On the Study of Microservices Antipatterns: a Catalog Proposal

Rafik Tighilt
tighilt.rafik@courrier.uqam.ca
Université du Québéc à Montréal
Montréal, Canada

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

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

Hafedh Mili
mili.Hafedh@uqam.ca
Université du Québéc à Montréal
Montréal, Canada

Ghizlane El Boussaidi
ghizlane.elboussaidi@etsmtl.ca
École de Technologie Supérieure
Montréal, Canada

Jean Privat
privat.jean@uqam.ca
Université du Québéc à Montréal
Montréal, Canada

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

ABSTRACT

Microservice architecture has become popular in the last few years as it allows the development of independent, highly reusable, and fine grained services. However, a lack of understanding of its core concepts and the absence of a ground-truth lead to design and implementation decisions, which might be applied often and introduce poorly designed solutions, called antipatterns. The definition of microservice antipatterns is essential for improving the design, maintenance, and evolution of microservice-based systems. Moreover, the few existing specifications and definitions of microservice antipatterns are scattered in the literature. Consequently, we conducted a systematic literature review of 27 papers related to microservices and analyzed 67 open-source microservice-based systems. Based on our analysis, we report in this paper 16 microservice antipatterns. We concisely describe these antipatterns, how they are implemented, and suggest refactoring solutions to remove them.

KEYWORDS

Microservices, architecture, antipatterns

ACM Reference Format:
Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hafedh Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. 2020. On the Study of Microservices Antipatterns: a Catalog Proposal. In *European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20), July 1–4, 2020, Virtual Event, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3424771.3424812

EuroPLoP '20, July 1–4, 2020, Virtual Event, Germany

ACM ISBN 978-1-4503-7769-0/20/07…$15.00
https://doi.org/10.1145/3424771.3424812

1 INTRODUCTION

Microservice architecture is an *architectural style and approach to developing a single application as a suite of small services, each running on its own process and communicating with lightweight mechanisms, often an HTTP resource API* [1].

This architectural style has been broadly adopted by several major actors in industry (e.g., Amazon, Netflix, Riot Games, etc.) as it allows the development of independent, reusable, and fine grained services. The dynamic and distributed nature of microservice-based systems allow them to offer greater agility and operational efficiency to enterprises, which are challenging to be obtained with monolithic applications[2].

Microservices recently demonstrated to be an effective architectural paradigm for migrating and modernizing monolithic applications. They allow the developers to decompose monolithic applications into small and independent services. Each service (1) is developed by a team; (2) represents a single business capability; and, (3) can be delivered and updated autonomously without impacting other services and their releases.

However, the highly dynamic nature of microservice-based systems as well as the continuous integration and continuous delivery of microservices can lead to design and implementation decisions, which might be applied often and introduce poorly designed solutions, called antipatterns. These antipatterns may significantly affect the maintainability of the systems and degrade their design and operational quality [3].

Unlike object-oriented antipatterns, the specifications and definitions of microservice antipatterns are still in their infancy. Only a few books and academic papers deal with microservice antipatterns. These antipatterns are often mixed with Service Oriented Architecture (SOA) antipatterns and not described thoroughly: while microservices antipatterns are mainly concerned with service granularity, deployment, and monitoring, SOA antipatterns often focus on the architectural design of services and service-based systems. Besides, the current state-of-the-art microservices antipatterns (1) are scattered in the literature, (2) are called under different names, and (3) are done with incomplete/imprecise definitions/description. This makes it difficult for both practitioners and researchers to

gather relevant information to their context. It also underlines the lack of a solid foundation to refer to while investigating or working with microservices. Hence, in this paper, we want to unify the definitions and naming of microservice antipatterns and go beyond granularity, deployment, and monitoring.

In this paper we propose a catalog of the state-of-the art microservice antipatterns based on (1) a systematic literature review of 27 studies related to microservice architectural design, and (2) the analysis of 67 concrete microservice-based systems to have a better understanding of the implementation of these antipatterns and their possible refactored solutions. The paper provides a uniform and structured description of each antipattern as well as refactoring solutions to remove them. The paper also studies the impact of these antipatterns on both developers and microservices end-users.

This paper is structured as follows. Section 2 describes the related works. Section 3 details our study design. Section 4 provides our catalog of microservice antipatterns. Section 5 synthesizes our work and provides several recommendations. Section 6 discusses threats to validity. Section 7 concludes with future work.

2 BACKGROUND AND RELATED WORK

Definition of Microservices: Microservices are an architectural style for developing applications as sets of small services, each running in its own process and communicating through lightweight mechanisms [1]. While microservices emerged as a variant of the SOA paradigm, there are several differences between them such as (1) Microservices promotes multidisciplinary teams (DBAs, engineers, DevOps etc.) while SOA teams are more specialized, (2) SOA services are larger, modular services compared to microservices fine-grained services, (3) Microservices rely on lightweight APIs while SOA uses ESB, and (4) SOA promotes sharing between services of code, resources and functionalities as much as possible to support service reuse. Microservices promote a share-nothing philosophy to easily apply iterative, continuous development and delivery processes (DevOps) and to increase the system agility [4].

Microservice Patterns and Antipatterns: Several research work exist on the design of microservice-based systems. Pahl and Jamshdi [5] conducted a systematic mapping study of 21 works on microservice design published between 2014 and 2016. They studied and classified the works based on a characterization framework defined in their work. They concluded that microservice research is still in a formative stage. They also reported a lack of research tool supporting microservice-based systems.

Zimmerman [6] mapped microservice tenets to SOA patterns and concluded that microservices are just one special implementation of the SOA paradigm.

Garriga [7] proposed a taxonomy of the concepts of microservices, which includes the whole microservice life-cycle (design, implementation, deployment, runtime, and cross-cutting concerns) as well as organizational aspects.

Soldani et al. [8] studied the industrial *grey* literature and identified and compared benefits and limitations of microservices. They also studied the design and development practices of microservices to bridge academia and industry in terms of research focus.

Marquez and Astudillo [9] extended their previous work with Osses [10] to determine whether architectural patterns are used in the development of microservice-based systems. They 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.

Taibi et al. [11] proposed a taxonomy of microservice antipatterns based on a survey of 27 practitioners. They compiled 20 antipatterns and estimated their degree of harmfulness from the practitioners' point of view. They did not describe the antipatterns, their symptoms, or their practical use and implementations.

We rely on these previous works to perform a systematic literature review of microservice antipatterns. We describe these antipatterns, their implementations, and refactoring solutions. We also describe the impact of each antipattern from the point of view of microservice end users and developers.

