Towards a Tool-based Approach for Microservice Antipatterns Identification

Rafik Tighilt
Université du Québéc à Montréal
Montréal, Québec, Canada
Email: tighilt.rafik@gmail.com

Manel Abdellatif
Polytechnique Montréal
Montréal, Québec, Canada
Email: manel.abdellatif@polymtl.ca

Naouel Moha
Université du Québéc à Montréal
Montréal, Québec, Canada
Email: moha.naouel@uqam.ca

Yann-Gaël Guéhéneuc
Concordia University
Montréal, Québec, Canada
Email: yann-gael.gueheneuc@concordia.ca

***Abstract*—Microservice architecture has become popular in the last few years because it allows the development of independent, reusable, and fine-grained services. However, a lack of understanding of its core concepts and the absence of reference or consensual definitions of its related concepts may lead to poorly designed solutions called antipatterns. The presence of microservice antipatterns may hinder the future maintenance and evolution of microservice-based systems. Assessing the quality of design of such systems through the detection of microservice antipatterns may ease their maintenance and evolution. Several research works studied patterns and antipatterns in the context of microservice-based systems. However, the automatic identification of these patterns and antipatterns is still at its infancy. We searched for re-engineering tools used to identify antipatterns in microservice-based systems in both academia and industry. The results of our search showed that there is no fully-automated identification approach in the literature. In this paper, we aim to reduce this gap by (1) introducing generic, comprehensive, and consensual definitions of antipatterns in microservice-based systems, and (2) presenting our approach to automatically identify these antipatterns. Currently, this work is still in progress and this paper aims to present the approach and the metamodel used for future implementation.**

Keywords–Microservices; Antipatterns; Identification.

I. Introduction

A microservice is defined as a small service, with a single responsibility, running on its own process, and communicating through lightweight mechanisms [1], such as representational state transfer application programming interfaces (REST APIs) and message brokers. Each microservice in a microservice-based system fulfills a single business function, manages its own data, runs on its own process, is managed by a single team, and is not tied to the system itself for its evolution or deployment. Microservices are built around business requirements, deployed by a fully automated deployment machinery with a minimum centralized management [2] and are loosely coupled.

Several major actors of the software industry have adopted microservice-based systems, such as Netflix and Amazon. The popularity of this architecture still grows, mainly due to its dynamic and distributed nature, which offers greater agility and operational efficiency and reduces the complexity of handling applications scalability and deployment cycles wrt. monolithic applications [2]. Software maintenance is one of the most important fields in the software industry, whether in expenses or in resources [3].

However, like any other architectural style, microservice-based systems also face challenges with maintainability and evolution due to "poor" solutions to recurring design and implementation problems, called antipatterns [4]. These antipatterns can degrade the overall quality of design and quality of service of the microservices themselves and the system as a whole [5].

The nature of microservice systems makes them very dynamic (multi-language, multi-operating environments, etc.) [2]. This makes the identification of antipatterns difficult, especially because there is a lack of automated approaches in the literature to help fulfil this task.

We contribute to the maintenance and evolution of microservice-based systems with generic, comprehensive, and consensual definitions of antipatterns in microservice-based systems and an automatic tool-based approach for the identification of antipatterns in these systems. Our automatic tool-based approach relies on a meta-model we established and described in this paper. The meta-model covers the needed information to apply our heuristics and identification rules yet can be extended for future work.

However, we must overcome some challenges introduced by microservice-based systems.

1) **Microservices are, by definition, independent [2].** Microservice-based systems are deployed on multiple providers using different tools and configurations.
2) **Microservices can be built using different programming languages [1]**. This makes the identification process more challenging compared to single-language systems.

Thus, for the first step of our work and to validate our approach, we only consider systems built with the Java programming language and using Docker as container technology as they are among the most popular tools to build microservice-based systems.

The remainder of this paper is structured as follows. Section II describes previous work related to microservices antipatterns cataloguing and identification. Section III outlines our methodology for antipatterns identification. It also introduces our catalogue of microservice antipatterns and the metrics and hints to identify these antipatterns. Section IV presents some limitations that we identified in our approach. Finally, Section V concludes this paper and presents the future work.

