2023 IEEE 20th International Conference on Software Architecture (ICSA) | 979-8-3503-9749-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSA56044.2023.00013

From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization

Gianluca Filippone, Nadeem Qaisar Mehmood, Marco Autili, Fabrizio Rossi, Massimo Tivoli
Department of Information Engineering, Computer Science and Mathematics
University of L'Aquila
L'Aquila, Italy
gianluca.filippone@graduate.univaq.it, {nadeemqaisar.mehmood, marco.autili, fabrizio.rossi, massimo.tivoli}@univaq.it

***Abstract*—Migrating from a legacy monolithic system to a microservice architecture is a complex and time-consuming process. Software engineers may strongly benefit from automated support to identify a high-cohesive and loose-coupled set of microservices with proper granularity. The automated approach proposed in this paper extracts microservices by using graph clustering and combinatorial optimization to maximize cohesion and minimize coupling. The approach performs static analysis of the code to obtain a graph representation of the monolithic system. Then, it uses graph clustering to detect high-cohesive communities of nodes using the Louvain community algorithm. In parallel, the tool clusters the domain entities (i.e., classes representing uniquely identifiable concepts in a system domain) within bounded contexts to identify the required service granularity. Finally, it uses combinatorial optimization to minimize the coupling, hence deriving the microservice architecture. The approach is fully implemented. We applied it over four different monolithic systems and found valuable results. We evaluated the identified architectures through cohesion and coupling metrics, along with a comparison with other state-of-the-art approaches based on features such as granularity level, number of produced services, and methods applied. The approach implementation and the experimental results are publicly available.**

***Index Terms*—microservices, architecture migration, system decomposition, graph clustering, combinatorial optimization**

I. Introduction

Micro-Service Architecture (MSA) promotes the adoption of a new migration paradigm to re-engineer the business capabilities of existing monolithic systems into "cloud-ready", flexible, and loosely-coupled components of specialized services [1], [2]. Enterprises can take advantage of the MSA principles (e.g., agility, operational efficiency, scalability, portability, and continuous deployment) when refactoring a monolithic system into self-contained services, each of them running as a separate process that has been designed for a specific function.

Refactoring is a complex task and, when performed manually, is time-consuming and error-prone. The quality of the decomposed system depends on the experience and knowledge of experts [3]. As a result, automated support is desirable.

There exist different strategies to refactor a monolithic system into an MSA [4]. They can be classified into three categories: (i) top-down, forward-engineering strategies, in which high-level monolithic domain artifacts (e.g., use cases, activity diagrams, etc.) are accounted for and decomposed to model and implement the targeted microservices; (ii) bottom-up, monolithic re-engineering strategies, in which the dependencies of a monolithic system are analyzed to extract reusable components, remove dependencies, and rewrite some existing applications by using the newly created microservices; and (iii) hybrid strategies, which suitably combine (i) and (ii).

One of the main challenges that any monolithic-to-MSA strategy has to face is the identification of high-cohesive and loose-coupled microservices with the *right granularity* and within the *right bounded context* [5]. Hassan et al. define granularity as "the service size and the scope of functionality a service exposes" [6]; similarly, Homay et al. state that "the problem in finding service granularity is to identify a correct boundary (size) for each service in the system" [7]. As for the bounded context, the problem is to identify related functionalities and combine them into a single business capability, or responsibility, which is then implemented as a service [5].

As discussed in Section II, during the last decade, several researchers have been proposing monolithic-to-MSA solutions, e.g., [4], [8]–[14]. Most of the proposed approaches are challenged by the difficult task of establishing the proper granularity of microservices or supporting the so-called "single responsibility principle", i.e., a microservice must conform to and stay within a bounded context [5], which may contain one or more aggregates [1]. As already anticipated, this task needs much effort and specific skills to be accomplished. Moreover, many of the proposed approaches require different types of inputs (e.g., use cases, domain models, activity diagrams, or business models) [4], hence limiting their applicability to the availability of these inputs. Things are worse when the required inputs must be provided manually [11], [14].

In order to address the aforementioned challenges, in this paper we present a bottom-up decomposition approach that fully automates the decomposition process and identifies microservices having functionalities combined together for each application's bounded context, without requiring manual inputs, system models, specific skills, or knowledge. Our approach first performs Static Code Analysis (SCA) of the monolithic system to produce a graph representation of the

979-8-3503-9749-9/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSA56044.2023.00013

system with a method-level resolution, in which arcs are weighted in order to represent the relevance of each of the relationships between nodes. Then, as detailed in Section III, graph nodes are clustered in two different ways to obtain the granularity and the functionality scope of the services. In particular, we first cluster the whole graph to obtain high-cohesive communities of nodes likely linked to the same functionalities. Then, we cluster the sub-graph of the domain entities only (i.e., classes representing uniquely identifiable concepts in a system domain) to identify the service granularity and the system's bounded contexts. Finally, still accounting for the relevance of the relationships between nodes and the communities found in the previous step, a combinatorial optimization problem is solved to obtain the final set of microservices having the minimum coupling.

We fully implemented our approach[1] and run experiments on four well-known publicly-available monolithic systems that have been widely used for evaluating a number of monolithic-to-MSA approaches, e.g., [3], [9]–[11], [14]–[21]. We assessed the obtained microservice architectures against different dimensions, i.e., number of microservices, cohesion and coupling metrics, independence of functionality, and bounded contexts also in comparison with results obtained with other state-of-the-art approaches. Experimental results are promising and show that:

1) the resulting architecture shows high cohesion and low coupling;
2) the obtained microservices are independent and meet the single responsibility principle;
3) bounded contexts are identified with good precision from the reverse-engineered system.

The rest of this paper is organized as follows. Section II discusses related work. Section III presents our approach by covering details about performing analysis, clustering, and optimization. Section IV reports on the validation of our approach and presents the experimental results. Limitations of the approach are discussed in Section V. Section VI concludes by also giving future insights.

II. Related work

In this section, we discuss relevant related work by considering the inputs required by the approaches, the usage of static code analysis, graph-based clustering, data-driven decomposition, and the leveraging of layered architectures.

Discussing the *input requirements*, some researchers using a top-down approach have involved the users to manually provide high-level domain information, such as domain models, data-flow diagrams, or use cases [9], [19], [22]. Similarly, Levcovitz et al. [23] manually categorized the database tables, established a dependency graph of the code, and extracted the microservices from the bottom up [4]. In contrast, our approach is not dependent on any manual input effort, system models, and/or user's system knowledge as it requires the monolith's code only.

