the curve may not exhibit a clear elbow shape. This was confirmed by our experiments. Therefore, in this work, we require from users to specify the expected number of clusters based on their knowledge of the software system to migrate.

Cluster construction, After constructing, preprocessing, enriching, and vectorizing documents, this step involves creating clusters using an appropriate clustering technique. In our process, we opted for **agglomerative clustering**, which groups a set of observations based on their similarities [14]. The algorithm considers each document/table as a single cluster. It iteratively merges pairs of clusters based on a defined similarity or distance metric until all documents/tables are assigned to their relevant clusters. In our implementation, we used Euclidean distance.

Figure 4 shows an example of database clustering results of the `Petclinic` project, where the input data is illustrated in figure 3 as an Entity-Relationship diagram of seven tables. The data-centric process identifies four clusters, each representing a potential microservice.

2.2. Source Code Assignment

At this phase, the source code of the monolithic software system is processed according to the following steps:

2.2.1. Data Collection

In this step, we perform a static analysis on the source code, where we use the Java Parser library[2] to generate the AST (Abstract Syntax Tree) of the source code. Based on the generated AST, we extract class features and method calls as follows:

- Feature Extraction. In our approach, we consider as a feature every term in the source code that is related to the domain of the monolithic software system where features involve class names, method names, method types, class fields and their types, and method parameters and their types.

  In addition, we modeled the structural relationship between classes as features:

  1. Inheritance relations: we include extended class names and implemented interface names as features of a given class.
  2. Method invocations: we include the method name and its original class name as features of the caller class. Furthermore, we add the caller class name to features of the class containing the called method.
  3. Instance creations: we add the original class name of the created instance to the creator class's features.

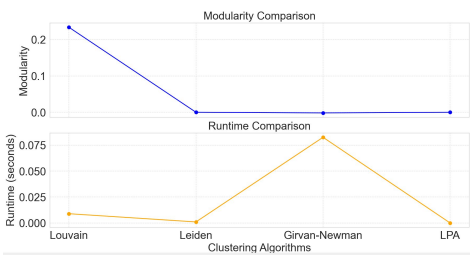
  In the end, we obtain a set of class features.

- Method Calls Extraction. Based on the Abstract Syntax Tree(Ast) of the source code, we identify and extract the method invocations list that will be employed in the distance calculation.

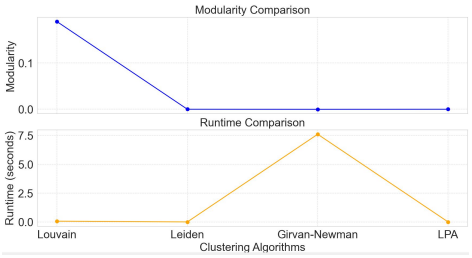
2.2.2. Feature Vectorization

Similar to the vectorization in the Document Clustering subsection 2.1.3 in Phase 1, we convert class features into numerical vectors using **Tf-Idf** and **BERT**, then we concatenate the obtained vectors in order to improve the quality of the results.

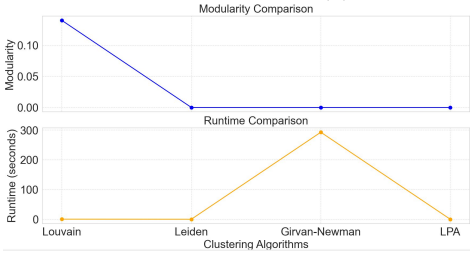
[2] `https://javaparser.org/`



(a) Petclinic Modularity and Runtime



(b) SpringBlog Modularity and Runtime



(c) Jforum3 Modularity and Runtime

Figure 5: Comparison of Clustering-based Modularity Algorithms

2.2.3. Class Assignment

In this step, we selected a set of common clustering-based modularity algorithms: Louvain algorithm, Leiden algorithm, Girvan-Newman algorithm, and Label Propagation Algorithm (LPA). Then we evaluated them by comparing their performance in terms of modularity and runtime. We performed these algorithms on three Java projects, the evaluation results are illustrated in Figure 5.

Based on the analysis of state-of-the-art studies [15][16][17][18][19], which demonstrate the effectiveness of the Louvain algorithm in various problems, and considering our evaluation results (Figure 5), we conclude that the Louvain algorithm provides an optimal trade-off between modularity and execution time.

As a result, we used the Louvain community detection algorithm to assign classes to their appropriate clusters produced in phase 1. The Louvain algorithm[20] is a heuristic algorithm that requires a graph as input and extracts clusters/communities based on the graph (network) modularity maximization. It aims to detect partitions with high modularity. This corresponds to strong relationships within a partition's nodes and weak relationships between different partitions.

The Louvain algorithm consists of two phases repeated iteratively:

- Initially, each node i is assigned to a community/cluster, then for each node i and its neighbor j the gain of modularity is calculated. If it is a positive gain, the node i is moved to community of node j. If not, the node i stays in its original community. This phase is repeated for all nodes until no additional modularity improvement can be attained[20].

- In the second phase, a new network/graph is built where its nodes are the detected communities/clusters in Phase 1. The edges between the communities/clusters are weighted by the sum of the weight of the edges between nodes/classes within the corresponding two communities/clusters[20].

In order to apply the Louvain algorithm, we represented our data as a weighted graph G = (V, E), where vertices refer to the monolith's classes and edges refer to relationships between them. The edge weights are calculated using a combination of cosine similarity between classes, which reflects the semantic relationship, and method calls that depict the structural relationship. Also, we take into consideration the database clusters obtained from Phase 1, where we define the data relationship that represents the links between data entities/classes based on their clustered tables.

