2022 IEEE 19th International Conference on Software Architecture (ICSA) | 978-1-6654-1728-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSA53651.2022.00021

Leveraging the Layered Architecture for Microservice Recovery

Pascal Zaragoza
MAREL, LIRMM
DRIT, Berger-Levrault
Montpellier, France
zaragoza@lirmm.fr

Abdelhak-Djamel Seriai
MAREL, LIRMM
Montpellier, France
seriai@lirmm.fr

Abderrahmane Seriai
DRIT, Berger-Levrault
Montpellier, France
abderrahmane.seriai
@berger-levrault.com

Anas Shatnawi
DRIT, Berger-Levrault
Montpellier, France
anas.shatnawi@berger-levrault.com

Mustapha Derras
DRIT, Berger-Levrault
Montpellier, France
mustapha.derras@berger-levrault.com

***Abstract*—The microservice-oriented architecture (MSA) is an architectural style which involves organizing an application as of small independent services, each oriented towards one business functionality while being data autonomous. In pursuit of modernizing their software to take advantage of the Cloud, companies have been eager to migrate their monolithic legacy software towards an MSA. This migration necessitates an identification phase to reorganize classes around the monolith's functionalities as a set of microservice candidates. However, most identification approaches fail to utilize the monolith's internal multilayered architecture to identify those functionalities, and thus the microservices. As a consequence, ignoring the internal multilayered architecture increases the risk of identifying microservice by their technical layer which is recognized as a conceptual anti-pattern. In this paper, we explore the impact of the multi-layer architecture in monolithic applications during the identification to develop a semi-automatic approach that relies on it to identify an MSA. Particularly, we analyze the presentation layer to determine the endpoints of each business functionality of the monolith. From these endpoints, we apply a vertical decomposition to identify the necessary classes to implement each feature as a microservice. In the process, we also define the bounded context of each microservice during the vertical decomposition of the data-access layer. For the evaluation, we implemented a model-driven process and applied it on a set of varying open-source applications commonly used in the literature. We compared the results of approach with and without the reverse-engineering of the internal architecture to measure the impact of our approach on the identification of quality microservices. Using decomposition metrics (e.g., MoJoFM, $c2c_{cvg}$), we were able to measure a significant positive impact.**

***Index Terms*—identification, extraction, decomposition, software re-engineering, software migration, software architecture, microservices, monolithic, object-oriented**

I. Introduction

Microservice-Oriented Architecture (MSA) is one of the latest trends in software development that has emerged from service-oriented architecture styles. Microservices are a set of small, autonomous, highly-cohesive, and loosely-coupled services, each independently deployable and communicating with lightweight mechanisms (such as HTTP, AMQP, etc.) [1]. These characteristics are well-suited for the Cloud as they can be developed independently, are more easily deployable, and are re-usable by other applications. Furthermore, these small services facilitate both maintenance and scalability. Finally, a microservice architecture's modularity allows for a smoother adoption of Development & Operation (DevOps) techniques [2].

In general, enterprise software are often built as a 3-part system: a client-side user interface, a database, and a server-side application [1]. The server-side application is often built as a single logical executable and is often referred to as the monolith [1]. These monoliths are often built as a single-tiered system with multiple layers, namely: presentation, business logic, and data-access layers. In the early stages of developing monolithic applications, the architecture is simpler to develop, test, and deploy. However, as they grow and age, they become more complex and harder to maintain [3]. For these reasons, many enterprises are looking at migrating their existing legacy *monolithic* applications towards an MSA to benefits from its advantages.

The software migration towards a microservice-oriented architecture involves two phases : (1) the identification of a microservice-based architecture from the existing application, and (2) transformation of the monolithic source code to conform to microservices-oriented architecture principles. The purpose of the first phase is to analyze the existing legacy software artifacts and extract microservice candidates to guide the transformation phase.

Several approaches have been proposed to migrate monolithic applications towards microservice-oriented ones. These approaches often focus on identifying microservices by promoting highly-cohesive and loosely-coupled structural modules. However, they often do not consider that enterprise applications are built upon standardized frameworks which promote a layered architecture [4]. As a result, they can fall into the trap of creating microservice based on technical layers. Indeed, according to experts, one of the most harmful anti-

978-1-6654-1728-0/22/$31.00 ©2022 IEEE
DOI 10.1109/ICSA53651.2022.00021

patterns to consider during the identification is the Wrong Cuts [5]. This anti-pattern proscribes for microservices to be split based on technical layers (presentation, business, and data layers). Instead, popular decomposition patterns prescribe that microservices should be split based on business capabilities with a vertical integration of all layers within one small autonomous application [4]. Consequently, their implementation has to be vertical or 'cross-layer' (i.e., composed of the three layers).

Popular frameworks such as Spring[1], ASP.NET Core[2], or Node.js-based ones[3] all rely on a technically-layered architecture to promote a separation of concerns. These layered architectures often take the form of a 3-layer architecture with a presentation layer, business layer, and a data-access layer. Alternatively, the Model-View-Controller (MVC) pattern also promotes a similar separation of concern between the 3 different elements of the pattern.

In this paper, we leverage the underlying architecture found in industrial monoliths to propose a decomposition technique that prioritizes the identification of microservices based on their business capabilities first. We leverage the architecture in two ways: (1) we use the presentation layer to drive the clustering process from a usage perspective, (2) we use the data-access layer to ensure a level of data autonomy for each microservice. Regarding the presentation layer, we use its high-level business functions to guide the decomposition based on business capabilities to avoid creating the Wrong Cuts antipattern. With regard to the data-access layer, we are able to avoid dangerous anti-patterns such as the Shared Persistence [5] which pertains to microservices that end up using the same data, thus reducing team and service independence [5].

This paper proposes an approach in three parts. The first contribution is a reverse-engineering step for extracting the layered architecture found in most monolithic applications. The second contribution consists of proposing a fitness function to evaluate the quality of a microservice based on their cohesion, internal coupling, and the manipulated data. The third contribution is an identification/clustering process guided by the fitness function and which takes into consideration the recovered internal multi-layered architecture, as well as the design patterns and antipatterns discussed previously. To aid in the extraction of a the layered architecture of the monolith, we propose a metamodel to represent the extracted architecture.

To evaluate our approach, we apply it to a set of monolithic applications using different architectural styles and frameworks. By applying our approach onto these applications we aim to answer the following questions:

- **RQ1**: Does the extraction of the layered architecture have a positive impact on the results of the identification process?
- **RQ2**: How does our approach compare to the state of the art?

[1]https://spring.io/
[2]https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-5.0
[3]https://nodejs.org/

With regards to the **RQ1**, we aim to establish whether or not the addition of an intermediate architecture extraction step increases the extraction accuracy of the microservice candidate regarding an expert decomposition. In **RQ2**, we wish to compare the existing approaches with the proposed approach with a case study, in an effort to highlight the advantages of an architecture-level decomposition.