[1]https://github.com/sosygroup/from-monolith-to-microservices

Several researchers applied SCA to exploit the semantic information for clustering using proximity measures or topic modeling [3], [11], [15], [16], [24]–[26]. In contrast, we perform SCA to understand the pre-existing system's architecture to identify the structural elements (i.e., classes, methods, etc.) and the relationships between them (i.e., method calls, class inheritance, etc.), like, e.g., [11], [14], [20], [27]–[30].

Representing the structural elements of the original software system in a processable format, such as trees or graphs, is a widely exploited strategy, as *graph-based clustering* may lead toward the identification of microservice candidates [14]. Having a graph-based representation, some approaches [11], [14], [16] applied the Louvain algorithm [31] to find candidates to increase modularity value. Gysel et al. [9] in their visualization tool of Service Cutter applied Girvan–Newman's algorithm [32] to find communities. Mazlami et al. [24] designed a graph-cutting algorithm to generate microservices. Kamimura et al. [3] perform graph clustering to find features based on program groups and data. In [33] four self-developed graph-based algorithms are used to generate microservices. Tyszberowicz et al. [34] represent a monolithic system in a bipartite graph and then use the shortest path to find nodes closer to each other to find density-based clusters. In contrast to the aforementioned approaches, we do not rely solely on graph clustering to find candidates. We apply the Louvain algorithm on two different graph-based representations of the system to find highly cohesive communities of nodes and clusters of domain entities. Instead of considering those clusters as candidate microservices, they are used as input in the optimization phase to obtain high-cohesive and low-coupled microservices.

The *data-driven* approaches are primarily top-down or hybrid, in which the database is partitioned or the data-flow graphs of business logic are analyzed [18]. Romani et al. [25] have applied semantic analysis on the database schema to find clusters, which are then mapped to topics for microservice identification. They do not form any graph representation of the extracted information from the database schema, however, they relate to us by focusing on the persistence layer and persisted domain entities. Li et al. [19] extended the dataflow-driven semi-automatic decomposition top-down approach of Chen et al. [22]. They map the business logic's code toward the data storage or the persistence layer. They build a virtual data flow from the manual input to apply an algorithm to find clusters or modules as microservices. However, differently from our approach, they require manual intervention and system knowledge to analyze the business logic and use cases and build the data-flow diagrams.

Some approaches leverage systems' layered architectures [2] to identify microservices. Trabelsi et al. [11] extend the work in [35] and classify the source code classes into three layers and identify three service types: entity, application, and utility. After building a weighted graph representation of the system, they apply the Louvain algorithm to group classes of each layer into services according to their static relationships. Then, a fuzzy clustering algorithm is applied to cluster services

across different layers into candidate microservices. In [10], authors assign each class to its related layer. By using two different similarity metrics that favor a vertical decomposition, a hierarchical clustering algorithm builds a dendrogram that is then suitably cut into a set of candidate microservices. In contrast, we exploit layers to build the graph representation and assign weights to the graph's edges. Both clustering and optimization are performed on the graph without considering the system layers explicitly, and allow methods from all the layers to be included in each microservice. Vertical decomposition is obtained through the inter-relationships between methods and methods and entities.

Finally, differently from all the approaches reported above, we represent and decompose the system at the method-level resolution, even splitting classes if needed, rather than considering classes as atomic units.

III. Methodology

The methodology comprises three main phases (Figure 1):

- *Static Code Analysis*: the source code is analyzed to produce an information graph representing the system at the method level;
- *System Decomposition*: the nodes of the graph are clustered conveniently to find cohesive communities of both the whole graph and the domain entities subgraph;
- *Optimization*: an optimization model is built to distribute different communities into microservices in a loose-coupled way.

A. Static Code Analysis

In analyzing the system, we leverage the fact that most of the languages and frameworks used for building web services rely on layered architectures [2], [10], [11]. In such architectures, methods in the presentation layer expose public interfaces and catch incoming requests. Then, methods of the lower layers are called, down to the persistence layer and the database [2]. Given this, we analyze the system and produce a graph representation that holds the relationships among components (methods and classes) by taking into account the role of each layer in the system. For sake of generality, we abstract the exact number of system layers by considering only two main layers: a "logic" layer that comprises both presentation and business layers and a "persistence" layer.

The analyzer tool we have aptly realized parses the code to allocate the classes containing the system's business logic into each of the two layers and find classes that define the domain entities of the system (i.e., classes that represent a concept of the domain model, e.g, `User`, `Person`, `Pet`, etc.). In doing this, we leverage the annotations that are used in most of the frameworks to define the technical role of the classes (e.g., `@Controller`, `@Service`, `@Repository`, `@Entity` annotations provided by Spring Framework) to identify the layer of each class or whether it represent a domain entity. The automated identification of such layers and entity classes can be manually refined to obtain a more accurate analysis.

After having allocated classes in the layers, the tool inspects the classes to find the declared methods and, going recursively deep in the code's syntax tree, collects the method calls and the references to the entities. Then, a node of the graph is created for: (i) each method from classes of the logic and repository layer, (ii) each entity class. Relationships between methods and entities are put in the graph as directed arcs. We define five types of arcs, each representing a particular relationship type:

- *Calls* arcs are added between method nodes when there is a method call from one method to another;
- *Uses* arcs are added between methods and entities if there is a reference to an entity in a method's code, e.g., if an object is instantiated or its values are accessed;
- *Persists* arcs are added between methods and entities if a method from the persistence layer reads or writes a given entity on the database;
- *References* arcs are added between entities if an entity references another entity, thus representing the association, aggregation, and composition relationships in the domain model;
- *Extends* arcs are added between entities if an entity extends another entity, thus representing the generalization relationship.

