

# **Investigating Practices and Challenges in Microservice-Based Development**

by

Yingying Wang

B.SE., Dalian University of Technology, China, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF

**MASTER OF APPLIED SCIENCE**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL  
STUDIES

(Electrical and Computer Engineering)

The University of British Columbia  
(Vancouver)

April 2019

© Yingying Wang, 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, a thesis/dissertation entitled:

INVESTIGATING PRACTICES AND CHALLENGES IN MICROSERVICE-BASED DEVELOPMENT

submitted by YINGYING WANG in partial fulfillment of the requirements for

the degree of MASTER OF APPLIED SCIENCE

in ELECTRICAL AND COMPUTER ENGINEERING

**Examining Committee:**

JULIA RUBIN, ELECTRICAL AND COMPUTER ENGINEERING  
Supervisor

PHILIPPE KRUCHTEN, ELECTRICAL AND COMPUTER ENGINEERING  
Supervisory Committee Member

KONSTANTIN BEZNOSOV, ELECTRICAL AND COMPUTER ENGINEERING  
Supervisory Committee Member

**Additional Supervisory Committee Members:**

\_\_\_\_\_  
Supervisory Committee Member

\_\_\_\_\_  
Supervisory Committee Member

# Abstract

Microservice-based architecture is a principle inspired by service-oriented approaches for building complex systems as a composition of small, loosely coupled components that communicate with each other using language-agnostic APIs. This architectural principle is now becoming increasingly popular in industry due to its advantages, such as greater software development agility and improved scalability of deployed applications.

In this thesis, we report on a broad interview study we conducted, which involved practitioners developing microservice-based applications for commercial use for at least two years. By deliberately excluding “newcomers” and focusing the study on “mature” teams, our goal was to collect best practices, lessons learned, and technical challenges practitioners face.

Our study helps inform researchers of challenges in developing microservice-based applications, which can inspire novel software engineering methods and techniques. The study also benefits practitioners who are interested to learn from each other, to borrow successful ideas, and to avoid common mistakes.

# Lay Summary

In traditional software development processes, relatively large teams work on a single, monolithic deployment artifact. However, such monolithic applications can evolve into a “big ball of mud”. For better scalability and development agility, practitioners start developing complex applications as a set of loosely coupled components that communicate with each other through lightweight interfaces. Such an approach is referred to as *microservice*-based development and is now becoming increasingly popular in the industry. In this thesis, we report on the results of an interview study we conducted with industrial practitioners developing successful microservice-based applications. The goal of the study is to understand the best practices, lessons learned, and challenges in microservice-based development. Our study helps inform researchers about challenges in microservice-based development and share thoughts on possible future directions. The study also benefits practitioners who are interested to learn from each other, to borrow successful ideas, and to avoid common mistakes when using microservices.

# Preface

This thesis is an original intellectual product of its author, Yingying Wang, done in collaboration with Harshavardhan Kadiyala, under the supervision of Prof. Julia Rubin. This work is covered by UBC Behavioural Research Ethics Board Number: H18-00733.

Part of the work was presented as an invited talk at a Vancouver Microservices Meetup event, attended by around 50 practitioners. This work was also presented at a workshop of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON'18).

# Table of Contents

<b>Abstract . . . . .</b>	<b>iii</b>
<b>Lay Summary . . . . .</b>	<b>iv</b>
<b>Preface . . . . .</b>	<b>v</b>
<b>Table of Contents . . . . .</b>	<b>vi</b>
<b>List of Tables . . . . .</b>	<b>ix</b>
<b>List of Figures . . . . .</b>	<b>x</b>
<b>List of Acronyms . . . . .</b>	<b>xi</b>
<b>Acknowledgements . . . . .</b>	<b>xiii</b>
<b>Dedication . . . . .</b>	<b>xiv</b>
<b>1 Introduction . . . . .</b>	<b>1</b>
1.1 Overview . . . . .	1
1.2 Findings Summary . . . . .	4
1.3 Novelty and Contribution . . . . .	5
1.4 Structure of This Thesis . . . . .	6
<b>2 Background and Related Work . . . . .</b>	<b>7</b>

