From Monolithic Architecture Style to Microservice one Based on a Semi-Automatic Approach

Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Rahina Oumarou Mahamane, Pascal Zaragoza, Christophe Dony

From Monolithic Architecture Style to Microservice one Based on a Semi-automatic Approach

Anfel Selmadji*†, Abdelhak-Djamel Seriai*, Hinde Lilia Bouziane*, Rahina Oumarou Mahamane*, Pascal Zaragoza*, and Christophe Dony*
* *LIRMM, University of Montpellier, CNRS*
Montpellier, France
† *MISC Laboratory, Abdelhamid Mehri University*
Constantine, Algeria
{selmadji, seriai, bouziane, zaragoza, dony}@lirmm.fr
om.rahina@gmail.com

***Abstract*—Due to its tremendous advantages, microservice architectural style has become an essential element for the development of applications deployed on the cloud and for those adopting the DevOps practices. Nevertheless, while microservices can be used to develop new applications, there are monolithic ones, that are not well adapted neither to the cloud nor to DevOps. Migrating these applications towards microservices appears as a solution to adapt them to both. In this context, we propose an approach aiming to achieve this objective by focusing on the step of microservices identification. The proposed identification, in this paper, is based on an analysis of the relationships between source code elements, their relationships with the persistent data manipulated in this code and finally the knowledge, often partial, of the architect concerning the system to migrate. A function that measures the quality of a microservice based on its ability to provide consistent service and its interdependence with others microservice in the resulting architecture was defined. Moreover, the architect recommendations are used, when available, to guide the identification process. The conducted experiment shows the relevance of the obtained microservices by our approach.**

***Index Terms*—Object-Oriented, microservices, software migration, identification, architect recommendations, software architecture, quality.**

I. Introduction

Recently, microservices [21, 23] have emerged as a technology and architectural style, in which an application consists of a set of small services that are independently deployable and scalable. Each microservice manages its own data and communicates with others relying on lightweight mechanisms.

Due to their characteristics (e.g., independently deployable, autonomous, etc.), microservices can allow efficient development, deployment, and maintenance. Indeed, since they are autonomous they can be developed, maintained, and tested independently, which facilitates continuous integration and continuous testing (i.e., DevOps practices). Moreover, microservices are typically packed and deployed using containers, which eases their deployment in different platforms such as the cloud.

While microservices can be used to develop new applications, there are monolithic ones (i.e., monoliths) [13, 27] built as a single unit, which hardens their development, deployment, and maintenance. For instance, maintaining a part of a monolith requires testing the entire application, and since we do not know the impact of the change on the rest of the application, there is the chance that parts that have not been updated will fail to start correctly.

Some existing monolithic applications have high business value. Therefore, to overcome their limitations, they can be migrated towards recent architectural styles, such as microservices. The migration process consists mainly of two steps: 1) microservice identification where available software artifacts (e.g., source code, documentation, execution traces, etc.) of the monolithic application are analyzed to determine the corresponding microservices, and 2) microservice packaging where the necessary transformations are performed on the source code of the identified microservices in order that they become executable entities.

Several approaches have been proposed to address the first step of the migration process [5, 7, 12, 14, 15, 16, 19, 20, 22]. Nevertheless, they suffer from three main limitations. Firstly, they are based on ad-hoc and limited heuristics, that are not specifically adapted to the concept "microservice" [5, 7, 12, 15, 16, 22]. Secondly, they do not consider all the characteristics of microservices, especially data autonomy [5, 12, 14, 20]. Therefore, the extracted microservices by these approaches are not the most relevant, and do not reflect all the semantics of the concept "microservice". Finally, the approaches that rely on experts suffer fro certain limitations: 1) they cannot be applied unless an expert is available, 2) they require a profound knowledge of the monolith, and 3) they require intensive expert intervention to perform some steps of the approach, if not all of them.

In this paper, we propose an approach that tackles these limitations. Our approach identifies microservices from monolithic Object-Oriented (OO) applications based on a quality function that, unlike existing approaches, have been defined after an analysis of microservice characteristics to ensure that the produced results are relevant, and reflect the semantics of the concept microservices. It is noteworthy that this function takes into account data autonomy of microservices (Section III). Furthermore, our approach uses architect recommendations to guide the identification process. Nevertheless, these

recommendations are not necessary, they are used when they are available. Moreover, they are related to the use of the application. Thus, they do not require a profound knowledge of the monolithic OO application's source code. Furthermore, the role of the architect is limited to providing recommendations at the beginning of the identification.

To validate our approach, we experimented on several OO applications of different sizes (i.e., from small to relatively large). The obtained results showed the relevance of our choices. On the one hand, the relevance of the microservice-based architectures that have been identified is higher than the results of our automatic approach. On the other hand, taking into account the data autonomy of microservices considerably improves the obtained results.

The remainder of this paper is organized as follows. Section II outlines related works. Section III presents our well-defined quality measurement of microservices. Section IV shows how our approach identifies microservices using a clustering algorithm, and how does it inject software architect recommendations, when available, in the identification process. Section V presents the conducted experimentations to evaluates our proposal. Finally, section VI concludes the paper and provides future directions.

II. Related Works

In literature, several approaches have been proposed to identify microservices from monolithic applications. Nevertheless, they suffer from four main limitations.

Firstly, some approaches [7, 15, 16, 22] require software artifacts, which unlike source code, may be unavailable or not up-to-date, limiting their applicability. For instance, the proposed approach in [15] analyzes and clusters execution traces to identify classes cooperating to perform the same functionality. Whereas, in [22], the authors rely on the change history of the monolith.

Secondly, when microservices are automatically identified from source code, existing approaches do not consider all the semantics of microservices, especially data autonomy [22]. Therefore, the produced results by these approaches may not match those that can be manually identified by an expert. To the best of our knowledge, only two approaches identify microservices from the source code while considering data autonomy [20, 24]. The first one [20] is manual. It identifies microservices from enterprise applications relying on dependencies between their facades and database tables connected by business functionalities. Facades represent the entry points of the system, that invoke business functionalities. The second approach is our previous one [24] in which even though data autonomy is considered, the frequency of data manipulation and the access mode to data have not been taken into account, despite their importance (Section III-B3).