3 STUDY DESIGN

We use the following steps to build our catalog of microservice antipatterns. We believe that the following study design can be used for replication purpose and for future studies investigating other microservice patterns and antipatterns.

3.1 Systematic Literature Review Methodology

We follow the procedures proposed by Kitchenham et al. [12] for performing systematic literature reviews. First, we collected research papers based on search queries. We started by identifying relevant query terms that are related to microservices. Then, for each keyword, we identified a set of related terms and synonyms using an online synonym finder.[1] Thus, we defined the following search string:

> (microservice OR micro-service OR service) AND (antipattern OR anti-pattern OR bad smell OR pitfall OR poor design OR code smell OR bad practice)

Second, we executed this search query in different scientific search engines: ACM Digital Library, Engineering Village, Google Scholar, IEEE Xplore Digital Library, which returned a total of 1,195 unique references.

Third, we filtered these references based on (1) their titles, (2) their abstracts, and (3) their contents. Two of the authors manually and independently analyzed all the papers and then reconciled any differences through discussions. We excluded from our review papers that met one of the following criteria:

- Papers not written in English.
- Papers not related to microservices.
- Papers not related to microservice antipatterns.

Thus, we reduced the number of references to 21 papers on the design of microservice-based systems.

Fourth, we applied forward and backward snowballing [13, 14] to minimize the risk of missing important papers. Forward snowballing refers to the use of the bibliographies of the papers to identify new papers. Backward snowballing refers to the identification of new papers citing the identified papers. We iterated the backward and forward snowballing and applied for each new candidate paper

[1] https://www.synonym-finder.com/

our exclusion criteria. We stopped the iteration process when we could not found new candidate paper. We performed a total of five iterations and added six papers. We thus obtained 27 papers that describe microservice antipatterns, presented in Table 1.

3.2 Analysis of Microservice-Based Systems

We searched for open-source microservice-based systems to have a better understanding of microservice antipatterns. We manually analyzed 67 systems[2] to detect potential violations of microservice design practices and thus potential antipatterns. We analyzed the implementation of each detected antipattern in the source code and noted textually its related symptoms/hints that helped us to detect their presence in systems. This analysis also allowed us also to identify the concrete refactoring solutions or practices that should be applied to remove the antipatterns

3.3 Validation and Generation of the Catalog of Microservice Antipatterns

Based on our systematic literature review and the analysis of microservice-based systems, we generated a catalog of microservice antipatterns. Through discussions among the authors as well as prior studies on SOA patterns and antipatterns, we generalized the specifications and definitions of each microservice antipattern, their symptoms, and their possible refactoring solutions.

Several templates are used in the literature to define patterns and antipatterns [34, 35]. We adapted the template provided by Dudney et al. [36] to describe the list of microservice antipatterns as follows:

- Antipattern: Name of the antipattern.
- Context: Context in which the antipattern could occur.
- General form: How the antipattern manifested itself.
- Symptoms: Hints/elements that show the presence of the antipattern.
- Consequences: Drawbacks related to the presence of the antipattern.
- Refactored solution: Steps to remove the antipattern and apply best practices.
- Advantages of refactoring: Benefits of removing the antipattern by applying the refactoring solution.
- Trade-offs: Trade-offs between keeping/removing this antipattern.

Finally, we obtained a total number of 16 antipatterns in our catalog that we detail in the following section.

3.4 Categorization of Microservice Antipatterns

We organized our antipattern catalog into four categories, based on the development cycle of a microservice-based system:
Design: Antipatterns related to the specification of the architectural design of a microservice-based system.
Implementation: Antipatterns related to how the microservices are implemented.
Deployment: Antipatterns related to packaging and deployment of microservice-based systems.

[2]https://github.com/davidetaibi/Microservices_Project_List

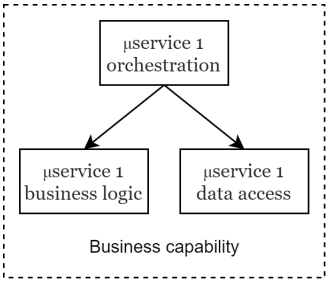

Figure 1: Wrong Cut

Monitoring: Antipatterns related to the monitoring of microservice-based systems, their behavior and changes.

We relied on the scale of Taibi et al. [11] to assign a developer's and a end user's *impact* (high, moderate, or low) to each antipattern based on our observations:
High: Antipattern consequences directly affect end users.
Moderate: End user may indirectly experience some of the impact, either in performance or in application evolution.
Low: Antipattern consequences has little to no impact on end users, only as an increase in maintenance or deployment costs.

4 MICROSERVICES ANTIPATTERNS

We now provide our catalog of 16 microservices antipatterns.

4.1 Design Antipatterns

We found four antipatterns related to the design of microservice-based systems.

4.1.1 Wrong Cut

Also known as: Layered microservices architecture
Developer impact: High
End user impact: Low
Refactored solution name: Decompose on business capabilities

Context: A microservice should encapsulate a group of functionalities to allow the independent delivery of business capabilities. It should be owned, developed, and deployed by a single team. It should fulfill a single purpose.

General Form: The system is decomposed in microservices following technical aspects, such as presentation layer, business layer, and data access layer, as illustrated by Figure 1.

Symptoms: Some of the following aspects can indicate the presence of the Wrong Cut antipattern in a microservice-based system: (1) high microservice coupling; (2) process calls; (3) front-end/ORM microservices; or, (4) deployment dependencies.

Consequences: Delivering business capabilities becomes harder because it involves multiple teams that need coordination. Microservices become highly coupled together and thus failures become more difficult to handle.

Refactored Solution: Microservices should be decomposed based on business capabilities and organization teams. They should be