The output of this phase is a directed graph $G = (V, A)$ in which a type $t_i \in \{\texttt{Method, Entity}\}$ is assigned to each node $i \in V$, a relationship type $r_{ij} \in \{\texttt{Calls}, \texttt{Uses}, \texttt{Persists}, \texttt{References}, \texttt{Extends}\}$ is assigned to each arc $(i, j) \in A$. Moreover, a weight w_{ij} is assigned to each arc (i, j). Weights define the relevance of the relationships between nodes: the higher the weight of an arc, the more likely methods and/or entities at its endpoints should be put into the same microservice. After the analysis, default weights are assigned according to the type of the relationship. The default weight of the `Persists` relationship is 1, while those of the `Calls`, `Uses`, `References`, and `Extends` relationships are 0.8, 0.6, 0.2, and 0, respectively. In this way, we set the `Persists` relationship as the most important, and the method call relationship next to it. The reference and extends relationship weights are lower to favor domain entities to be distributed into different microservices. As for the system layers, weights can be manually refined to comply with specific application domains and requirements.

Figure 2 shows a portion of the information graph obtained after the analysis of the project Spring Petclinic[2], which we will use as a reference use case throughout this section. For the sake of simplicity, the figure only portrays a small portion of the complete graph. Circles represent methods: empty circles are the methods from the logic layer, while dot-filled ones are from the persistence layer. Hexagons represent entities. Black continuous arrows depict `Calls` relationships, while grey continuous arrows depict `Uses` relationships. Dashed black arrows outgoing from persistence-layer methods represent `Persists` relationships, while the ones between entities

[2]https://github.com/spring-projects/spring-petclinic

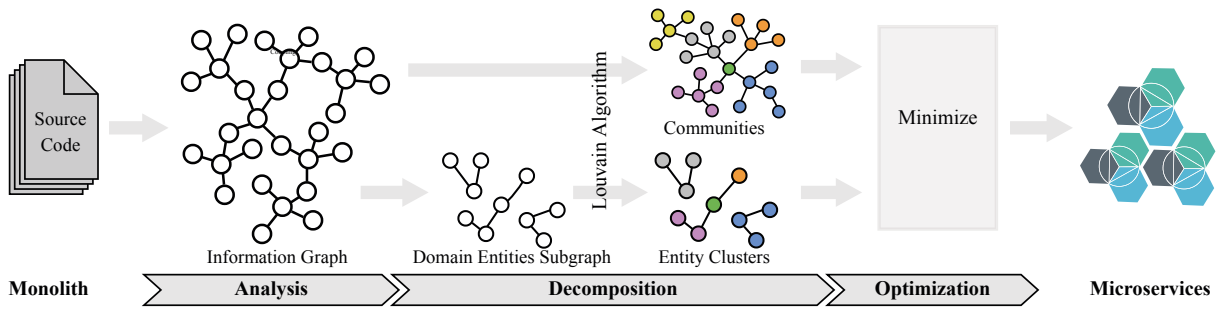

Fig. 1. Methodology: Monolithic to Microservices Migration

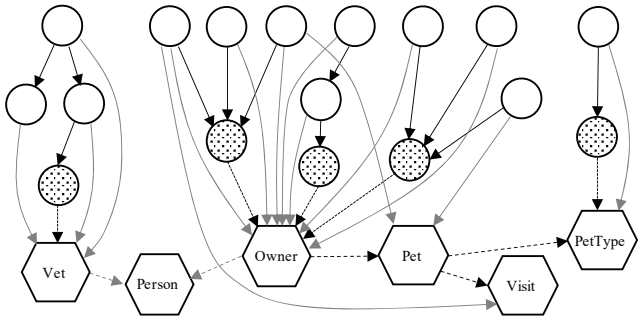

Fig. 2. Static code analysis output for Spring Petclinic

depict `References` relationships. Dashed grey arrows depict `Extends` relationship.

Such a graph can be seen as a reverse-engineered model of the whole system. The `Calls`, `Uses`, and `Persists` relationships represent all the inter-dependencies in the code, while the entity-to-entity relationships (`References`, `Extends`) reproduce the structure of the domain model of the system. This allows the identification of clusters of entities that possibly belong to the same bounded context and portions of the application in which methods are strongly related to each other and to a precise set of entities.

B. System Decomposition

The problem of microservice extraction from a monolithic system naturally becomes a community detection problem when we represent it in the form of an information graph [14]. Newman describes the term community as: "a subgraph containing nodes which are more densely linked to each other than to the rest of the graph, or equivalently, a graph has a community structure if the number of links into any subgraph is higher than the number of links between those subgraphs" [36]. Therefore, selecting an appropriate community detection algorithm is very important to identify microservice candidates of high quality.

To identity community candidates within a large network, the Louvain community algorithm has some better features than its related algorithms [31] such as the GN algorithm [32], [36], the K-L algorithm [37], and the Spectral Bisection algorithm [38]. In fact, it has the lowest time complexity, higher cohesion, and stability [14]. To detect communities, the Louvain algorithm maximizes a modularity score, i.e., evaluating how much more densely connected the nodes are within a single community but fewer connections between different communities [39]. Equations (1) and (2) define the modularity heuristic function [36] that the Louvain algorithm optimizes to evaluate the partition of a community:

$$Q = \frac{1}{2m} \sum_{(i,j)} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j), \tag{1}$$

$$\delta(u, v) = \begin{cases} 1 \ when \ u = v \\ 0 \ when \ u \neq v \end{cases}, \tag{2}$$

where A_{ij} represents the weight on the edge connecting nodes i and j. For an unweighted graph, a constant value is set such as one. The k_i represents the sum of the weights of all edges connected to node i:

$$k_i = \sum_j A_{ij}, \tag{3}$$

where m is the sum of the weights of all edges in the graph, and c_i is the community to which node i currently belongs.

Being unsupervised, the Louvain algorithm does not require the number of communities to be found or the size of the communities [16].

By applying the Louvain algorithm to our representation of the system, we obtain a cohesive set of communities whose nodes are from all the system's layers. While this is good since communities consider relationships between nodes with different roles in the system (i.e., entities, controller interfaces, business logic, all required to offer complete functionalities) they cannot be generally taken as acceptable microservice candidates. In fact, since there are no constraints over the structure of the candidate microservices to be identified, communities may be built only of `Entity` type nodes, or – on the contrary – only of `Method` type nodes, hence, they may not realize complete and standalone microservices. Moreover, there are no assurances about the right granularity of the communities. Since there are no constraints over the number of microservices to be identified, they are likely to be too fine-grained, especially if obtained from graphs representing

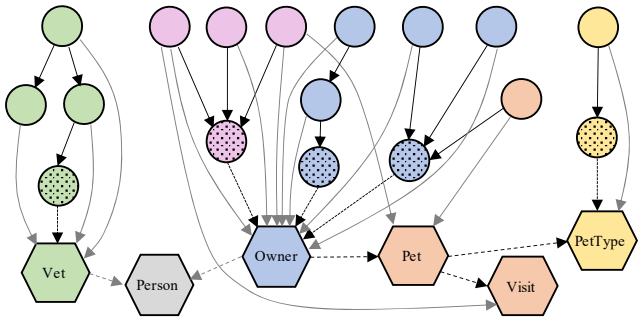

Fig. 3. Communities obtained from the complete information graph

large systems with a high number of methods. Finally, given the above, communities may not succeed in wrapping all the nodes required to realize a business functionality, i.e., to build microservices around bounded contexts.