Thirdly, existing approaches either do not benefit from expert knowledge [5, 7, 15, 16, 22] or require expert intervention to perform some of their steps [12, 14, 19], if not all of them [20], which is tedious, error-prone, and time-consuming. Moreover, it limits their applicability. For instance, in the first step of the proposed approach in [12], a purified and detailed DFD is constructed manually from the business logic of the system to be decomposed. The build DFD is the base of the following steps.

It is noteworthy that, to the best of our knowledge, none of the investigated approaches rely on a well-specified set of recommendations that can be easily provided by an expert and allows to enhance the identification results. Furthermore, none of them combine expert recommendations with source code information.

Finally, some approaches [20] rely on a restrictive hypothesis about the architecture of the monolith to be decomposed. For example, in [20], the authors suppose that the application has three main parts: a client side user interfaces, a server side, and a database. Moreover, they consider that a large system consists of smaller subsystems and each one has a well-defined set of business functionalities. In addition, they assume that each subsystem has a separate data store.

III. Measurement of the Quality of Microservices

Our identification process partitions into clusters the classes of the original monolithic software architecture to identify the microservices of the targeted one. To determine from all the possible partitions, those that reflect a relevant microservice-based architecture, it is necessary to measure the "relevance" of this architecture. Note that the term quality of architecture or microservice is used to refer to their relevance.

To quantitatively measure the quality of a candidate microservice, similarly to the proposed approach in [2], we were inspired by the *ISO/IEC 25010:2011* [1] model, which links the quality characteristics of a software product to the corresponding metrics for measuring each one. Therefore, the characteristics that reflect the quality of a microservice were identified and then refined to obtain the metrics that allow measuring them.

The identification of microservice characteristics is based on an analysis of their most commonly used definitions. In literature, several ones have been proposed [21, 23, 25]. Based on these definitions and others [26], the main characteristics of a microservice are the following:

- *Small and focused on one functionality:* even if *small* is not a sufficient measure to describe microservices, it is used as an attempt to imply their granularity [21, 23, 26]. However, a question that is often asked is *how small is small?* A microservice is typically responsible for a granular unit of work (i.e., encapsulates a simple business functionality). Therefore, it must be small enough so that its whole design and implementation can be understood. Moreover, it can be maintained or rewritten easily.
- *Autonomous:* microservices are separate entities that can be developed, tested, upgraded, replaced, deployed, and scaled independently from each other. All communications between the microservices themselves are via network calls, to enforce separation between them and ensure that they are loosely coupled (i.e., structural and

behavioral autonomy) [21, 23, 25]. Moreover, each one manages its own database (i.e., data autonomy) [21, 23].
- *Technology-neutral:* with a system composed of a set of collaborating microservices, it is possible to use different technologies inside each one. This allows picking the right tool for each job [21, 23].
- *Automatically deployed:* with a system consisting of a small number of microservices, it might be acceptable to manually provision machines to deploy them. Nevertheless, if this number increases, at some point, the use of a manual approach might not be possible. Hence, automatic deployment is required.

These characteristics can be classified into two categories: 1) those related to the structure and behavior of microservices, and 2) others related to the implementation technologies and deployment platform of microservices. The first category concerns characteristics that are independent of the microservices implementation technologies and deployment platforms. They are related to design (resp., identification) phase of the microservice development (resp., migration) process. It includes "small and focused on one functionality and "autonomous" characteristics. The second category depends on implementation technologies and development platforms. It includes "technology-neutral" and "automatically deployed" characteristics, which are related to the packaging phase of the development/migration process.

In order to evaluate the quality of candidate microservices, from the above-mentioned characteristics, only the ones that define microservice structure and behavior are selected: "small and focused on one functionality" and "autonomous". Note that the autonomy of a microservice includes its structural and behavioral autonomy as well as its data autonomy.

To measure these characteristics, a set of metrics were chosen and used to define a quality function. It is a weighted aggregation of two sub-functions. The first one evaluates the strength of all the structural relationships between source code elements, that will constitute the implementation of a potential microservice. The second sub-function measures the degree of dependence of microservice classes on persistent data. The goal of our approach is to identify microservices with the maximized quality function values.

A. Measuring the Quality of a Microservice Based on Structural and Behavioral Dependencies

Regarding their structural and behavioral dependencies, the quality of microservices is measured based on two elements. The first one consists in determining the characteristics that a microservice should have and how they can be evaluated based on metrics. The second element consists in identifying among all the existing implementations of the used metrics, those that best reflect the characteristics to be assessed.

1) Quality Function based on the Assessment of Appropriate Characteristics:

1.1) Measuring Focused on One Functionality Characteristic of a Microservice: in our approach, a microservice is viewed as a set of classes collaborating to provide a given function. This collaboration can be determined from source code through the internal coupling measure, which represents the degree of direct and indirect dependencies between the set of classes, representing a candidate microservice. The more two classes of a candidate microservice use each other's methods, the more they are internally coupled. Furthermore, the collaboration can be determined by measuring the number of volatile data (i.e., not persistent data) such as attributes whose use is shared by these classes. It reflects the internal cohesion measure.

To evaluate the characteristic "Focused on One Function" of a set of classes representing a candidate microservice *M*, the function *FOne* was defined as follows:

$$FOne(M) = \frac{1}{2}(InterCoup(M) + InterCoh(M)) \quad (1)$$

1.2) Measuring the Structural and Behavioral Autonomy of a Microservice: as explained earlier, microservices are separate entities that can be developed, tested, upgraded, replaced, deployed, and scaled independently from each other. Therefore, in order that a set of classes represents a microservices, they should be self-sufficient. In other words, their dependencies on external classes should be minimal. This can be measured using external coupling, which evaluates the degree of direct and indirect dependencies between the classes belonging to the microservices and the external classes.

To evaluate the structural and behavioral autonomy of a candidate microservice *M*, the function *FAutonomy* was defined as follows:

$$FAutonomy(M) = ExterCoup(M) \quad (2)$$

1.3) Quality Measurement Relying on Structural and Behavioral Dependencies: the two functions that measure the quality of a microservice based on structural and behavioral dependencies were aggregated in one *FStructBeh* as follows:

$$FStructBeh(M) = \frac{1}{n}(\alpha FOne(M) - \beta FAutonomy(M)) \quad (3)$$

Where α and β are coefficient weights specified by a software architect and $n = \alpha + \beta$. The default value of each term is 1.