- **Semantic relationship**. It is the aspect that we focus on most in our approach since we target the identification of microservices through database splitting. Therefore the source code classes are expected to be grouped based on their sub-domains, where classes that share multiple features/domain vocabularies tend to belong to similar domains and are semantically related. To measure the similarity between classes we use Cosine similarity [12] which quantifies the similarity between the class vectors by calculating the cosine of the angle between them. The cosine similarity values belong to the range [-1,1], where -1 indicates exact dissimilarity and 1 indicates exact similarity. In the following equations, we use `cos_sim` to denote the cosine similarity.

$$cos\_sim(c_i, c_j) = \frac{c_i \cdot c_j}{\|c_i\| \cdot \|c_j\|} \tag{1}$$

  Where:

  - c_i and c_j refer to distinct class vectors.

- $c_i \cdot c_j$ refers to the dot product of the vectors c_i and c_j.
- $\|\mathbf{c_i}\|.\|\mathbf{c_j}\|$ denotes the Euclidean norms of vectors.

- **Structural relationship**. It is important to involve this aspect in microservices identification as it represents the dependencies/interactions between source code classes. Previously, we modeled some dependencies (inheritance, method call, and instance creation) as features to enrich the semantic aspect. Furthermore, in this step, we leverage method calls between classes in the distance calculation to enhance the class assignment results.

  We use `struct_rel` to refer to the structural relationship between two classes and we use the following formula to calculate it:

$$struct\_rel(c_i, c_j) = \frac{\text{call}(c_i, c_j)}{\text{total\_calls}} \tag{2}$$

  Where $\text{call}(c_i, c_j)$ is the number of calls between the two classes c_i and c_j. By calls between classes, we mean method invocations between this pair of classes.

  Consequently, the weight between two classes c_i and c_j is calculated as follows:

$$\begin{aligned} weight(c_i, c_j) = cos\_sim(c_i, c_j) + \\ struct\_rel(c_i, c_j) \end{aligned} \tag{3}$$

- **Data relationship**. To assign classes to their database-obtained clusters, we propose to join the corresponding data classes/entities of each database-obtained cluster in the weighted graph G, where we raise the edge weights that link these data classes/entities to guarantee a dense connection between them, while we reduce weights that link data classes/entities in different database-obtained clusters. The weight between two data entities e_i and e_j, representing two data tables within the same cluster, is defined as the maximum value of all weights in the graph G:

$$weight(e_i, e_j) = \max(weight(c_i, c_j)) \tag{4}$$

  The weight between two data entities e_i and e_j, representing two data tables in different clusters, is defined as the minimum value of all weights in the graph G:

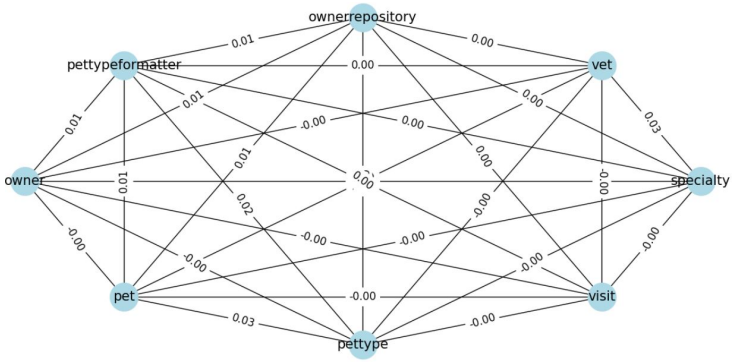


Figure 6: Weighted SubGraph of Petclinic

$$weight(e_i, e_j) = \min(weight(c_i, c_j)) \tag{5}$$

Consequently, source code classes with a strong relationship with data classes/entities will be within the same cluster/community.

In order to standardize the graph weights and avoid significance variance between the values, we normalize them using "Row Normalization" technique [21]. This method guarantees that each row in the adjacency matrix of the graph G sums to 1, thereby standardizing the scale of the weights. This normalization improves the readability and interpretability of results.

An example of a weighted subgraph of the `Petclinic` application is represented in Figure 6, where data entities like `pet` and `pettype` that refer to `pets` and `types` tables respectively, are connected by an edge with the maximum weight. In contrast, data entities, such as `vet` and `visit` which refer to `vets` and `visits` tables respectively, are connected by an edge with the minimum weight.

The prototype implementation of our process is publicly available on the following Github repository:
`https://github.com/Yamina-Romani/Data_CentricProcess.git`.

3. Evaluation

In order to evaluate the proposed method, we address the following research questions.

3.1. Research Questions

- **RQ1** How does the effectiveness and efficiency of our approach compare to alternative methods for migrating monolithic to microservices architectures?

- **RQ2** To what extent do the identified potential microservices align with ground truth?

In order to answer these questions, we conducted two experiments.

3.2. Experiment 1

The goal of this experiment is to assess the performance of our approach compared to other competitive methods. We selected four metrics that are commonly used in the literature. The first two metrics are chosen to evaluate the cohesiveness of obtained microservices. The chosen metrics are calculated according to the identified interfaces of microservices, a microservice interface consists of a set of methods that can be exposed through endpoints.

The second two metrics are chosen to evaluate the modularity of obtained microservices. It represents the trade-off between cohesion and coupling.

