Towards Microservice Smells Detection

Ilaria Pigazzini
University of Milano-Bicocca
Milano, Italy
i.pigazzini@campus.unimib.it

Francesca Arcelli Fontana
University of Milano-Bicocca
Milano, Italy
arcelli@disco.unimib.it

Valentina Lenarduzzi
Lappeenranta-Lahti University
Lappeenranta-Lahti
Finland
valentina.lenarduzzi@lut.fi

Davide Taibi
Tampere University
Tampere, Finland
davide.taibi@tuni.fi

ABSTRACT

With the adoption of microservices architectural styles, practitioners started noticing increasing pitfalls in managing and maintaining such architectures, with the risk of introducing architectural debt. Previous studies identified different *microservice smells* (also named *anti-patterns*) that harm microservices architectures. However, according to our knowledge, there are no tools that can automatically detect microservice smells, so their identification is left to the experience of the developer. In this paper, we extend an existing tool developed for the detection of architectural smells to explore microservices architecture through the detection of three microservice smells: Cyclic Dependencies, Hard-Coded Endpoints, and Shared Persistence. We detected the smells on five open-source projects implemented with microservices and manually validated the precision of the detection results. This work aims to open new perspectives on facing and studying architectural debt in the field of microservices architectures.

KEYWORDS

Microservices, Anti-patterns, Microservice bad smells detection

1 INTRODUCTION

Microservices are becoming more and more popular. Companies are currently migrating their systems to microservices for different reasons - for example because they expect to improve the quality of their system or to facilitate software maintenance [21][19]. However, the migration to microservices is not an easy task, mainly because of its novelty and because microservice migration patterns are not clear yet [20], [19].

Some of the recent works on migration from monolithic systems to microservices [9][21][12]) highlighted that the migration process generally increases Technical Debt. The major reason behind this could be the need to rewrite the vast majority of the code to be migrated. Moreover, the monitoring of the migration process, the large number of point-to-point connections between services, and the presence of business logic in the communication layer increase the dependency between services and consequently the TD [7].

In order to identify which factor can affect TD in microservices-based systems, a set of microservices-specific anti-patterns and "smells" have been identified [20], [23]. Bogner et al. [5] conducted a Systematic Literature Review on the subject and created a public catalog of anti-patterns/smells. Other practitioners and researchers have also proposed anti-patterns (or smells) [16][1][17], highlighting that they should be removed from the code since they could decrease software maintainability, increase bug-proneness, and generate different types of issues.

While various tools exist for monolithic systems that can detect code smells and architectural smells, to the best of our knowledge, there are no tools that support the identification of microservice smells, which means that developers need to manually check whether their systems comply with standards and do not contain smells. This is due to the usage of recent technologies for the implementation of this kind of architectures and the difficulties that must be faced when monitoring network communications among services at runtime [14].

In order to help practitioners with the detection of smells, in this paper we propose an extension of an existing tool, initially developed for the detection of architectural smells, by implementing an initial set of three microservice smells.

Hence, this paper provides the following contributions:

- A set of strategies for detecting three microservices smells (Cyclic Dependency, Hard-Coded Endpoints, Shared Persistence).
- Implementation of the three smells strategies in an existing tool to automatically detect them in microservices-based systems.
- Application of the newly implemented detection strategies on five open-source projects developed with microservices and manual validation of the detection results.
- Roadmap for the implementation of the next set of smells with new useful insights that emerged from this first evaluation.

Thanks to the new extension of our tool, developers will be able to spot the three microservice smells, thereby reducing the risk of TD due to Hard-Coded Endpoints and reducing coupling among services by removing Shared Persistence and Cyclic Dependencies smells.