The rest of this article is organized as follows: Section II introduces the context of the paper by presenting the typical 3-tier architecture with JPetStore as a motivating example; Section III explains the identification process step-by-step; Section IV presents the implementation and evaluation of the approach; Section V presents the current state-of-the-art with regards to the identification of microservice; Section VI concludes the paper and draws attention to some possible future works.

II. Motivating Example : JPetStore

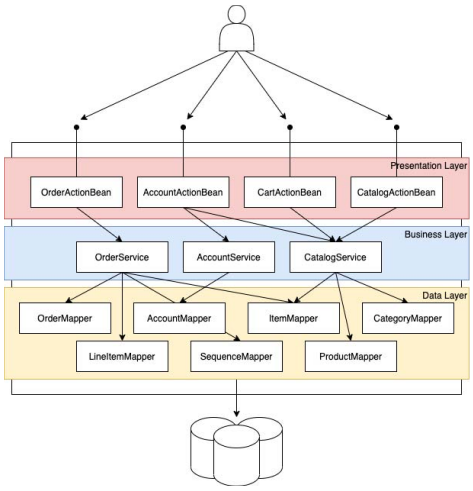

Fig. 1. Layered Architecture of JPetStore.

To better illustrate the problems and solutions related to microservice identification for object-oriented monolithic applications, we introduce JPetStore[4]. JPetStore is a typical web application implementing a 3-Layer architecture that acts as an online pet commerce. In essence, JPetStore contains 4 main features (functionalities): account, catalog, shopping cart, and order management.

Structurally, it contains 24 classes, of which 14 are reusable structural classes and 9 data entities. They are used in conjunction with the Inversion of Control (IoC) pattern and the Dependency Injection (DI) pattern to create loosely-coupled and highly-cohesive applications.

Each class can be mapped to one of the three layers described in the typical 3-tier architecture: the presentation layer, the business layer, and the data layer. The presentation layer is the topmost level of the application. This layer is responsible for displaying the information related to JPetStore's features.

[4]https://github.com/mybatis/jpetstore-6

The business layer acts as an intermediary between the presentation layer and the data layer. This layer is responsible for performing the business logic of the application. Finally, the data layer is responsible for applying the domain logic of the application as well as the data persistence mechanisms. In the case of JPetStore, we map the 14 structural classes to the three layers presented in Figure 1.

Ideally, we want to extract a microservice for each feature and their underlying implementation. This implies decomposing the application vertically to provide each microservice with a business functionality, its underlying business logic, and data access. In Figure 2, we illustrate the risk of identifying microservices based on the technical layers (i.e., a wrong cut) by showing the decomposition proposed in [6]. In this example, the authors propose 4 microservice candidates. Two microservice candidates (MS1 & MS2) offer a vertical decomposition while being business-oriented and containing some data autonomy. However, there is a horizontal slice that creates a microservice candidate (MS4) which only contains data-access classes, and a microservice candidate (MS2) with the business functionality.

In the next section, we present our approach using JPetStore as an example.

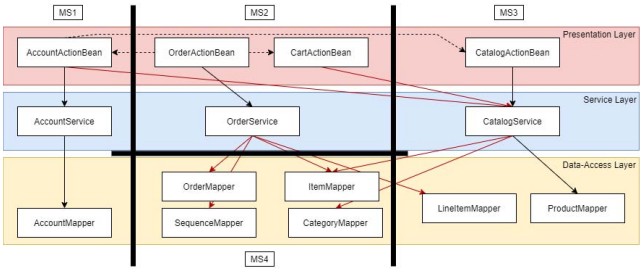

Fig. 2. Proposed decomposition of JPetStore by [6].

III. Layered Identification Approach

The layered architecture identification approach leverages well-established design patterns and antipatterns to guide the process of identifying microservices. Just like the metaphor of the carrot and stick, we motivate the identification process through observed design patterns while inhibiting it through bad decomposition patterns. Before explaining this identification approach, it is necessary to cover some patterns proposed to design MSA [4], [5].

One pattern proposed by [4], is to decompose monoliths based on the business capabilities of the monolithic application. This pattern promotes microservice decomposition based on the business-oriented functionalities. For example, in the context of an e-commerce, it can include order management, account management, and product management. It is important to note that a business capability is often focused on a particular business objects (i.e., data entity) [4].

Among the 20 microservice anti-patterns surveyed by [5], two can be attributed to the identification phase of the migration process. The first anti-pattern is that of the Wrong

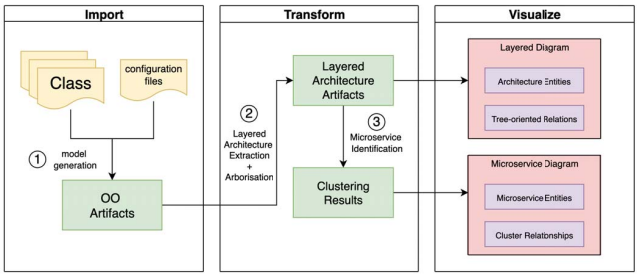

Fig. 3. The microservice architecture identification process.

Cuts. This anti-pattern appears when microservices are split based on technical layers (i.e., presentation, business, and data layer) instead of its business capabilities. This can increase data-splitting complexity according to [5], and lead to increased communication between microservices. The second antipattern (i.e., Shared Persistence), involves different microservices accessing the same database. This anti-pattern creates a strong coupling between microservices manipulating the same data, thus reducing both development team and service independence [5]. Generally, this goes against one of the main characteristics of microservices : data autonomy. To avoid this antipattern, it is important to assign data ownership to specific microservices. One way to do so, is by assigning the business classes responsible for the data-access to specific microservices.

The objective of this paper is to propose an approach which takes into consideration these design patterns and antipatterns. The process of our proposed identification approach is illustrated in Figure 3, and can be described as a three-step process. The initial step involves (i) serves to recover the class artifacts from the object-oriented source. Then, (ii) the class artifacts are categorized into different layers of the architecture (i.e., presentation, business, and data-access). Finally, from the extracted internally layered architecture's artifacts, (iii) the microservice candidates are identified through an automatic clustering approach.

Concretely, we present our approach in two parts. First, we present the extraction of the layered architecture and how it is represented. Second, we present the identification process that takes into consideration the extracted artifacts, as well as the design patterns and the anti-patterns discussed previously. In addition, a quality metric is proposed to assist in the identification process.