The evaluation metrics are defined as follows:

- CHM (Cohesion at Message Level): it is the average cohesion of the obtained microservices, where chm_i measures the cohesiveness of each microservice based on the cohesiveness of its interface at the message level. Specifically, it calculates the similarity between two exposed functions at the message level, given that the two functions are considered related when their input parameters and outputs (returns) are similar [22]. The higher the CHM, the greater the cohesion of a microservice.

$$CHM = \frac{1}{N} \sum_{i=1}^{n} chm_i \tag{6}$$

- CHD (Cohesion at Domain Level): similar to CHM, CHD represents the average cohesion of the obtained microservices, where chd_i measures the cohesiveness of each microservice based on the cohesiveness of its interface at the domain level. Here, the similarity between two exposed functions at the domain level is evaluated by examining the intersection

between the set of domain terms in their function signatures [22]. The higher the CHD, the better the cohesion of a microservice.

$$CHD = \frac{1}{N}\sum_{i=1}^{n} chd_i \tag{7}$$

- SMQ (Structural Modularity Quality): it represents the modularity from a structural insight, where it measures the intra-connectivity and interconnectivity of the obtained architecture. A higher SMQ reflects better modularization of microservices[23].

$$SMQ = \frac{1}{N}\sum_{i=1}^{N} scoh_i - \frac{1}{N(N-1)/2}\sum_{i\neq j}^{N} scop_{ij} \tag{8}$$

$$scoh_i = \frac{u_i}{N_i^2}, \quad scop_{ij} = \frac{\sigma_{ij}}{2(N_i \times N_j)}$$

  scoh calculates cohesiveness of a given microservice while *scop* calculates coupling between microservices.u_i is the number of edges within a microservice i. σ_{ij} is the number of edges between microservice i and microservice j. Ni/Nj is the number of classes within a microservice i/j. An edge exists if there is a structural call dependency between two classes.

- CMQ (Conceptual Modularity Quality): it represents the modularity from a conceptual insight. A higher CMQ reflects better modularity.

$$CMQ = \frac{1}{N}\sum_{i=1}^{N} ccoh_i - \frac{1}{N(N-1)/2}\sum_{i\neq j}^{N} ccop_{ij} \tag{9}$$

$$ccoh_i = \frac{u_i}{N_i^2}, \quad ccop_{ij} = \frac{\sigma_{ij}}{2(N_i \times N_j)}$$

  CMQ uses a similar equation as SMQ, but it differs in using text terms rather than structural call dependencies. An edge exists if the intersection between the textual term set of two classes is not empty [23].

Afterward, we selected two well-known competitive approaches for microservice identification, namely Topic Modelling [8] and MAGNET [24].

Comprehensive descriptions of these approaches are provided in the related works section (Section 5). The rationale behind our choice is the common points between our approach, Topic Modelling, and MAGNET, as they are all automated and rely on static and semantic analysis.

To perform this experiment, we applied our approach to three common open-source projects, PetClinic [3], a Spring-based sample application for managing a veterinary clinic, SpringBlog[4], a blog system implemented with Spring Boot Java framework, and JForum3[5], a forum software created with Java. Table 1 shows the properties of each project.

Project	# Classes	# DB Tables	SLOC
PetClinic	23	7	752
Springblog	85	10	3583
JForum	243	33	22418
PartsUnlimitedMRP	53	5	4407

Table 1: Projects Properties

3.3. Experiment 2

We conducted the second experiment to address **RQ2**. The aim is to evaluate the results obtained by applying our approach to a set of monolithic software systems compared to their existing microservice variants. Accordingly, we selected two common measures used to evaluate the distance between two software architectures: architecture-to-architecture ($a2a$) [25] and cluster-to-cluster coverage ($c2c_{cvg}$)[26].

- Architecture-to-architecture ($a2a$), is a similarity metric that measures the change level in a given software system. A higher value of $a2a$ indicates a better similarity between the obtained microservice architecture and the ground truth [25].

$$a2a(A,B) = (1 - \frac{mto(A,B)}{mto(A_0,A) + mto(A_0,B)}) \ * 100\% \qquad (10)$$

[3] `https://github.com/spring-projects/spring-petclinic`
[4] `https://github.com/bvn13/SpringBlog`
[5] `https://github.com/rafaelsteil/jforum3`

where `mto` stands for minimum-transform-operation, is the minimum number of operations required to move from an architecture to another.

$$mto(A,B) = remC(A,B) + addC(A,B) + remE(A,B) \\ +addE(A,B) + movE(A,B) \tag{11}$$

The five operations needed to change architecture A into architecture B are: additions (addE), removals (remE), and movements (movE) of entities from one cluster to another, as well as additions (addC) and removals (remC) of clusters.

A_0 stands for an initial empty architecture.

- Cluster-to-cluster coverage ($c2c_{cvg}$), it is a metric that quantifies the accuracy at the component(cluster) level by measuring the degree of overlap between the implementation-level entities within two clusters[26].

$$c2c(c_i, c_j) = \frac{|entities(c_i) \cap entities(c_j)|}{|entities(c_i)|, |entities(c_j)|} * 100\% \tag{12}$$

Where c_i is a given cluster in the obtained architecture, c_j is a given cluster in the ground truth architecture, and $entities(c)$ is the set of entities in cluster c.

The calculated value using Equation 12 is then compared to a certain overlap threshold to determine whether these two clusters are similar or not. Where the th_{cvg} can be: 50% (majority match), 33% (moderate match), and 10% (weak match).

The overall coverage ($c2c_{cvg}$) between the identified architecture clusters and the ground truth architecture ones is calculated as follows:

$$c2c_{cvg}(A_1, A_2) = \frac{|simC(A_1, A_2)|}{|A_2.C|} * 100\% \tag{13}$$

$$simC(A_1, A_2) = \{c_i | (ci \in A_1, \exists c_j \in A_2) \wedge \\ (c2c(c_i, c_j)\ th_{cvg})\} \tag{14}$$

Where A_1 is the identified architecture, A_2 is the ground-truth architecture and $A_2.C$ are the clusters of A_2.

The higher the $c2c_{cvg}(A_1, A_2)$ value, the better the match between the identified microservices and the ground truth.

We applied our approach to two open-source projects, each offering monolithic and microservice variants, PetClinic, a spring-based application for managing a veterinary clinic, and PartsUnlimitedMRP[6], a manufacturing resource planning system. Table 1 shows the properties of each project.

For comparison purposes, we excluded microservices and technical modules that do not have direct equivalents in the monolithic architecture (such as the API Gateway, the Config service, the Discovery service, etc.) from the set of microservice variants in the aforementioned projects.