Author Version. Please cite as:
Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi. 2020. Towards Microservice Smells Detection . In *International Conference on Technical Debt (TechDebt '20), October 8–9, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3387906.3388625

2 RELATED WORK

Different microservice patterns for solving common microservices-related problems have been proposed by researchers [22][15] and practitioners [17], [16], [18], [1]. Zimmerman et al. [24] proposed a joint collaboration between academia and industry to collect microservices patterns. However, all these works focus on patterns that companies should follow when implementing microservices-based systems rather than on anti-patterns and bad smells to avoid. Balalaie [4] conducted an industrial survey to understand how companies migrated to microservices, obtaining 15 migration patterns. However, they did not focus on bad practices or anti-patterns. Toledo et al. [7] conducted an industrial case study investigating architectural Technical Debt in microservices, highlighting a list of architectural issues, including architectural smells, especially on the microservice communication layer. Lenarduzzi et al. [10] also conducted a case study to investigate the impact of technical debt while migrating to microservices, highlighting that projects migrating to microservices are willing to increase the code TD, due to the large amount of refactoring activities involved in the migration process.

Bogner et al. [5] reviewed microservices smells and anti-patterns proposed in the literature, extracting 36 anti-patterns from 14 peer-reviewed publications. Their survey includes the vast majority of anti-patterns and smells highlighted also by practitioners. However, they did not report or classify their harmfulness.

Taibi et al. [21] conducted an industrial survey in order to investigate the migration processes adopted by companies. The results showed that practitioners are not aware of the patterns they should adopt and of the anti-patterns or smells to avoid. The authors then investigated microservice architectural patterns [22] and defined a set of 20 organizational and technical smells [20] that are specific to systems developed using a microservice architectural style, together with possible solutions adopted by 72 experienced developers to overcome these smells. Practitioners reported "Wrong Cuts" (wrong separation of concerns), "Hard-Coded Endpoints", "Cyclic Dependencies", and "Shared Persistence" as the most harmful smells during the development of microservices-based systems.

Soldani et al. [19] also conducted an industrial survey, classifying the existing gray literature and identifying a migration and re-architecting patterns catalog to facilitate migration to a cloud-native microservices-based architecture.

A comparison of the microservice smells proposed by practitioners and researchers can be found in [23] and [20], where the authors identified a set of microservice-specific smells. The researchers collected evidence of bad practices by interviewing 72 developers with experience in developing systems based on microservices. Then they classified the bad practices into a catalog of eleven microservice-specific smells frequently considered harmful by practitioners. Moreover, a taxonomy of 20 anti-patterns has been proposed through an industrial survey [23], including organizational (team-oriented and technology/tool-oriented) and technical (internal and communication) ones. The results of this work confirm the highly perceived harmfulness of "Wrong Cuts" (wrong separation of concerns), "Hard-Coded Endpoints", "Cyclic Dependencies", and and "Shared Persistence".

Unlike from previous works, in this paper we do not define new types of smells, but rather focus our attention on the definition of a detection strategy and implementation for three of the most common microservice smells cited above. With this aim, we provide an extension of an existing tool developed for the detection of architectural smells by exploiting different features provided by this tool (see Section 3).

3 MICROSERVICE SMELLS IDENTIFICATION

In this Section, we describe the detection strategies of three microservice smells: *Shared Persistence*, *Hard-Coded Endpoints*, and *Cyclic Dependency*. We selected these smells to be included in our Minimum Viable Product (MVP) [11] for two main reasons:

1) The three selected smells are easy to implement technically, and they can be detected through static code analysis. Other smells would require the collection of data at runtime (e.g., "Service Intimacy" [23]) or analyzing other information on the organizational side (e.g., "Microservice Greedy" [23] or "Too Many Standards" [23]). The detection of other smells is far more complex. As an example, "Wrong Cuts" [23] would require analyzing the different business processes, the architectural structure of the whole system, and further information. Several researchers are working on the identification of decomposition strategies and other methods to understand if the system was decomposed properly. However, the identification of detection strategies for wrong decomposition (or "Wrong Cuts") is beyond the scope of this paper.