2) Measuring Microservice Characteristics Based on Appropriate Metrics: earlier in this section, two microservice characteristics have been evaluated using some metrics. The rest of this section presents how these metrics are computed to reflect the semantics of the corresponding measures.

2.1) Internal coupling: the internal coupling measures the degree of direct and indirect dependencies between the classes of a microservice. These dependencies correspond to method calls. The more two classes use each other's methods, the more they are internally coupled (i.e., higher internal coupling values). This can be evaluated by measuring the frequency of internal calls between classes. Hence, the internal coupling is computed as follows:

$$InterCoup(M) = \frac{\sum CoupP(P)}{NbPossiblePairs} \quad (4)$$

Where *P= (C1, C2)* is a pair of classes of the microservice *M*, *NbPossiblePairs* is the number of possible pairs of classes in *M*, while *CoupP* is computed as follows:

$$CoupP(C1, C2) = \frac{NbCalls(C1, C2) + NbCalls(C2, C1)}{TotalNbCalls} \quad (5)$$

Where *NbCalls(C1, C2)* is the number of calls of the methods of *C1* by the methods of *C2* and *TotalNbCalls* represents the total number of method calls in the OO application.

In fact, computing internal coupling using *Equation 4* considers the frequency of calls between methods. Nevertheless, it does not promote clusters in which the values of the *CouplingPair* are close (i.e., all the classes are coupled). To tackle this problem, the standard deviation between the coupling values was introduced in the computing of the internal coupling as follows:

$$InterCoup(M) = \frac{\sum CoupP(P) - \sum_{PV \in PairsV} \sigma(PVal)}{NbPossiblePairs} \quad (6)$$

Where $\sigma(PV)$ is the standard deviation between the *CoupP* values of the pair PV belonging to all the possible pairs of values $PairsV$.

Note that introducing the standard deviation in the computing of the internal coupling ensures that very big or very small coupling values between a small sub-set of microservice classes do not impact the overall evaluation of the internal coupling.

2.2) External coupling: external coupling measures the degree of direct and indirect dependencies of the classes belonging to a candidate microservice on external classes. It is computed as shown in *Equation 7*. Where P is a pair of classes such that only one class belongs to the microservice *M*, but not both, *CoupP* is measured using *Equation 5*, $\sigma(PV)$ is the standard deviation between the *CoupP* values of the pair PV belonging to all the possible pairs of values $PairsV$. Whereas, *NbPossibleExternalPairs* is the number of pairs of classes in which only one class belongs to the microservice *M*. It is noteworthy that the main difference in the evaluation of internal and external coupling is the used pairs of classes.

$$ExterCoup(M) = \frac{\sum CoupP(P) - \sum_{PV \in PairsV} \sigma(PV)}{NbPossibleExternalPairs} \quad (7)$$

2.3) Internal cohesion: internal cohesion evaluates the strength of interactions between classes. Generally, if the methods of two classes manipulate the same attributes, these classes are more interactive. Hence, internal cohesion is computed as follows:

$$InterCoh(M) = \frac{NbDirectConnections}{NbPossibleConnections} \quad (8)$$

Where *NbPossibleConnections* is the number of possible connections between the methods of the classes belonging to the microservice *M*, while *NbDirectConnections* is the number of connections between these methods. Two methods *method1* and *method2* are directly connected if they both access the same attribute or the call trees starting at *method1* and *method2* access the same attributes.

Note that when measuring the internal cohesion using *Equation 8*, the connections between the methods of the same class are taken into account. However, our goal is to evaluate the cohesion between the classes of the candidate microservices. To solve this problem, the connections between the methods of the same class are not considered. It is noteworthy that the proposed internal cohesion evaluation metric is a variation of the metric *TCC (Tight Class Cohesion)* [9].

B. Measuring the Quality of a Microservice Based on Data Autonomy

One of the main characteristics of a microservice is its data autonomy [21, 23]. A microservice can be completely data autonomous if it does not require any data from others. In order that a microservice needs less data, the internal data manipulations (i.e., reading and writing operations) between its classes should be maximized, while the accesses to external data should be minimized.

To identify such microservices, *FData* is defined as shown in *Equation 9*. It is based on measuring data dependencies between the classes of the microservice (*FIntra*), as well as their dependencies with external classes (*FInter*). Note that the microservices having high accesses to data manipulated by external classes (i.e., high values of *FInter*) are penalized. Indeed, the greater *FInter* is, the lower *FData* will be.

$n = \alpha + \beta$, while α and β are coefficient weights specified by a software architect based on his/her knowledge of the system to be migrated and the aims of the migration. For instance, if partitioning the database is one of the aims, to facilitate it, he/she can give more weight to $FIntra$, which allows maximizing the shared data between the classes of a microservice. The default value of each term is 1.

$$FData(M) = \frac{1}{n} \quad (\alpha FIntra(M) - \beta FInter(M)) \quad (9)$$

1) Computing FIntra: $FIntra$ function applied on a microservice M represents the average of data dependencies measurement between all the possible pairs of classes belonging to M (*Equation* 10). We chose the average of the data manipulations measurement instead of the sum because the latter promotes large microservices (i.e., consisting of a high number of classes) even if their classes do not manipulate the same data.

$$FIntra(M) = \frac{\sum_{c_i, c_j \in M} DataDepend(c_i, c_j)}{NbPossiblePairsInMicroservice} \quad (10)$$

To better understand, Fig. 1 shows an example, in which the microservice *M2* contains four classes (*C4*, *C5*, *C6*, and *C7*) manipulating the same data (*D2* and *D3*). Whereas, *M1* has only two classes (*C1* and *C2*) accessing to *D1*. Therefore, the sum of data manipulations measurement for *M2* will be higher than *M1*. This indicates that *M2* is better, while it is not the case, since all the classes of *M1* manipulate the same data, whereas merely a one-third of *M2*'s classes do that.