4. Results & Discussion

In this section, we present and discuss our obtained results from the conducted experiment, where, we focus on the qualitative evaluation of our approach.

Therefore, we compared our approach with competing methods in terms of cohesion and modularity. We calculated CHM and CHD metrics to measure the cohesiveness of the identified microservices at both the message and domain levels. Also, we calculated the SMQ and CMQ metrics to measure the modularity of obtained microservices.

Table 2 shows the obtained results of our approach, Topic Modelling[8], and MAGNET[24] for each project.

Regarding Table 2, we obtained the best CHM values for both `Petclinic` and `SpringBlog` projects, indicating an effective microservice cohesion. The **MAGNET** approach achieved the best CHM for `JForum3`. The **Topic Modelling** approach had the lowest CHM values.

In terms of CHD, our approach outperformed both **MAGNET** and **Topic Modelling** for all projects. These results are logical since our approach focuses on semantic similarity between classes in microservice identification, aligning well with the microservices domain.

Overall, our CHM and CHD findings from Experiment 1 reflect the effectiveness of our approach in extracting a set of coherent microservices at both message and domain levels.

Concerning structural modularity, the **Topic Modelling** method obtained the best SMQ for `Petclinic`, but the difference with our method is quite marginal. Our approach yielded the highest SMQ values for `SpringBlog`

[6] `https://github.com/microsoft/PartsUnlimitedMRP`

Project	Metrics	Our Approach	Topic Modelling	MAGNET
Petclinic	# MS	4	4	13
	CHM	**0.56**	0.41	0.55
	CHD	**0.68**	0.43	0.65
	SMQ	0.225	**0.267**	0.05
	CMQ	**0.077**	0.073	0.003
SpringBlog	# MS	5	11	33
	CHM	**0.55**	0.53	0.39
	CHD	**0.64**	**0.63**	0.55
	SMQ	**0.228**	0.200	0.0431
	CMQ	**0.012**	-0.222	-0.002
JForum3	# MS	18	13	75
	CHM	0.56	0.54	**0.62**
	CHD	**0.64**	0.62	0.5
	SMQ	**0.79**	0.52	0.04
	CMQ	-0.63	-0.207	**0.003**

Table 2: Comparison with Competitive Approaches

and `Jforum3` projects, reflecting the use of the `Louvain` algorithm that focuses on detecting modular partitions/microservices. Furthermore, in terms of conceptual modularity, our approach produced the highest CMQ values for `Petclinic` and `SpringBlog` projects, while **MAGNET** achieved the best CMQ value for `JForum3` project. Our CMQ value for `JForum3` indicates greater coupling between microservices in the context of term intersection.

Overall, our SMQ and CMQ findings from Experiment 1 reflect a well-balanced approach between coupling and cohesion in microservices.

In addressing **RQ2**, we quantitatively evaluate our approach, **MAGNET** and **Topic Modelling** by measuring the distance between the obtained microservices and the ground truth using the $a2a$ measure, as well as using $c2c_{cvg}$. The microservices variants of each project are publicly available:

- PetClinic:
  `https://github.com/spring-petclinic/spring-petclinic-microservices`,
- PartsUnlimitedMRP:
  `https://github.com/microsoft/PartsUnlimitedMRPmicro`

The results obtained from the $a2a$ measure are shown in Table 3 for each

Project	**Approaches**	**# GT MS**	**# MS**	$a2a$ **(%)**
PetClinic	Our Approach	3	4	84.3
	Topic Modelling		4	80.6
	MAGNET		13	54.2
PartsUnlimitedMRP	Our Approach	5	6	79
	Topic Modelling		6	75
	MAGNET		31	60

Table 3: $a2a$ Results

project, while Table 4 presents the results of the $c2c_{cvg}$ scores with different overlap thresholds for each project.

According to Table 3, our approach obtained the best $a2a$ scores for the `PetClinic` and `PartsUnlimitedMRP` projects, closely matching the **Topic Modelling** scores. These scores indicate the high similarity between the identified microservices and the ground truth, in addition to highlighting the necessary changes to match the ground truth.

However, the lowest $a2a$ scores are observed for **MAGNET**, reflecting important changes between the obtained microservices and the ground truth architecture compared to the other approaches.

In addition to how each approach identifies microservices, the rationale for these results lies in the exclusion of certain monolith classes from the microservices variant and the duplication of others across different microservices, which increases the gap between the two architectures' microservices.

Projects	**Approaches**	$c2c_{cvg} \geq 10\%$	$c2c_{cvg} \geq 33\%$	$c2c_{cvg} \geq 50\%$
PetClinic	Our approach	100	100	67
	Topic Modelling	100	100	33
	MAGNET	100	33	0
PartsUnlimitedMRP	Our approach	100	100	100
	Topic Modelling	100	100	80
	MAGNET	100	20	0

Table 4: $c2c_{cvg}$ Results

According to Table 4, for $th_{cvg} = 10\%$, all approaches achieved the highest $c2c_{cvg}$ score of 100% across all projects. This indicates perfect alignment in terms of weak matches between the microservices produced by all approaches

and the ground truth.

for $th_{cvg} = 33\%$, our approach and **Topic Modelling** achieved the highest $c2c_{cvg}$ score of 100% across all projects, while **MAGNET** obtained the lowest scores. These scores indicate a perfect alignment in terms of moderate matches between the microservices obtained by our approach and **Topic Modelling**, and the ground truth. In contrast, **MAGNET** shows a weaker alignment in terms of moderate matches between its derived microservices and the ground truth.

for $th_{cvg} = 50\%$, our approach achieved the highest $c2c_{cvg}$ score of 67% for `Petclinic` and 100% for `PartsUnlimitedMRP`, reflecting a well-balanced majority match between the obtained microservices and the ground truth. **Topic Modelling** approach achieved a $c2c_{cvg}$ score of 33% for `PetClinic` and 80% for `PartsUnlimitedMRP`, indicating an acceptable majority match between its obtained microservices and the ground truth. **MAGNET** approach achieved a $c2c_{cvg}$ score of 0%, indicating a dissimilarity between its obtained microservices and the ground truth.