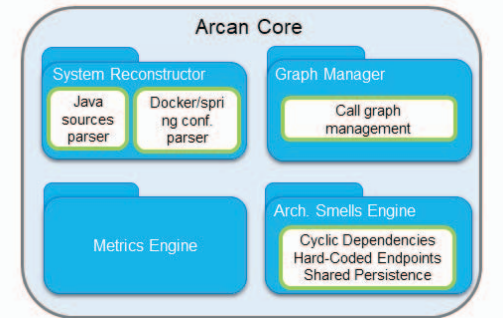
2) The goal of the detection of these smells is to validate the feasibility of our approach and to test the MVP. Therefore, we preferred selecting a limited set of smells and testing the product on Open Source projects.

We implemented the detection of the smells in a new version of an existing tool for the detection of architectural smells called Arcan [2]. As described in [8], Arcan consists of four components, which manage the different steps of the provided analysis: Figure 1 shows these components with the new additions. To extend this tool and make it suitable for the detection of microservice smells, we *1)* added new parsers to the component dedicated to architecture reconstruction (*System Reconstruction*) in order to scan Java source files and docker/Spring configuration files. Moreover *2)*, we developed three new detectors, one for each microservice smell, and added them to the component that collects all the architectural smell detectors (*Architectural Smells Engine*). Arcan relies on graph database technology: All the computations are based on the *dependency graph*, which is the representation of the project under analysis in the form of a directed graph. Currently, the tool allows storing the graph in a Neo4j[1] graph database, which also offers a browser for visualizing and querying the graph. We extended the dependency graph representation (*Graph Manager*) in order to include microservices and called it *call graph*: each node represents a microservice and each edge represents a microservice call.

Some of the proposed detection strategies are limited to specific programming languages, technologies, and frameworks. We chose a selection of them depending on two constraints: availability of

[1] https://neo4j.com

Figure 1: New Arcan core components for the detection of microservice smells



compliant open-source projects and ease of automation of the detection strategy. Next, we will describe each strategy by providing its *definition* and the description of the related *detection*.

3.1 Cyclic Dependency

Definition: A cyclic chain of calls among microservices. e.g., A calls B, B calls C, and C calls back A. Microservices involved in a cyclic dependency can be hard to maintain or reuse in isolation.

Detection: Arcan already automatically detects Cyclic Dependencies in monolithic Java applications by exploiting the dependency graph representation and graph algorithms. For the detection in microservices, we exploit the newly introduced *call graph*. Hence, in order to reuse the original Arcan detector, we implemented the identification of microservices dependencies. Microservices communication can be managed in different ways depending on the technologies, frameworks, and strategies that have been used; hence the call graph generation must be adapted depending on each specific case. In this paper, we focus our attention on Java projects exploiting the Spring framework[2] and/or the Docker platform. The nodes representing the different microservices are generated automatically from the information provided by the configuration files. In particular, the considered Docker files are `docker-compose.yml` and `Dockerfile`; the used Spring file is `application.yml`: they are all useful for service names identification. Concerning the microservices dependencies identification, in Spring applications a class named *RestTemplate* is used to handle http requests from a service to the others, hence we detect its usages to identify services communication. For the same reason, Arcan looks for the usage of *Feign*[3], which is a Java library for web services development compatible with Spring.

Dependency detection:

- **Input**: Java project folder
- **Exec**: Scan {docker-compose.yml, Dockerfile, application.yml} files to find microservices names
  - *FOR EACH* service_X, explore service source files and look for pattern matching of Feign annotations and *RestTemplate*.
    - *IF MATCH* dependency_pattern = "@FeignClient(name = "service_y")"
      - dependency = $service\_x \rightarrow service\_y$
    - *IF MATCH* dependency_pattern = "RestTemplate"
      - get service_y name from the methods of RestTemplate class.
      - dependency = $service\_x \rightarrow service\_y$
- **Output**: A file with the detected dependencies

Cyclic Dependency detection: Once the microservice dependencies are identified, the output results of the algorithm are processed by Arcan, which creates the dependency graph. Nodes represent microservices and edges represent their dependencies. Thanks to the graph representation, it is possible to detect Cyclic Dependency smells.