Therefore, to overcome these issues, we applied the Louvain community detection algorithm on the information graph in two different ways: (i) on the complete information graph, which overlaps all the architecture layers, to obtain cohesive sets of nodes; (ii) on the subgraph having only the entities related with the persistence layer, as it plays the most important role in maintaining the domain context [19], [22], [23], [25].

1) Graph Communities from the complete graph: the execution of the Louvain algorithm on the complete information graphs returns a set C of n communities $C_1, ..., C_n$ of nodes representing entities and methods spreading across all architectural layers. Figure 3 shows the communities obtained for the Spring Petclinic application. As discussed above, they can not be considered as complete microservices since they are too fine-grained. For instance, methods related to the *Owner* entity are spread in three different communities. This means each of them may not contain the full set of functionalities required to realize a business capability, i.e., they do not identify a bounded context. Moreover, while being cohesive (e.g., blue nodes are related to each other and all referencing the *Owner* entity) they would result in a high-coupled microservice architecture. However, we can leverage the cohesiveness of the communities to obtain cohesion in the final microservice architecture.

2) Entity Clusters: we obtain the sub-graph of the domain entities by considering the set of nodes of type `Entity` having relationships with the persistence layer and the arcs of type `References` and `Extends`. The Louvain algorithm on this graph returns a set K of m communities $K_1, ..., K_m$ of nodes of type `Entity`. For sake of clarity, from here on, we will use the term *community* for the communities obtained from the whole graph, while we will use *entity cluster* for the communities obtained with the domain entities subgraph.

Figure 4 shows the entity clusters obtained for Spring Petclinic. We obtained two clusters. The first includes *Vet* and *Person* entities, and the second includes *Owner*, *Pet*, *Visit*, and *PetType* entities. These two clusters identify as many application contexts.

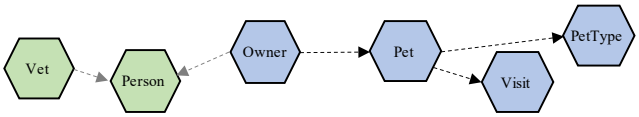

Fig. 4. Communities obtained from the sub-graph of persisted entities

We use communities and entity clusters to, respectively, obtain cohesion and identify the granularity of the microservices in the optimization phase.

C. Optimization

The optimization of the microservice architecture is realized through an *Integer Linear Programming* (*ILP*) model that finds a solution to a variant of the Multiway Cut problem formulation [40] to which we added constraints over the structure of the microservices to be identified. The goal of the optimization is to partition the nodes of the information graph into a set of m microservices – where m is the number of entity clusters identified in the previous step – in such a way that the coupling among microservices is minimized. We define coupling as the sum of the weights of the arcs that connect microservices, as we will better describe in this section.

We state the problem as follows:
Given a graph $G = (V, E)$, a type $t_i \in \{\texttt{Method}, \texttt{Entity}\}$ for each node $i \in V$, a weight w_{ij} for each edge $(i, j) \in E$, **find** a partition of V in m sets $\{M_1 ... M_m\} = M$ such that:
(i) $\bigcap M_i = \emptyset$;
(ii) $\bigcup M_i = V$;
(iii) each $M_k \in M$ contains at least one node i s.t. $t_i \neq \texttt{Entity}$;
(iv) the sum of weights w_{ij} s.t. $i \in M_h, j \in M_k$, $h \neq k$ is minimized.
Each of the sets $M_1, ..., M_m$ is a candidate microservice.

The problem stated above has been modeled with an *ILP* formulation through the following decision variables, as in [40]:

$$x_{ik} = \begin{cases} 1 \text{ if node } i \text{ is in the microservice } M_k \\ 0 \text{ otherwise} \end{cases} \tag{4}$$

$$y_{ij} = \begin{cases} 1 \text{ if edge } (i,j) \text{ has its endpoints into the same MS} \\ 0 \text{ otherwise} \end{cases} \tag{5}$$

$$z_{ij}^k = \begin{cases} 1 \text{ if edge } (i,j) \text{ has both its endpoints into MS } M_k \\ 0 \text{ otherwise} \end{cases} \tag{6}$$

We define the following constraints:

$$z_{ij}^k - x_{ik} \leq 0 \quad \forall M_k \in M \; \forall (i,j) \in E \tag{7}$$

$$z_{ij}^k - x_{jk} \leq 0 \quad \forall M_k \in M \; \forall (i,j) \in E \tag{8}$$

$$x_{ik} + x_{jk} - z_{ij}^{k} \leq 1 \;\; \forall M_k \in M \; \forall (i,j) \in E \tag{9}$$

$$y_{ij} = \sum_{M_k \in M} z_{ij}^{k} \quad \forall (i,j) \in E \tag{10}$$

$$\sum_{k} x_{ik} = 1 \quad \forall i \in V \tag{11}$$

$$\sum_{i \in V \mid t_i = \text{Method}} x_{ik} \geq 1 \quad \forall M_k \in M \tag{12}$$

Constraints (7) to (11) are from the Multiway Cut problem formulation [40], while constraint (12) is a new one. We added it to ensure that methods are put inside the microservices, since they are needed to expose interfaces, offer functionalities, realize microservice logics, etc.

The objective function (13) minimizes the *coupling* among microservices. As mentioned before, it is defined as the sum of the weights of the arcs whose endpoints belong to different microservices.

$$\min \sum_{(i,j) \in E} w_{ij} \; (1 - y_{ij}) \tag{13}$$

However, microservices obtained from the solution of this optimization model, while having the lowest coupling, show a series of problems. In fact, the model does not lead to an actual domain-driven decomposition, since domain entities are free to be placed into any of the identified microservices. Hence, it may happen that all the entities are put into a single all-containing microservice, while other microservices may be small and built of only a few low-cohesive nodes, without a real meaning from both an architectural and functionalities point of view.