2) Computing FInter: $FInter$ represents the average of measuring data dependencies between all the pairs of classes in which only one class belongs to the microservice M (*Equation* 11). Note that the main difference between *FIntra* and *FInter* is the used pairs of classes.

$$FInter(M) = \frac{\sum_{c_i, c_j \in Classes} DataDepend(c_i, c_j)}{NbPossibleExternalPairs} \quad (11)$$

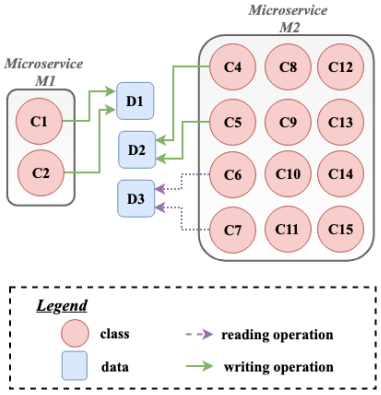

Fig. 1. Example motivating the use of the average to compute *FIntra*

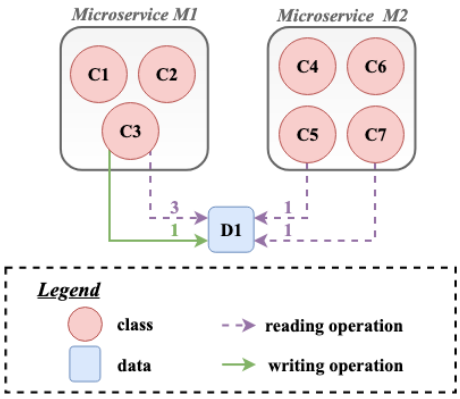

Fig. 2. Example of the frequency of data manipulations

3) Computing DataDepend: Both *FIntra* and *FInter* rely on *DataDepend*. This function measures data dependencies between two classes based on their read and written data. Considering the access mode to data (i.e., read and/or write) while measuring the data autonomy of a microservice is important. Based on the data ownership pattern proposed to decompose a monolith into microservices [6], data can be modified or created just through its owner (i.e., corresponding microservice). Other microservices are allowed to have a copy of the data that they do not own, but they should be careful about its staleness.

DataDepend is measured as follows:

$$DataDepend(c_i, c_j) = \frac{\sum_{k \in Data} D(c_i, c_j, k)}{NbDataManipulatedInMicro} \quad (12)$$

Where $Data$ is the set of data manipulated in the microservice *M* and *NbDataManipulatedInMicro* is its size. D is defined, inspired by the proposed approach in [5], as follows:

$$D(c_i, c_j, k) = \begin{cases} \text{- 1 if } c_i \text{ and } c_j \text{ write } k. \\ \text{- 0.5 if a class writes } k \text{ and the other one reads it.} \\ \text{- 0.25 if } c_i \text{ and } c_j \text{ read } k. \\ \text{- 0 otherwise.} \end{cases}$$

In fact, *DataDepend* is the average of data manipulation measurement for a given pair of classes. We chose the average instead of the sum to promote microservices manipulating data more strongly (i.e., microservices having higher values of *D*). Note that if a class reads and writes the same data only the writing (major) operation is considered.

Measuring *DataDepend* using *Equation* 12 does not take into account the frequency of data manipulations. This frequency has a substantial impact on the autonomy of the identified microservices. For instance, if a class of a candidate microservice $M1$ frequently manipulates a data $D1$ associated to another microservice $M2$ (i.e., $D1$ was associated to $M2$ because this microservice contains more classes manipulating it less often (Fig. 2)). Once the microservice identification and packaging are done, each manipulation will be interpreted as a communication between $M1$ and $M2$. Frequent manipulations produce frequent communications, which reduce the autonomy of the identified microservices.

Hence, to identify autonomous microservices, the frequency was introduced in the *DataDepend* measurement, as shown in *Equation* 13.

$$DataDepend(c_i, c_j) = \frac{\sum_{k \in Data}(D(c_i, c_j, k) * Freq(c_i, c_j, k))}{NbDataManipulatedInMicro} \quad (13)$$

Where $Freq(c_i, c_j, k)$ is the number of times the classes c_i and c_j manipulate k. It is defined as follows:

$$Freq(c_i, c_j, k) = FreqCl(c_i, k) + FreqCl(c_j, k) \quad (14)$$

Measuring the frequency using *Equation* 14 does not promote clusters in which the data manipulation frequency of k for the two classes c_i and c_j is close. To tackle this problem, the standard deviation was introduced in the frequency measurement, as shown in *Equation* 15.

C. Global Measurement of the Quality of a Microservice

The global evaluation of the quality of a microservice depends on the measurement of its quality based on structural and behavioral dependencies, as well its quality relying on its data autonomy. To measure this quality, the function *FMicro* was defined as follows:

$$FMicro(M) = \frac{1}{n}(\alpha FStructBeh(M) + \beta FData(M)) \quad (16)$$

Where α and β are coefficient weights specified by a software architect and $n = \alpha + \beta$. The default value of each term is 1.

Note that the coefficient weights show the importance of the relationships between code entities (*FStructureBehavior*) and their relationships with persistent data (*FData*). A software architect, according to his/her knowledge of the system to migrate, can decide to give more or less importance either to *FStructBeh* or *FData*.

For example, if the source code was developed respecting the rules of separation of responsibility, modularity, and so on, the architect can give a high coefficient weight to *FStructure-Behavior*. Otherwise, he/she can lower it.

$$Freq(c_i, c_j, k) = FreqCl(c_i, k) + FreqCl(c_j, k) - \sigma(FreqCl(c_i, k), FreqCl(c_j, k)) \quad (15)$$

IV. Identification of Microservices Based on Software Architect Recommendations

A. Identification Guided by Architect Recommendations

It is clear that nothing is worth the human expert to understand software applications. However, companies aiming to migrate their applications are often confronted with the problem of turnover and the absence of these experts. Furthermore, when they are present, their cost makes the migration process extremely expensive. This cost depends on the time spent by the experts to realize/control the migration process. Hence, clear evidences emerge regarding this process: 1) the more automated this process is, the better it is, 2) the migration process should use the knowledge of the experts, when they are available and 3) specifying the recommendations should not require neither a profound knowledge of the source code nor an intensive intervention of the architect. Regarding the last point, the required knowledge should be related, mainly, to the use of the application.