II. Related Work

In their study, Pahl and Jamdi [6] aim to identify, taxonomically classify and systematically compare the existing research body on microservices and their application in the cloud. They conducted a systematic mapping study of 21 works on microservice design published between 2014 and 2016. They defined a characterization framework and used it to study and classify the works. Their study reports a lack of research tools supporting microservice-based systems and conclude that microservice research is still in a formative stage. The study results in a discussion of the microservice architectural style concerns, positioning it within a continuous development context and moving it closer to cloud and container technology.

In his overview and vision paper, Zimmerman [7] reviews popular introductions to microservices to identify microservices tenets. It then compares two microservices definitions and contrasts them with SOA principles and patterns. this paper compiles practitioner questions and derives research topics from the differences between SOA and microservices architectural style. The author conclude his paper with the opinion that microservices are one special implementation of the SOA paradigm.

Garriga [8] defines a preliminary analysis framework in the form of a taxonomy of concepts including the whole microservices lifecycle, as well as organizational aspects. The author claims that this framework is necessary to enable effective exploration, understanding, assessing, comparing, and selecting microservice-based models, languages, techniques, platforms, and tools. He then analyzed state of the art approaches related to microservices using this taxonomy to provide a holistic perspective of available solutions. Additionnaly, the paper identified open challenges for future research from the results of litterature analysis.

Soldani et al. [9] identified and compared benefits and limitations of microservices by studying the industrial *grey* literature. They also studied the design and development practices of microservices to bridge academia and industry in terms of research focus. Marquez and Astudillo [10] provided (1) a catalog of microservice architectural patterns from academia and industry, (2) a correlation between quality attributes and these patterns, (3) a list of technologies used to build microservice-based systems with these patterns and (4) a comparative analysis of SOA and microservice architectural patterns. They did that to determine whether architectural patterns are used in the development of microservice-based systems. This work extended their previous work with Osses [11].

Taibi et al. [12] introduced a catalog and taxonomy of the most common microservices anti-patterns to identify common problems resulting from the migration of monolithic applications to microservice-based systems. Their catalog is based on the experience of 27 interviewed practitioners. The authors identified a taxonomny of 20 anti-patterns including organizational and technical anti-patterns and estimated their level of harmfulness through a survey. They conclude that splitting a monolith is the most critical issue.

Borges and Khan [13] selected 5 well known anti-patterns in microservice-based systems and proposed an algorithm to automatically detect them. The authors claim that their solution can avoid common mistakes when deploying microservices-based projects and can help project managers to get an overview of the system as a whole. They tested their algorithm on a well known open source microservice-based project and revealed possible improvements.

Microservice antipatterns have been discussed in the literature, but very little work has been done in the field of their automatic identification. To the best of our knowledge, only Borges and Khan [13] proposed an algorithm to automatically identify antipatterns in microservice-based systems. However, they only identify 5 antipatterns. Our approach do not focus on the same antipatterns eventhough we may share some.

III. Study Design

This section presents the design of our study. First, we explain the approach we used to construct our research. Then, we detail the metamodel used to automatically identify antipatterns in microservice-based systems. Finally, we list the detection rules for each antipattern.

A. Approach

This section presents our approach to fulfill our objectives. First, we reviewed the literature and studied 67 open-source projects to build a catalogue of microservice antipatterns. Second, we study each antipattern to provide a concise description and extract hints of its presence in microservice-based systems using source-code, configuration files, deployment files and git repositories. The catalogue and the description of the microservice antipatterns have been presented in our previous work [14]. Finally, we build an automated tool-based approach for the identification of microservice antipatterns.

1) Step 1: Catalogue of Microservice Antipatterns:

a) Literature review: To build our catalog, we reviewed the literature following the procedures proposed by Kitchenham et al. [15] for performing systematic literature reviews. We excluded papers not written in English and papers not related to microservices antipatterns. We obtained a total of 27 papers describing microservice antipatterns.