Overall, our findings from Experiment 2 demonstrate the effectiveness of our approach in identifying microservice boundaries that closely align with expert-designed architectures. The consistently high $c2c_{cvg}$ scores across different projects and thresholds, particularly at the stringent 50% threshold, provide strong evidence for the performance of our data-centric decomposition strategy. These results suggest that our approach can serve as a valuable tool for architects and developers in the decomposition of a monolithic system.

4.1. Threats to Validity

Internal validity. The internal validity of our study faces several important challenges that merit discussion. Primarily, our comparison with other methods relies on results generated using specific clustering hyperparameters, which could impact the validity of our conclusions. To address this concern, we conducted extensive testing with multiple parameter configurations, documenting results across various runs with different parameter values. This systematic variation of parameters helped establish the robustness of our approach across different settings. Although we acknowledge that additional evaluation metrics could provide further information, our choice of the $a2a$ metric, supplemented by CHM, CHD, SMQ and CMQ metrics, provides a solid foundation for comparing microservice identification approaches.

External validity. Our study's generalizability faces potential limitations in two aspects. First, while our sample size of test projects is relatively small, the selected projects represent a substantial range in terms of complexity and scale. Our test cases span from small applications with just a few hundred lines of code to larger systems with over 20,000 lines of code, and database schemas ranging from 5 to 33 tables. These projects also represent diverse business domains, including veterinary management, content management, and forum systems, providing a broad spectrum for validation.

The second aspect of external validity concerns our focus on Java applications for the class analysis phase. However, our approach's fundamental principles transcend specific programming languages. The database schema analysis, which forms the foundation of our method, is inherently language-agnostic. Our successful preliminary tests with TypeScript/NodeJS applications, particularly those following similar multi-layered architectural patterns (controller, service, and data access layers), provide encouraging evidence of our approach's broader applicability. These applications exhibited comparable characteristics in terms of code organization and data access patterns, suggesting our method's effectiveness across different technology stacks.

5. Related Work

In recent years, migration to microservices architecture has gained a lot of attention. Many approaches of microservice identification and monolithic software decomposition have been proposed. For instance, the authors in[27] propose a variety of strategies to decompose monolithic software into independent services. Their paper compares 10 existing strategies of code refactoring. These strategies are classified based on their decomposition techniques and they are illustrated as a flow chart to simplify the identification of the best approach for a given scenario.

From our side, we classified a set of current works of monolithic software decomposition. We distinguished two categories based on the point of view of the microservice identification approaches.

5.1. Business-Logic-Oriented Approaches

This category involves the microservice identification approaches that count on the business logic perspective.

Carvalho et al. [28] used a code analysis technique for analyzing Java legacy system and then visualized it as a graph. Methods are represented

as vertices and relationships between methods are denoted by edges. Then, they conducted a multi-objective search on that graph to extract a set of microservices using Non-dominated Sorting Genetic Algorithm III (NSGA-III) considering five criteria: cohesion, coupling, feature modularization, reuse, and communication overhead.

Similarly, Sellami et al. [29] modeled the microservices identification process as a multi-objective optimization problem, where they provided the MSExtractor technique that analyzes the source code of a given OO application and uses the Indicator-Based Evolutionary Algorithm (IBEA) to search and to assess different combinations of classes in order to obtain the optimal microservices candidates with the best cohesion, coupling and granularity. Furthermore, this work combined the semantic and structural dependencies between classes, where the dependencies are estimated using two different types of similarity.

Zhang et al. [30] presented an automated microservice identification approach that aimed to breakdown a legacy software system into parts using execution and performance logs of the system. The approach considered two types of objects: controller objects (COs) and subordinate objects (SOs). Based on the measured relation between each pair of CO and SO, the system classes were grouped into microservices using NSGA by optimizing the objectives associated with functional (coupling and cohesion) and non functional (load-balancing) metrics.

The Mono2Micro approach developed at IBM to migrate monolithic applications towards a microservice architecture has been proposed by Kalia et al.[31]. Mono2Micro employed a hierarchical spatio-temporal decomposition using well-defined business use cases (the space dimension) and its runtime traces (time dimension) to partition the application classes. This approach takes into consideration cohesion and coupling criteria to split Java legacy software systems.

Brito et al.[8], Topic Modelling was used to detect microservices based on domain terms, where it extracted relevant terms from the source code and used them to obtain topics. Then a weighted graph was generated based on the structural dependencies and the distribution of topics among classes. At last, microservices were identified by applying the Louvain community detection algorithm on the generated graph.

The authors in [7] relied on the analysis of the source code of a monolithic software decompose it into a set of microservices. They classified the source code classes by labeling them according to their software layers (Application,

Entity, Utility) using Support Vector Machine (SVM) and Graph Convolutional Network (GCN). The classes of the same layer (i.e., of the same label) were then grouped into services based on static relationships using the Louvain community detection algorithm. Microservices were generated based on the obtained services by analyzing the static and semantic relationships among them. The Application service was chosen as the core business component of each microservice, and then the related Entity and Utility services were merged to form the final microservices.

Chen et al.[32] presented a method for identifying microservices by performing semantic, static, and dynamic analysis on the monolith, transforming it into multiple views. These views are merged into a fusion view representing a set of comprehensive features of the monolith source code. Deep Fusion Graph Clustering model and a Partitioning Quality Assessment module were utilized to assess and identify the optimal partition.