While respecting these evidences, we defined a set of recommendations (Fig. 3). The list of these recommendations is specified to lighten the expert task as much as possible, allowing to use its knowledge to enhance the quality of the identified microservices while reducing its cost. Depending on the availability of these recommendations, several types of identification can be carried out.

In our approach, we decided to exploit the recommendations presented in Fig. 3. There are two main ones: gravity centers and number of microservices.

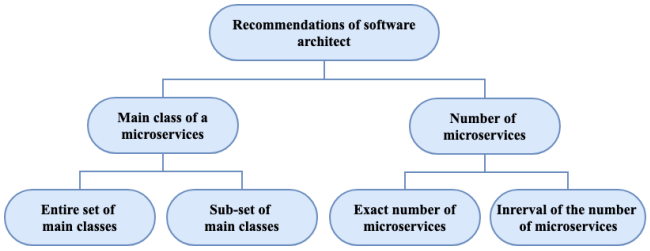

Fig. 3. The used recommendations of software architect

Why relying on gravity centers? in many cases, a microservice is built around a class that constitutes its "gravity center". It represents the functional core of the microservice (i.e., the main class). Other classes revolve around this "core" class to constitute the complete implementation of the microservice. This information, when available, is exploited to build microservices, where each center of gravity will be the starting point to group the classes.

Why relying on the number of microservices? the granularity of the microservices constituting the target architecture is a determinant element of their relevance. Nevertheless, this element is very dependent on the architect's style (i.e., coarse grains versus fine grains). For this reason, the granularity can be very variable in the approaches based only on the source code. Two pieces of information from the architect can be indicators of this granularity: the number of classes per microservice and the number of microservices in the target architecture. We believe that defining the same number of classes for all microservices goes against a partitioning that depends on their quality (i.e., in the same architecture, it is not excluded that some microservices can be relatively larger than others). Giving the exact number of classes for each microservice amounts to manually identifying them (i.e., against a semi-automatic approach). Hence, the retained recommendation as an indicator of the granularity of microservices is their number in the target architecture.

B. Examples of Identification Based on Architect Recommendations

In our approach, depending on the available recommendations, several identifications can be carried out. Due to space limitations, only two examples will be presented.

1) Identification Based on the Entire Set of Gravity Centers: to identify microservices based on the entire set of gravity centers, the idea is to consider each gravity center as a cluster. Then, the remaining classes of the OO application are partitioned iteratively on these clusters based on their quality evaluation. At each iteration, a remaining class is associated to the cluster for which adding this class produces the highest value of the quality function. In the end, the OO application classes are partitioned into clusters. Each one of them contains one gravity center. Algorithm 1 shows the presented clustering.

Algorithm 1: Clustering based on the entire set of gravity centers

input : A set of gravity centers $S_{Gcenters}$
A set of remaining OO classes $S_{Rclasses}$
output: A set of clusters $S_{Clusters}$

for *each* $class \in S_{Gcenters}$ **do**
  let $class$ be a $cluster$;
  add $cluster$ to $S_{Clusters}$;
end
for *each* $class \in S_{Rclasses}$ **do**
  for *each* $cluster \in S_{Clusters}$ **do**
    let $tempCluster$ be a cluster containing all the classes of $cluster$;
    add $class$ to $tempCluster$;
    let $quality$ be the quality function value of $tempCluster$;
    save the pair ($quality$,$cluster$);
  end
  let $best_{cluster}$ be the cluster of the pair ($quality$,$cluster$) such that $quality$ is the highest value in all pairs;
  add $class$ to $best_{cluster}$;
end
return $S_{Clusters}$;

2) Identification Based on the Exact Number of Microservices: to make use of the availability of the exact number of microservices, our idea is to firstly identify microservices, and then compose/decompose the identified ones to obtain

the exact number, while taking into account the quality of the identified microservices. Thus, there are two steps: 1) microservices identification, and 2) microservice composition/decomposition (Algorithm 2).

Algorithm 2: Clustering based on the exact number of microservices

input : A number of microservices Nb
A set of OO classes $S_{Classes}$
A dendrogram $dendrogram$
output: A set $S_{Clusters}$ consisting of Nb cluster

let $initial_{clusters}$ be the identified clusters using the hierarchical clustering algorithm;
if *sizeOf*$(initial_{clusters}) = Nb$ **then**
  $S_{Clusters} \leftarrow initial_{clusters}$;
else
  if *sizeOf*$(initial_{clusters}) < Nb$ **then**
    $S_{Clusters} \leftarrow$ decomposeMicro(Nb,$initial_{clusters}$,$dendrogram$);
  else
    $S_{Clusters} \leftarrow$ composeMicro(Nb,$initial_{clusters}$,$dendrogram$);
  end
end
return $S_{Clusters}$;

Step 1: Microservices identification: to identify microservices, a hierarchical clustering algorithm [17] is used.
Step 2: Microservices composition/decomposition: once the microservices are identified, their number is compared to the exact one:

- If they are equal, the identification is completed.
- If the number of identified microservice is lower than the exact number, these microservices are decomposed. The question is which ones should be decomposed? An adequate answer to this question should take into account the quality of the produced decomposition. Therefore, our idea is to decompose a microservice at a time and compute the sum of the quality function values for the new set of microservices. The decomposition which produces the best result is chosen. This step is repeated as long as the number of microservices is lower then the requested one.
- If the number of identified microservice is higher than the requested one, instead of decomposing them, they are composed until the exact number is obtained.

Composing/decomposing microservices this way allows us, on the one hand, to obtain the exact number of microservices, and on the other hand, produce a decomposition with the highest quality function values.

V. Experimentation and Validation

This section presents the conducted experiments on case studies to validate qualitatively our identification approach of microservices from OO applications. We start by presenting the research questions that we have attempted to answer empirically. We also present the experimental protocol of the experiments carried out to answer the research questions. Finally, we discuss the internal and external threats to validity with respect to the conducted experiments.