We grouped antipatterns having similar definitions under a single name and excluded antipatterns that are only related to the organizational structure of the company or too specific and that cannot be generalized (e.g., the Frankenstein antipattern that is related to switching from waterfall to agile development).

b) Open-source systems review: After reviewing the literature, we manually analyzed 67 open source systems [16] to assess the concrete presence of the identified antipatterns in these microservice-based systems. Table I shows examples of identified antipatterns inside microservice-based systems.

After reviewing the litterature and the open source microservice-based systems, we obtained a total of 16 antipatterns described below.

1) **Wrong Cuts:** This antipattern consists of microservices organized around technical layers (Business layer, Presentation layer, Data layer) instead of functional capabilities, which causes strong coupling of the microservices and impedes the delivery of new business functions.
2) **Cyclic Dependencies:** This antipattern occurs when multiple services are co-dependent circularly and, thus, no longer independent, which goes against the very definition of microservices.

TABLE I. EXAMPLES OF IDENTIFIED ANTIPATTERNS IN MICROSERVICE-BASED SYSTEMS

System name	Identified antipatterns
ACME Air	Manual Configuration, Shared Persistence, Hardcoded Endpoints, No Healthcheck
Cinema microservice	Manual Configuration, Hardcoded Endpoints, No Healthcheck
Delivery system	Hardcoded endpoints, Local logging, Insufficient monitoring, No Healthcheck
E-commerce microservices sample	Manual Configuration, Hardcoded Endpoints, No API gateway, Local Logging
Microservices demo	Hardcoded Endpoints, No API Gateway, No API versioning
Beer Catalog	Hardcoded Endpoints, Shared Libraries, Multiple Service Instances Per Host
Springboot microservices example	Manual Configuration, Hardcoded Endpoints, No Healthcheck

3) **Mega Service:** This antipattern appears when a microservice serves multiple business functions. A microservice should be manageable by a single team and bounded to a single business function.
4) **Nano Service:** This antipattern results from a too fine-grained decomposition of a system in which multiple microservices work together to fulfill a single business function.
5) **Shared Libraries:** This antipattern consists of libraries and files (ex. binaries) used by multiple microservices, which breaks the microservices independence as they rely on a single source to fulfill their business function.
6) **Hardcoded endpoints:** This antipattern relates to URLs, IP addresses, ports and other endpoints being hardcoded in the microservice source code including configuration files. This may interfere with the load balancing and the deployment of the microservices.
7) **Manual Configuration:** This antipattern happens with configurations that must be manually pushed to each microservice of a system. Microservice systems evolve rapidly and their management should be automated, including their configuration.
8) **No Continuous Integration (CI) / Continuous Delivery (CD):** Continuous integration and delivery are important for microservices to automate repetitive steps during testing and deployment. Not using CI/CD undermines microservices, which encourages automation wherever possible.
9) **No API Gateway:** This antipattern occurs when consumer applications (front ends, mobile applications, etc.) communicate directly with microservices. Each application must know how the whole system is decomposed and must then manage endpoints and URLs for each microservice.
10) **Timeouts:** This antipattern happens when timeout values are set and hardcoded in HTTP requests, which leads to spurious timeouts or unnecessary delays.
11) **Multiple Service Instances Per Host:** This antipattern happens when multiple microservices are deployed on a single host, which prevents their independent scaling and may cause technological conflicts inside the host.
12) **Shared Persistence:** This antipattern happens when multiple microservices share a single database: they no longer own their data and cannot use the most suitable database technology for it.
13) **No API Versioning:** This antipattern happens when no information is available about a microservice version, which can break changes and force backward compatibility when deploying updates.
14) **No Health Check:** This antipattern occurs when microservices are not periodically health checked. Unavailable microservices may not be noticed and cause timeouts and errors.
15) **Local Logging:** This antipattern occurs when microservices have their own logging mechanism, which prevents the aggregation and analyses of their logs and may slow down the monitoring and recovery of a system.
16) **Insufficient Monitoring:** This antipattern relates to microservice systems performances/failures, which are not tracked and cannot help maintain the functions of the systems.