Trabelsi et al.[24] proposed a fully automated approach for microservices identification. They relied on static analysis of the source code of a monolithic software using the MSDISCO eclipse plugin to generate the Knowledge Discovery Metamodel (KDM) that was used in generating a method dependency graph. Also, they integrated the semantic analysis where each method in the dependency graph was vectorized based on its semantic features derived from its definition and body, the obtained vectors were used to create a feature matrix. A Feature-Rich Dependency Graph was built using the method dependency graph and the feature matrix. Microservices were generated based on the obtained Feature-Rich Dependency Graph by an unsupervised clustering through a graph neural network (GNN).

Ding et al.[33] proposed a microservice extraction method that considered both logical independence and performance. They used a meta-heuristic search-based solution algorithm to identify the best microservices. The algorithm aims to optimize the response time of all requests, reflecting performance while improving the IFN(Interface Number), CHM(Cohesion at Message Level), and CHD(Cohesion at Domaine Level) values, which reflect the logical independence of extracted microservices.

The aforementioned works mainly rely on source code analysis and functional aspects to split monolithic. In contrast, our proposal takes a data model-centric perspective. The data model is an artifact that exists in all situations in enterprise applications. We argue that using only this artifact as a preliminary step facilitates the microservices identification task.

5.2. Data-Oriented Approaches

The following approaches took into consideration the data in their microservice identification processes.

Newman. [34] proposed some architectural patterns to manually breakdown a database to fit a microservice architecture. This work considered a set of seeds (*already identified* microservices) on which the approach worked. Some of these patterns achieved the logical separation of databases in case of shared data such as : i) the "Database view" pattern that considers a view as a constrained projection from a core database schema and a service can access only the data provided by its view, this pattern is used to reduce coupling-related issues, ii) the "Database Wrapping Service" pattern which aims to hide the database behind a service that performs as a lightweight wrapper and data can be accessed only through it(whenever using this pattern, updating data implies switching from direct DB access to an API call), iii) the "Database-as-a-Service Interface" pattern that provides a separate database as an endpoint for read-only, while the internal database is hidden. The other patterns faced the physical database decomposition, where the author recommended `splitting the data before the code` in order to identify potential problems and determine whether a database separation is necessary. After that, the application code may be separated into services. This separation can be realized with the "Database per Bounded Context" pattern where the database is divided around the boundaries of the defined bounded contexts. Moreover, in case of `splitting the code first`, the data of each identified microservice should be associated with the microservice's own schema. However, partitioning a relational database introduces new issues including `shared tables` between services and breaking `foreign key connections`, which are handled by the "Split Table" and "Move Foreign-Key Relationship to Code" patterns.

In our work, we are interested in the physical decomposition of databases to automate the identification of microservices. To this end, we proposed splitting the database models of a given software system into a set of submodels that would then be bundled with their corresponding business logic to a set of microservices.

Barbosa et al. [35] introduced a manual approach to identify microservice candidates using business rules that were executed in stored procedures. This technique identified the system requirements using an expert and examined the source code to map the stored procedures that correspond to the implementation of these business requirements. Then, database artifacts

(stored procedures) were analyzed to detect all business rules. Then, rules that handled the same business requirements were merged into the same microservices.

The authors in [36] presented a semi-automatic decomposition process of monolithic software. The decomposition was based on dataflow diagrams and aimed to identify microservices. This approach created a use case and business logic specification by means of a business requirement analysis. Then, it generated a fine-grained Data Flow Diagram (DFD) and the process-datastore version of this DFD which designated the business logic. The dependencies between processes and data stores were extracted and put in decomposable sentence sets that were grouped into individual modules to formulate the microservice candidates.

From their perspectives, the authors in[37] suggested a method based on the business functionalities that were captured via use cases. They used two relationship matrices as: i) *Use Case-Use Case Relationship Matrix*, which illustrated the degree of interdependence between all use cases. **Include**, **Extend**, and **Generalization** were the possible relationships. Depending on their priority, each relationship was assigned a weight. ii) *Use Case-Database Entities Relationship Matrix*, the **Create**, **Read**, **Update**, and **Delete** tags were used to indicate the link between use cases and database entities in this matrix, which were subsequently transformed into numeric values in accordance with their priority. After that, a *Resemblance Matrix* for *Use Case-Database Entities Relationship Matrix* was created to identify the degree of similarity or dissimilarity between use cases in terms of accessing the database entities. The *Resemblance* and *Use Case-Use Case Relationship* matrices were aggregated to produce a *Combined Similarity Matrix* that was used as input for Hierarchical Agglomerative Clustering to generate a consistent set of use cases that can be combined into microservices.

Quattrocchi et al. [38] presented Cromlech, a semi-automated tool for decomposing a monolithic application into microservices. Cromlech took as input a high-level description of a software system in terms of operations and data entities, the maximum number of microservices, and a parameter indicating the relative importance of organizational aspects over operational ones. It utilized a Mixed Integer Linear Programming (MILP) solver to generate the microservice decomposition, optimizing the placement of operations and data entities according to the user's needs.

In the aforementioned works, the authors exploited data artifacts besides the business logic to find candidates for microservices within monoliths. In

our work, we focus on the use of database artifacts as the initial drivers to identify microservices. Business logic is used later to complete the obtained microservices.

Authors in [39] proposed a microservice identification technique based on examining database access requests. They relied on the belief that various service endpoints accessing the same data tables/collections would have similar duties. Thus, similar service endpoints must be in the same microservice. To this end, they created a new annotation named "AMIADA" to identify service endpoints indicated in the *Controller Layer* of a given application, where if any service endpoint receives a request, all pertinent information, including the service endpoint that accessed the database, the kind of operation (GET, POST, ...), and the name of the associated database table/collection, will be immediately logged into the *log DB*. After that, based on the request methods and a Likert 5-point rating system, the strength of relationships between service endpoints and data tables/collections was evaluated. The Post method received the highest score, followed by the PUT method for the second place, and the GET method at last. The obtained scores of service endpoints visiting the data tables were employed to generate score vectors. Later, the relationships between the service endpoints were measured using the cosine similarity between the resulting vectors and the hierarchical clustering was applied to group these vectors (service endpoints) into a set of microservice candidates.