Table 1: Microservices design references

Reference	Title
Newman [2]	Building Microservices: Designing Fine-Grained Systems
Di Francesco et al. [4]	Research on architecting microservices: Trends, focus, and potential for industrial adoption
Pahl and Jamshidi [5]	Microservices: A Systematic Mapping Study
Garriga [7]	Towards a Taxonomy of Microservices Architectures
Soldani [8]	The pains and gains of microservices: A Systematic grey literature review
Marquez And Astudillo [9]	Actual Use of Architectural Patterns in Microservices-Based Open Source Projects
Osses et al. [10]	Exploration of academic and industrial evidence about architectural tactics and patterns in microservices
Taibi et al. [11]	Microservices Anti Patterns: A Taxonomy
Salah [15]	Microservices Antipatterns
Bogard [16]	Avoiding microservices megadisaster
Shadija [17]	Towards an understanding of microservices
Taibi et al. [18]	Architectural patterns for microservices: a systematic mapping study
Taibi et Lenarduzzi [19]	On the Definition of Microservice Bad Smells
Brogi et al. [20]	Design principles, architectural smells and refactorings for microservices: A multivocal review
Bogner et al. [21]	Towards a collaborative repository for the documentation of service-based antipatterns and bad smells
Carrasco et al. [22]	Migrating towards microservices: migration and architecture smells
Alshuqayran et al. [23]	A systematic mapping study in microservice architecture
Balalaie et al. [24]	Microservices migration patterns
Carnell [25]	Spring microservices in action
Di Francesco et al. [26]	Architecting with microservices: a systematic mapping study
Dragoni et al. [27]	Microservices: yesterday, today, and tomorrow.
Ghofrani and Lubke [28]	Challenges of microservices architecture: a survey on the state of the practice.
Nadareishvili et al. [29]	Microservice architecture: aligning principles, practices, and culture
Pautasso et al. [30]	Microservices in practice, part 1: reality check and service design
Richards [31]	Microservices antipatterns and pitfalls
Stocker et al. [32]	Interface quality patterns—communicating and improving the quality of microservices APIs.
Wolff [33]	Microservices: flexible software architecture

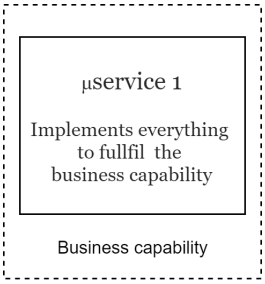

Figure 2: Refactored Wrong Cut

atomic business entities and implement all the functionalities that fulfill given business capabilities (see Figure 2).

Advantages of Refactoring: The decomposition of microservices based on business capabilities allows teams to focus on single responsibilities. The microservices become easier to maintain, deployments are more frequents, the microservice boundaries are clear.

Trade-offs: Decomposing a system by business capabilities results in multiple independent teams that can be expert in these capabilities and focus on their tasks. However, implementing new functionality that spans multiple business capabilities requires coordination and efficient communication tools between teams. It is also challenging to keep all the teams aligned to the system vision.

4.1.2 Cyclic Dependencies

Also known as: N/A
Developer impact: High
End user impact: Low
Refactored solution name: Group dependent microservcies

Context: Microservices should be independent processing units that communicate through lightweight mechanisms [1] to avoid managing dependencies and the "distributed monolith" pitfall [19].

General Form: Microservices depend on each other in a cyclic way, as illustrated by Figure 3.

Symptoms: Cyclic dependencies manifest through (1) direct calls between microservices; (2) frequent communications between microservices; or, (3) presence of HTTP requests in callbacks.

Consequences: Microservices are no longer independent. The deployment of a microservice depends on deploying its coupled microservices. A failure in one of the cyclic-dependent microservices causes a failure in the others.

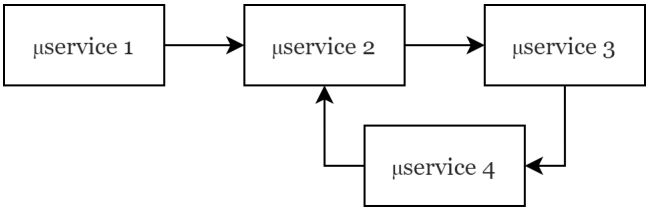

Figure 3: Cyclic Dependencies

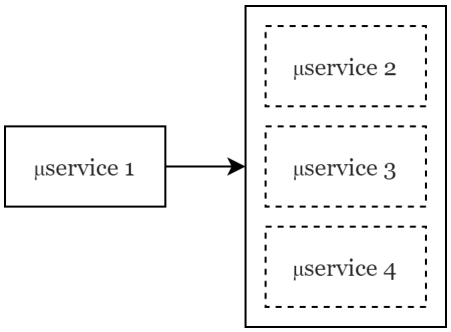

Figure 4: Refactored Cyclic Dependencies

Refactored Solution: We can avoid cyclic dependencies by grouping strongly dependent microservices into a single microservice, see Figure 4.

Advantages of Refactoring: Refactoring into units remove dependencies between microservices and transforms cyclic-dependent microservices into a singular unit easier to maintain and deploy.

Trade-offs: Cyclic dependencies may be necessary when the output of one service should trigger an input microservice, for example when processing product orders result in orders for sub-products to be processed. However, cyclic dependencies make reasoning about the business processes more difficult as well as make changes to the involved microservices harder. Refactoring this antipattern requires updating all the microservices that communicate with the dependent microservices. Developers should also ensure that the resulting microservices each fulfill one business capability and do not become *Mega Services*.

4.1.3 Mega Microservice

Also known as: N/A
Developer impact: Moderate
End user impact: Moderate
Refactored solution name: Decompose on business capabilities

Context: Microservices should be small and independent units, independently deployable and serving a single purpose [1].

General Form: A microservice that serves multiple purposes.

Symptoms: A mega microservice is a microservice with a high number of lines of code, modules or files, as well as a high fan-in.

Consequences: Having a mega microservice creates maintenance issues, reduced performance, and difficult testing, in addition to the complexity of the microservices infrastructure.

Refactored solution: We must decompose the mega microservice into smaller microservices that serve a single purpose.

Advantages of Refactoring: Decomposing a mega microservice into smaller pieces simplifies the deployment and testing process and, isolates business capabilities into well defined microservices.

Trade-offs: Having one cohesive microservice providing a set of related business capabilities helps developers focus and coordinate their work. However, it also prevents the evolution of the systems in new directions, the reuse of microservices in other scenarios or systems, and developers forming loosely coupled, independent teams.

4.1.4 Nano Microservice

Also known as : Break the piggy bank
Developer impact: Low
End user impact: Low
Refactored solution name: Decompose on business capabilities

Context: Refactoring a monolith system into a microservices-based system is a complex problem. Microservices should fulfill single business capabilities, no more but also no less.

General Form: A monolith system broken into a large number of small microservices.

Symptoms: The nano microservice antipattern exists when (1) the system has a large number of microservices; (2) microservices exchange a lot of information; or, (3) cyclic dependencies exist.

Consequences: When decomposing a monolith system into microservices without considering domain or granularity, services often become coupled.

Refactored Solution: We must decompose a system around business capabilities and consider microservices as the base unit for a business entity.

Advantages of Refactoring: Refactoring nano microservices increases microservices independence, system efficiency, and ease maintenance.

Trade-offs: Nano microservices increase the opportunities for reuse and help developers focus on important microservices (those bringing coordination and business values). However, at development time, they require more developers or that developers switch contexts more often; at deployment time, they yield more work because more services must be deployed; and at runtime, they imply more communication overhead among services. However, grouping a set of nano microservices may lead to microservices that no longer have only one business capability. The refactoring of *nano microservices* antipatterns can also lead to Wrong Cuts or Mega Microservices.