2) Step 2: Detection of the Microservice Antipatterns: We present an approach to detect the antipatterns catalogued in Section III-A1. Figure 1 shows that our approach takes as input a microservice-based project or a list of microservices (both either as Git repositories or local source code folders). Then, from each microservice, it extracts the relevant files by excluding binaries (e.g., .jar, .exe, .bin files) and vendor files (e.g., node_modules, composer vendor etc.). Then, our approach splits the extracted files into four categories based on their extension, content, and programming language:

1) **Code:** These are source-code files of the microservice. We split these files into programming files (Java, PHP, Go, etc.), configuration files (XML, JSON, YAML, etc.), markup files (HTML, CSS, EJS, etc.), and data files (CSV, GitAttributes, Properties, etc.).
2) **Environment:** If available, these files store environment variables for the microservices. Usually in key value pairs.
3) **Deployment:** These are deployment scripts for the microservices (Dockerfiles, docker-compose, etc.). We do not consider configuration files in this category. We only save files directly related to deployment (Docker files, Docker-compose, etc.).
4) **Configuration:** These are configuration files for the microservice (JSON files, XML files, etc.). Source code files that only contain configuration are also added to this category. That means that a source code file "xxx-config.java" will be considered in both the source code category and the configuration category.

From each category, we can extract some information to build

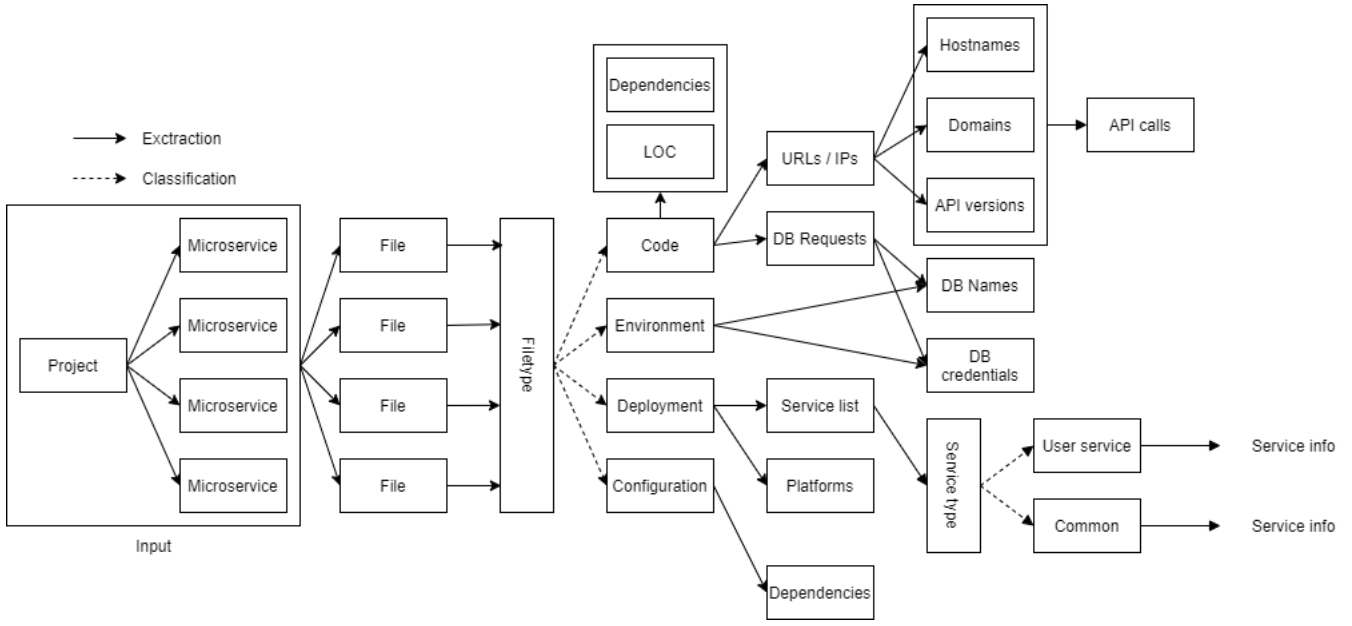

Figure 1. Antipatterns Identification Process Architecture

our model of the microservices, which contains information needed to apply our detection heuristics to identify antipatterns.