2.1	Microservices and Their Characteristics . . . . .	7
2.2	Claimed Advantages of Microservices . . . . .	9
2.3	Microservices vs. Collaborative Software Development . . . . .	10
2.4	Existing Studies on Microservice-Based Development . . . . .	10
2.4.1	Exploratory Studies . . . . .	11
2.4.2	Literature Surveys . . . . .	13
<b>3</b>	<b>Study Methodology . . . . .</b>	<b>15</b>
3.1	Subjects . . . . .	15
3.2	Data Collection . . . . .	17
3.3	Data Analysis . . . . .	19
3.4	Ethical Considerations . . . . .	21
3.5	Threats to Validity . . . . .	22
3.5.1	External Validity . . . . .	22
3.5.2	Internal Validity . . . . .	22
3.5.3	Construct Validity . . . . .	23
<b>4</b>	<b>Architecture . . . . .</b>	<b>24</b>
4.1	Microservice Granularity . . . . .	24
4.2	Microservice Ownership . . . . .	26
4.3	Language Diversity . . . . .	27
<b>5</b>	<b>Infrastructure . . . . .</b>	<b>29</b>
5.1	Logging and Monitoring . . . . .	29
5.2	Distributed Tracing . . . . .	30
5.3	Automating Processes . . . . .	31
5.4	Tools . . . . .	32
<b>6</b>	<b>Code Management . . . . .</b>	<b>33</b>
6.1	Common Code . . . . .	33
6.2	Managing API Changes . . . . .	35
6.3	Managing Variants . . . . .	37

<b>7 Discussion and Future Work . . . . .</b>	<b>40</b>
<b>8 Conclusion . . . . .</b>	<b>43</b>
<b>Bibliography . . . . .</b>	<b>45</b>



# List of Tables

Table 3.1	Interviewee Demographics . . . . .	17
Table 3.2	Concept Frequency . . . . .	20
Table 4.1	Microservice Granularity . . . . .	24
Table 6.1	Common Code . . . . .	34
Table 6.2	Managing API Changes . . . . .	36
Table 6.3	Managing Variants . . . . .	38

# List of Figures

Figure 1.1	Popularity of Searches for the Term “Microservices” (Google Trends [62], March 2019)	2
Figure 3.1	Data Collection and Analysis Process	18
Figure 3.2	Mapping of Categories to Concepts	19

# List of Acronyms

**ACM** Association for Computing Machinery

**API** Application Program Interface

**BREB** Behavioural Research Ethics Board

**CD** Continuous Delivery

**CI** Continuous Integration

**CPU** Central Processing Unit

**DevOps** Development and Operation

**DRY** Do Not Repeat Yourself

**gRPC** gRPC Remote Procedure Calls

**HTTP** HyperText Transfer Protocol

**IoT** Internet of Things

**JAR** Java Archive

**MASc** Master of Applied Science

**REB** Research Ethics Board

**REST** Representational State Transfer

**RPC** Remote Procedure Call

**SaaS** Software as a Service

**SIGSOFT** The Association for Computing Machinery's Special Interest Group  
on Software Engineering

**SOA** Service-Oriented Architecture

**UBC** University of British Columbia

# Acknowledgements

First I would like to thank all study participants for volunteering their time to participate in the study and sharing their experience with us. Their help and time are much appreciated.

Back to my overall master studies, I would like to express my sincere gratitude to my supervisor, Professor Julia Rubin, for introducing me the research world, moreover, patiently supporting, guiding, and encouraging me along my studies. I enjoy the venture a lot and look forward to continuing the expedition in the research world under her excellent supervision.

Next, I would like to thank my committee members, Professor Philippe Kruchten and Professor Konstantin Beznosov, for reviewing my thesis and providing constructive comments. I also would like to thank Professor Mieszko Lis for his insightful feedback on shaping my defence presentation.

I also want to thank my collaborator, Harshavardhan Kadiyala, as well as all my (previous and current) labmates from ReSeSS research group for all the inspiring discussions we had, their insightful feedback on my presentations, and the invaluable support during my defence.

Special thanks to my friends Sirou Zhuo, Xinwen Zhang, Lina Qiu, and Wendy Ma for taking care of me during the hectic deadlines, leaving a light on for me during my late nights, and borrowing an ear for me all the time.