Li et al.[40] presented an automatic method for microservice identification from a validated requirement model that was generated by a tool called RM2PT. The model consists of a conceptual class diagram, a use case diagram, system sequence diagrams for use cases, and the formal contracts of their system operations in OCL (Object Constraint Language). From the validated requirement model, a function-data dependency diagram was extracted and used by the multi-objective optimization algorithm (NSGA-II) to generate a set of microservices with desirable cohesive coupling properties.

Bajaj et al.[9] suggested a comprehensive approach to extract microservices from a monolithic software system. They applied a static analysis of the source code to create both a Class and Method Dependency Graph. In the class dependency graph, interface classes were linked to internal classes, with the edge weights defined by the frequency of calls between classes. Classes with strong dependencies were grouped to generate microservice candidates. The same procedure was applied to the method dependency graph. Later, data entities were associated with their microservices using a set of guiding

rules that addressed ownership conflicts.

Compared to the above-mentioned works, our approach is different in several aspects. First, while other approaches either rely primarily on source code analysis or combine multiple artifacts from the start, our process uniquely positions database tables/collections as the primary starting point for microservices identification. This data-centric perspective leverages the database schema as a natural domain-concept model, reflecting established business boundaries that often remain more stable than code structures. Second, our approach innovatively combines both semantic and structural relationships between database entities, where semantic analysis captures the business domain knowledge embedded in table names and relationships, while the structural analysis reflects the logical organization of the data model. This dual analysis provides a more robust foundation for identifying microservice boundaries than approaches focusing solely on code dependencies or runtime behavior. Furthermore, by using the database model as an initial driver and only later incorporating code analysis, our method reduces the complexity of the identification process while maintaining alignment with the database-per-service pattern fundamental to microservice architecture. This staged approach not only simplifies the decomposition process but also helps identify potential data sharing and consistency challenges earlier in the migration process, leading to more maintainable and cohesive microservice architectures.

6. Conclusion

In this paper, we presented an approach for microservice identification during the migration of monolithic applications. The approach leverages data models by employing topic modeling to decompose the monolithic database schema into thematically cohesive sub-models. These sub-models then serve as potential candidates for individual microservices with their own independent databases. Furthermore, we analyze the source code of the monolith and we assign classes to their relevant clusters using the Louvain algorithm, which receives a weighted graph, its nodes refer to the monolithic source code classes and its edges refer to the relationships between them. The edge weights are calculated using semantic, structural, and data relationships. The qualitative evaluation of our approach showed that our process is effective in identifying a set of well-modularized microservices that exhibit strong cohesion in terms of both domain- and message-level functionalities. Meanwhile,

the quantitative evaluation demonstrated the competence of our process in obtaining a set of microservices that strongly match the ground truth.

In the future, we aim to enhance our approach by analyzing transaction patterns in source code, focusing on operations that span multiple tables and for which ACID properties must be preserved. By analyzing CRUD operation frequencies and table modification patterns, we can refine microservice boundaries based on runtime data access patterns. This analysis will enable more precise service decomposition recommendations that better reflect actual data usage patterns.

In addition, we plan to develop automated support for the migration process itself, particularly around API design and communication patterns. By analyzing data access layer code, we aim to provide concrete recommendations for service interfaces and communication modes (synchronous, asynchronous, or message broker-based). This tooling will help bridge the gap between microservice identification and practical implementation, making the migration process more systematic and reliable.

More generally, in our future project, we would like to integrate other extra-functional characteristics, like security (access control), scalability, and resilience in the identification of microservices and source code assignment. Indeed, in some projects, these characteristics may have a strong influence on the migration to microservices. Thereby they should be considered as drivers in the modernization of existing legacy monolithic software systems, together with those considered in our process.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in some parts to improve syntax and grammar. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

References

[1] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd Edition, O'Reilly Media, 2021.

[2] P. Di Francesco, P. Lago, I. Malavolta, Architecting with microservices: A systematic mapping study, Journal of Systems and Software

150 (2019) 77–97. URL: https://www.sciencedirect.com/science/article/pii/S0164121219300019. doi:https://doi.org/10.1016/j.jss.2019.01.001.

[3] I. Vayansky, S. A. Kumar, A review of topic modeling methods, Information Systems 94 (2020) 101582.

[4] M. Hagiwara, Real-World Natural Language Processing: Practical Applications with Deep Learning, Manning, 2021. URL: https://books.google.dz/books?id=Ok5NEAAAQBAJ.

[5] G. Bonaccorso, Machine Learning Algorithms: Popular Algorithms for Data Science and Machine Learning, 2nd Ed., Packt Publishing, 2018. URL: https://books.google.dz/books?id=4xjAuQEACAAJ.

[6] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).

[7] I. Trabelsi, M. Abdellatif, A. Abubaker, N. Moha, S. Mosser, S. Ebrahimi-Kahou, Y.-G. Guéhéneuc, From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis, Journal of Software: Evolution and Process 35 (2023) e2503.

[8] M. Brito, J. Cunha, J. Saraiva, Identification of microservices from monolithic applications through topic modelling, in: Proceedings of the 36th Annual ACM Symposium on Applied Computing, 2021, pp. 1409–1418.

[9] D. Bajaj, A. Goel, S. C. Gupta, A comprehensive microservice extraction approach integrating business functions and database entities., Int. Arab J. Inf. Technol. 21 (2024) 32–45.

[10] C. Sas, A. Capiluppi, Using structural and semantic information to identify software components, in: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2021, pp. 546–550.

[11] G. Mazlami, J. Cito, P. Leitner, Extraction of microservices from monolithic software architectures, in: 2017 IEEE International Conference on Web Services (ICWS), IEEE, 2017, pp. 524–531.