To obtain a domain-driven decomposition and build microservices inside bounded contexts, we distribute entities into microservices according to the entity clusters identified in the previous phase (Section III-B2). Hence, we force each entity i in the entity cluster $K_k \in K$ to be in the microservice k. We realize this by fixing the x variables as follows:

$$x_{ik} = 1 \quad \forall i \in V \text{ iff } i \text{ is in the entity cluster } K_k. \tag{14}$$

In order to gain cohesion, we force all the nodes from the same community to be in the same microservice, so as to build microservices as a composition of the high-cohesive communities found in the previous phase (Section III-B1). Hence, we fix the y variables as follows:

$$y_{ij} = 1 \quad \forall (i,j) \in E \text{ iff } i \text{ and } j \text{ are in the same community.} \tag{15}$$

Notice that fixing x and y as in (14) and (15) can produce an infeasible model if two entities from the same community are in different entity clusters. To avoid this, we use a *Fix-and-Relax* approach. As shown in Algorithm 1, we first fix x variables according to the entity clusters in K. Then, after fixing y variables for a given community $C_c \in C$, we check the model feasibility. If infeasible, we relax the fixed y variables for all the nodes in the community C_c to keep the model feasible.

Algorithm 1 Fix-and-Relax algorithm for x and y variables

```
for entity cluster K_k ∈ K do
    for i ∈ k do
        Fix x_ik = 1
    end for
end for
for community C_c ∈ C do
    for (i,j) ∈ E do
        if i ∈ C_c & j ∈ C_c then
            Fix y_ij = 1
        end if
    end for
    if model is infeasible then
        for (i,j) ∈ E do
            if i ∈ C_c & j ∈ C_c then
                Relax y_ij
            end if
        end for
    end if
end for
```

Results obtained as a solution of this optimization model allow to assign nodes of the information graph (i.e., methods and entities) to microservices. Figure 5 show the architecture resulting from the optimization phase of the Spring Petclinic application. Communities shown in Figure 3 have been merged in bigger sets of nodes according to the entity clusters (Figure 4) and in such a way that the coupling among obtained microservices is the minimum. We can also notice that the identified microservices represent a vertical decomposition of the system, in which each microservice contains methods from all the layers. Moreover, nodes of each microservice are mainly connected to the domain entities that are persisted by methods of the same microservice. These characteristics allow microservices to be autonomous, to be focused on a single responsibility and within a bounded context, and to fully realize functionalities since each of them contains methods to expose interfaces, realize the business logic, and access the database.

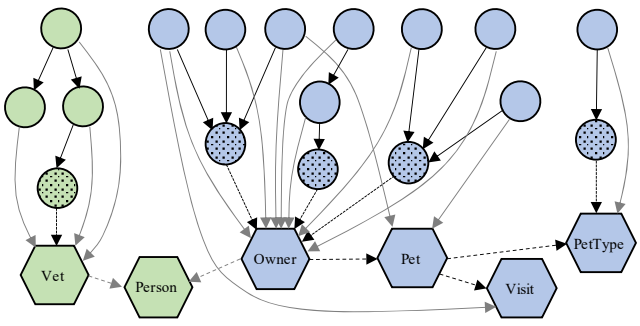

Fig. 5. Result of the optimization

IV. Evaluation

The main goal of the evaluation is to assess whether our approach can produce a decomposed system that follows the main principles of microservice architectures. In particular, we check that each microservice: (i) has a well-modularized structure with low coupling and high cohesion, (ii) follows the single responsibility principle, and (iii) independently offers functionalities. To do this, we use cohesion and coupling metrics to quantify how much the design principles are respected and evaluate the structure of the decomposition. Also, we use the *IFN* metric [18] to evaluate how much independently functionalities are provided, as detailed below. Moreover, we assess the composition of the bounded contexts realized tanks to the entity clustering. We compare our results with those of other approaches at the state-of-the-art and reference architectures proposed in the literature.

The complete implementation of the approach and all the reported results are publicly available[3].

In the following, we define the evaluation metrics. Then, we describe the performed experiments. Finally, we report the obtained results.

A. Evaluation Metrics

We compute the *average coupling* to quantify how much coupled are the microservices in the obtained architecture. It is computed from the graph representation of the obtained architecture as the ratio of the sum of the weights of the arcs whose endpoint nodes are in different microservices and the number of identified microservices:

$$AverageCoupling = \frac{1}{|M|} \sum_{(i,j)\in E | i\in M_h, j\in M_k, h\neq k} w_{ij}, \quad (16)$$

where M is the set of identified microservices, $M_h, M_k \in M$, and w_{ij} is the weight of the arc (i, j). The less the average coupling, the less coupled a microservice architecture is.

Similarly, we compute the *average cohesion* to quantify how much cohesive are the obtained microservices. It is computed from the graph representation of the obtained architecture as the ratio between the sum of the weights of inners arcs (i.e., those arcs whose endpoints are in the same microservice) and the sum of the weights of all arcs outgoing from the nodes of a microservice:

$$AverageCohesion = \frac{1}{|M|} \sum_{M_k\in M} \frac{inner_k}{outer_k}, \quad (17)$$

where M is the set of identified microservices. The values of $inner_k$ and $outer_k$ are obtained as follows:

$$inner_k = \sum_{(i,j)\in E | i,j\in M_k} w_{ij}, \quad (18)$$

$$outer_k = \sum_{(i,j)\in E | i\in M_k} w_{ij}. \quad (19)$$

The optimal value that a microservice architecture can obtain for the average cohesion is 1, indicating that the elements of each microservice are strongly interrelated. A high cohesion value suggests that all the methods inside a microservice have strong relationships with entities included in the same microservice. That implies that a microservice with high cohesion may be well concerned with a specific bounded context. Conversely, a poor microservice decomposition may show a low cohesion since, e.g., many methods need to reference entities placed in other microservices.

Regarding the independence of functionality, we use the *IFN* metric [18], which measures the average number of interfaces exposed by a microservice. It is defined as follows:

$$IFN = \frac{1}{|M|} \sum_{M_k\in M} ifn_k, \quad (20)$$

where $|M|$ is the set of identified microservices and ifn_k is the number of interfaces exposed by the microservice k. According to our system representation, we define an interface as a set of methods of the logic layer, without ingoing edges, that are connected to the same set of entity nodes. A microservice focused on the single responsibility principle is expected to have only one interface, i.e., to offer functionalities related to a single entity. Hence, the lower the value of *IFN* (down to 1), the most likely the microservice architecture is supposed to follow the single responsibility principle.