4.2 Implementation Antipatterns

We now describe antipatterns related to the implementation of microservice-based systems.

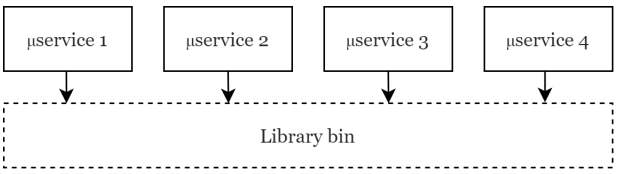

Figure 5: Shared Libraries

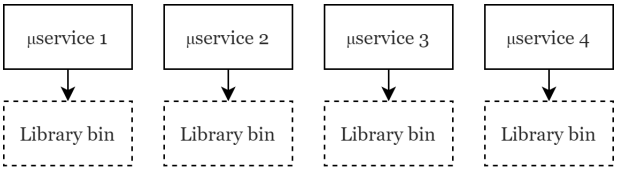

Figure 6: Refactored Shared Libraries (1)

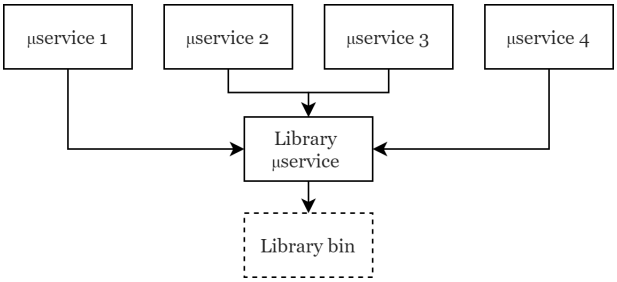

Figure 7: Refactored Shared Libraries (2)

4.2.1 Shared Libraries

Also known as: I was taught to share
Developer impact: Moderate
End user impact: Low
Refactored solution name: Share as little as possible

Context: Microservices should avoid sharing runtime libraries and code directly.

General Form: Microservices share libraries or files as illustrated by Figure 5

Symptoms: The presence of executable files or runtime libraries shared among multiple microservices, added at compile or packaging time, can indicate this antipattern.

Consequences: This antipattern couples microservices together and breaks the boundary between microservices. It also impedes change, evolution, testing, and deployment [31].

Refactored Solution: Runtime assets should not be shared even at the cost of the DRY principle [31] as illustrated by Figure 6. Shared libraries should be their own microservices as shown in Figure 7.

Advantages of refactoring: Microservices become independent and isolated with clear boundaries.

Trade-offs: Sharing libraries among microservices is useful to ensure consistency and have a single location where to update a library. However, sharing libraries creates a dependency on a single point of failure and defeats the purpose of having independent, lightly coupled microservices. However, when refactoring the Shared Libraries antipattern, the libraries must be copied into the related microservices, which increases the complexity of updating/maintaining the libraries and microservices. Refactoring could be done by adding one microservice that wraps one or more shared libraries.

```
module.exports = {
  catalogueUrl:  util.format("http://catalogue%s", domain),
  tagsUrl:       util.format("http://catalogue%s/tags", domain),
  cartsUrl:      util.format("http://carts%s/carts", domain),
  ordersUrl:     util.format("http://orders%s", domain),
  customersUrl:  util.format("http://user%s/customers", domain),
  addressUrl:    util.format("http://user%s/addresses", domain),
  cardsUrl:      util.format("http://user%s/cards", domain),
  loginUrl:      util.format("http://user%s/login", domain),
  registerUrl:   util.format("http://user%s/register", domain),
};
```

Figure 8: Hardcoded Endpoints

4.2.2 Hardcoded Endpoints

Also known as: Hardcoded IPs and ports [15]
Developer impact: High
End user impact: Low
Refactored solution name: Service discovery

Context: Microservices must communicate with one another. They are independently deployed and usually communicate through REST APIs. Microservices can reach one another endpoints via IP addresses and port numbers.

General Form: Microservice IP addresses, ports, and endpoints are explicitly/directly specified in the source code. Figure 8 show hardcoded endpoints in one microservice-based system[3].

Symptoms: Hardcoded endpoints antipattern show via the presence of IP addresses or fully qualified domain names in source code, configuration files, or environment variables[4].

Consequences: When there are many microservices in a system, it becomes more difficult to track all the endpoints and URLs. Running multiple instances of a microservice with a load-balancer becomes impossible. Changing the IP address or port number of a microservice requires changing and redeploying other microservices.

Refactored Solution: Service discovery prevents hardcoding IP addresses and port numbers. It tracks microservices endpoints and ease the communications among microservices. Two strategies may be used for service discovery: (1) client-side service discovery, in Figure 9, and server-side service discovery, in Figure 10.

Advantages of Refactoring: With service discovery, microservice endpoints can change dynamically without affecting other microservices. It also simplifies the deployment of microservices on containers and virtual machines with dynamic IP addresses. It also provides a single, centralized registry for all microservice endpoints.

[3]https://github.com/microservices-demo
[4]https://github.com/oktadeveloper/spring-boot-microservices-example

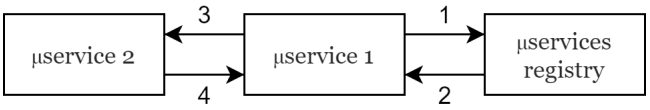

Figure 9: Client side service discovery. Service 1 asks Registry for the endpoint of Service 2 (1,2); then Service 1 uses the returned endpoint to communicate with Service 2 (3,4).

Figure 10: Server side service discovery. Service 1 asks Registry to fulfill the service (1,4); whereas Registry just proxies the communication to Service 2 (2,3).

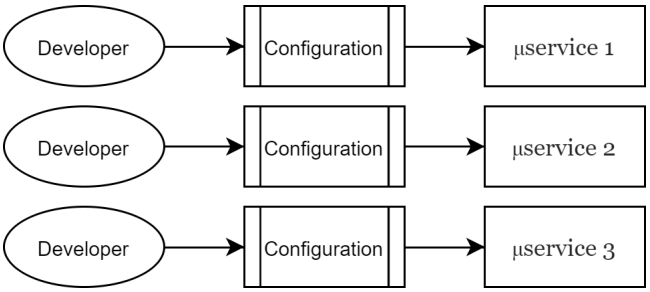

Figure 11: Manual Configuration

Trade-offs: In a tightly controlled, slow changing environment, hardcoding endpoints is an easy shortcut to speed development, deployment, and testing. However, it also makes evolution and scalability more difficult. In particular, client-side service discovery requires writing discovery logic inside each microservice. Server-side discovery requires adding a microservice to manage, deploy and maintain endpoints, which may affect the system's performance. In both refactoring scenarios, a network call is added. Therefore, a trade-off between dynamic and hardcoded endpoints should be considered when refactoring this antipattern.