Last but not least, I would like to express my deepest gratefulness to my parents for their unconditional and continuing understanding, support, and encouragement, closely and remotely, along the journey.

# Dedication

To my beloved parents

致我亲爱的父母

# Chapter 1

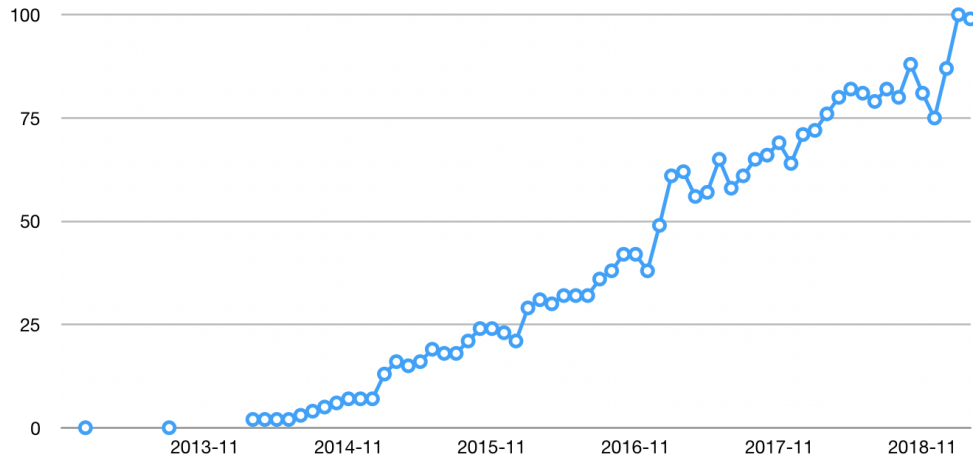
## Introduction

### 1.1 Overview

Service-oriented architecture (SOA) is an architectural approach to designing applications around a collection of independent services. A service can be any business functionality that completes an action and provides a specific result, such as processing a customer order or compiling an inventory report. Services can be stitched together to create composite applications, providing greater functionality to end users [48]. Service-orientation concept had received a lot of attention in early 2000s [17], but lost momentum in 2009. Since 2014, the interest in service-oriented paradigms was renewed under the *microservices* moniker (see Figure 1.1).

The term *microservice* was coined at a workshop of software architects near Venice in May 2011 [65]. It described what the participants saw as a common architectural style that many of them had been exploring. Yet, microservice-based architecture became really popular around 2014, when Lewis and Fowler published their blog on the topic [33] and Netflix shared their expertise and lessons learned from a successful transition to microservices at a Silicon Valley Microservices Meetup event [42]. In 2015, researchers from Gartner [32] found that 68% of organizations implementing or actively investigating transition to microservices [28]. Gartner also ranked microservice-based architecture as one of the top 10 strate-

**Figure 1.1:** Popularity of Searches for the Term “Microservices” (Google Trends [62], March 2019)



gic technology trends for 2016, which were predicted to shape opportunities in digital business through 2020 [67]. In 2018, DZone [16], a programming and DevOps community, conducted an internal survey, which received 732 replies; nearly 50% of respondents stated that they are using microservices, and another 38.7% are considering microservices [25]. The Jakarta EE 2018 survey, covering 1,805 participants, reported that nearly 46% of respondents are now using microservices, and another 21% plan to start using microservices within one year [44]. In another survey of 354 enterprises across 51 countries and 12 industries [13], 63% of companies surveyed are currently using microservice-based architectures. “Famous” adopters of microservice-based architectures include Amazon, IBM, Uber, LinkedIn, Groupon, and eBay [56].

