Migration of Monoliths through the Synthesis of Microservices using Combinatorial Optimization

Gianluca Filippone
University of L'Aquila
gianluca.filippone@graduate.univaq.it

Marco Autili
University of L'Aquila
marco.autili@univaq.it

Fabrizio Rossi
University of L'Aquila
fabrizio.rossi@univaq.it

Massimo Tivoli
University of L'Aquila
massimo.tivoli@univaq.it

***Abstract*—Microservices are an emerging architectural style that is gaining a growing interest from companies and research. They are small, distributed, autonomous and loosely coupled services that are deployed independently and work together by communicating through lightweight protocols. Microservices are easy to update, scale, deploy, and reduce the time-to-market thanks to continuous delivery and DevOps. Several existing systems, in contrast, are difficult to maintain, evolve, and scale. For these reasons, microservices are the ideal candidates for the refactoring and modernization of long-lived monolithic systems. However, the migration process is a complex, time-consuming and error-prone task that needs the support of appropriate tools to assist software designers and programmers from the extraction of a proper architecture to the implementation of the novel microservices. This paper proposes a possible solution for the automated decomposition of a monolithic system into microservices, which exploits combinatorial optimization techniques to manage the decomposition. Our proposal covers the whole decomposition process, from the microservice architecture definition to the generation of the code of the microservices and their APIs, in order to assist developers and ensure by construction the correct behavior of the refactored system.**

***Index Terms*—microservices, system decomposition, microservices architecure, software synthesis**

I. Introduction

In the last years, Microservices are gaining a growing interest from companies and research [1], [2]. The spread of Cloud-native platforms and the need for modernizing legacy systems are leading to the adoption of the microservice architectural style for developing and deploying modern applications capable of scaling [3]. Microservices are small, well-defined, distributed, loosely coupled and autonomous services that are independently deployed and that work together by communicating through lightweight mechanisms. Each microservice runs its own process and implements a single business capability [4], [5].

Importantly, microservices are independently scalable and can be replaced and upgraded independently, with the objective to support scalability [6]. Instead, in monolithic systems all the interfaces, the business logic, and the databases are packaged together into a single executable artifact, e.g., deployed and running on a single server [5]. Due to these characteristics, monoliths are affected by a number of issues that are calling for a renovation process. In fact, they are difficult to maintain, evolve and scale: adding a new functionality or updating an existing one is often a complex task that can erode the modularity and the architecture of the system while causing the "blowing up" of its complexity [7].

Microservices are the ideal candidates to solve the aforementioned issues. In fact, tanks to their independent nature, the evolution of the system can be gained by simply adding new microservices or updating a small set of existing ones, thus without either affecting the whole system or degrading its architecture. Also the scalability of the system is increased, while continuous integration and continuous delivery are enabled, since microservices can be deployed independently and their updating does not require the reboot of the whole system [5], [8].

However, migrating an application from a monolith to a microservice architecture is a complex, time-consuming and error-prone task, which requires the use of tools which ease and drive the decomposition process [9], [10]. In the literature, many approaches are proposed to analyze a monolithic system, decompose its functionalities and extract the candidate microservices [11]–[13]. Specific metrics like *cohesion* and *coupling* are used to drive the decomposition process and evaluate the quality of the resulting architecture [11], [14]–[17]. Once extracted the new system architecture, developers are required to define and implement the APIs that allow the communication between microservices. Even if the new architectural style enables DevOps and continuous delivery, the definition and implementation of interfaces remains a time-consuming activity that needs tool support [3]. Moreover, the synchronization and the coordination among microservices must also be ensured in such a way that the behavior of the new system complies with the behavior of the legacy one [18].

This paper proposes an approach that aims to fill the gap between the identification of microservices and the next steps of the migration process, by covering all the phases of the migration (i.e., system analysis, architecture extraction, microservices implementation) in an automated way. The extraction of the microservice architecture is performed by using combinatorial optimization techniques that allow architects to obtain optimal cohesion and coupling, and minimal communication overhead, among the identified microservices. Given the resulting architecture, a synthesis algorithm automatically generates the code of the microservices by suitably moving methods and classes of the monolith into the correct microservice, further producing the implementation of the APIs that have to be exposed. The automated generation guarantees

978-1-6654-2603-9/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSREW53611.2021.00056

by construction that the resulting microservices are able to properly coordinate and to behave as the monolithic system.

The paper is structured as follows: Section II overviews the state of the art, Section III presents our approach, and Section IV discusses conclusions and future works.

II. Related Works

As already introduced, in the literature, several approaches have been proposed with the aim of addressing the monolithic system decomposition into microservices. As far as we know, approaches that, after finding an optimal decomposition, automatically produce an implementation or automatically define the microserivice APIs are lacking. As discussed in the following, the ones that are mostly related to our approach follow a series of well-defined steps that, starting from the legacy system, produce a representation of it and then extract the microservice architecture or a set of candidate microservices by applying decomposition rules or by using metrics or clustering algorithms.