Concerning the evaluation of the bounded contexts, we check the entities belonging to each microservice and compute the *precision* and the *recall* score with respect to a reference microservice architecture used as a baseline. To obtain precision and recall, we reverse-engineer each microservice to identify its bounded context(s) and collect its domain entities. We define the *precision* for the microservice k in the architecture d as the ratio between (i) the number of domain entities that are both in the microservice k and in the baseline microservice(s) concerning the same bounded contexts, and (ii) the number of domain entities of the microservice k:

$$Precision(k, d) = \frac{|E_d^k \bigcap E_B^k|}{|E_d^k|}, \quad (21)$$

where E_d^k is the set of domain entities into the microservice k in the decomposition d, and E_B^k is the set of domain entities into the microservice k of the baseline.

Similarly, we define the *recall* for the microservice k in the architecture d as the ratio between (i) the number of domain entities in the microservice k provided by the baseline and (ii) the number of domain entities that are both in the microservice k and in the microservice k from the baseline:

$$Recall(k, d) = \frac{|E_B^k|}{|E_d^k \bigcap E_B^k|}. \quad (22)$$

A high value of precision and recall means that a given microservice owns the complete set of entities required to fully realize the bounded context, without including "spurious" entities that may be related to different bounded contexts.

[3]https://github.com/sosygroup/from-monolith-to-microservices

B. Performed Experiments

To run the experiments, we have chosen publicly-available systems, implemented in Java, whose architectures are monolithic, and that have been already used in evaluating related works on the decomposition to microservices. Table I lists the applications that we considered. For each of the considered projects, we show the number of classes (test classes excluded), the number of lines of code, and the number of domain entities and methods identified in the system analysis phase.

TABLE I
APPLICATION USED IN THE REPORTED EXPERIMENTS

Project	Classes	Lines of code	Entities
JPetStore	24	1409	9
Spring Petclinic	23	741	10
SpringBlog	46	1487	9
Cargo Tracking	104	4001	35

JPetstore[4] is a Java web application built on top of the Spring Framework realizing a simple e-commerce system. It is one of the most popular systems and it is used as a reference application in many works [10], [16], [17], [21] (just to mention a few). In [10], an expert decomposition has been proposed for this system. We use it as a baseline for the comparison of the results.

Spring Petclinic[5] is a sample Spring Boot application realizing a veterinary management system. Spring also released a version of this system featuring a microservice architecture, which can be considered a baseline for comparing our results.

SpringBlog[6] is a simple blog system realized using many frameworks such as Spring Boot, Hibernate, and Thymeleaf.

Cargo Tracking[7] is a medium-sized system built on top of the Spring framework. It is used as a reference system for many works, e.g., [9], [14], [15], [19]. In [15], the expected decomposition of the domain model of this system is presented and discussed. We use it as a baseline to evaluate the identification of the bounded contexts.

We performed experiments by running our approach for each of the four applications described above and obtained the decomposed microservice architecture of the system. We collected the results and computed the value of the metrics defined in Section IV-A. Moreover, to allow the comparison of our results with the other state-of-the-art (*sota*) approaches and baseline solutions, we: (i) collected the architectures resulting from each *sota* and the baselines, (ii) represented them through the graph-based representation that we employ in this work, and (iii) computed the value of the metrics by using the same default arc weights that we used for the decomposition with our approach.

[4]https://github.com/mybatis/jpetstore-6
[5]https://github.com/spring-projects/spring-petclinic
[6]https://github.com/Raysmond/SpringBlog
[7]https://github.com/citerus/dddsample-core

C. Experimental Results

Table II reports the results of the decomposition of JPetstore. We report the values of the defined metrics for the obtained microservice architecture according to: our approach, four different *sota* approaches [10], [16], [17], [20], and the baseline obtained from [10]. Besides the *average coupling*, *average cohesion*, and *IFN* metrics, we also report (i) the number of method calls (`Calls` relationships) across different microservices and (ii) the number of references from methods to entities (`Uses` relationships) across different microservices. These two values allow to asses how much microservices are autonomous and how much the offered functionalities are self-contained in a single microservice. All the considered approaches produced 2 to 4 microservices. Interestingly, our approach outperforms the four approaches and the baseline in all the metrics. It obtained the lowest average coupling (0.87), the highest average cohesion (0.97, very close to the optimal value, 1), and the lowest *IFN*. The number of method calls is 3, as those of the baseline, while there are no entity references from one microservice to another. This suggests that: (i) the decomposition complies with the high-cohesion and loose-coupling principles of microservice architectures, (ii) the functionalities of each microservice are independent and self-contained, and (iii) the microservices are built in the right way according to the bounded contexts of the system.

Table III reports and compares the results for the decomposition of Spring Petclinic application. We report the metrics computed for the microservice version of this application[8] provided by Spring (baseline) and for the decomposition approach presented by Kamimura et al. [3]. Our approach was able to find automatically two totally-independent microservices: the decomposition obtained the lowest possible value for the coupling (0) and the highest possible value for the cohesion (1), while there are no method calls or entity references across microservices. In contrast, both Kamimura et al. and the baseline show a higher coupling and a lower cohesion. On the other hand, the *IFN* obtained by our approach is slightly higher (2.0) than the others (1.7). This suggests that, in this case, we found microservices with a lower granularity, being slightly less accurate in following the single responsibility principle, but obtaining complete independence.

Regarding the results obtained for the SpringBlog application, our approach produced three microservices with a satisfiable low coupling (0.67) and high cohesion (0.99) with two method calls and two entity references across different microservices. The *IFN* value is 1.67. As a comparison, Jin et al. [18] reported the value of 2.167 for *IFN* for the decomposition obtained with their approach.