4.3 Deployment Antipatterns

We now describe antipatterns related to the deployment of microservices.

4.3.1 Manual Configuration

Also known as: N/A
Developer impact: N/A
End user impact: Low
Refactored solution name: Automated configuration

Context: Microservices efficiency relies on automation and everything that can be automated should be automated.

General form: Configuration of instances, services and hosts is done manually by developers as illustrated in Figure 11

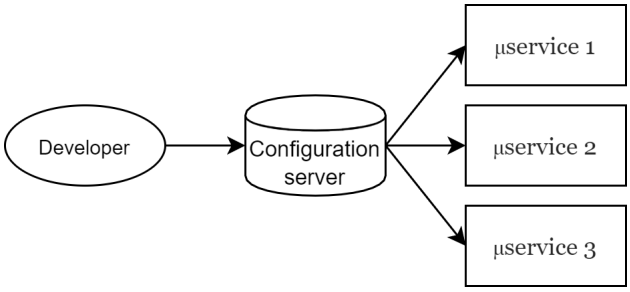

Figure 12: Refactored Manual Configuration

Symptoms: Configuration files in every microservice and the reliance on environment variables can indicate the presence of this antipattern[5].

Consequences: The process of configuring microservices is time consuming and error prone as we need to manage, for each microservice, different configurations for different environments (development, testing and production).

Refactored solution: Use configuration servers and services to automate the configuration process like depicted in Figure 12, this can be done through continuous delivery tools.

Advantages of refactoring: Using configuration management tools allow developers to manage configurations in a centralized way, making it easier to share common elements across configurations, and more generally, easier to maintain consistency across configurations of different services.

Trade-offs: Manually configuring microservices and their interactions makes it easy to write, deploy, and run the microservices and deliver the system. Enabling automatic configuration requires to set up new tools and pipelines, which may be complex depending on the system. It also requires monitoring to check that all the configurations are correctly loaded for each service. However, it is necessary when the number of microservices grows and-or when the system must scale (in deployment and at runtime).

4.3.2 No Continuous Integration / Continuous Delivery (CI/CD)

Also known as: No DevOps tools [11]
Developer impact: Low
End user impact: Low
Refactored solution name: CI/CD usage

Context: The independent deployment of microservices allows relatively small teams -within a single enterprise- to easily apply iterative continuous development and delivery (DevOps) processes, and thereby increase system agility. The integration of Development and Operations, and the continuous delivery result in (1) reducing delivery time; (2) increasing delivery efficiency; (3) decreasing time between releases; and (4) maintaining software quality [2].

General form: No automated solution to manage tests and deployment processes.

[5] https://github.com/venkataravuri/e-commerce-microservices-sample

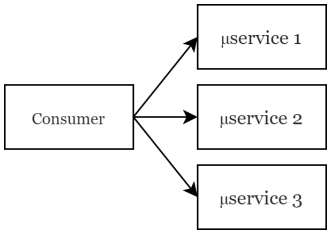

Figure 13: No API gateway

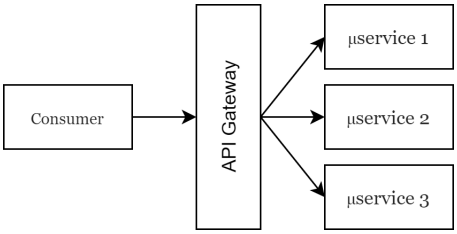

Figure 14: API gateway

Symptoms: Some of the following symptoms can indicate the presence of the no CI/CD antipattern: (1) no version control repositories on microservices; (2) no unit/integration/functional tests; (3) no automated delivery tools; or (4) no staging environments.

Consequences: Testing and deployment processes are slower and error prone [37]. Bugs and build failures are harder to catch. Also, a lot of coordination is needed to ensure the cohesion of microservices deployments.

Refactored solution: Use continuous integration and continuous delivery tools to build an efficient pipeline of testing and deployment.

Advantages of refactoring: When using continuous integration and continuous delivery tools, code changes can be smaller and deployed as early as possible. Faults in the system become easier to track and to identify. Rollbacks are easier because the root cause of a problem can be found in a efficient way. Also, tests become more efficient as they are automated and executed before each deployment.

Trade-offs: Despite all their benefits, there are some challenges when adopting CI/CD. Substantial effort, time, and money are required to install and run CI/CD-based development. Also an organization may face internal resistance from its developers to learn and integrate CI/CD in their well-honed development process. A trade-off between refactoring this antipattern and the aforementioned challenges should be considered when considering this antipattern.

4.3.3 No API Gateway

Also known as: N/A
Developer impact: Moderate
End user impact: Low
Refactored solution name: API gateway

Context: When building microservices-based systems, consumer applications needs to communicate with a lot of microservices, and every consumer needs a very specific set of information.

General form: Microservices are exposed and consumers communicate with them directly, as shown in Figure 13.

Symptoms: Consumer applications sending multiple HTTP requests, or requests to multiple different URLs, and systems that have multiple front ends (Web, mobile, etc.) can be indicative of the presence of this antipattern[6].

[6]https://github.com/venkataravuri/e-commerce-microservices-sample

Consequences: Consumer applications need to be "aware" of how the application is partitioned into microservices. They also need to manage/maintain endpoints for many microservices. Finally, authentication and authorization are done per microservice.

Refactored solution: Use an API gateway as the single entry point for all the microservices as illustrated in Figure 14.

Advantages of refactoring: As a single entry point for all microservices, the API gateway will be in charge of: 1) routing requests to the corresponding microservices, and 2) common tasks such as authentication, authorization, input validation, and response transformation.

Trade-offs: Implementing an API gateway requires to create, manage, and maintain a new microservice. It also requires choosing the type of API gateway to implement (to handle or not security for example). Implementing an API gateway may also slow down the microservices response times as it adds additional network operations. However, without an API gateway, the clients of the microservice-based systems are dependent on particular microservices, which limit the systems evolution.

4.3.4 Timeouts

Also known as: Dog piles [31]
Developer impact: N/A
End user impact: High
Refactored solution name: Circuit breaker

Context: Service *availability* refers to the possibility for a service consumer to connect and send a request to a service. Service *responsiveness* is the time taken by the service to respond to that request [31]. It is common practice in distributed systems to have consumer applications/tasks use timeouts to handle service unavailability or unresponsiveness.