In particular, in [19] an approach that leverages on a graph-based clustering algorithm is presented. The monolithic system is represented as a graph in which nodes represent classes and edges are weighted in order to determine their coupling. Then, a clustering algorithm is used to cut the graph into a set of microservice candidates. Clustering algorithms are exploited also in [20], in which a call graph is generated from both static and dynamic analysis, and microservices are identified through the algorithm. Gysel et al. [10] propose an approach that exploits 16 coupling criteria. The input of the process is a system representation that is suitably translated in a graph on which a clustering algorithm finds service cuts.

Jin et al. [21] monitor the system in order to extract execution traces, then an algorithm identifies the entities and search-based algorithm identifies the candidates in such a way that the grouped entities optimize their inter- and intra-connectivity. Assunção et. [22] implemented a feature-driven strategy that produces a graph representation of the legacy system through static analysis of the source code, then a search-based optimization algorithm outputs a set of redesigned architecture candidates that optimize five criteria. Their approach is similar to the one we propose in this paper. However, they divide the system according to business features that are defined by users and then used in the optimization process. In contrast, we do not require these definitions and we also consider domain entity classes. Moreover, our approach goes beyond by finding an optimal decomposition, further defining the microservice APIs and generating their implementation.

In [23] static analysis is performed in Java projects in order to recover the project architecture by removing architectural erosion. Then, candidate microservices are detected by performing the analysis of the dependency graph and by extracting the domain concerns of the project through topic detection considering the code as text.

III. Approach Overview

Fig. 1 overviews our approach. The decomposition is performed in three steps: *system analysis*, *architecture optimization*, and *system refactoring*. The first step analyzes the source code in order to produce a graph representation of the system. The optimization step takes the graph as input and outputs the microservice architecture. Then, the refactoring step produces the skeleton implementation of the microservices. In the following, we present the approach in detail.

A. System Analysis

The first step of the decomposition process requires as input the source code of the legacy system. The methods that implement the API controllers of the legacy system are marked as *entry points* for the system analysis. Moreover, the classes that represent the entities of the domain model are marked as *domain entities*. The analysis inspects the code at the methods level, starting from the defined entry points and following the execution trace. The process goes through all the method calls of the classes that are not marked as domain entities, thus stopping when there are calls to library methods or to methods that manipulate the state of domain objects [24]. The output of the analysis is a graph representation of the system. A node is created for each inspected method and for each domain entity class. A directed edge connects the nodes of the graph in order to describe: i) the method calls that occur from one method to another (*method-to-method edge*); ii) the reference of a method to an object of an entity class (*method-to-entity edge*); iii) the associations between two entities (*entity-to-entity edge*). Method-to-method and method-to-entity edges are weighted with the number of method calls or references to entity objects.

B. Architecture Optimization

This step takes as input the graph obtained in the previous step and outputs a candidate microservice architecture by solving a combinatorial optimization problem. The optimization problem finds a solution that groups the nodes of the graph (i.e., methods and entity classes of the legacy system) into a set $M = \{M_1, ..., M_n\}$ of microservices in such a way that cohesion is maximized, and coupling and communication overhead are minimized. In the following, we describe how these metrics have been defined in order to be used as objective function for our optimization problem.

Cohesion – Cohesion is defined as the ratio between the number of method calls across the same microservice and the number of all the method calls performed by a microservice. In other words, it is the ratio between the number of edges (i, j) whose endpoints belong to the same microservice $M_k \in M$ and the number of outgoing edges from all the nodes of M_k:

$$Cohesion(M_k) = \frac{\sum\limits_{(i,j)|i,j \in M_k} 1}{\sum\limits_{i \in M_k} |neighbours(i)|}, \qquad (1)$$

where the function $neighbours(i)$ identifies a set of nodes j s.t. the edge (i, j) exists.

Then, the objective function maximizes the value of the cohesion of the whole system, which is defined as the average of the cohesion value of each microservice:

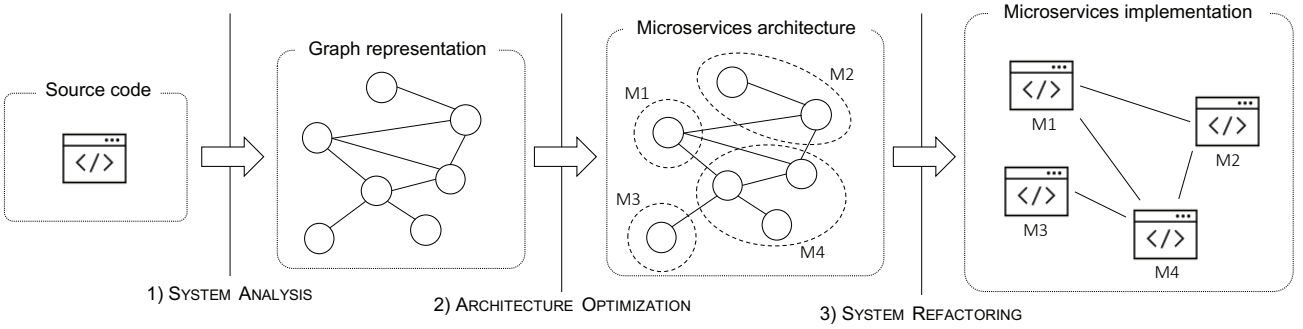

Fig. 1. Approach overview

$$Cohesion = \frac{\sum_{M_k \in M} Cohesion(M_k)}{|M|}. \quad (2)$$

Coupling – Coupling is defined as the number of method calls that are performed from a microservice to another microservice. In other words, it is the number of edges (i, j) whose endpoints belong to different microservices:

$$Coupling = \sum_{M_k \in M} \sum_{(i,j)|i \in M_k, j \notin M_k} 1. \quad (3)$$

Since a microservice architecture is better when microservices have low coupling, the objective function minimizes the value of coupling.

Communication overhead – Microservices communicate across the network through their APIs. With respect to the method calls that occur into the same microservice, API calls introduce a non-negligible overhead due to the network communication. Therefore, the objective function should also consider the overhead introduced by API calls. In order to estimate the overhead of a function call, we introduce a heuristic function $h(i)$ that computes the overhead for the call to a method $i \in M_k$ through an API (i.e., from a method $j \notin M_k$). The value for $h(i)$ is computed according to the number and the type of the attributes of each argument of the method i. Since the coupling function sums the method calls across different microservices (i.e., API calls), we can update the function (3) as follows:

$$Coupling = \sum_{M_k \in M} \sum_{(i,j)|i \in M_k, j \notin M_k} h(j)w(i, j), \quad (4)$$

where $w(i, j)$ is the weight of the edge (i, j) that represents the number of method calls from method i to method j.

C. System refactoring

The solution of the optimization problem produces a graph representation of the microservice architecture, in which methods and domain entity classes (i.e., nodes) are grouped into microservices. The refactoring step generates the implementation of the identified microservices by: i) adding new methods for exposing and consuming APIs (*API controller synthesis* and *API consumer synthesis*); ii) "moving" the code of each method into the right microservice; iii) placing the domain entity classes into all the microservices requiring them.

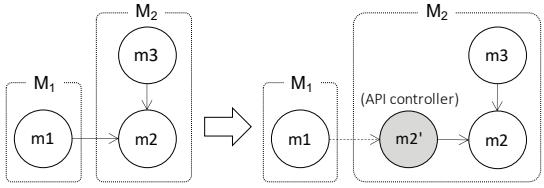

Fig. 2. Example of application of Rule 1

In the following, we describe the rules that allow the generation of API controllers and API consumer methods.

Rule 1: API Controller synthesis – An API controller method is generated for each method that is called from at least one different microservice. That is, in each microservice M_k, an API controller method i' is generated for each method $i \in M_k$ s.t. exists an edge $(j, i)|j \notin M_k$. The generated API controller will receive the API calls from outside the microservice and will invoke the method i. If needed, the methods in the microservice M_k will invoke the method i without using the API.

Fig. 2 shows an example of the application of this rule: the node representing the method $m2$ in the microservice M_2 has an ingoing edge from $m1$, which belongs to a different microservice (left hand side). The method $m2'$ is generated as an API controller for $m2$ (right hand side).

Rule 2: API Consumer synthesis – An API consumer method is generated to call methods that have been placed into different microservices. For this reason, in each microservice M_k, a method j' is generated for each method $i \in M_k$ s.t. exists an edge $(i, j)|j \notin M_k$. The new method contains the code needed to invoke the API related to the method j.

Fig. 3 shows an example of the application of this rule. The nodes representing the methods $m1$ and $m2$ in the microservice M_1 has an outgoing edge towards $m3$, which belongs to a different microservice (left hand side). The method $m3'$ is generated as an API consumer for $m3$ (right hand side).

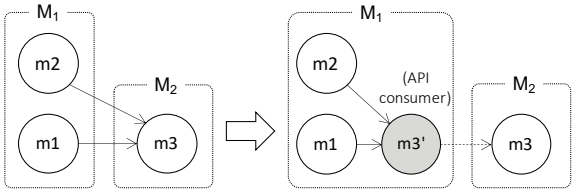

Fig. 3. Example of application of Rule 2

Alongside the API controllers and consumers, methods must have the access to all the required entity classes. Thus, all the entity classes that are referenced by a method are copied into the microservice that hosts the method, together with all the other entities connected to the first entity class. In other words, for each microservice M_k, an entity class e is put into the microservice M_k iff exists a method-to-entity or an entity-to-entity edge (i, e) s.t. $i \in M_k$.