Besides the evaluations discussed above, we also assess the results in the identification of bounded contexts and their consequent identified microservices. While JPetstore, Spring Petclinic, and SpringBlog are relatively simple systems with few entities, Cargo Tracking System features a more complex domain model and a wider set of functionalities. Thus, the

[8]https://github.com/spring-petclinic/spring-petclinic-microservices

TABLE II
RESULT COMPARISON FOR THE DECOMPOSITION OF JPETSTORE

Approach	# Microservices	Average Coupling	Average Cohesion	IFN	# Method Calls	# Entity References
Baseline	3	1.26	0.95	2.0	*3*	2
Zaragoza et al. [10]	3	3.13	0.90	2.0	6	7
Selmadji et al. [20]	2	3.8	0.93	2.5	*3*	8
Jin et al. [17]	4	2.25	0.82	1.75	7	5
Brito et al. [16]	4	3.1	0.85	1.75	9	8
Our	3	*0.87*	*0.97*	*1.7*	*3*	*0*

TABLE III
RESULT COMPARISON FOR THE DECOMPOSITION OF SPRING PETCLINIC

Approach	# Microservices	Average Coupling	Average Cohesion	IFN	# Method Calls	# Entity References
Baseline	3	1.2	0.75	*1.7*	2	3
Kamimura et al. [3]	3	2.07	0.76	*1.7*	2	7
Our	2	*0.0*	*1*	2.0	*0*	*0*

TABLE IV
COMPARISON OF THE MICROSERVICES IDENTIFIED FOR CARGO TRACKING SYSTEM

Approach	# Microservices	Identified services	Included domain entities
Baseline	4	Voyage	Voyage, CarrierMovement
		Location	Location
		Planning	Cargo, Leg, Itinerary, RouteSpecification
		Tracking	HandlingEvent, Delivery
Baresi et al. [15]	4	Trip	CarrierMovement, RouteSpecification
		Tracking	HandlingEvent, Delivery, Voyage
		Planning	Location, Itinerary, Leg, Voyage
		Product	Cargo
Service Cutter [9]	3	Location	Location
		Tracking	HandlingEvent, Delivery
		Voyage & Planning	CarrierMovement, Itinerary, Voyage, Leg, RouteSpecification, Cargo
Our	3	Tracking	HandlingEvent
		Planning & Location	Itinerary, RouteSpecification, Delivery, Cargo, Leg, Location
		Voyage	Voyage, CarrierMovement

identification of bounded contexts for this system is not a trivial task, and it may require specific knowledge of the system. In order to perform the evaluation, we applied our approach and reverse-engineered the obtained decomposition to find the distribution into microservices of the domain model entities. We then labeled the microservice with the name of its emerging bounded context. Table IV reports the distribution of domain entities in each identified service by: our approach, Service Cutter [9], the approach by Baresi et al. [15], and the manually-obtained expected decomposition provided in [15] as a baseline. The baseline consists of 4 microservices (i.e., 4 bounded contexts according to microservices are built): *Voyage*, *Location*, *Planning*, and *Tracking*. The interface analysis run by Baresi et al. was able to find 4 microservices as well, although with different bounded contexts (except for *Planning*). Service Cutter found 3 microservices: *Location*, *Tracking*, and *Voyage & Planning*. As one can notice, bounded contexts are distributed into microservices with a lower granularity (e.g., instead of having *Voyage* and *Planning* in separate microservices, they have been put together). Also, our approach found 3 microservices: *Tracking*, *Planning & Location*, and *Voyage*, thus resulting in a lower granularity than the baseline.

The *Voyage* microservice identified with our approach got 1 for both precision and recall, while *Tracking* microservice got 1 for the precision and 0.5 for the recall. Furthermore, if we compute the metric's values for *Planning & Location* by considering the union of the services *Planning* and *Location* from the baseline solution, we obtain a precision of 0.83 and a recall of 1. The average value for the precision and the recall on the three identified microservice is 0.94 and 0.83, respectively. In comparison, Service Cutter was able to get 1 for the values of both precision and recall for the microservices *Location* and *Tracking*. Also, if we compute the metric's values for *Voyage & Planning* by considering the union of the services *Voyage* and *Planning* from the baseline solution, we

obtain 1 for both the metrics. These results suggest that, at least in this case, and despite not being optimal in the identification of bounded contexts, our approach still obtained comparable results. However, we remark that all the results have been obtained in an automatic way, without requiring any further input or specific knowledge of the system.

V. Threats to validity

1) Internal validity: The evaluation has been performed on a set of metrics that are computed on the same graph representation that we used in our approach. Besides the number of used metrics (that we plan to expand in the future for a more accurate evaluation), computing all of them by relying on the same system view may lead to biased results. However, in the comparison of our solution with other microservice architectures, in order to avoid other biases, we have taken baselines from other works, so we have not been involved in their definition. Also, cohesion and coupling-related metrics, together with *IFN*, despite being widely used for evaluating decomposed architectures [11], [14], [16], do not guarantee that the obtained architectures are the best-fitting for the applications domains and needs, as many different architectures there may exist [41]. Moreover, we have tested our approach only on a small set of projects, that were not very big both in the number of classes and entities. Thus, we still need to test it on a wider variety of systems in terms of both dimension and domain model complexity in order to better assess the scalability of the approach.

2) External Validity: By default, our tool assigns predefined weights to build the information graph after the static code analysis. This permits to fully automate the process, while sparing developers from the task of manual weight specification. However, the default values cannot be generally taken as the best-fitting weights, since they may be less accurate for specific systems, e.g., very big or complex systems. As mentioned in Section III-A, our tool allows developers to refine weights, e.g., according to application-specific requirements.

The use of static analysis as the only mean of system analysis may represent a limitation of the approach. Without dynamic analysis, the tool is not aware of the actual number of method calls that are performed at runtime. This may lead to a decomposition that may not optimize the number of runtime inter-microservices calls. This may affect the system's performance since inter-microservices calls introduce overhead due to network communication. However, the approach can be easily extended to consider data coming from dynamic analysis, by, e.g., including the number of method calls performed at runtime into the information graph and considering it in the optimization's objective function.

The quality of the decomposition may naturally be affected by the bad quality of the system's source code (e.g., it does not follow OOP principles, it has very badly designed classes, or there are code smells that reduce the separation of concerns between classes). In fact, if the relationships among classes representing the domain model are not reported in the code according to the OOP paradigm, we may lose cohesiveness and fail to identify the clusters of entities. This may lead to a bad identification of the system's domain contexts and, as a consequence, to failure in adhering to the single responsibility principle. The same holds if the domain model is very big, complex, and highly interconnected. However, we allow developers to adjust arc weights individually to identify the domain entities that are more closely related, and we also allow them to adjust the entity clusters to overcome these issues.