A. Research Questions

To validate our proposal, we conducted two experiments to answer the following research questions:

- **RQ1:** does the proposed quality function produce an adequate decomposition of an OO application into microservices? Our approach partitions an OO application into microservices based on the proposed quality function and software architect recommendations, when available. This question aims to check whether the defined function enables obtaining relevant microservices without considering the recommendations of a software architect.
- **RQ2:** is the definition of the quality function, without considering data autonomy, adequate? The goal behind this research question is to check whether the assessment of the characteristics "focused on one function" and "structural and behavioral autonomy" produce appropriate microservices.
- **RQ3:** does the evaluation of data autonomy characteristic enhance the quality of microservices? This question aims to check whether the function *FData* related to the evaluation of data autonomy characteristic allows improving the quality of the identified microservices compared to those identified only based on the assessment of "focused on one function" and "structural and behavioral autonomy" characteristics.
- **RQ4:** does the use of software architect recommendations enhance the identification results? The goal behind this research question is to check whether software architect recommendations guide our approach to produce better results.
- **RQ5:** what are the software architect recommendations that generate the best decomposition of an OO application into microservices? Since our approach uses several recommendations, this question aims to determine the ones that produce the best results.

B. Experimental Protocol

Our experiments conducted to answer the previous research questions are based on a prototype plug-in that we developed in Java. It carries out the identification process defined in our approach. This section presents the experimental protocols followed to answer these questions based on a qualitative evaluation of the identified microservices using our plug-in.

In order to answer the **RQ1**, we used our plug-in to partition three Java applications, that will be presented in Section V-C1. Since the goal of the **RQ1** is to evaluate the correctness of the proposed quality function, we set the plug-in to apply the clustering algorithm that does not take as inputs any software architect recommendations (i.e., fully automatic). Then, we compared the produced microservices with those identified manually. To carry out the manual identification, we analyzed the source of these applications and thoroughly understood their known features. Thus, we can be considered as their experts. It is noteworthy that to avoid biasing the results, we firstly partitioned these applications manually. Then, we carried out the identification using our plug-in.

The protocol for answering the **RQ2** is similar to the one used to answer the **RQ1** with only one difference: we set our plug-in to identify microservices based on the sub-function *FStructureBehavior* related only to the characteristics "focused on one function" and "structural and behavioral autonomy".

To answer the **RQ3**, we simply compare the recall values obtained respectively from the experiment related to answering the **RQ1** and the **RQ2**.

The protocol for answering the **RQ4** is based on comparing the results generated by our identification approach using the software architect recommendations (i.e., semi-automatic) with those obtained without using them (i.e., fully automatic). The results of the manual identification remain of course the reference of confidence to calculate the distance between the two modes of identification (i.e., fully automatic or semi-automatic).

To answer the **RQ5**, we simply compare the identification results using the different software architect recommendations (i.e., entire set of gravity centers, sub-set of them, exact number of microservices, interval of numbers, as well as a combination of the exact number and a sub-set of gravity centers). The recommendations that produce the most relevant microservices are the best ones to guide the identification process.

As explained earlier, to evaluate the produced microservices, we compare them with those identified manually. Thus, we classify the microservices obtained manually in three categories:

- **Category 1: Excellent microservices:** this category includes the microservices that exactly match the ones identified by our approach.
- **Category 2: Good microservices:** the microservices that can be obtained by at most three composition/decomposition operations of the ones identified by our approach are considered as good microservices.
- **Category 3: Bad microservices:** they are the ones that are neither in the first nor the second categories.

C. Qualitative Evaluation

In this section, we firstly outline the Java applications used to validate our approach qualitatively. After that, we present the microservice identification results from these applications. Finally, we interpret the obtained results.

1) Data Collection: to have a codebase for partitioning OO applications into microservices, we collected several Java projects from *GitHub*. These projects have different sizes: small (*FindSportMates*[1]), average (*SpringBlog*[2]), and relatively large (*InventoryManagementSystem*[3]). The source code of these applications used in our experiment, as well as their libraries, have been gathered in https://seafile.lirmm.fr/d/2bb141de92c9420092b9/. Table I provides some metrics on these applications.

[1]https://github.com/chihweil5/FindSportMates

[2]https://github.com/Raysmond/SpringBlog

[3]https://github.com/gtiwari333/java-inventory-management-system-swing-hibernate-nepal

TABLE I
APPLICATIONS METRICS

Application	No of classes/ interfaces	No of classes representing database tables	Code size (LOC)
FindSportMates	17	2	895
SpringBlog	43	5	1617
InventoryManagement System	104	19	13449

FindSportMates is an application which allows users to find groups of people with whom they can play certain sports. *SpringBlog* is a clean-design blog system implemented with Spring Boot. *InventoryManagementSystem* is an application that supports the main inventory management operations.

2) Microservices Identification Results: the source code of each of the previous applications was partitioned into a set of clusters. The results of classifying the identified microservices based on our protocol are described in Table II and expressed in term of recall in Table III. Recall assesses the ratio between the number of excellent and good microservices to the number of the manually identified ones.

3) Interpreting Results: firstly, the recall values related to the results obtained based only on the quality function *FMicro* (i.e., without architect recommendations) are equals to or greater than 80% (80% and 100%). This shows that a large part of the produced microservices are those identified manually. However, for *InventoryManagementSystem*, the number of bad microservices is relatively high (i.e., 4 bad microservices). When analyzing results, we found out that usually, the bad microservices are the ones containing utility classes. Manually, we identified them as microservices because they participate in the realization of several functionalities of *InventoryManagementSystem* application. The utility classes generally do not use each other's methods or attributes. Moreover, they do not manipulate any data. Therefore, our approach could not identify them as microservices.

Secondly, the recall values obtained relying on the sub-function *FStructBeh* are between 65% and 100% (65%, 70%, and 100%). They are equal to or less than those obtained based on the entire quality function *FMicro*. Nevertheless, they remain high. An analysis of Table I allows us to understand that the same values are related to *FindSpotMates* that does not have many persistent data.