- For each microservice, create a node and add it to the call graph.
- For each microservice call, create an edge to model the dependency.
- Run the Depth First Search (DFS) algorithm: each detected cycle is an instance of a Cyclic Dependency smell.

3.2 Hard-Coded Endpoints

Definition: Hard-coded IP addresses and ports of the services between connected microservices. This smell leads to problems when the service locations need to be changed [20].
Detection:

- Scan the source code and look for pattern matching. The first part of the pattern (until the semicolumn) identifies IPv4 addresses and the second part matches ports.
  - pattern = \b\d1,3\d1,3\.\d1,3\.\d1,3:(6553[0-5]|655[0-2][0-9]\d|65[0-4] (\d)2|6[0-4](\d)3|[1-5](\d)4|[1-9](\d)0,3)\b

This strategy can be applied to every microservice implementation since it does not depend on any specific technology or framework. In this study, we experimented with the detection in Java projects (see Section 4).

3.3 Shared Persistence

Definition: Different microservices access the same database. In the worst case, different services access the same entities of the same database [20]. This smell highly couples the microservices connected to the same data, reducing team and service independence.
Detection:

- For each microservice, collect all database references/usages (database name or database url).
- If two or more different microservices refer to the same database, then those services are affected by the smell.

The detection of this smell is strictly linked to the technologies used. At the moment, we are focusing our attention on Java projects exploiting the Spring framework[4], which allows storing configuration information regarding databases in dedicated files (*YAML*[5] and *.properties* files) and we only detect accesses to the same databases (not entities) i.e. we look for the same connection string.

4 VALIDATION

In this Section, we report on the validation of the proposed detection strategies on open-source projects.

[2]https://spring.io/
[3]https://github.com/OpenFeign/feign
[4]https://spring.io/
[5]YAML is a human-friendly data serialization standard. https://yaml.org/

Table 1: Analyzed projects

Name	# Services	Link	Lang
Micro-company	4	github.com/idugalic/micro-company	J
Service Commerce	8	github.com/antonio94js/servicecommerce.git	JS
Sharebike	4	github.com/JoeCao/qbike.git	J
Sitewhere	19	github.com/sitewhere/sitewhere.git	J
Task Track Support	5	github.com/yun19830206/CloudShop-MicroService-Architecture.git	J

Legend: J = Java, JS = JavaScript

4.1 Design

We selected projects from the data set proposed by Marquez et al. [13]. Projects had to be developed with microservices. They had to be implemented using Java with Spring to validate the Shared Persistence detection and/or using Docker to build the call graph needed for Cyclic Dependency detection. There were no constraints for the detection of Hard-Coded Endpoints. We first ran Arcan on all projects coming from Marquez work (30 projects), detecting possible microservice smells. We identified 5 projects affected by at least one type of smell (Table 1).

Then we validated the results of the detection strategies by manually inspecting the detection of the smells in the five projects. The manual validation was performed separately by two of the authors; cases of disagreement were discussed. We obtained a precision value of 100% since all the found instances represent true instances of microservice smells. On the other hand, we were not able to compute the recall value due to the lack of detailed project documentation and project developers feedback. We aim to extend our validation by analyzing more open-source or industrial projects, possibly with the assistance of developers and information about known microservices defects.

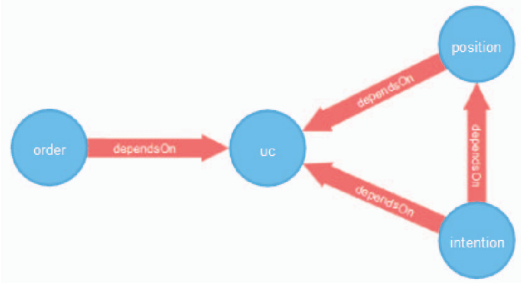
4.2 Validation Results

Out of the five selected projects, Arcan was able to detect 2 instances of the Shared Persistence smell and 6 instances of the Hard-Coded Endpoints smell. Regarding Cyclic Dependency, the projects under analysis were not affected by this smell. However, Arcan was able to create the dependency graph of the Java projects developed with Spring and using Feign, and store it in a Neo4j[6] graph database: Figure 2 shows the example of the *Sharebike* project. The image was captured by the Neo4j browser.