B. Metamodel Definition

This section describes the meta-model that we use to encapsulate the needed information to identify antipatterns. The metamodel is divided in 13 components, each one containing some information related to the systems and the microservices. The components are as follow:

1) System: This component holds information about the system itself.

- **isGitRepository**: True if the provided system to analyze is a git repository.
- **importedAt**: The timestamp when the analysis was performed.

2) GitRepository: If the provided system is in a Git repository, this component stores information about it.

- **url**: Repository URL.
- **owner**: The owner of the Git repository.
- **nbContributors**: The number of different developers that contributed to the repository.
- **nbCommits**: The number of commits to the repository.
- **branch**: Current branch of the repository.
- **buildStatus**: If available, the build status of the repository.

3) Microservice: This component stores information about a single microservice.

- **languages**: List of programming languages used in a microservice.
- **loc**: The number of lines of code for a microservice.

4) Dependency: This component holds information about a single dependency.

- **name**: Dependency name.
- **source**: The source from where the dependency is installed.
- **category**: The dependency category (e.g., ORM, Logging, Monitoring, etc.).
- **type**: The type of the dependency (binary, framework, library, etc.).

5) Deploy: This component contains generic deployment information.

- **area**: If available, the area of the deployment (development, staging, production, etc.).
- **instructions**: List of deployment instructions (e.g., Dockerfile commands).

6) Config: This component contains configuration information.

- **type**: The configuration file type.
- **path**: The path of the configuration file.
- **values**: Actual configuration key/value pairs.

7) Env: If available, this component contains environment variables information.

- **type**: The environment file type.
- **values**: Actual environment variables key/value pairs.

8) Code: This component holds information about a given source code file.

- **languages**: List of programming languages of the current file.
- **mainLanguage**: Main programming language used in the file.
- **loc**: Lines of code of the file.

9) Image: If available, this component holds information about container images of the system.

- **name**: Image name.
- **type**: If available, the image type (e.g., database system, monitoring,etc.).

10)Server: This component is related to deployment server information.

- **address**: Server address.
- **port**: Deployment port number.

11)HTTP: This component stores information about HTTP requests.

- **sourceFile**: File from where the HTTP request was performed.
- **endpointURL**: HTTP requests destination endpoint.
- **port**: HTTP requests port.
- **type**: The type of the HTTP request.
- **parameters**: HTTP requests parameters.

12) Database: This component stores information about database queries.

- **dbQuery**: The database query string.
- **queryType**: The type of the database query.
- **dbName**: Database name.
- **dbLocation**: Database location address.
- **dbUsername**: If available, the database user name.
- **dbPassword**: If available, the database user password.

13) Import: This component stores information about imported packages in the source code.

- **package**: The imported package.
- **path**: The imported path.
- **fileType**: Imported file type.

Figure 2 illustrates the relations between each component of our metamodel.

C. Detection rules

We now describe the detection rules of our approach for each antipattern.

1) Wrong Cuts: Microservices have one file type in the source code and connect to multiple microservices having also one file type. An example would be a microservice containing only presentation related code connecting to a microservice containing only business logic code. We rely on the files extensions, contents, and programming languages to identify this antipattern.

2) Cyclic dependencies: Microservices performing API call to other microservices circularly. We detect this antipattern using the API calls, endpoints, and dependencies extracted in our model.

3) Mega Service: Such a microservice has more lines of code, connects to multiple databases, has a high fan in and fan out, and has a lot of dependencies compared to other microservices.

4) Nano Service: Such a microservice has less lines of code, connects to zero or one database, has a low fan in and fan out, and has no or a few dependencies compared to other microservices.

5) Shared Libraries: Multiple microservice source files, dependency binaries, and libraries are shared between multiple microservices.