At the core of microservice-based architecture is the SOA-inspired principle of building complex applications as a composition of small, loosely coupled components that encapsulate individual business capabilities and communicate with each other using language-agnostic APIs. The current popularity of microservices can possibly be explained by a combination of several factors:



- **Agile software development:** Agile movement promotes frequent software releases, performed by small, independent teams. However, under the “traditional” software development model, teams need to synchronize on common product release dates. Microservice-based architecture helps reduce cross-team coordination, allowing each team to independently deploy changes to production at the time appropriate for them. In a sense, it takes the already popular agile development paradigm to the next level of flexibility, providing the “architectural phase of the agile progression” [41].
- **Cloud:** Software hosted on a cloud, a.k.a. Software as a Service (SaaS), has become a popular method of software delivery. A survey involving 2,151 Java Virtual Machine developers and IT professionals from Lightbend [37] reported that 31% of developers are already running most of their applications in the cloud and 29% are in the process of creating a cloud-native strategy. The flexible pay-as-you-go offering allows developers and companies to dynamically increase the number of machines when the application is “busy” and decrease when it is used less. Breaking large applications into microservices further supports this goal because of the ability to spin up more instances of the specific “busy” microservices when needed, rather than scale up the entire application. Such support is ideal to further decrease the operation costs.
- **Tools:** The elasticity of microservices is further amplified by the availability of open-source tools, such as Docker container [31], which makes it possible to deploy and duplicate each individual microservice; Prometheus [7] and Grafana [3], which provide monitoring, alerting, and visualization framework for analyzing the running microservices; and Kubernetes cluster manager [20], which facilitates automated deployment, scaling, and management of microservices. For many organizations, availability of these tools enables (or substantially simplifies) the transition to microservices.

Yet, just “jumping on the microservices trend”, expecting that the transition

itself will allow a company to achieve improvements similar to those reported by Amazon and Netflix, is a false belief [47]. “Proper” adoption of microservices requires several technical and organizational changes companies need to consider. The goal of this research is thus to investigate best practices, lessons learned, and current challenges in employing microservice-based architectures in industry.

To achieve this goal, we conducted a broad interview study, which involved practitioners developing microservice-based applications for several years in successful industrial companies. By deliberately excluding microservice newcomers and focusing the study on “mature” teams, we believe our work benefits practitioners who are interested to learn from each other, to borrow successful ideas, and to avoid common mistakes. The work also informs researchers of challenges in developing microservice-based applications, which can inspire novel software engineering methods and techniques.

## **1.2 Findings Summary**

In a nutshell, our participants learned that following the “standard” advice of splitting microservices based on business capabilities and using the most appropriate programming language for each microservice is not always fruitful. While they started with such an approach, they later had to redefine and merge microservices, e.g., because the resulting product consumed an unacceptable amount of computing resources, and also restrict the number of languages that they use, e.g., because the end product was difficult to maintain.

They also learned that an early investment in a robust infrastructure to support automated setup and management, extensive logging and monitoring, tracing, and more, are some of the main factors contributing to the success of their development processes. Several practitioners indicated that while in practice they delayed setting up a solid infrastructure, they regretted such decisions in the future. They found that having a well-defined owner for each microservice is essential for the success of their development process, as this owner is the one to ensure the architectural integrity of the microservice, be the first to efficiently troubleshoot the microservice,

etc.

Our study also identifies several common challenges related to managing code of microservices, where no explicit guidelines are available as part of the microservices “cookbooks”. These include decisions on how to deal with code shared by several microservices, e.g., for authentication and logging, how to manage products consisting of multiple customer offerings, and more. Even though solutions to these challenges might exist in the software engineering field in general, they are still to be studied and adapted in the particular context of microservices. Our study lists different possible solutions applied by the studied companies and discusses the trade-offs related to each solution.

### **1.3 Novelty and Contribution**

In the past few years, several studies focused on the process of transitioning to microservices [61, 22, 27, 12, 9, 10, 24, 64]. These studies identified a number of challenges related to the transition, such as difficulties around identifying the desired service boundaries, costs involved in setting up the infrastructure to deploy and monitor microservices, and the need for skilled developers to design, implement, and maintain a large distributed system.

In our work, rather than focusing on the transition to microservices, we investigate the experience of companies that have been successfully running microservices for several years. Specifically, we focus on fully operational development teams of companies serving the needs of their commercial customers by using microservice-based solutions.

We report on best practices, lessons learned, and trade-offs we collected by conducting 21 semi-structured interviews with participants from their subject companies. All of the study participants are experienced software developers, with 12 years of experience on average, and all have at least two years of experience developing microservice-based applications (see Table 3.1).