A. Reverse-engineering the layered architecture

The first step towards identifying microservice candidates is to extract the layered architecture artifacts from the existing source code of the monolithic application. This step aims at analyzing the object-oriented source code to extract OO artifacts. It involves identifying the monolithic application's structural elements (e.g., classes, methods, etc.) and the relationships between them (e.g., method calls, class inheritance, etc.) by analyzing the existing source code. Additionally, the

project configurations may be analyzed to extract additional information on the structure of the source code.

From these OO artifacts, the layered architecture can be revealed through reverse-engineering techniques. In most modern frameworks, the OO artifacts are annotated based on the technical function they serve. In the case of the Spring framework, annotations such as `@RestController`, `@Service`, and `@Repository` serve to label the classes based on their responsibility. When appropriately labeled, the extraction of a layered architecture can be automatically induced. However, in the case of applications which do not use such frameworks, a manual labeling process is required.

In either cases, to facilitate the labeling process and to represent the extracted layered architecture of monolithic applications, we propose the Layered Architecture Metamodel (LAMM). LAMM is illustrated in Figure 4, and can be divided into three viewpoints.

The first viewpoint (Layered Architecture) is responsible for representing the three layers present in the typical 3-Layer architecture: presentation, business-logic, and data-access layer. The presentation layer contains the classes responsible for interacting with the user interface (UI), and it handles the requests generated by the user (i.e., the *Controller* entity). The business-logic layer contains the classes responsible for the business-logic of the application (i.e, the *Service* entity). It is often described as the service layer, and it acts as the middleman between the presentation layer and the data-access layer. Finally, the data-access layer is composed of the classes responsible for the data persistence mechanism and the data-access that encapsulates the persistence mechanism and exposes the data (i.e., the *DAO* entity).

The second viewpoint (DI/IoC) is responsible for representing the decoupling mechanism found in most frameworks to create highly-cohesive layer artifacts. The entity *LayerArtifact* is extended by the 3 entities found in the Layered Architecture viewpoint. It is implemented by a class entity represented by the OO artifacts, and can be described by one or more interfaces.

The third viewpoint (Data Persistence) is responsible for representing the various data types found in web applications. Particularly, we denote two types the *Data Entity* and *DTO* (i.e, Data-Transfer Object) which specialize the *DataType* entity. *Data Entities* represent the implementation of a data table, while the *DTO* represents a data structure that can be easily serialized and transferred to the client.

A semi-automatic process is applied to place the application's classes into one of these groups by mapping them to one of the LAMM entities. In the case of *JPetStore*, the application uses a annotation-based framework which encourages the annotation of controller, service, DAO, and data entity classes. 23 of the 24 classes in the project are categorized into four distinct categories, see figure 5. The remaining class is an abstract class (*AbstractActionBean*) inherited by all controller classes.

Once we have mapped the classes in the monolithic application into the appropriate entity, we analyze the dependencies

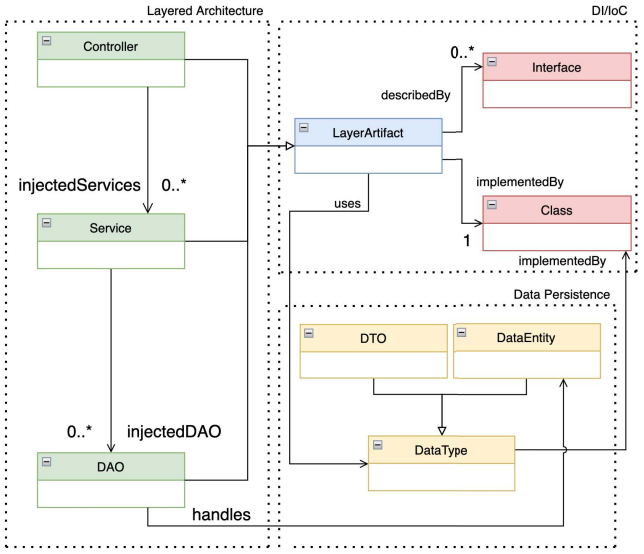

Fig. 4. Layered Architecture Meta-model (LAMM) that represents the layered architecture found in most frameworks.

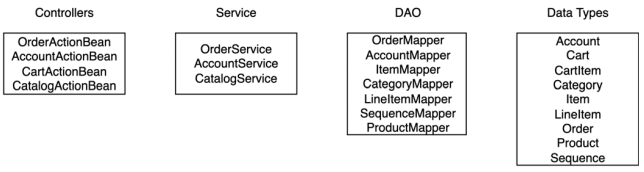

Fig. 5. Class categorization of JPetStore.

of the classes to map them to their layer counterparts (e.g., if a Controller's implementing class references a class implementing a Service, then we can add a dependency between the Controller and the Service). This results in an oriented graph that models the architecture found in Figure 1. Once all entities and their relationships have been mapped to the LAMM model, the microservice identification step begins.

B. Microservice Identification process using extracted artifacts

We use a hierarchical clustering algorithm to create a dendrogram. The dendrogram is cut following a depth-first algorithm, and presented to the user for validation.

For the clustering algorithm, we present two similarity measures to guide the clustering:

$$F_{Struct}(MS) = \frac{intradependencies(MS)}{nbPossibleDependencies(MS)} \quad (1)$$

$$F_{Data}(MS) = \frac{\sum_{(Cl_k,Cl_m)\in MS} f_{simData}(Cl_k, Cl_m)}{\frac{|MS|\times(|MS|-1)}{2}} \quad (2)$$

$$f_{simData}(Cl_k, Cl_m) = \frac{|Data_k \cap Data_m|}{|Data_k \cup Data_m|} \quad (3)$$

The first similarity measure ($F_{Struct}(MS)$) calculates the structural cohesion between the classes of a microservice

candidate. Particularly, the $intradependencies(MS)$ function calculates the number of intra-relationships between classes of the microservice candidate (e.g., method calls, attribute access, inheritance). While the denominator calculates the total possible incoming dependencies from the microservices. Together, they produce a similarity which promotes structural cohesion by rewarding clusters in which there are more internal relationships. The second similarity measure is described in Eq. 1 ($F_{Data}(MS)$). It calculates the data cohesion between the classes of a microservice candidate. This is to encourage clusters with classes that manipulate the same class, and promote clusters who are data autonomous. The numerator of the equation is the sum of $f_{simData}$ for each pair of class possible within the microservice. More specifically, Eq. 3 measures the data cohesion between two classes based on the the data entities used by both classes. Eq. 3 returns a value close to 1 whenever the two classes manipulate the same data entities. The sum of $f_{simData}$ is divided by the total number of class pairs to produce an average value.

When the first similarity measure is applied to the Layered Architecture Metamodel, it will ideally propose decompositions that favor vertical microservices focused on business capabilities. This is done by using a model that highlights these vertical dependencies. Furthermore, by favoring the vertical dependencies between the business classes of the application, we limit the risk of creating Wrong Cuts.