VI. Conclusion and future work

In this paper, we proposed a decomposition approach that targets the problem of automating the migration of monolithic systems to microservice architectures. The approach automates the main labor-intensive processes, i.e., analyzing the system, extracting the information graph, and identifying the microservices. We exploit the Louvain community detection algorithm to obtain the granularity of the microservices and high-cohesive communities of nodes, while combinatorial optimization produces the final loosely coupled microservices. Experimental results show that the methodology allows obtaining architecturally meaningful microservices that satisfy the main principles of a microservice architecture, i.e., high cohesion, loose coupling, and single responsibility principle. The approach was able to outperform the other approaches to which we compared while being fully automated.

As future work, we plan to address the limitations discussed in the previous section and improve the approach. In particular, we plan to examine more legacy systems for a better evaluation. We are looking towards industrial partners that could adopt our methodology and apply it to their existing systems for further validation and improvements. Also, we plan to allow experts to provide their inputs during or after the community detection phase as a further refinement step. This will improve the quality of the obtained domain contexts, overcome the limitations concerning the clustering phase, and take into account possible application-specific requirements. Moreover, we plan to improve the analysis phase by considering dynamic analysis to optimize the overhead of inter-MS communications. This will also call for more accurate metrics to be used as the objective function of the optimization phase. Finally, we will extend our tool by using machine learning techniques to classify classes more accurately, especially when the system is large and complex, and by including a further synthesis phase that identifies the new interfaces and automatically refactors the system according to the obtained microservice architecture, thus producing the (skeleton) code of the new microservices.

Acknowledgment

This work was partially supported by: the Italian Government under CIPE resolution n. 135 (December 21, 2012), project INnovating CIty Planning through Information and Communication Technologies (INCIPICT); the Italian MIUR SISMA national research project (PRIN 2017, Contract 201752ENYB); and Italian PNRR MUR Centro Nazionale HPC, Big Data e Quantum Computing, Spoke9 - Digital Society & Smart Cities.

REFERENCES

[1] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[2] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Inc., 2015.

[3] M. Kamimura, K. Yano, T. Hatano, and A. Matsuo, "Extracting candidates of microservices from monolithic application code," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 571–580, IEEE, 2018.

[4] J. Fritzsch, J. Bogner, A. Zimmermann, and 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*, pp. 128–141, Springer, 2018.

[5] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How to make your application scale," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pp. 95–104, Springer, 2017.

[6] S. Hassan, R. Bahsoon, and R. Kazman, "Microservice transition and its granularity problem: A systematic mapping study," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1651–1681, 2020.

[7] A. Homay, M. de Sousa, A. Zoitl, and M. Wollschlaeger, "Service granularity in industrial automation and control systems," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 132–139, IEEE, 2020.

[8] R. S. Huergo, P. F. Pires, and F. C. Delicato, "Mdcsim: A method and a tool to identify services," *IT Convergence Practice*, vol. 2, no. 4, pp. 1–27, 2014.

[9] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing* (M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, eds.), (Cham), pp. 185–200, Springer International Publishing, 2016.

[10] P. Zaragoza, A.-D. Seriai, A. Seriai, A. Shatnawi, and M. Derras, "Leveraging the layered architecture for microservice recovery," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pp. 135–145, 2022.

[11] I. Trabelsi, M. Abdellatif, A. Abubaker, N. Moha, S. Mosser, S. Ebrahimi-Kahou, and 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*, 2022.

[12] S. Hassan, N. Ali, and R. Bahsoon, "Microservice ambients: An architectural meta-modelling approach for microservice granularity," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 1–10, IEEE, 2017.

[13] S. Klock, J. M. E. Van Der Werf, J. P. Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 11–20, IEEE, 2017.

[14] Z. Li, C. Shang, J. Wu, and Y. Li, "Microservice extraction based on knowledge graph from monolithic applications," *Information and Software Technology*, vol. 150, p. 106992, 2022.

[15] L. Baresi, M. Garriga, and A. D. Renzis, "Microservices identification through interface analysis," in *European Conference on Service-Oriented and Cloud Computing*, pp. 19–33, Springer, 2017.

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

[17] 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)*, pp. 211–218, 2018.

[18] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, 2021.

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

[20] A. Selmadji, A.-D. Seriai, H. L. 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)*, pp. 157–168, IEEE, 2020.

[21] 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)*, pp. 135–145, 2020.

[22] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 466–475, IEEE, 2017.

[23] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," *arXiv preprint arXiv:1605.03175*, 2016.

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

[25] Y. Romani, O. Tibermacine, and 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)*, pp. 15–19, IEEE, 2022.

[26] I. Pigazzini, F. Arcelli Fontana, and A. Maggioni, "Tool support for the migration to microservice architecture: An industrial case study," in *Software Architecture* (T. Bures, L. Duchien, and P. Inverardi, eds.), (Cham), pp. 247–263, Springer International Publishing, 2019.

[27] H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," *IEEE Software*, vol. 35, no. 3, pp. 44–49, 2018.

[28] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, "Microservice decomposition via static and dynamic analysis of the monolith," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 9–16, IEEE, 2020.

[29] L. Nunes, N. Santos, and A. Rito Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *European Conference on Software Architecture*, pp. 37–52, Springer, 2019.

[30] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," in *2016 XLII Latin American computing conference (CLEI)*, pp. 1–11, IEEE, 2016.

[31] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[32] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[33] V. Raj and S. Ravichandra, "A service graph based extraction of microservices from monolith services of service-oriented architecture," *Software: Practice and Experience*, 2022.

[34] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying microservices using functional decomposition," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pp. 50–65, Springer, 2018.

[35] M. Abdellatif, R. Tighilt, N. Moha, H. Mili, G. El Boussaidi, J. Privat, and Y.-G. Guéhéneuc, "A type-sensitive service identification approach for legacy-to-soa migration," in *International Conference on Service-Oriented Computing*, pp. 476–491, Springer, 2020.

[36] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[37] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[38] E. R. Barnes, "An algorithm for partitioning the nodes of a graph," *SIAM Journal on Algebraic Discrete Methods*, vol. 3, no. 4, pp. 541–550, 1982.

[39] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[40] D. Bertsimas, C.-P. Teo, and R. Vohra, "Analysis of lp relaxations for multiway and multicut problems," *Networks*, vol. 34, no. 2, pp. 102–114, 1999.

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