General form: Timeouts are set by developers when sending requests or waiting for responses.

Symptoms: Request retrial and timeout values are good signs of the presence of this antipattern.

Consequences: Finding the right timeout value is difficult. A too short timeout will fall too quickly on timeout exception handling, not giving "busy" services enough time to respond, leading to thrashing; a too long timeout will keep consumer applications wait too long before bailing out on a down service.

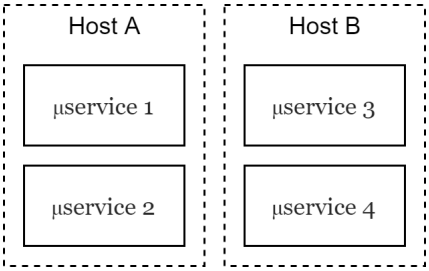

Figure 15: Multiple Microservices Instances Per Host

Refactored solution: Use *circuit breakers*, which are interceptors that check regularly for service "health" (availability and responsiveness), and decline requests automatically if the microservice is down.

Advantages of refactoring: Consumer applications no longer need to wait until timeout to fall back on service unavailability exception handling, and avoids unnecessary retrials. Circuit breakers close themselves (become pass-through) automatically once the service is up again.

Trade-offs: Implementing circuit breakers requires updating all microservices to call one another via the circuit breakers as proxies. Choosing the right timeout values for the circuit breaker is also challenging. Yet, circuit breakers are necessary to maintain and improve the responsiveness of the systems.

4.3.5 Multiple Service Instances Per Host

Also known as: N/A
Developer impact: N/A
End user impact: Low
Refactored solution name: Single service instance per host

Context: When microservices are built, multiple deployment strategies could be applied. We can choose to deploy either each microservice instance in its own host or multiple microservices instances in a single host.

General form: A single host contains multiple microservices instances are deployed to the same host (See Figure 15).

Symptoms: The hints of the presence of this antipattern could be : (1) a single deployment platform; (2) a single version control repository; or (3) a global deployment script.

Consequences: Microservices have to share the same resources that are available inside the host. Moreover, scaling up or down a given host involves scaling all the instances that are inside this host. Finally, possible technology-related conflicts may happen between the microservices instances that share the same host.

Refactored solution: Each microservice instance should be independently deployed on a different host as illustrated in Figure 16.

Advantages of refactoring: Single microservice instance per host allows the independent use and scale of resources for each one of them. It also provides an additional isolation layer to microservices, and prevents conflicts between the technologies that are present in the same host.

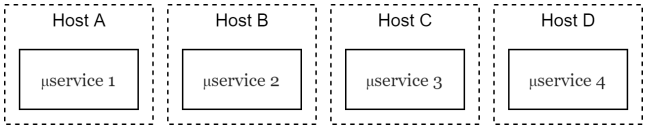

Figure 16: Single Microservice Instance Per Host

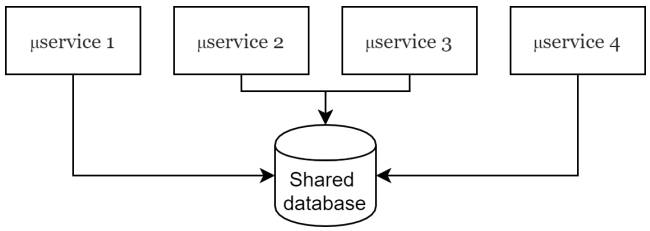

Figure 17: Shared Database Between Microservices

Trade-offs: Running multiple microservices on the same host eases deployment and debugging. Besides, having a single microservice per host requires managing load balancing, service discovery, and resource consumption for each microservice. It also requires managing multiple environments with potentially multiple systems and languages, and monitoring each instance independently.

4.3.6 Shared Persistence

Also known as: Data ownership [16]
Developer impact: Moderate
End user impact: Low
Refactored solution name: Database per service

Context: microservices architecture is a way of building systems that decompose application code into small independent services [1]. Each of these small services may need to persist and access data. However, in order to fully benefit from the microservices architecture, software architects need to handle data storage in a way where each microservice can store and access its data without affecting other microservices [38].

General form: A single database being accessed by multiple microservices, as illustrated in Figure 17.

Symptoms: This antipattern is characterized by one or more of the following symptoms: (1) multiple microservices share the same configuration files and deployment environments; (2) database tables are prefixed; or (3) databases have a lot of schemas.[7]

Consequences: Microservices become coupled together [38], and the maintenance becomes harder. Also, data must be adapted to fit a single data store. In addition, microservices become no longer independently deployable because they share the same database.

Refactored solution: Data should be separated regarding of how it is accessed and how it is used. We have to choose the appropriate data store for every kind of data (Polyglot persistence [39]), and

[7] https://github.com/acmeair/

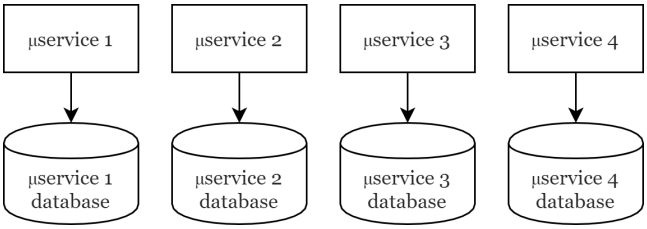

Figure 18: Database Per Microservices

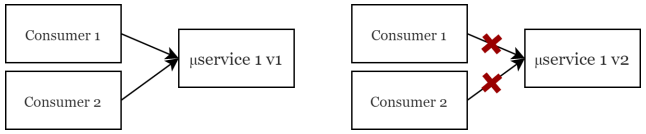

Figure 19: No API Versioning

use a single database for each microservice. Figure 18 illustrates the refactoring solution.

Advantages of refactoring: Refactoring to the database per service pattern offers the following advantages: (1) separation of concerns as each microservice owns its data; (2) each microservice can be fully managed by a single team; and (3) flexibility in terms of storage technologies.

Trade-offs: A persistence service (or microservice) shared among many microservices eases deployment and increases performance while ensuring the consistency of the data among the microservices. Having one database per microservice requires managing multiple database systems, implementing queries across multiple microservices, and ensuring the consistency among the different databases.

4.3.7 No API Versioning

Also known as: Static contract [31]
Developer impact: Moderate
End user impact: Moderate
Refactored solution name: API versioning

Context: Sometimes, multiple versions of the exposed API of a given microservice must be supported. This is generally the case when a service API has undergone major changes and we need to support both the new and old versions for some period of time.