Our study can benefit industrial practitioners who develop such applications and are interested to learn from each other, to borrow successful ideas, and to

avoid common mistakes. Moreover, a description of current practices and challenges practitioners face can inspire researchers and tool builders in devising novel software engineering methods and techniques.

To summarize, this work makes the following contributions:

- It outlines the state-of-the-art in developing microservices, as described in the literature (Chapter 2);
- It describes the main characteristics of microservice-based applications and outlines the rationale for using microservices (Chapter 2);
- It identifies best practices, lessons learned, and challenges in developing microservices, as viewed by our study participants, in terms of architectural considerations (Chapter 4), infrastructure support (Chapter 5), and code management (Chapter 6);
- It discusses the findings and outlines possible next steps for both researchers and practitioners (Chapter 7).

## **1.4 Structure of This Thesis**

The thesis is structured as follows:

- Chapter 2 introduces the concept of microservices, describes their main advantages, and reviews the related studies of microservice-based development practices;
- Chapter 3 describes the methodology of the study;
- Chapter 4, 5, and 6 report the best practices, lessons learned, and challenges in developing microservices, as viewed by our study participants;
- Chapter 7 discusses the findings of the study and possible future directions for microservice-based development;
- Chapter 8 summarizes and concludes the study.

# Chapter 2

## Background and Related Work

In this chapter, we provide a brief introduction to microservices, outline their promises, and then discuss the related studies of microservice-based development practices.

### 2.1 Microservices and Their Characteristics

A microservice-based approach promotes building a single application as a suite of (micro-)services<sup>1</sup> which run in separate processes and can be deployed and scaled independently [33]. Microservices communicate with each other – synchronously or asynchronously – via lightweight language-agnostic protocols, such as HTTP REST [18]. The services are split following business capabilities. Each service has a fully automated pipeline and is independently deployable [33].

As discussed by Lewis and Fowler [33], a typical microservice-based architecture has the following characteristics:

- **Componentization via services.** A component is defined as a software unit that is independently replaceable and upgradeable. In microservice-based architectures, the primary way of componentizing the software is by breaking it down into services. Moreover, microservice practitioners usually

---

<sup>1</sup>We use *service* and *microservice* interchangeably.

see service decomposition as a further tool to enable application developers to control changes in their application without slowing down a change.

- **Smart endpoints and dumb pipes.** Applications built from microservices aim to be as decoupled and as cohesive as possible – they own their domain logic, receive a request, apply logic as appropriate, and produce a response. The two protocols used most commonly are HTTP REST and lightweight messaging (e.g., RabbitMQ [51]).
- **Decentralized data management.** This includes decentralizing decisions about conceptual models and data storage decisions. Distributed transactions are difficult to implement and as a consequence, microservice has explicit recognition that consistency may only be *eventual* consistency.
- **Design for failure.** Applications need to be designed to tolerate service failures. Any service call could fail due to unavailability of the supplier and the client has to respond to this as gracefully as possible. Moreover, it is important to be able to detect the failures quickly and automatically restore services.
- **Infrastructure automation.** Infrastructure automation techniques, including continuous integration (CI) and continuous delivery (CD) are used to reduce the operational complexity of building, deploying, and operating microservices.
- **Teams organized around business capabilities.** Teams developing microservice-based applications are generally cross-functional and organized around business capabilities.
- **Products not projects.** A team owns the software/product over its full lifetime, rather than looking at the software as a set of functionalities to be completed.

- **Decentralized governance.** Rather than using organizational-level guidelines, teams can choose the languages and tools that work best for the functionality they want to implement.

## 2.2 Claimed Advantages of Microservices

In this section, we discuss advantages of microservices from two aspects: technical and organizational.

**Technical.** Microservices aim at shortening the development lifecycle while improving the quality, availability, and scalability of applications at runtime. From the development perspective, cutting one big application into small independent pieces reinforces the component abstraction, and makes it easier for the system to maintain clear boundaries between components [21]. Developers are able to reason about the system without people having the opportunity to break abstractions. In monolithic application development, developers might “take shortcuts” and access code at the wrong layers, while in microservices, APIs specified in the service contract are the only channel for other teams to access the service.

Moreover, as different microservices can be developed using different languages, frameworks, and tools, developers have the freedom to choose languages that work best for the functionality to implement and also easily experiment with new technology stacks on a small piece of the system, before migrating the whole application.