4.2.1 Shared Persistence results. We detected Shared Persistence smells in two projects: *Micro-company* and *Sharebike*. Table 2 shows the details of the detected smell instances. As for the Micro-company project, the data is shared by two different services that access the same Mongo database[7], named "blogpost". This information can be found in the configuration files of the two services, stored in the configuration service of the application. The Mongo database allows isolating data through the *collection* feature, where a collection is a set of stored document. A set of collections makes up a database, where collections can be accessed by specific services. However, this is not the case for Micro-company because all services access the same collection. In the case of the ShareBike project, the smell affects all the services. In particular, every configuration file (*application.yml*) contains the declaration of the same mysql database, whose name is "qbike". From the manual validation we obtained a precision of 100%, since all the found instances represent true instances of microservice smells. On the other hand, we were not able to compute the recall value due to the lack of detailed project documentation and expert support.

[6]https://neo4j.com/
[7]https://www.mongodb.com

Figure 2: Sharebike call graph



The call graph shows microservices as nodes and microservices dependencies as edges. The *Sharebike* project is a platform for renting and sharing electric vehicles. The **Order** server manages vehicle requests and trip information; the **UC** (User Center) manages user registration; **Position** handles vehicle discovery and user trip history; **Intention** manages the match between user and nearby vehicles.

Table 2: Shared Persistence results

Db Type	Affected Services	Shared Database Name
Micro-company		
mongoDB	query-side-blog	blogposts
	command-side-blog	
Sharebike		
mysql	<all>	qbike

4.2.2 Hard-Coded Endpoints. This smell was detected in the projects *Task Trak Support*, *Sitewhere*, and *Service Commerce*. Table 3 shows for every affected Java class the detected IPs and ports and the description of the smell instance. Concerning the *Task Track Support* project, the *DictRequestUrl* class contains a set of http requests such as http://192.168.1.108:9003/api/gateway/getOrderByUserIdNormal. The *CuratorFrameworkFactoryBean* Java class contains the address to the ZooKeper service[8], which is used for maintaining configuration information. The last two rows of the table report hard-coded http requests in two test classes. As regards Sitewhere, the tool detected one occurrence of the smell, in the class *EventSourceTests* with the value 0.0.0.0:1234, caused by the presence of the configuration of a test server. Considering the JavaScript project, *Service Commerce*, the tool detected one occurrence of the smell in a file containing an http URL. As in the case of Shared Persistence, all the smell instances found represent true instances of microservice smells, therefore the precision obtained is 100%. In this case, too, it was not possible to calculate recall due to the lack of detailed project documentation and expert support.

[8]https://zookeeper.apache.org/

Table 3: Hard-Coded Endpoints results

File name	Matched ip:port	Description
Task Track Support		
DictRequestUrl.java	192.168.1.108:9003	hard-coded multiple times in the list of all the http requests that can be made versus the *API gateway*
CuratorFrameworkFactoryBean.java	127.0.0.1:2181	zookeper address
CloudShopRequestConcurrentTest.java	192.168.1.108:9002	hard-coded in an http request to the *order* API
RequestTestMainApp.java	192.168.4.181:9002	hard-coded in an http request to the *order* API
Sitewhere		
EventSourceTests.java	0.0.0.0:1234	test server configuration
Service Commerce		
mercadopago.js	190.207.117.222:3000	hard-coded assignation of URLl to a service

4.3 Validation Discussion

The application of Arcan to the selected projects demonstrates that it is possible to automatically detect smells in projects, and therefore proves that we can propose our implementation to practitioners and researchers. The lack of access to industrial projects developed with microservices, and the lack of documentation on existing open-source projects did not allow us to compute the recall. We aim to extend our validation by analyzing more open-source or industrial projects, possibly with the support of developers and information about known microservices defects.