When the second similarity measure is applied to a clustering algorithm, it will ideally propose a decomposition that favors grouping classes that manipulate the same data. Furthermore, by taking into consideration the data-access classes when partitioning the monolith, we limit the risk of creating Shared Persistence between microservice by creating data ownership.

$$F_{Quality}(MS) = F_{Struct}(MS) + F_{Data}(MS) \qquad (4)$$

We combine both similarity measures to propose a multi-objective function (see Eq. 4), and apply them to i) build a hierarchical clustering (Algorithm 1) and ii) decompose it into a set of microservice candidate (Algorithm 2).

In Algorithm 1, to create the hierarchical order, a new parent cluster is created by merging two child clusters based on their similarity score. Thus, a parent cluster contains the classes of its two child clusters. The obtained hierarchy is represented as a dendrogram, i.e., a binary tree structure, in which each node represents a cluster of classes (see example Figure 6).

To begin, we initialize a set of classes $S_{classes}$ extracted during the layered architecture recovery (line 1), and an empty set $S_{Clusters}$ which will contain the clusters of classes created during the algorithm (line 2). Then, for each class belonging to $S_{classes}$, a cluster is created in which the class is placed into, and the cluster is added to $S_{Clusters}$ (line 3-5). Afterward, we iterate over the $S_{Clusters}$, finding the best two pair of clusters, that when merged, return the best score using $F_{Quality}(MS)$ (line 7-8). Once the best pair is identified, they are merged to create a cluster containing the pair of clusters as children

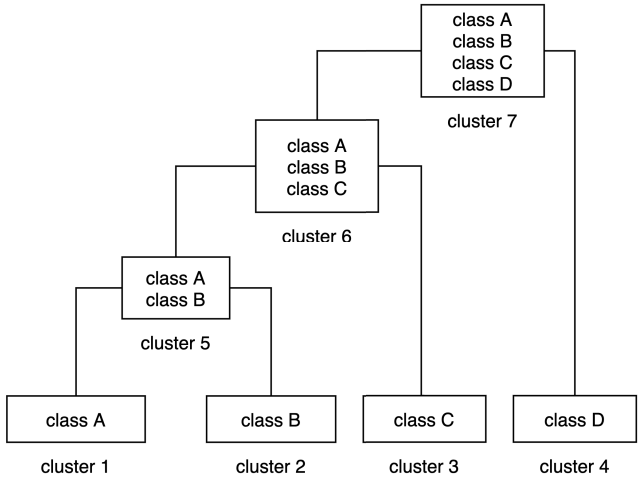

Fig. 6. Example of a dendrogram.

(line 9). Then the pair of clusters are removed from the $S_{Clusters}$, and the new cluster is added to $S_{Clusters}$ (lines 10-11). When two clusters are merged, a new cluster is created which contains the two clusters as child entities. The iteration ends when all clusters of $S_{Clusters}$ have been merged into one cluster. The end result is a binary tree in which each node represents a cluster, and which the leafs are the original clusters containing one class. This binary tree is referred to as a dendrogram and is the main input of Algorithm 2.

Algorithm 1: Hierarchical Clustering

Data: OO Source code *code*
Result: A dendrogram *dendro*
1 let $S_{classes}$ be the set of classes extracted from *code*;
2 let $S_{Clusters}$ be the set of clusters of classes;
3 **for *each*** $class \in S_{classes}$ **do**
4 | let *class* be a *cluster*;
5 | add *cluster* to $S_{Clusters}$;
6 **end**
7 **while** $size(S_{Clusters}) > 1$ **do**
8 | let $(cluster1, cluster2)$ be the closest pair of clusters based on $F_{Quality}(MS)$;
9 | let $New_{cluster} \leftarrow merge(cluster_1, cluster_2)$;
10 | remove $cluster_1$ and $cluster_2$ from $S_{Clusters}$;
11 | add $New_{cluster}$ to $S_{Clusters}$;
12 **end**
13 *dendro* $\leftarrow get(0, S_{Clusters})$;
14 return *dendro*;

In Algorithm 2, the main goal is to decompose the dendrogram created previously in an way as to generate a pseudo-optimal set of microservice candidates. The first step is to create an empty stack ($Stack_{Clusters}$) which will contain clusters to be decomposed, and we place the dendrogram inside (lines 1-2). Then, we initialize msa as an empty set of clusters, which will contain the final microservice candidates

(line 3). We pop the first cluster from the stack, and fetch the children ($child_1, child_2$) of that cluster (lines 5-6). If the average score of the fitness function applied on both $child_1$ and $child_2$ is higher than the score of the parent cluster, then we push both children to the stack (line 7-8). Otherwise, we add the parent cluster to the final microservice candidate set (msa) (line 10). We iterate over the previous instructions until $Stack_{Clusters}$ is empty (lines 4-11). Intuitively, the algorithm will iterate until it empties the stack or reaches the leaves of the dendrogram. In the latter case, the leaves are automatically added to the msa set. The end result is a set of clusters (or microservice candidates) which is a decomposition of the monolith (line 13).

Algorithm 2: Identifying the microservice candidates

```
   Data: A dendrogram dendro
   Result: A set of clusters S_Clusters
1  let Stack_Clusters be an empty stack of clusters;
2  push dendro to Stack_Clusters;
3  let msa be an empty set of clusters;
4  while size(Stack_Clusters) > 0 do
5  |  let cluster ← pop(Stack_Clusters);
6  |  let child_1, child_2 ← getChildren(cluster) ;
7  |  if avg(F_Quality(child_1), F_Quality(child_2)) >
   |     F_Quality(cluster) then
8  |  |  push child_1, child_2 to Stack_Clusters;
9  |  else
10 |  |  add cluster to msa
11 |  end
12 end
13 return msa;
```

IV. Evaluation

A. Research Questions & Methodology

To validate our approach, we conducted a set of experiments with the goal of answering the following research questions:

- **RQ1**: What is the impact of the extraction of the layered architecture on the identification of microservice candidates? While we propose to extract layered architecture artifacts from the existing source code, it does not necessarily contribute to the identification of microservice candidates. Indeed, naive approaches using a clustering technique with an objective function may be enough to extract microservice candidates close to the ground truth. With this question, we want to measure the impact of recovering the layered architecture of the monolithic application on the identification of a microservice-oriented architecture when compared to a naive approach.
- **RQ2**: How does our approach compare to the state of the art? The goal of **RQ2** is to compare the proposed approach in relation to other approaches found in the literature. Particularly, we wish to highlight the difficulties in identifying microservices.

The results from the proposed approach, the tools used for measuring our results, as well as the implementation of our approach are available in our repository[5].