Since the refactoring process considers the methods of the monolith as atomic entities, the behavior of the resulting system remains unchanged. In fact, the methods placed into the same microservice can interact as they do in the monolithic system; instead, API controllers and API consumer methods are generated to allow the correct interaction of the methods displaced in different microservices.

IV. Conclusions and Future Work

In this work, we proposed an approach for the automated migration of monolithic systems into microservice-based ones. Our approach focuses on the optimization of the cohesion, coupling and communication overhead by solving an optimization algorithm applied to a method-level graph representation of the monolith. With respect to the approaches analyzed at the state of the art, our approach is not limited to the extraction of the microservice architecture only. It goes beyond by i) defining the APIs that microservices have to expose in order to allow their interaction, and ii) generating the code that implements the API and that allow to consume them.

The approach can be refined by explicitly considering the persistence layer and applying a step of refactoring of the monolithic application in order to avoid that the degradation of the monolith affects the quality of the decomposed system. The refactoring process can be improved by also considering the different technologies and languages through which both the monolithic system and the microservices can be implemented.

Future works concern i) implementation and evaluation with real-world use cases, as well as, ii) validation and comparison with other approaches in the state of the art.

References

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

[2] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?," *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.

[3] D. Wolfart, W. K. G. Assunção, I. F. da Silva, D. C. P. Domingos, E. Schmeing, G. L. D. Villaca, and D. d. N. Paza, "Modernizing legacy systems with microservices: A roadmap," in *Evaluation and Assessment in Software Engineering*, EASE 2021, p. 149–159, ACM, 2021.

[4] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 1st ed., 2015.

[5] J. Lewis and M. Fowler, "Microservices a definition of this new architectural term." https://martinfowler.com/articles/microservices.html, 2014. Accessed on: July 15, 2021.

[6] M. Autili, A. Perucci, and L. D. Lauretis, "A hybrid approach to microservices load balancing," in *Microservices, Science and Engineering*, pp. 249–269, Springer, 2020.

[7] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[8] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*, pp. 195–216. Springer, 2017.

[9] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating towards microservices: Migration and architecture smells," in *Proceedings of the 2nd International Workshop on Refactoring*, p. 1–6, ACM, 2018.

[10] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing*, pp. 185–200, Springer, 2016.

[11] L. Carvalho, A. Garcia, W. K. G. Assunção, R. de Mello, and M. J. de Lima, "Analysis of the criteria adopted in industry to extract microservices," in *Proceedings of the Joint 7th International Workshop on Conducting Empirical Studies in Industry and 6th International Workshop on Software Engineering Research and Industrial Practice*, CESSER-IP '19, p. 22–29, IEEE Press, 2019.

[12] C. Schröer, F. Kruse, and J. Marx Gómez, "A qualitative literature review on microservices identification approaches," in *Service-Oriented Computing*, pp. 151–168, Springer, 2020.

[13] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pp. 128–141, Springer, 2019.

[14] L. Briand, S. Morasca, and V. Basili, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.

[15] M. Perepletchikov, C. Ryan, and Z. Tari, "The impact of service cohesion on the analyzability of service-oriented software," *IEEE Transactions on Services Computing*, vol. 3, no. 2, pp. 89–103, 2010.

[16] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, "Cohesion-driven decomposition of service interfaces without access to source code," *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 550–562, 2015.

[17] N. Santos and A. Rito Silva, "A complexity metric for microservices architecture migration," in *2020 IEEE International Conference on Software Architecture (ICSA)*, pp. 169–178, 2020.

[18] A. De Iasio and E. Zimeo, "A framework for microservices synchronization," *Software: Practice and Experience*, vol. 51, no. 1, pp. 25–45, 2021.

[19] 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, 2017.

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

[21] 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.

[22] W. K. G. Assunção, T. E. Colanzi, L. Carvalho, J. A. Pereira, A. Garcia, M. J. de Lima, and C. Lucena, "A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 377–387, 2021.

[23] I. Pigazzini, F. Arcelli Fontana, and A. Maggioni, "Tool support for the migration to microservice architecture: An industrial case study," in *Software Architecture*, pp. 247–263, Springer, 2019.

[24] M. Autili, P. D. Benedetto, and P. Inverardi, "A hybrid approach for resource-based comparison of adaptable Java applications," *Journal of Science of Computer Programming*, vol. 78, no. 8, pp. 987–1009, 2013.