5 ROADMAP

In this work, we proposed detection strategies for three microservices smells and implemented them in the tool Arcan. As reported in Section 4, there is room for improvement regarding the detection accuracy of some rules. Moreover, as the goal of this work was to evaluate the feasibility of the detection of the smells in microservice-based systems, we did not implement all existing smells, but we focused only on a subset of three smells.

Our roadmap includes the following points:

- Validation of the MVP with companies. We are planning to involve companies to evaluate their interest in the detection of the smells and to prioritize the smells that will be developed in the next versions of Arcan.
- Validation of the MVP with researchers. We will involve researchers to identify better strategies to detect the smells, or to include other types of smells.
- Detection of a larger set of smells, based on the feedbacks received from the users.
- Detection of additional frameworks and technologies to extend the tool support. The approach adopted to identify the three smells is open to extension. Each strategy requires a set of information, whose source depends on the different technologies. Once we are able to extract such information from a new techonology, we are able to detect the existing smells. For example, if we discover how to retrieve service names from Kubernates[9], then we can build the call graph in the same way we do for Docker.
- Detection of microservice smells using tracing information collected at runtime. This will allow more accurate detection of a set of smells and will abstract from the programming language. As an example, using Open Tracing[10] will make it possible to generate a call graph between microservices, independent of their language.
- Extensive validation of the microservice smells on larger projects, including industrial closed-source projects.
- Identification and implementation of metrics for coupling, cohesion, and other relevant metrics for microservices [6]. This approach will allow us to detect other types of smells such as “High Cohesion” or “Inappropriate Intimacy”, but also to give information to developers on the actual level of coupling and cohesion.

6 CONCLUSION AND FUTURE WORKS

In this paper, we introduced the detection of three microservices smells described in the previously proposed catalog [20], outlining for each smell the detection strategy and the output data. We integrated the detection of these smells into the Arcan tool developed for the detection of architectural smells. Moreover, we showed some preliminary results obtained with Arcan for the analysis of 5 open-source projects with a microservices nature and reported a total of 8 instances of microservices smells.

Practitioners will benefit from our work by being able to keep the Technical Debt related to the three bad smells detected by the tool under control. Researchers can use the data provided by Arcan to further validate the importance of these smells.

This work is subject to some threats to validity. First of all, the validity of our approach for microservice smell detection has been proven only on a small number of open-source projects, but with high precision of the results. All the proposed strategies except for Hard-Coded Endpoints rely on the presence of the Spring framework and the usage of the Docker platform, since we exploit their configuration files to retrieve information for detecting the microservice smells.

In the future, we aim to extend the tool to automatically detect other microservice smells and improve the implementation of the detection strategies to generalize our current approach for different kinds of microservices architectures, different programming languages and technologies. Finally, we plan to validate our detection results on larger projects, also in industrial settings. Depending on the industrial experimentation, we may be able to identify new microservice smells not yet included in the catalog, also considering architectural smells for monolithic systems that can be applied to microservices [3] and investigating approaches for evaluating microservice decomposition[22] with the aim of detecting “wrong cuts”. Future work will also include the detection of different smells considering data collected at runtime.

REFERENCES

[1] V. Alagarasan. 2016. Microservices Antipatterns. In *Microservices-Summit, NY.*
[2] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D.A. Tamburri, M. Zanoni, and E. Di Nitto. 2017. Arcan: A Tool for Architectural Smells Detection. In *ICSA Workshops.*
[3] U. Azadi, F. Arcelli Fontana, and D. Taibi. 2019. Architectural Smells Detected by Tools: a Catalogue Proposal. In *International Conference on Technical Debt.*

[9] another container platform similar to Docker: https://kubernetes.io/

[10] Opentracing https://opentracing.io

[4] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. Tamburri, and T. Lynn. 2018. Microservices migration patterns. *Software: Practice and Experience* 48 (2018).