Thirdly, the recall values obtained using software architect recommendations are higher than 80%. These values are equal to or higher than those obtained without using software architect recommendations. Furthermore, for *SpringBlog* and *InventoryManagementSystem*, the highest values are produced relying on software architect recommendations (100%). For FindSportMates, the highest value is 100%. Nevertheless, since this application is small, it was partitioned correctly with and without software architect recommendations. Additionally, by analyzing the results of Table III, we can see that precise (i.e., exact number) and complete (i.e., entire set of gravity centers) produce better results (i.e., higher recall values or more excellent microservices). For instance, the recall values

TABLE II
MICROSERVICE CLASSIFICATION RESULTS

Microservice identification				Applications		
				findSportMates	SpringBlog	InventoryManagementSystem
Automatic identification	Based on FMicro		Number of excellent microservices	1	0	1
			Number of good microservices	2	8	15
			Number of bad microservices	0	2	4
	Based on FStructure Bahavior		Number of excellent microservices	0	0	0
			Number of good microservices	3	7	13
			Number of bad microservices	0	3	7
Semi-automatic identification	Gravity centers	Entire set	Number of excellent microservices	1	2	5
			Number of good microservices	2	8	15
			Number of bad microservices	0	0	0
		Sub-set	Number of excellent microservices	0	1	5
			Number of good microservices	3	8	14
			Number of bad microservices	0	1	1
	Number of microservices	Exact number	Number of excellent microservices	1	1	2
			Number of good microservices	2	7	15
			Number of bad microservices	0	2	3
		Interval of number	Number of excellent microservices	0	0	1
			Number of good microservices	3	8	16
			Number of bad microservices	0	2	3
	Exact number and a sub-set of gravity centers		Number of excellent microservices	1	3	5
			Number of good microservices	2	6	14
			Number of bad microservices	0	1	1

TABLE III
RECALL MEASUREMENT

Microservice identification			Recall		
			Find Sport Mates	Spring Blog	Inventory Management System
Automatic identification	FMicro		100%	80%	80%
	FStructureBehavior		100%	70%	65%
Semi-automatic identification	Gravity centers	Entire set	100%	100%	100%
		Sub-set	100%	90%	95%
	Number of microservices	Exact number	100%	80%	85%
		Interval of number	100%	80%	85%
	Exact number and a sub-set of gravity centers		100%	90%	95%

related to the results obtained from *SpringBlog* relying on the exact number and interval of numbers are the same (80%). However, the identification guided by the exact number produced 1 excellent microservices and 7 good ones, whereas the one guided by an interval of numbers produced 8 good microservices.

Finally, the recall values obtained based on more recommendations (i.e., entire-set of gravity centers, which imply the availability of the number of microservices, or the exact number and a sub-set of gravity centers) are equal or higher than those obtained based on one recommendation (i.e., sub-set of gravity center, exact number or interval of numbers). Usually, when the values are the same, more recommendations produce a higher number of excellent microservices. For instance, the identification results related to *SpringBlog* application based on software architect recommendations can be ordered as follows: 1) entire set of gravity centers (100%), 2) exact number and a sub-set of gravity centers (90% and 3 excellent microservices), 3) sub-set of gravity centers (90% and 1 excellent microservices), 4) exact number (80% and 1 excellent microservice) and 5) interval of numbers (80% and no excellent microservice). It is noteworthy that the identification guided by the entire set of gravity centers produce the best results for the three applications (100%).

D. Answers Based on the Qualitative Evaluation

based on the qualitative evaluation of the obtained results by our approach, the research questions can be answered as follows:

1) Since a large part of the identified microservices without using the architect recommendations are those identified manually, we answer the **RQ1** as follows: the proposed quality function produces an adequate decomposition of an OO application into microservices.
2) Even though the obtained recall values based on *FStructBeh* are equal to or lower than those produced by *FMicro*, they still high. Therefore, we answer the **RQ2** as follows: the definition of the quality function, without considering data autonomy, is adequate.
3) The obtained recall values based on *FMicro* are equal to or higher than those obtained relying on *FStructBeh*. Moreover, the same values are related to the application that does not have many persistent data. Hence, we answer **RQ3** as follows: the evaluation of data autonomy characteristic enhance the quality of microservices.
4) The recall values obtained using software architect recommendations are equal to or higher than those obtained without using them. Furthermore, the best results are always produced relying on these recommendations. Therefore, the answer to **RQ4** is the following: the use of software architect recommendations enhances the identification results.
5) Since the best identification results are produced based on the entire set of gravity centers, we answer **RQ5** as follows: the software architect recommendations that generate the best decomposition of an OO application into microservices is the entire set of gravity centers.

E. Threats to Validity

1) Threats to Internal Validity: the proposed approach may be affected by the following internal threats:

- Our decomposition of an OO application into microservices realizes a partition of the classes. Therefore, each class belongs to one and only one cluster. This may not reflect the reality of some applications where some classes may participate in the realization of several functionalities. In this case, the results of the identification could be negatively impacted. Nevertheless, this threat is limited because it generally concerns only certain classes that the architect can duplicate.
- We rely on a hierarchical clustering algorithm to partition the classes of an OO application. This algorithm does not allow to obtain optimal values of the quality function. Indeed, some grouping choices may not be the best considering the whole process of grouping and not just the one at a given moment. Nevertheless, it makes it possible to obtain values close to optimal ones because it performs optimization at each clustering step.

2) Threats to External Validity: our approach could be concerned with the following main external threats:

- The quality of the OO source code can impact the results of the microservice identification. However, since our quality function is a weighted aggregation of two sub-functions (*FStructBeh* and *FData*), in the case of poor quality code, the impact of this factor can be reduced by lowering the coefficient weight of *FStructBeh*.
- To validate our approach, we manually identified reference microservices from the three applications presented above, instead of relying on their experts due to their unavailability. Nevertheless, since we carried out the identification after analyzing the source code of these applications, and understanding their main features, we can be considered as their experts.
- The matching between the microservices that can be obtained by our approach and those obtained manually can vary according to the granularity of microservices obtained by a manual identification.
- Our approach was experimented only on Java applications. To apply it on the ones implemented using other languages, we have to adapt it to take into consideration the specificities of these languages.