General form: No information is available on the microservice version.

Symptoms: Some of the following are hints to the presence of this antipattern: (1) microservices endpoint urls do not contain version numbers[8]; (2) no custom header information are sent by the client; and (3) multiple microservices have similar names.

Consequences: Changes to a microservice API will impact all consumers and may lead to breaking changes; i.e. several consumers may fail to use the new microservice.

[8]https://github.com/GoogleCloudPlatform/microservices-demo/

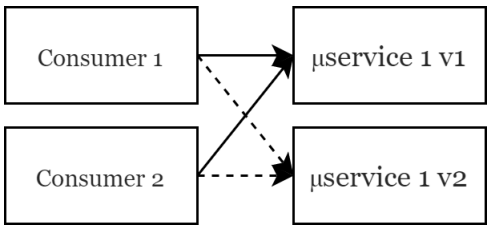

Figure 20: API Versioning

Refactored solution: Using API versioning allows the same microservice to provide multiple versions. In an HTTP-based microservices architecture, the version number can be part of the URI. Otherwise, it can be included in the the header of the requests/responses.

Advantages of refactoring: Microservices can be upgraded without breaking changes, and consumers can continue using an older version of a microservice while switching gradually to the newer version as depicted in Figure 20 [40].

Trade-offs: API versioning requires managing and documenting the versions of each microservice. It also changes resource names and URIs with version changes. Thus, it slows down development and deployment and leads to having to maintain multiple versions of the same APIs. Yet, it is necessary for large systems whose APIs evolve, for example to increase security and provide new features while retiring old, unnecessary ones.

4.4 Monitoring Antipatterns

We describe in this section the antipatterns that are related to the operation of microservices-based systems.

4.4.1 No Health Check

Also known as: N/A
Developer impact: N/A
End user impact: Low
Refactored solution name: Health check endpoint

Context: The nature of microservices is volatile. A microservice can be deployed anywhere, and can be unavailable for a particular amount of time or in a particular context.

General form: No endpoint is exposed to check the health of the given microservice[9].

Symptoms: No periodic HTTP request, no API gateway or no service discovery can be hints of the presence of this antipattern.

Consequences: Consumers of a given microservice may experience timeouts and long waiting time without getting a response in case the microservice is down.

Refactored solution: Add health check API endpoints to the microservices to periodically verify their status and their ability to answer requests.

[9]https://github.com/Crizstian/cinema-microservice

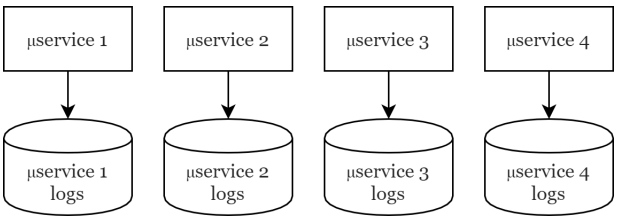

Figure 21: Local Logging

Advantages of refactoring: When using health check endpoints, microservices that are down no longer receive requests, and consumers get the information before even sending a request. This prevent useless timeouts and waiting times.

Trade-offs: Implementing a health-check endpoint allows periodically testing the availability of a microservice. Developers must carefully think about how often the health-check is performed to keep the performance of the systems while a microservice may fail between two health-checks. Besides, health-checks increase the code to be maintained and deployed and the communication among services.

4.4.2 Local Logging

Also known as: N/A
Developer impact: Moderate
End user impact: Low
Refactored solution name: Distributed logging

Context: Each microservice produces a lot of information that is being logged in different file systems. This information is very useful in a monitoring context and should be easily accessed and stored.

General form: Each microservice writes its logs to a local storage. See figure 21.

Symptoms: Some indications of this antipattern are (1) the presence of log files inside microservices; (2) files being written by the microservice; (3) the usage of time aware databases; and (4) logging frameworks and tools[10].

Consequences: Local logs can be very difficult to aggregate and to analyze. This slows down the monitoring process proportionally to the number of microservices and log size.

Refactored solution: Distributed logging allows microservices to use a logging systems which will be responsible of aggregating logs for all the microservices. See figure 22.

Advantages of refactoring: Using a distributed logging mechanism allow to have a single log repository, forces microservices to use the same log formatting, and simplifies the monitoring and analysis processes.

Trade-offs: Local logging is easy to implement and helps with debugging each microservice independently. Using a distributed logging system requires setting up an infrastructure to aggregate the logs. It also requires having uniform logs generated by all microservices.

[10]https://github.com/venkataravuri/e-commerce-microservices-sample

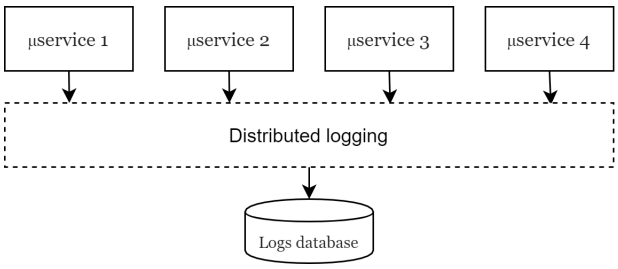

Figure 22: Distributed Logging

4.4.3 Insufficient Monitoring

Also known as: Lack of monitoring [11]
Developer impact: Moderate
End user impact: High
Refactored solution name: Application metrics

Context: Because provided microservices are often subject to service level agreements (SLA), monitoring their behavior and performance is crucial.

General form: Performance and failure of the microservices are not tracked.

Symptoms: Some indications of this antipattern include the use of local logging[11] for some microservices or the absence of health check endpoints.

Consequences: Insufficient monitoring may hinder the maintenance activities in a microservices-based system. Failures become more difficult to catch and tracking performance issues become more tedious. It may even affect the ability to compy with the SLA.

Refactored solution: Use a monitoring tool that will gather performance and failure data and statistics about every microservice in the system.

Advantages of refactoring: Monitoring every aspect of a microservice based system provides a full understanding of the behavior of the application. It also provides data that can be used to enhance the performance and quality of the whole application.

Trade-offs: Monitoring microservices requires setting up new tools and monitoring infrastructure. It also requires the refactoring of the microservices to include the monitoring processes. Also aggregating monitoring data and metrics could be challenging: it requires the unifying heterogeneous microservices logs and data analysis methods.

5 SYNTHESIS

We proposed a catalog of 16 microservice antipatterns categorized into: design, implementation, deployment, and operation-related antipatterns. We found that most of these antipatterns relate to deployment (7/16), design (4/16) and monitoring (3/16). Only two antipatterns relate to implementation.