1) RQ1 Methodology: We perform 3 different experiments to evaluate the impact of our proposed approach:

Experiment 1: Manual Microservice Identification. We perform a manual identification of microservice candidates. For each monolithic application, we use the source code as the primary artifacts for identifying microservice candidates. The identification was performed by 1 researcher and 3 R&D engineers. They have 12, 4, 7, and 8 years of experience in software decomposition, particularly towards microservice and component identification. We applied the pre-established identification pattern proposed in [4], namely *Decompose by Business Capabilities*. This pattern involves identifying business capabilities (and sub-capabilities) from the existing application (e.g., user, product, and catalog management), and mapping them to services. From there, the classes are placed into a service based on their relevance to the business capability and dependence between each other. We used static analysis to generate the dependency graph to help in the decision-making for classes that overlap between two different services. In the case of FindSportMates, JPetStore, and SpringBlog we followed this pattern. In the case of ProductionSSM, we were able to reuse the manual identification proposed by [7]. In their work, they describe an identification strategy similar to the one we followed. Indeed, they first identify a set of end-user functionalities. Then, they manually partition the set of classes belonging to the application into the different functionalities. Similarly, in our process, they use a debugging method to find which classes are related. As they did not consider java-interfaces in their partition, we supplemented their identification by adding them based on which classes implemented them. This was to have a global view of the application. All partitions are available in the aforementioned repository.

Experiment 2: Microservice Identification without the layered architecture recovery. In this experiment, we identify microservices based on the hierarchical clustering presented in this paper. As input artifacts, we use the java-interfaces and classes of each monolithic application. These artifacts are clustered according to Algorithm 1 & Algorithm 2 and the objective function of Eq. 4 presented in Section III-B. The results of this clustering algorithm are groups of java-interfaces/classes in which each group represents a candidate microservice.

Experiment 3: Microservice Identification using the proposed approach. In this experiment, we apply the proposed approach to its fullest. First, we apply the layered architecture recovery step (presented in Section III-A) on each monolithic application, in order to map each java-interface and class to a LayerArtifact entity. Then, we apply the hierarchical clustering presented in Section III-B on the LayerArtifact entities to produce groups of LayeredArtifact entities. Finally,

[5]https://gitlab.com/LeveragingInternalArchitecture/IdentificationApproach

for each group, we replace the LayerArtifact entities with their implementation (i.e., java-interfaces and classes). The partition resulting from this replacement is then used to evaluate the impact of the proposed approach.

All partitions are available on the provided repository, as well as the code used to perform Experiment 2 & 3. We propose a comparative study between the results of the proposed approach (Experiment 3) and the results obtained without considering the layered architecture recovery (Experiment 2). By comparing both experiments, we aim to highlight that the added layered architecture recovery has a positive impact on the identification of a microservice architecture. To measure the impact of the layered architecture recovery, we measure the similarity of each recovered architecture (Experiment 2 & 3) with regards to the proposed ground-truth (Experiment 1). Inspired by the experimental method proposed in [8], we use two different accuracy measures to evaluate the identification approach. Particularly, we apply MoJoFM [9], and $c2c_{cvg}$ [10], proposed to evaluate modularization techniques by measuring the similarity of a recovered architecture with regards to the proposed ground-truth. We then can compare the similarity measures of Experiment 2 & 3 to see which performs better. We reason that by using two different metrics, if Experiment 3 consistently outperforms Experiment 2, we reduce the bias of selecting a metric that favors one particular experiment.

The MoJoFM metric is used to evaluate modularization techniques [9]. This metrics is commonly used in the literature to evaluate architecture recovery approaches [8], [11]. During the evaluation, the identified microservice-oriented architectures are compared with the ones prepared by experts. The MoJoFM is calculated by Eq. 5, where $mno(A, B)$ is the minimum number of operations (e.g., move or join) required to transformed the architecture proposed by the approach (A) into the ground truth (B) [9]. In addition, $max(mno(\forall A, B)$ calculates the actual maximum distance to partition B.

$$MoJoFM(M) = (1 - \frac{mno(A,B)}{max(mno(\forall A,B))}) \times 100\% \quad (5)$$

The higher the value of MoJoFM, the more similar the identified architecture is to the ground truth. Inversely, a lower score indicates that the identified architecture is further from the ground truth.

The cluster-to-cluster coverage ($c2c_{cvg}$) [10] is a metric used to measure the degree of overlap of the implementation-level entities between two clusters , using the following equation as a base:

$$c2c(c_i, c_j) = \frac{|entities(c_i) \cap entities(c_j)|}{max(|entities(c_i), entities(c_j)|)} \times 100\% \quad (6)$$

Eq. 6 is used in Eq. 7 to evaluate the similarity between two clusters. More precisely, Eq. 7 calculates the best similarity for each cluster of the recovered architecture with regards to the ground truth. It is then compared to a pre-determined overlap threshold (th_{cvg}) to count the cluster as covered. The authors of [8], define the thresholds values of 50%, 33%, and 10%. The first value depicts $C2C_{cvg}$ for $th_{cvg} = 50\%$ which is referred to as the majority match [8]. This threshold is used to measure the clusters produced by the proposed approach approach which mostly resemble the clusters proposed by the ground truth [8]. While, the remaining threshold highlight the moderate (33%) and weak (10%) matches [8].

$$simC(A_1, A_2) = \{c_i | (c_i \in A_1, \exists c_j \in A_2) \wedge (c2c(c_i, c_j) > th_{cvg})\} \quad (7)$$

$$c2c_{cvg}(A_1, A_2) = \frac{|simC(A_1, A_2)|}{|A_2.C|} \times 100\% \quad (8)$$

The final count of the number of covered clusters is used in Eq. 8 (i.e., $c2c_{cvg}$) to calculate the total coverage of the recovered architecture with regards to the ground truth architecture. The result of this metric is highly dependent on the threshold used. The higher the threshold, the higher the extent to which the clusters produced by the approaches mostly resemble the clusters in the ground truth.

We use these two metrics to compare the similarity score of Experiment 2 and Experiment 3. If the results of Experiment 3 consistently outperform the results of Experiment 2, then the intermediary step of recovering the layered architecture of the monolith has a positive impact on the identification of microservices.

2) RQ2 Methodology: To highlight the difficulties in identifying microservices, we have extracted the class partitions of JPetStore from different approaches to compare them. First, we have extracted the partitions presented, as is, in [6], [12]. In addition, we have contacted the first author of [13] to apply their identification approach on JPetStore. For the approach proposed by Selmadji et al., they reported that their approach with the quality metric's ($FMicro(M)$) coefficients calibrated to $\alpha = 1$ and $\beta = 3$ [13]. Finally, we use the partition of the proposed approach (Experiment 3) on JPetStore. We compare all these partitions with the expert partition proposed in Experiment 1. We analyze each proposed architecture and compare them with the expert architecture to highlight the common pitfalls of microservice identification.