6) Hardcoded Endpoints: REST API calls inside microservices source code, deployment files, configuration files, or environment files contain hard-coded IP addresses, port numbers, and URLs. There is no service discovery present in the system.

7) Manual Configuration: Microservices have their own configuration files. No configuration management tools are present in the dependencies of the system and no microservice is responsible of configuration management.

8) No CI/CD: Configuration files and version control repositories do not contain continuous integration/delivery-related information. We rely on an extensible list of CI/CD tools to perform our analysis.

9) No API Gateway: Microservice source code does not contain signatures of common API gateway implementations (e.g., Netflix Zuul). No frameworks or related tools are present in the dependencies of the microservice. API calls are direct calls to microservices.

10) Timeouts: Timeout values are present in REST API calls. No signatures of common circuit breaker implementations (e.g., Hystrix) are present in the source code. No circuit breaker is present in the dependencies of the microservice.

11) Multiple Service Instances Per Host: We analyze and compare deployment scripts of all microservices to find the ones that share the same hosts.

12) Shared persistence: We extract the databases used by the microservices and then assess if any database is used by more than one microservice.

13)No API Versioning: Endpoints and URLs do not contain version numbers. No version information present in the headers when performing HTTP requests.

14) No health check: No "healthcheck" or "health" endpoint in microservices. No common implementation of health checks present in the source code (e.g., Springboot actuator).

15) Local Logging: No distributed logging present in the dependencies. No common logging microservice. Each microservice has its own log file paths.

16) Insufficient Monitoring: No monitoring framework or library in the microservices dependencies (e.g., Prometheus).

IV. Approach Limitations

In this section, we discuss the limitations of our approach and the measures that we took to reduce them.

A. Internal limitations

Although we intensively reviewed the literature to find the most common antipatterns in microservice-based systems, they are potentially other antipatterns that we did not include in our study. Yet, with the antipatterns described in our catalogue, we aim to establish a foundation for future work. Other researchers should perform similar reviews to confirm/infirm ours.

The detection rules we established to identify antipatterns are subject to our interpretation of antipatterns. Mega service for example is subjective, and can be discussed. However, we tried to minimize this limitation by considering every microservice as a part of the system instead of a stand-alone application. This way, we can say that a Mega service is *relative* to the system.

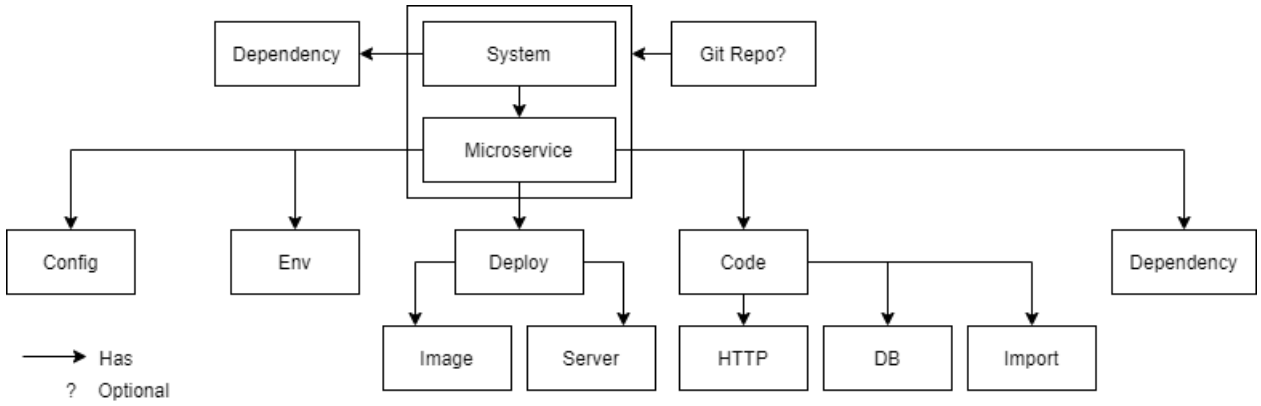

Figure 2. Relations between all metamodel components

B. External limitations

Microservice-based systems are volatile. They can be built using multiple technologies and deployed to multiple providers. Even though we tried to identify the most common technologies in the field of microservices, we might have omitted some. We will minimize this limitation by building a tool that can be extended by providing more parsers and deployment environments.