At runtime, microservices can be individually scaled by adding more instances of only those microservices that experience increasing traffic. Moreover, while a small change in part of the monolithic application could cause the whole deployment to fail, single defective microservice would not crash the entire application, as long as the service is resilient.

**Organizational.** Independent deployment reduces communication and coordination effort needed to align on common delivery cycles. In monolith, teams have to release everything at once, whereas microservices can be deployed in different

speed, following different timetables. The release of one microservice would not get blocked by unfinished work in another microservice. Moreover, the failure domains of microservices are rather independent; if one microservice failed, only the specific team responsible for the microservice need to be notified.

Developers can also focus on small parts of an application, without the need to reason about the complex dependencies and large code base. This is especially beneficial for junior developers and for onboarding new team members. Dealing with legacy software also becomes easier. As the application is broken down into smaller, replaceable pieces, they become easier to understand and upgrade.

## **2.3 Microservices vs. Collaborative Software Development**

Microservice-based systems add additional complexity to “typical” collaborative software development. These systems are distributed, their components are loosely-coupled and communicate over the network, and are composed at runtime. Therefore, tracing the interaction between the components, identifying inter-service faults, and monitoring the behaviour of the entire system are more difficult.

Moreover, such systems consist of multiple independent executables, allowing the flexibility of adding new components to the system and requiring setting up the development pipeline and workflow for each such component. Thus, automation is especially important to reduce operation cost.

These properties of microservices complicate development and call for software development tools and practices uniquely suitable to support microservice-based engineering, which we investigate in this thesis.

## **2.4 Existing Studies on Microservice-Based Development**

We divide the discussion of related work on microservice-based development into exploratory studies and literature surveys. Overall, most existing studies focused



on the process of migration to microservices. In contrast, we collected experiences, challenges, and lessons learned by organizations in the post-migration phase. We also included companies that did not perform any migration but built their systems using a microservice-based architecture from the start.

### **2.4.1 Exploratory Studies**

The closest to ours is the study of Taibi et al., [61] who surveyed 21 practitioners from industry who adopted a microservice-based architecture style for at least two years. The goal of that survey was to analyze the motivation as well as pros and cons of migration from monolithic to microservice-based architectures. The authors found that maintainability and scalability were ranked as the most important motivations, that the return on investment in the transition is not immediate, and that the reduced maintenance effort in the long run is considered to compensate for non-immediate return on investment. Unlike that work, our study does not focus on the migration process. We also conducted in-depth interviews rather than structured surveys, which allowed us to identify and discuss challenges in a higher level of technical detail.

Like Taibi et al., Francesco et al. [22] also investigated the practices and challenges in migration to microservices. The authors did that by conducting five interviews and then followed up with a questionnaire. Similar to our study, they also observed that finding a proper service granularity and setting up an initial infrastructure for microservices, are some of the challenges that the developers face. Again, while that study only focused on the initial stages of a transition to microservices, our work covered organizations in advanced adoption stages. Performing in-depth interviews with a substantially higher number of practitioners in post-migration stages led us to discover additional challenges not mentioned in earlier work, e.g., the need to redefine service granularity in resource-constrained environments, issues related to variant and common code management, handling API updates, and more.

Zhou et al. [68] focused particularly on fault analysis and debugging of mi-

croservices. The authors interviewed 16 practitioners from 12 companies, investigating the fault cases these practitioners encountered in their microservice-based systems, their debugging practices, and practical challenges they faced when debugging microservices. The authors further created a benchmark microservice-based system and replicated all fault cases extracted from the interviews. Then, the authors evaluated the effectiveness of current debugging practices via an empirical study. This study particularly focused on the debugging aspect of microservices, while our study is more general and it investigates practices and challenges in the development, management, and maintenance of microservices.

Gouigoux et al. [27] described the lessons learned during migration from monolith to microservices in a French company named MGDIS SA. The study focused on how the company determined a suitable granularity of services, and their deployment and orchestration strategies. Rather than one case study, we collect practices and challenges of multiple companies developing different types of applications. Moreover, apart from collecting the list of challenges, we make extra efforts enumerating and comparing all emerged alternative solutions for each such challenge, to help practitioners and researchers get a better view of current options for addressing the challenges.