B. Data Collection

We selected a set of monolithic applications of various sizes (small, medium, and large) found in the literature. In Table I, 4 different applications are presented, which are commonly used in the literature.

C. Results and Discussion

1) RQ1: Table II and III respectively show the results of MoJoFM and $C2C_{cvg}$ to evaluate the overall accuracy of the different experiments with regards to the ground truth. Experiment 2 corresponds to the proposed identification approach without the use of the layered architecture artifacts. On the other hand, Experiment 3 corresponds to the proposed identification approach with the use of the layered architecture artifacts. In both experiments, we applied the approach to the

TABLE I
APPLICATIONS USED IN THE FOLLOWING EXPERIMENTS.

Application name	No of classes	Lines of Code (LOC)
FindSportMates[6]	21	4.061
JPetStore[7]	24	4.319
SpringBlog[8]	87	4.369
ProductionSSM[9]	226	31.368

same set of applications presented in Table I. Generally, our results indicate that the use of the layered architecture artifacts improved the accuracy of the identification approach. According to the results of MojoFM, the use of layered architecture artifacts increased the accuracy of the identification approach **by at least 14%** (in the case of SpringBlog). On average, **Experiment 3 performed 23.98% better than Experiment 2**.

Table III shows $C2C_{cvg}$ for three different values of th_{cvg} (i.e., 10%, 33%, and 50%) for each application under both Experiment 2 & 3. The first value depicts $C2C_{cvg}$ for $th_{cvg} = 50\%$ which is referred to as the majority match [8]. The scores in which Experiment 3 performed better than Experiment 2 are in bold. We denote that generally, **the use of layered architecture artifacts increases the quality of the microservice architecture**. For FindSportMates, the use of layered artifacts was able to produce perfect matches which explains the 100% majority matches. With JPetStore, we notice a drop of matches once we reach 50% threshold. Indeed, in Experiment 2 the approach was unable to create pertinent microservice candidates. In the case of SpringBlog, the use of layered artifacts produced better moderate and weak matches but failed to produce strong matches. However, without the use of these artifacts, Experiment 2 produced few majority matches (7.41%). This seems to indicate that the low performance is due to the clustering technique. Indeed, we notice a similar drop in score with the MoJoFM, which seems to support that hypothesis. With regards to the largest application (ProductionSSM), we denote that the use of layered architecture artifacts greatly outperformed by producing more than 55% of majority matches, while without these artifacts, Experiment 2 produced 1.92% majority matches. Furthermore, when we increased the threshold to 66% Experiment 3 faced a small drop in the majority matches for ProductionSSM and remained at 48%. This seems to indicate that the identified clusters are of even higher quality.

TABLE II
MOJOFM RESULTS (IN %AGE).

Applications	Experiment 2	Experiment 3
FindSportMates	70.0%	**100.0%** ($\uparrow 30\%$)
JPetStore	55.0%	**89.47%** ($\uparrow 34,5\%$)
SpringBlog	58.06%	**72.09%** ($\uparrow 14\%$)
ProductionSSM	57.73%	**75.16%** ($\uparrow 17,4\%$)

Summary RQ 1

The overall conclusion from the results of these metrics is that the use of the layered architecture artifacts does indeed increase the accuracy of the proposed approach for all systems, in almost every case, independent of the accuracy measure chosen.

TABLE III
RESULTS OF $C2C_{cvg}$ (IN %AGE).

Applications		Experiment 2	Experiment 3
FindSportMates	th10	100%	100%
	th33	100%	100%
	th50	25%	**100%** ($\uparrow 75\%$)
JPetStore	th10	100%	100%
	th33	30%	**100%** ($\uparrow 70\%$)
	th50	0%	**66.66%** ($\uparrow 66,6\%$)
SpringBlog	th10	77.78%	**86.67%** ($\uparrow 8,9\%$)
	th33	48.15%	**53.33%** ($\uparrow 5,2\%$)
	th50	**7.41%**	0.0% ($\downarrow 7,4\%$)
ProductionSSM	th10	**90.38%**	86.21% ($\downarrow 4,1\%$)
	th33	30.77%	**72.41%** ($\uparrow 41,7\%$)
	th50	1.92%	**55.17%** ($\uparrow 53,3\%$)

2) RQ2: Figures 2, 7, 8, 9, and 10 illustrate five different decompositions of the JPetStore application: one decomposition proposed by [6], another by [12], and one by [13], the decomposition proposed by our approach, and a manual expert decomposition. The manual expert decomposition proposes 3 different microservices based on functionality and vertical decomposition. The first microservice is related to order management, the second microservice provides account management, and the third microservice relates to the product catalog. The decomposition tries to promote functional autonomy by limiting the inter-dependency between a microservice and the service layer of another microservice. This is the case except for the Account Microservice (MS2) which uses *CatalogService*. Furthermore, this decomposition tries to promote data autonomy by limiting the inter-dependency between a microservice and the data-access layer of another microservice. The expert decomposition is able to promote this except for the data-access related to the *Item* data-entity which is used by Order Microservice (MS1) to generate an order of items. In both cases, the decomposition limits the inter-microservice access to read-only features. For instance, the Account Microservice consults *CatalogService* to read the user's favorite category, and the Order Microservice to read the items being bought.

Regarding the decomposition proposed by Al-Debagy et al. [6], they propose 4 microservice candidates of which 2 offer a vertical decomposition (MS2 & MS3). The other two microservice candidates show a horizontal decomposition

[6]https://github.com/chihweil5/FindSportMates
[7]https://github.com/mybatis/jpetstore-6
[8]https://github.com/Raysmond/SpringBlog
[9]https://github.com/megagao/production_ssm

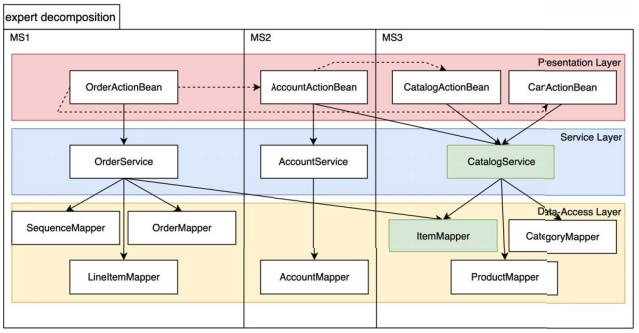

Fig. 7. Expert decomposition of JPetStore.