[11]https://github.com/matt-slater/delivery-system

We also studied the impact of these antipatterns on both developers and end users by assigning an impact level to each antipattern. We found that most of the antipatterns (10/16) have a low end-user impact. Three antipatterns have a moderate end-user impact. These antipatterns pertain to design, deployment, and monitoring. Finally, two antipatterns have a high end-user impact: *Insufficient Monitoring* and *Timeouts*.

We reported that no antipattern highly impacts both developers and end users. *Insufficient Monitoring* has a high impact on end users but a moderate one on developers. It should be avoided in microservice-based system. *API Versioning* and *Mega Microservice* have moderate impact on both developers and end users.

When developing microservice-based systems, developers should first decompose their systems into microservices, avoiding *Wrong Cuts*, *Mega* and *Nano Microservices*. They should also separate microservices responsibilities and the libraries that they share. They should share as little runtime libraries as possible and automate as much of their deployment as possible (*Manual Configuration*, *No CI/CD*). The evolution and maintenance of microservice-based systems is also important: monitoring, versioning, and logging must be taken seriously by developers.

6 THREATS TO VALIDITY

We now discuss threats to the validity of our catalog.

Threats to Internal Validity: These threats concern the causal relationship between the treatment and the outcome. We accept these threats because our goal was not an exhaustive list of all existing microservice antipatterns. To mitigate these threats, we studied 27 works on microservice design and few open-source microservice-based systems.

Our search query may not cover all terms related to microservice antipatterns and could miss important works. To minimize this threat, we (1) included in our search query the most important keywords related to the design of microservices and (2) applied forward and backward snowballing [13, 14] to minimize the risk of missing important papers.

Threats to External Validity: These threats concern the generalizability of our catalog. We minimized this threat by reporting antipatterns recurring three or more times.

Threats to Reliability Validity: These threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our catalog. We provided online access to all the references that we studied to build our catalog of microservices antipatterns and links to all the microservice-based systems that we analyzed.

7 CONCLUSION

We presented a catalog of microservice antipatterns based on a systematic literature review and the analysis of open-source microservice-based systems. We described these antipatterns, their implementations, and provided refactoring solutions to remove them. We also studied their impacts on developers and end users.

Our study can help both researchers and practitioners interested in microservices: practitioners by providing a better understanding of the bad practices that they should avoid when developing microservices and researchers by proposing new solutions for microservice development practices and establishing future research directions on the design of microservice-based systems.

In future work, we want to study patterns of microservice-based systems and propose an exhaustive and uniform catalog that describes these patterns. We also want to develop a tool to detect automatically the presence of microservice antipatterns. Finally, we will empirically study the effect of these antipatterns on the quality of systems.

REFERENCES

[1] J. Lewis and M. Fowler, "Microservices a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, accessed: January 2020.

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

[3] F. Palma, "Detection of soa antipatterns," in *Service-Oriented Computing - ICSOC 2012 Workshops*, A. Ghose, H. Zhu, Q. Yu, A. Delis, Q. Z. Sheng, O. Perrin, J. Wang, and Y. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 412–418.

[4] P. Di 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)*. IEEE, 2017, pp. 21–30.

[5] 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, 2016.

[6] O. Zimmermann, "Microservices tenets," *Computer Science - Research and Development*, vol. 32, no. 3-4, pp. 301–310, 2016.

[7] M. Garriga, "Towards a taxonomy of microservices architectures," in *Software Engineering and Formal Methods*. Springer International Publishing, 2018, pp. 203–218.

[8] 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.

[9] 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.

[10] 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, 2018.

[11] D. Taibi, V. Lenarduzzi, and C. Pahl, *Microservices Anti-patterns: A Taxonomy*. Cham: Springer International Publishing, 2020, pp. 111–128. [Online]. Available: https://doi.org/10.1007/978-3-030-31646-4_5

[12] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, no. 2004, pp. 1–26, 2004.

[13] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. ACM, 2014, p. 38.

[14] K. R. Felizardo, E. Mendes, M. Kalinowski, É. F. Souza, and N. L. Vijaykumar, "Using forward snowballing to update systematic reviews in software engineering," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, p. 53.

[15] T. Salah, "Microservices antipatterns," 2016.

[16] J. Bogard, "Avoiding microservices megadisaster," 2017.

[17] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," in *2017 23rd International Conference on Automation and Computing (ICAC)*, Sep. 2017, pp. 1–6.

[18] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: a systematic mapping study," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS, 2018.

[19] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, May 2018.

[20] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi, "Design principles, architectural smells and refactorings for microservices: a multivocal review," *SICS Software-Intensive Cyber-Physical Systems*, pp. 1–13, 2019.

[21] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann, "Towards a collaborative repository for the documentation of service-based antipatterns and bad smells," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2019, pp. 95–101.

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

[23] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2016, pp. 44–51.
[24] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," *Software: Practice and Experience*, vol. 48, no. 11, pp. 2019–2042, 2018.
[25] J. Carnell, *Spring microservices in action*. Manning Publications Co., 2017.
[26] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *Journal of Systems and Software*, vol. 150, pp. 77–97, 2019.
[27] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.
[28] J. Ghofrani and D. Lübke, "Challenges of microservices architecture: A survey on the state of the practice." in *ZEUS*, 2018, pp. 1–8.
[29] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.
[30] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1: Reality check and service design," *IEEE Software*, no. 1, pp. 91–98, 2017.
[31] M. M. Richards, "Antipatterns and pitfalls," 2016.
[32] M. Stocker, O. Zimmermann, U. Zdun, D. Lübke, and C. Pautasso, "Interface quality patterns: Communicating and improving the quality of microservices apis," in *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, 2018, pp. 1–16.
[33] E. Wolff, *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016.
[34] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.
[35] A. Rotem-Gal-Oz, E. Bruno, and U. Dahan, *SOA patterns*. Manning Shelter Island, 2012.
[36] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf, *J2EE antipatterns*. John Wiley & Sons, 2003.
[37] P. Duvall, S. Matyas, and A. Glover, *Continuous Integration, Improving Software Quality and Reducing Risk*. Pearson, 2007.
[38] Microsoft, "Microservices: Data considerations," https://docs.microsoft.com/en-us/azure/architecture/microservices/design/data-considerations, accessed: January 2020.
[39] M. Fowler, "Polyglot persistence," https://martinfowler.com/bliki/PolyglotPersistence.html, accessed: January 2020.
[40] V.Alagarasan, "Microservices antipatterns," https://www.youtube.com/watch?v=uTGIrzzmcv8, accessed: January 2020.