Likewise, Bucchiarone et al. [12] reported on experience transitioning from monolith to microservices in Danske Bank, showing that such transition improved scalability. Similarly, Luz et al. [40] shared their observations on the transition process of three Brazilian Government Institutions. The findings were cross-validated by surveying 13 practitioners in the studied institutions. The authors found that the lack of understanding on how to decompose a monolithic system into a service and how to evaluate quality properties of a microservice are commonly perceived as challenges in transition process. Balalaie et al. [9] also described the migration of a commercial mobile backend application to microservices, focusing on DevOps adoption practices that enabled a smooth transition for the company. Our study does not focus on the migration phase, collects best practices and challenges from multiple companies successfully completing the migration, and

discusses development challenges in a greater level of detail.

Balalaie et al. [10] reported on a set of migration and rearchitecting design patterns that the authors empirically identified and collected from industrial-scale software migration projects, such as decomposing monolith based on data ownership, transforming code-level dependency to service-level dependency, introducing internal and external load balancers, and more. The authors also presented three independent industrial case-studies, in which they observed the recurrence of the proposed pattern, demonstrating the practical value of these patterns.

Viggiato et al. [63] collected the general characteristics, advantages, and challenges of microservice-based development from literature and online posts. The authors then conducted an online survey to confirm, in practice, if practitioners developing microservices agree with the literature information. Similarly, Ghofrani and Lübke [24] also conducted an online survey with 25 practitioners who answered five questions related to goals and challenges in developing microservices, notations used by practitioner to describe microservices, and concerns practitioners have with using third-party libraries in their project. Our study has a broader scope; its comprehensive and open-ended nature not only allows us to discover issues not observed by previous work, but also sheds light on concerns of organizations of different type and size.

Razavian and Lago [53] explored migration activities when transitioning to SOA. The authors identified high-level activities, such as code analysis, architectural recovery, and business model recovery, and traced these activities to practices developed by the academic community. Our study focuses on more fine-grained development activities and investigates challenges specific to management of microservices.

## **2.4.2 Literature Surveys**

Pahl and Jamshidi [50] and Francesco et al. [23] performed systematic mapping studies aiming to classify research approaches to architecting microservices and proposed classification framework for categorizing, comparing, and evaluating

research work on microservices architectural solutions, methods, and techniques. Both work stated the demand for empirical studies involving practitioners to better understand the state of the practice on microservices, and our work serves such need.

Dragoni et al. [15] reviewed the history of software architecture, discussed characteristics of microservices, and outlined future challenges. The survey primarily addressed newcomers to the discipline and did not ground the discussion on any particular case study or experience.

Alshuqayran et al. [1] conducted a mapping study for identifying architectural challenges, popular microservice architectural diagram types, and quality measurement in microservices architectures mentioned in the literature. Vural et al. [64] also undertook a systematic mapping study aiming to find out current trends around microservices, the motivation behind microservices research, emerging standards, and the possible research gaps. Our work is orthogonal to that as we focus on identifying challenges of adopting microservices in industry rather than surveying the state-of-the-art in research.

# Chapter 3

## Study Methodology

This chapter describes our study methodology: the selection of subjects, the approach to data collection and analysis, and threats to the validity of the work.

### 3.1 Subjects

To gain a better understanding of current practices and challenges of microservice-based development, we recruited software developers with solid experience developing microservice-based applications. More precisely, our selection criteria were for participants who:

1. Have more than two years hands-on experience using microservice-based architecture in industry.
2. Are a member of a team that designs, develops, and deploys microservices for commercial use for at least two years.
3. Are familiar with processes and ways of interacting with other teams working on the same product.

Such selection criteria ensure that the interviewees are experienced developers from organizations that use microservices in a mature way.

For identifying the participants, we initially approached our network of collaborators and colleagues. We also reached out to developers who actively participate

in various microservice-related events and meetup groups [43]. In addition, we used the LinkedIn web platform [38] to recruit developers listing microservices as their core skills and holding active software development positions. Finally, we applied snowballing [26], asking the interviewees to distribute a call for participation in their professional networks. We interviewed each participant and stopped recruiting new participants when we reached *data saturation* [8]: we did not hear new concerns in the last five interviews.