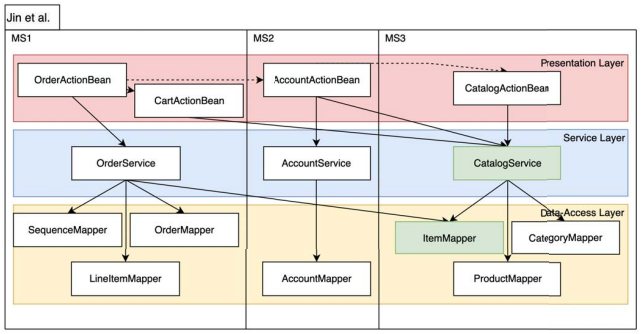

Fig. 8. decomposition of JPetStore proposed by [12]

along its technical layers. This is the typical example of a Wrong Cuts antipattern, where MS4 serves a technical need (that of data-access) to two different microservices, which may lead to increased network communication. The decomposition produced by Selmadji et al. [13], generates two microservices. While, they manage to avoid creating Wrong Cuts, they were unable to differentiate between the the account and the catalog service. Regarding the decomposition of Jin et al. [12], we observe a decomposition that respects the vertical decomposition and is similar to the expert decomposition. However, the approach places *CarActionBean* in a different microservice candidate. This is likely because of the dependency between the *OrderActionBean* and the *CartActionBean*. However, this decomposition creates another vertical dependency between MS1 and MS3. In contrast, our approach generated another decomposition that is similar to the expert decomposition. However, it placed *CartActionBean* in its own microservice. This is likely because our approach relies on data-oriented objective function as *CartActionBean* uses data exclusive to its function (e.g. Cart, CartItem). Since these data types were only used in this class it is likely that the approach opted to propose a separate microservice candidate.

Both the proposed approach and Jin et al.'s approach recovered a similar architecture to the expert decomposition while avoiding creating Wrong Cuts. However, Al-Debagy et al.'s approach decomposes JPetStore along technical layers, which usually results in highly dependent microservices.

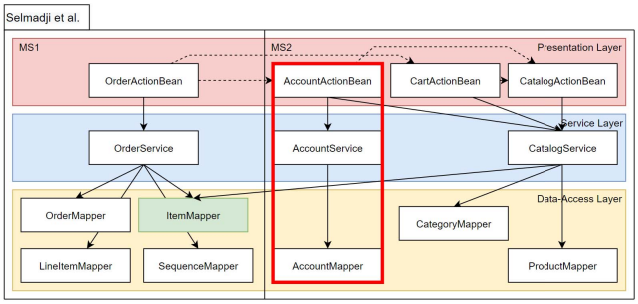

Fig. 9. decomposition of JPetStore proposed by [13]

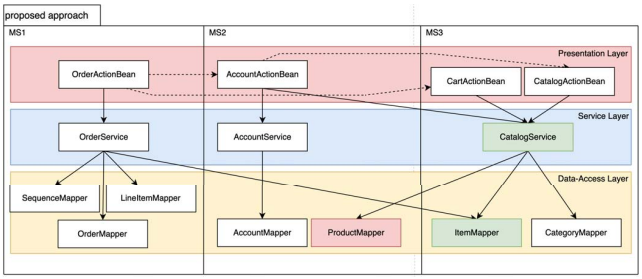

Fig. 10. decomposition of JPetStore using the proposed approach

Summary RQ 2

There is a recurring identification antipattern found in several approaches which highlights the difficulty in the identification process. We also conclude that the proposed approach can return comparable results to other approaches while avoiding common antipatterns.

D. Threats to Validity

1) Internal Validity: First, there is not necessarily a unique architecture for a system [14]. Since we only use one ground-truth architecture, there is a threat that our study is biased. To reduce this threat, we used two different metrics when comparing the ground truth with the identified architectures which both indicate a positive impact for the proposed approach. In the future, we intend to include other metrics to measure different aspects of the recovered architecture, such as system-level dependencies which is independent of the ground truth. Secondly, to limit the bias related to constructing the ground truth for the Experiment 1, we have performed the experiment using 4 people with different profiles. By doing so, we limit the influence of an individual's point of view. Furthermore, the approach and its results were not presented or produced until after the manual identification. Finally, we use the expert decomposition of `ProductionSSM` proposed by the authors of [7]. In this instance, we are able to completely remove the bias by using the expert decomposition of a team which is not involved in this experiment.

2) External Validity: Another threat to the validity of our experimentation is that we limit ourselves to applications implemented in JAVA. While JAVA is a popular language in the

industry, there are over types of languages that are not covered. We attempt to mitigate this threat by selecting applications of varying size and from different studies. However, the current literature does not provide many applications with a recovered architecture. In future works, we would like to extend the number of applications with a publicly available decomposition, and an evaluation framework to facilitate the evaluation of other identification approaches.

V. Related Work

Much of the existing work in the field of migration of monolithic applications towards microservice-oriented ones is related to the identification or recovery phase. In fact, several systematic reviews have been published on the subject and microservices in general [15] [16] [17]. Most recovery efforts do not consider the internal architecture of the application when identifying a microservice-oriented architecture from an existing monolithic application. Instead they focus on creating highly-cohesive and loosely-coupled clusters based on object-oriented relationships.

In [12], the authors presents a classic hierarchical clustering algorithm which clusters the class-level traces extracted from the execution monitoring of a monolithic application. From these trace clusters, the classes are partitioned into the appropriate cluster. Then, crossover processing is applied to resolve the cases where a class is placed into multiple clusters. Finally, the method-level traces are used to generate the interfaces, and corresponding API for each cluster.

The authors of [18] propose a model-driven approach to decompose monoliths implemented using Java Enterprise Edition . Their meta-model highlights the importance of session beans (i.e, similar to our components) and entities (i.e., similar to our DataType). For their clustering algorithm, they apply a static analysis to identify the dependencies between each type (i.e., classes and interfaces). This dependency graph is used to create a set of inheritance trees between session beans and entities. From there, each bean is placed into its own cluster. Furthermore, they consider the association between session beans and entities for their clustering approach. This involved identifying the inheritance tree between a session bean and entity, and creating a cluster for each session bean. Finally, the clusters a grouped based on the shared types. In our approach, we add the notion of an *internal architecture* where not all beans have the same responsibility. Furthermore, we consider the data-access components, and not the entities, are responsible for the management of the data layer and that the entities are merely representative of the stored data.

The internal architecture of the monolithic application is taken into consideration in [19]. In this work, the authors assume that monoliths apply the Model-View-Controller architectural style and propose a decomposition that takes into consideration the transactional context of these applications. Particularly, they consider the controllers and data entities when proposing a hierarchical clustering approach. In this approach, they propose to partition the data entities into clusters based on a similarity function calculated on the dependencies between the controllers and the data. This approach considers the structural importance of certain classes, however they limit their approach to the decomposition of data entities.