[12] P. Dangeti, Statistics for machine learning, Packt Publishing Ltd, 2017.

[13] Y. Romani, O. Tibermacine, C. Tibermacine, Towards migrating legacy software systems to microservice-based architectures: a data-centric process for microservice identification, in: 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C), IEEE, 2022, pp. 15–19.

[14] L. Billard, E. Diday, Clustering Methodology for Symbolic Data, Wiley Series in Computational Statistics, Wiley, 2019. URL: `https://books.google.dz/books?id=-eWnDwAAQBAJ`.

[15] T. Aynaud, J.-L. Guillaume, Static community detection algorithms for evolving networks, in: 8th international symposium on modeling and optimization in mobile, ad hoc, and wireless networks, IEEE, 2010, pp. 513–519.

[16] S. Rahiminejad, M. R. Maurya, S. Subramaniam, Topological and functional comparison of community detection algorithms in biological networks, BMC bioinformatics 20 (2019) 1–25.

[17] J. Mothe, K. Mkhitaryan, M. Haroutunian, Community detection: Comparison of state of the art algorithms, in: 2017 Computer Science and Information Technologies (CSIT), IEEE, 2017, pp. 125–129.

[18] M. G. Puxeddu, M. Petti, F. Pichiorri, F. Cincotti, D. Mattia, L. Astolfi, Community detection: Comparison among clustering algorithms and application to eeg-based brain networks, in: 2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), IEEE, 2017, pp. 3965–3968.

[19] S. Wibisono, D. Manongga, I. Sembiring, et al., Comparative analysis of louvain, leiden, and walktrap algorithms for community detection in the network of indonesian laws, in: 2024 IEEE 10th Information Technology International Seminar (ITIS), IEEE, 2024, pp. 232–239.

[20] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, E. Lefebvre, Fast unfolding of communities in large networks, Journal of statistical mechanics: theory and experiment 2008 (2008) P10008.

[21] M. J. Zaki, W. Meira Jr, W. Meira, Data mining and machine learning: Fundamental concepts and algorithms, Cambridge University Press, 2020.

[22] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, Q. Zheng, Service candidate identification from monolithic systems based on execution traces, IEEE Transactions on Software Engineering 47 (2019) 987–1007.

[23] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, E. R. Gansner, Using automatic clustering to produce high-level system organizations of source code, in: Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242), IEEE, 1998, pp. 45–52.

[24] I. Trabelsi, N. Moha, Y.-G. Gueheneuc, L. Geffard, Magnet: Method-based approach using graph neural network for microservices identification, in: 2024 IEEE 21st International Conference on Software Architecture (ICSA), IEEE Computer Society, 2024, pp. 1–11.

[25] P. Behnamghader, D. Le, J. Garcia, D. Link, A. Shahbazian, N. Medvidovic, A large-scale study of architectural evolution in open-source software systems, Empirical Software Engineering 22 (2017) 1146–1193.

[26] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, R. Kroeger, Measuring the impact of code dependencies on software architecture recovery techniques, IEEE Transactions on Software Engineering 44 (2017) 159–181.

[27] J. Fritzsch, J. Bogner, A. Zimmermann, S. Wagner, From monolith to microservices: A classification of refactoring approaches, in: International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment, 2018, pp. 128–141.

[28] L. Carvalho, A. Garcia, T. E. Colanzi, W. K. Assunção, J. A. Pereira, B. Fonseca, M. Ribeiro, M. J. de Lima, C. Lucena, On the performance and adoption of search-based microservice identification with

tomicroservices, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 569–580.

[29] K. Sellami, A. Ouni, M. Saied, S. Bouktif, M. Mkaouer, Improving microservices extraction using evolutionary search, Information and Software Technology 151 (2022) 106996.

[30] Y. Zhang, B. Liu, L. Dai, K. Chen, X. Cao, Automated microservice identification in legacy systems with functional and non-functional metrics, in: 2020 IEEE International Conference on Software Architecture (ICSA), IEEE, 2020, pp. 135–145.

[31] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, D. Banerjee, Mono2micro: a practical and effective tool for decomposing monolithic java applications to microservices, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 1214–1224.

[32] G. Chen, C. Li, S. Tyszberowicz, Z. Liu, B. Liu, Mono2ms: Deep fusion of multi-source features for partitioning monolith into microservices, in: Proceedings of the 15th Asia-Pacific Symposium on Internetware, 2024, pp. 259–268.

[33] Z. Ding, Y. Xu, B. Feng, C. Jiang, Microservice extraction based on a comprehensive evaluation of logical independence and performance, IEEE Transactions on Software Engineering (2024).

[34] S. Newman, Monolith to microservices: evolutionary patterns to transform your monolith, O'Reilly Media, 2019.

[35] M. H. G. Barbosa, P. H. M. Maia, Towards identifying microservice candidates from business rules implemented in stored procedures, in: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), IEEE, 2020, pp. 41–48.

[36] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, Z. Shan, A dataflow-driven approach to identifying microservices from monolithic applications, Journal of Systems and Software 157 (2019) 110380.

[37] D. Bajaj, A. Goel, S. Gupta, Greenmicro: Identifying microservices from use cases in greenfield development, IEEE Access 10 (2022) 67008–67018.

[38] G. Quattrocchi, D. Cocco, S. Staffa, A. Margara, G. Cugola, Cromlech: Semi-automated monolith decomposition into microservices, IEEE Transactions on Services Computing (2024).

[39] S. P. Ma, T. W. Lu, C. C. Li, Migrating monoliths to microservices based on the analysis of database access requests, in: 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE), IEEE, 2022, pp. 11–18.

[40] Y. Li, Y. Zhang, Y. Yang, W. Wang, Y. Yin, Rm2ms: A tool for automatic identification of microservices from requirements models, in: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), IEEE, 2023, pp. 50–54.