[5] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann. 2019. Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells. In *Int. Conf. on Software Architecture Companion (ICSA-C)*. 95–101.

[6] J. Bogner, S. Wagner, and A. Zimmermann. 2017. Automatically Measuring the Maintainability of Service- and Microservice-Based Systems: A Literature Review. In *Int. Conf. on Software Process and Product Measurement*. 107–115.

[7] S. de Toledo, A. Martini, A. Przybyszewska, and D. Sjøberg. 2019. Architectural Technical Debt in Microservices: A Case Study in a Large Company. In *International Conference on Technical Debt (TechDebt)*. 78–87.

[8] F. A. Fontana, I. Pigazzini, R. Roveda, and M. Zanoni. 2016. Automatic Detection of Instability Architectural Smells. In *International Conference on Software Maintenance and Evolution (ICSME)*. 433–437.

[9] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner. 2019. From Monolith to Microservices: A Classification of Refactoring Approaches. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer International Publishing, 128–141.

[10] V. Lenarduzzi, F. Lomio, N Saairimäki, and D. Taibi. 2020. Does Migrate a Monolithic System to Microservices Decreases the Technical Debt? *Journal of Systems and Software* (2020). to appear.

[11] V. Lenarduzzi and D. Taibi. 2016. MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 112–119.

[12] N. Lu, G. Glatz, and D. Peuser. 2019. Moving mountains – practical approaches for moving monolithic applications to Microservices. In *International Conference on Microservices (Microservices 2019)*.

[13] G. Marquez and H. Astudillo. 2018. Actual Use of Architectural Patterns in Microservices-based Open Source Projects. In *Asia-Pacific Software Engineering Conference (APSEC 2018)*.

[14] B. Mayer and R. Weinreich. 2018. An Approach to Extract the Architecture of Microservice-Based Software Systems. In *(SOSE)*. 21–30.

[15] Davide Neri, Jacopo Soldani, Olaf Zimmermann, and Antonio Brogi. 2019. Design principles, architectural smells and refactorings for microservices: a multivocal review. *SICS Software-Intensive Cyber-Physical Systems* (2019). https://doi.org/10.1007/s00450-019-00407-8

[16] M. Richards. 2016. Microservices AntiPatterns and Pitfalls. *O Reilly eBooks* (2016).

[17] C. Richardson. 2018. *Microservices Patterns: With Examples in Java*. Manning Publications Company. https://books.google.it/books?id=UeK1swEACAAJ

[18] R. Shoup. 2015. From the Monolith to Microservices: Lessons from Google and eBay. In *Craft-Con*.

[19] J. Soldani, D.A. Tamburri, and W-J Van Den Heuvel. 2018. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software* (2018), 215 – 232.

[20] D. Taibi and V. Lenarduzzi. 2018. On the Definition of Microservice Bad Smells. *IEEE Software* 35, 3 (2018), 56–62.

[21] D. Taibi, V. Lenarduzzi, and C. Pahl. 2017. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing* 4, 5 (2017), 22–32.

[22] D. Taibi, V. Lenarduzzi, and C. Pahl. 2018. Architectural Patterns for Microservices: a Systematic Mapping Study. *Int. Conf. on Cloud Computing and Services Science (CLOSER2018)* (2018).

[23] D. Taibi, V. Lenarduzzi, and C. Pahl. 2019. Microservices Anti-patterns: A Taxonomy. In *Microservices*. Springer International Publishing, 111–128. https://doi.org/10.1007/978-3-030-31646-4_5

[24] O. Zimmermann, M. Stocker, Uwe Zdun, D. Lübke, and C. Pautasso. 2018. Microservice API Patterns. https://microservice-api-patterns.org