We rely on lists of dependencies and frameworks to identify some antipatterns. We pre-define these lists by taking the most widely used technologies in that area, but we do not pretend to have exhaustive lists. However, we will build the system in a way these lists can be extended easily to cover more tools and frameworks.

Even though configuration files are widely written in JSON, XML, or YAML file formats, they can also be written in the programming language itself. This may lead us to misconsider a file and not include it in the configuration category. We reduce this limitation by not only relying on the file extension, but also on the file name and its content to do the classification.

V. Conclusion and Future Work

We describe in this paper our automated approach for the identification of antipatterns in microservice-based systems. We provide a list of previously identified antipatterns from the literature. We detail the meta-model we use in our approach and we finally define the heuristics and detection rules for each of the identified antipatterns.

We believe that our approach is robust enough to identify the described antipatterns yet still extensible and flexible to evolve with the evolution of programming languages and antipatterns themselves.

Future work includes first implementing our detection rules to identify antipatterns in Java microservices to validate and refine our approach. We will validate our approach by manually analysing the microservice-based systems and calculate precision and recall for each of the identified antipatterns. Then, we want to extend our approach to consider multi-language microservice-based systems. Finally, we aim to empirically study the effect of these antipatterns on the quality of systems.

References

[1] "Microservices: a definition of this new architectural term," 2019, URL: https://martinfowler.com/articles/microservices.html [retrieved: August, 2020].

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., Feb. 2015, ISBN: 978-1491950357.

[3] M. Hanna, "Maintenance burden begging for remedy," *Software Magazine*, vol. 13, pp. 53–53, Apr. 1993.

[4] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, pp. 56–62, May 2018.

[5] F. Palma, "Detection of SOA Antipatterns," in *Service-Oriented Computing - ICSOC 2012 Workshops*. Springer Berlin Heidelberg, Jan. 2013, pp. 412–418.

[6] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and and Technology Publications, Apr. 2016, pp. 137–146.

[7] O. Zimmermann, "Microservices tenets: : Agile approach to service development and deployment," *Computer Science - Research and Development*, vol. 21, pp. 301–310, Nov. 2016.

[8] M. Garriga, "Towards a Taxonomy of Microservices Architectures," in *Software Engineering and Formal Methods*. Springer International Publishing, Feb. 2018, pp. 203–218.

[9] J. Soldani, D. A. Tamburri, and W.-J. V. D. Heuvel, "The pains and gains of microservices: A Systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, Dec. 2018.

[10] G. Marquez and H. Astudillo, "Actual Use of Architectural Patterns in Microservices-Based Open Source Projects," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2018, pp. 31–40.

[11] F. Osses, G. Marquez, and H. Astudillo, "Exploration of academic and industrial evidence about architectural tactics and patterns in microservices," in *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE*. ACM Press, May 2018, pp. 256–257.

[12] D. Taibi, V. Lenarduzzi, and C. Pahl, *Microservices Anti-patterns: A Taxonomy*. Springer International Publishing, Jan. 2020, chapter 5, pp. 111–128, in Microservices: Science and Engineering, ISBN: 978-3-030-31646-4.

[13] R. Borges and T. Khan, "Algorithm for detecting antipatterns in microservices projects," in *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. CEUR-WS, Sep. 2019, pp. 21–29.

[14] R. Tighilt, M. Abdellatif, N. Moha, H. Mili, G. E. Boussaidi, J. Privat, and Y.-G. Guéhéneuc, "On the Study of Microservices Antipatterns: a Catalog Proposal," in *Proceedings of the 25th European Conference on Pattern Languages of Programs*, 2020, p. To appear.

[15] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, pp. 1–26, Jul. 2004.

[16] M. I. Rahman, S. Panichella, and D. Taibi, "A curated Dataset of Microservices-Based Systems," in *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. CEUR-WS, Sep. 2019, pp. 1–9.