In [7], the authors use a dynamic analysis along with a genetic algorithm to identify a microservice architecture. Particularly, they propose to differentiate between controller classes and subordinate classes. In their decomposition strategy, they use the controller classes to guide the decomposition of the subordinate classes into clusters based on functional and non-functional metrics. Alternatively, [13] uses a static analysis of the source code elements, and its relationship to persistent data. Furthermore, they proposed a semi-automatic approach that uses the recommendation of the architect, when available, to guide the identification process. Similarly, to [7], they use quality metrics based on the characteristics of the microservices. However, while [7] use genetic algorithms, and [13] use hierarchical clustering.

Finally, [6] proposes an identification approach based on a neural network model (code2vec), which creates a code embedding from the monolith's source code. By creating this code embedding, semantically similar code embeddings are clustered together using a hierarchical clustering algorithm.

VI. Conclusion and Future Works

The main contribution of this paper an approach for the identification of microservices from object-oriented source code by leveraging the internally layered architecture of monolithic applications. The proposed approach uses the layered architecture to vertically decompose a monolith while minimizing the coupling between microservices. Through experimentation we were able to demonstrate that adding artifacts about the layered architecture of the monolithic application had a positive impact on the identification process. In a practical sense, we highlight two takeaways (1) that extracting the internal architecture is essential towards understanding the monolith and making better microservice identification decisions, and (2) existing approaches that do not take this into consideration risk falling into known antipatterns. Additionally, for the experimentation we have limited ourselves to object-oriented systems. However, we believe that this architectural design can be found in other types of systems. As future work, we plan to investigate a number of directions. The first one concerns the use of search based algorithms for the identification of microservices and comparing the results with the clustering one as it was done in the context of software components [20], [21]. A second direction concerns the combination of static and dynamic analysis of source code to identify microservices [22]. Another direction concerns the application of the approach proposed in [23] to exploit available artefacts such as documentation during the architecture recovery process. A last future work is related to source code refactoring to be able to materialize the identified architecture based on the results of our previous work [24], [25].

REFERENCES

[1] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, 2014, accessed: 2020-06-20.

[2] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 05 2015.

[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[4] C. Richardson, *Microservices Patterns: With examples in Java*. Manning Publications, 2018. [Online]. Available: https://books.google.de/books?id=UeK1swEACAAJ

[5] D. Taibi, V. Lenarduzzi, and C. Pahl, *Microservices Anti-patterns: A Taxonomy*. Cham: Springer International Publishing, 2020, pp. 111–128. [Online]. Available: https://doi.org/10.1007/978-3-030-31646-4_5

[6] O. Al-Debagy and P. Martinek, "A microservice decomposition method through using distributed representation of source code," *Scalable Comput. Pract. Exp.*, vol. 22, no. 1, pp. 39–52, 2021. [Online]. Available: https://www.scpe.org/index.php/scpe/article/view/1836

[7] Y. Zhang, B. Liu, L. Dai, K. Chen, and X. Cao, "Automated Microservice Identification in Legacy Systems with Functional and Non-Functional Metrics," in *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, mar 2020, pp. 135–145. [Online]. Available: https://ieeexplore.ieee.org/document/9101217/

[8] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 69–78.

[9] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *12th International Workshop on Program Comprehension (IWPC 2004), 24-26 June 2004, Bari, Italy*. IEEE Computer Society, 2004, pp. 194–203. [Online]. Available: https://doi.org/10.1109/WPC.2004.1311061

[10] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 176–17 609.

[11] B. Pourasghar, H. Izadkhah, A. Isazadeh, and S. Lotfi, "A graph-based clustering algorithm for software systems modularization," *Information and Software Technology*, vol. 133, p. 106469, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920302147

[12] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai, "Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering," in *2018 IEEE International Conference on Web Services (ICWS)*, no. October. IEEE, jul 2018, pp. 211–218. [Online]. Available: https://ieeexplore.ieee.org/document/8456351/

[13] A. Selmadji, A. Seriai, H. Bouziane, R. O. Mahamane, P. Zaragoza, and C. Dony, "From monolithic architecture style to microservice one based on a semi-automatic approach," in *2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020*. IEEE, 2020, pp. 157–168.

[14] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 901–910.

[15] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, M. Cardoso, J and Ferguson, D and Munoz, VM and Helfert, Ed. SCITEPRESS - Science and and Technology Publications, 2016, pp. 137–146.

[16] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 21–30.

[17] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment - First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. Bruel, M. Mazzara, and B. Meyer, Eds., vol. 11350. Springer, 2018, pp. 128–141. [Online]. Available: https://doi.org/10.1007/978-3-030-06019-0_10

[18] D. Escobar, D. Cardenas, R. Amarillo, E. Castro, K. Garces, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," in *2016 XLII Latin American Computing Conference (CLEI)*. IEEE, oct 2016, pp. 1–11. [Online]. Available: http://ieeexplore.ieee.org/document/7833410/

[19] L. Nunes, N. Santos, and A. Rito Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *Software Architecture*, T. Bures, L. Duchien, and P. Inverardi, Eds. Cham: Springer International Publishing, 2019, pp. 37–52.

[20] S. Kebir, A.-D. Seriai, A. Chaoui, and S. Chardigny, "Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, ser. C3S2E '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–8. [Online]. Available: https://doi.org/10.1145/2347583.2347584

[21] A. Shatnawi, A.-D. Seriai, H. Sahraoui, and Z. Alshara, "Reverse engineering reusable software components from object-oriented apis," *Journal of Systems and Software*, vol. 131, pp. 442–460, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412121630098X

[22] A. Shatnawi, H. Shatnawi, M. A. Saied, Z. A. Shara, H. Sahraoui, and A. Seriai, "Identifying software components from object-oriented apis based on dynamic analysis," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 189–199. [Online]. Available: https://doi.org/10.1145/3196321.3196349

[23] S. Chardigny and A. Seriai, "Software architecture recovery process based on object-oriented source code and documentation," in *Software Architecture*, M. A. Babar and I. Gorton, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 409–416.

[24] P. Zaragoza, A. Seriai, A. Seriai, H. Bouziane, A. Shatnawi, and M. Derras, "Refactoring monolithic object-oriented source code to materialize microservice-oriented architecture," in *Proceedings of the 16th International Conference on Software Technologies, ICSOFT 2021, Online Streaming, July 6-8, 2021*, H. Fill, M. van Sinderen, and L. A. Maciaszek, Eds. SCITEPRESS, 2021, pp. 78–89. [Online]. Available: https://doi.org/10.5220/0010557800780089

[25] Z. Alshara, A.-D. Seriai, C. Tibermacine, H. L. Bouziane, C. Dony, and A. Shatnawi, "Materializing architecture recovered from object-oriented source code in component-based languages," in *Software Architecture*, B. Tekinerdogan, U. Zdun, and A. Babar, Eds. Cham: Springer International Publishing, 2016, pp. 309–325.