VI. Conclusion

The main contribution of this paper is the proposal of an approach for the identification of microservices from OO source code. The proposed approach is based mainly on two types of information: the source code information and the knowledge, often partial, of the architect concerning the system to migrate. On the one hand, the source code information includes relationships between its elements as well as their relationships with the persistent data manipulated in this code. In fact, source code information is used by a quality function to evaluate the relevance of a candidate microservice. This function was defined based on an analysis of microservice characteristics. On the other hand, architect recommendations are related, mainly, to the use of the applications. Our approach proposes to identify microservices using a hierarchical clustering algorithm and architect recommendations when available. The conducted experimentation shows the relevance of the identified microservices by our approach. Nevertheless, these results need to be consolidated by experimentations on other applications, especially those of considerable size. To further strengthen our validation, we intend to compare our approach with state-of-the-art approaches. We plan also, inspired by exiting works such as [11], to use a search-based algorithm instead of a clustering one. A search-based algorithm could treat microservice identification as a multi-objective problem, where more than one objective (i.e., metric) can be simultaneously optimized. Moreover, to benefit from both clustering and search-based algorithm, we intend to combine them [18]. Furthermore, we intend to use documentation, when available and up-to-date, to guide the identification process [10]. Finally, inspired by existing works [3, 4, 8], we want to propose an approach to package the identified microservices and deploy them in the cloud while taking into account the dynamic reconfiguration.

References

[1] ISO/IEC 25010:2011, systems and software engineering - systems and software quality requirements and evaluation (SQuaRE) - system and software quality models. Technical report, British Standards Institution, 2013.

[2] Seza Adjoyan, Abdelhak-Djamel Seriai, and Anas Shatnawi. Service identification based on quality metrics - object-oriented legacy system migration towards SOA. In Marek Reformat, editor, *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013*, pages 1–6. Knowledge Systems Institute Graduate School, 2014.

[3] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde-Lilia Bouziane, Christophe Dony, and Anas Shatnawi. Migrating large object-oriented applications into component-based ones: instantiation and inheritance transformation. In Christian Kästner and Aniruddha S. Gokhale, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015*, pages 55–64. ACM, 2015.

[4] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde-Lilia Bouziane, Christophe Dony, and Anas Shatnawi. Materializing architecture recovered from object-oriented source code in component-based languages. In Bedir Tekinerdogan, Uwe Zdun, and Muhammad Ali Babar, editors, *Software Architecture - 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings*, volume 9839 of *Lecture Notes in Computer Science*, pages 309–325, 2016.

[5] Mohammad Javad Amiri. Object-aware identification of microservices. In *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 253–256, 2018.

[6] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, and Theo Lynn. Microservices migration patterns. *Softw., Pract. Exper.*, 48(11):2019–2042, 2018.

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

[8] Gautier Bastide, Abdelhak Seriai, and Mourad Oussalah. Adapting software components by structure fragmentation. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1751–1758. ACM, 2006.

[9] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *ACM SIGSOFT Symposium on Software Reusability (SSR)*, pages 259–262, 1995.

[10] Sylvain Chardigny and Abdelhak Seriai. Software architecture recovery process based on object-oriented source code and documentation. In Muhammad Ali Babar and Ian Gorton, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings*, volume 6285 of *Lecture Notes in Computer Science*, pages 409–416. Springer, 2010.

[11] Sylvain Chardigny, Abdelhak Seriai, Mourad Oussalah, and Dalila Tamzalit. Search-based extraction of component-based architecture from object-oriented systems. In Ronald Morrison, Dharini Balasubramaniam, and Katrina E. Falkner, editors, *Software Architecture, Second European Conference, ECSA 2008, Paphos, Cyprus, September 29 - October 1, 2008, Proceedings*, volume 5292 of *Lecture Notes in Computer Science*, pages 322–325. Springer, 2008.

[12] Rui Chen, Shanshan Li, and Zheng Li. From monolith to microservices: A dataflow-driven approach. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, pages 466–475.

[13] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[14] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*, pages 185–200, 2016.

[15] Wuxia Jin, Ting Liu, Qinghua Zheng, Di Cui, and Yuanfang Cai. Functionality-oriented microservice extraction based on execution trace clustering. In *2018 IEEE International Conference on Web Services (ICWS 2018), San Francisco, CA, USA, July 2-7, 2018*, pages 211–218.

[16] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 2019.

[17] Stephen C Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

[18] Selim Kebir, Abdelhak-Djamel Seriai, Allaoua Chaoui, and Sylvain Chardigny. Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code. In Bipin C. Desai, Emil Vassev, and Sudhir P. Mudur, editors, *Fifth International C* Conference on Computer Science & Software Engineering, C3S2E '12, Montreal, QC, Canada, June 27-29, 2012*, pages 1–8. ACM, 2012.

[19] Gabor Kecskemeti, Attila Csaba Marosi, and Attila Kertész. The ENTICE approach to decompose monolithic services into microservices. In *International Conference on High Performance Computing & Simulation (HPCS 2016), Innsbruck, Austria, July 18-22, 2016*, pages 591–596, 2016.

[20] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio

Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *CoRR*, abs/1605.03175, 2016.

[21] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term, 2014. URL https://martinfowler.com/articles/microservices.html. Accessed: May 2019.

[22] Genc Mazlami, Jürgen Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In *IEEE International Conference on Web Services (ICWS 2017), Honolulu, HI, USA, June 25-30, 2017*, pages 524–531, 2017.

[23] Sam Newman. *Building microservices: Designing fine-grained systems, 1st Edition*. O'Reilly, 2015. ISBN 9781491950357.

[24] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde-Lilia Bouziane, Christophe Dony, and Rahina Oumarou Mahamane. Re-architecting OO software into microservices - A quality-centred approach. In Kyriakos Kritikos, Pierluigi Plebani, and Flavio De Paoli, editors, *Service-Oriented and Cloud Computing - 7th IFIP WG 2.14 European Conference, ESOCC 2018, Como, Italy, September 12-14, 2018, Proceedings*, volume 11116 of *Lecture Notes in Computer Science*, pages 65–73. Springer, 2018.

[25] Sourabh Sharma. *Mastering Microservices with Java*. Packt Publishing Limited, 2016. ISBN 978-1785285172.

[26] Sourabh Sharma, Rajech RV, and David Gonzalez. *Microservices: Building scalable software*. Packt Publishing, 2017. ISBN 1787280985.

[27] Rod Stephens. *Beginning software engineering*. Wrox, 2015. ISBN 978-1118969144.