As the result of this process, we interviewed 21 practitioners from 15 companies. Most participants hold a title of Software Engineer or Developer, Senior or Principal Software Engineer, or Software Architect. They have between 2.5 and 22 years of full-time software development experience, with a mean of 12.4 and a median of 12 years. In the microservices domain, the participants have between 2 and 10 years of development experience, with a mean of 3.7 and a median of 3 years. They are currently employed in teams that are practicing microservice-based development between 2 and 15 years, with a mean of 4.6 and a median of 4 years. Notably, some of the teams developed microservice-style systems long before the term was introduced and popularized.

The size of the teams ranges between 4 and 20 members, with a mean of 9.2 and a median of 8.5 members. The teams are responsible for 4 to 50 microservices, with a mean of 16.6 and a median of 15 microservices. They develop human resources applications, retail and social media portals, enterprise resource management and cloud resource management applications, cloud infrastructure management and IoT management applications, and more. Table 3.1 also presents demographic information about the participants. Their ages range between 25 and 47 years, with a mean of 35.3 and a median of 35 years. One participant holds a PhD degree, four hold a Master’s level degree, and 14 hold a Bachelor’s level degree. One participant, with 20 years of experience, is entirely self-taught.

**Table 3.1: Interviewee Demographics**

Item	Value
Industrial Experience [years]	Min=2.5; Max=22; Mean=12.4; Median=12
Microservice Experience [years]	Min=2; Max=10; Mean=3.7; Median=3
Team Microservice Experience [years]	Min=2; Max=15; Mean=4.6; Median=4
Number of Microservices (Team)	Min=4; Max=50; Mean=16.6; Median=15
Team Size	Min=4; Max=20; Mean=9.2; Median=8.5
Age	Min=25; Max=47; Mean=35.3; Median=35
Education Level	Bachelor's=14; Master's=5; PhD=1; Self-taught=1

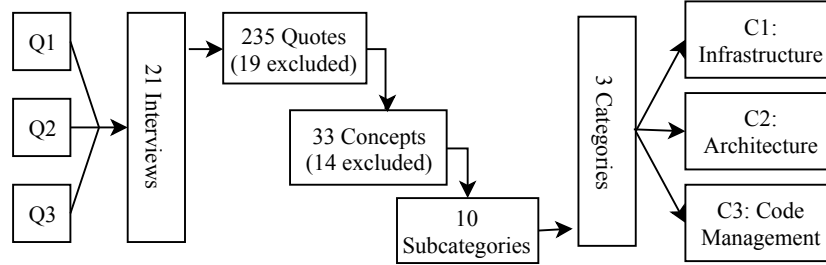
## 3.2 Data Collection

The essence of our data collection and analysis process is outlined in Figure 3.1. The study was based on semi-structured interviews with a set of open-ended questions. To identify an appropriate study protocol, we performed five pilot interviews with colleagues, friends, and student interns employed in organizations that develop microservices. These participants were not intended to satisfy our selection criteria but rather help us clarify, reorder, and refine the interview questions. We proceeded to the main study only when the pilot interviews ran smoothly; we discarded the data collected during the pilot study and did not include it in our data analysis.

For the main study, we conducted 21 semi-structured interviews that took around 50 minutes each (min=26; max=90; mean=49.9; median=47 minutes, total=17.5 hours). As typical for this kind of study, the interview length was not evenly distributed. At the beginning, the interviews were substantially longer, with many follow-ups on open-ended questions. As the study progressed, we repeatedly heard similar concerns and thus the interviews became shorter. We collected quantitative data about the participants' background, their project, and team offline, which also saved time from interviews. Each interview revolved around three central questions:

1. How, why, and when do you create new microservices?

**Figure 3.1:** Data Collection and Analysis Process



2. How are microservices maintained, evolved, tested, and deployed to production?
3. Which of your practices work well and what you think can be improved?

We followed up with subsequent questions and in-depth discussions that depended on the interviewees' responses. Our goal was to identify best practices, lessons learned, and the set of challenges practitioners face when developing and maintaining microservice-based applications.

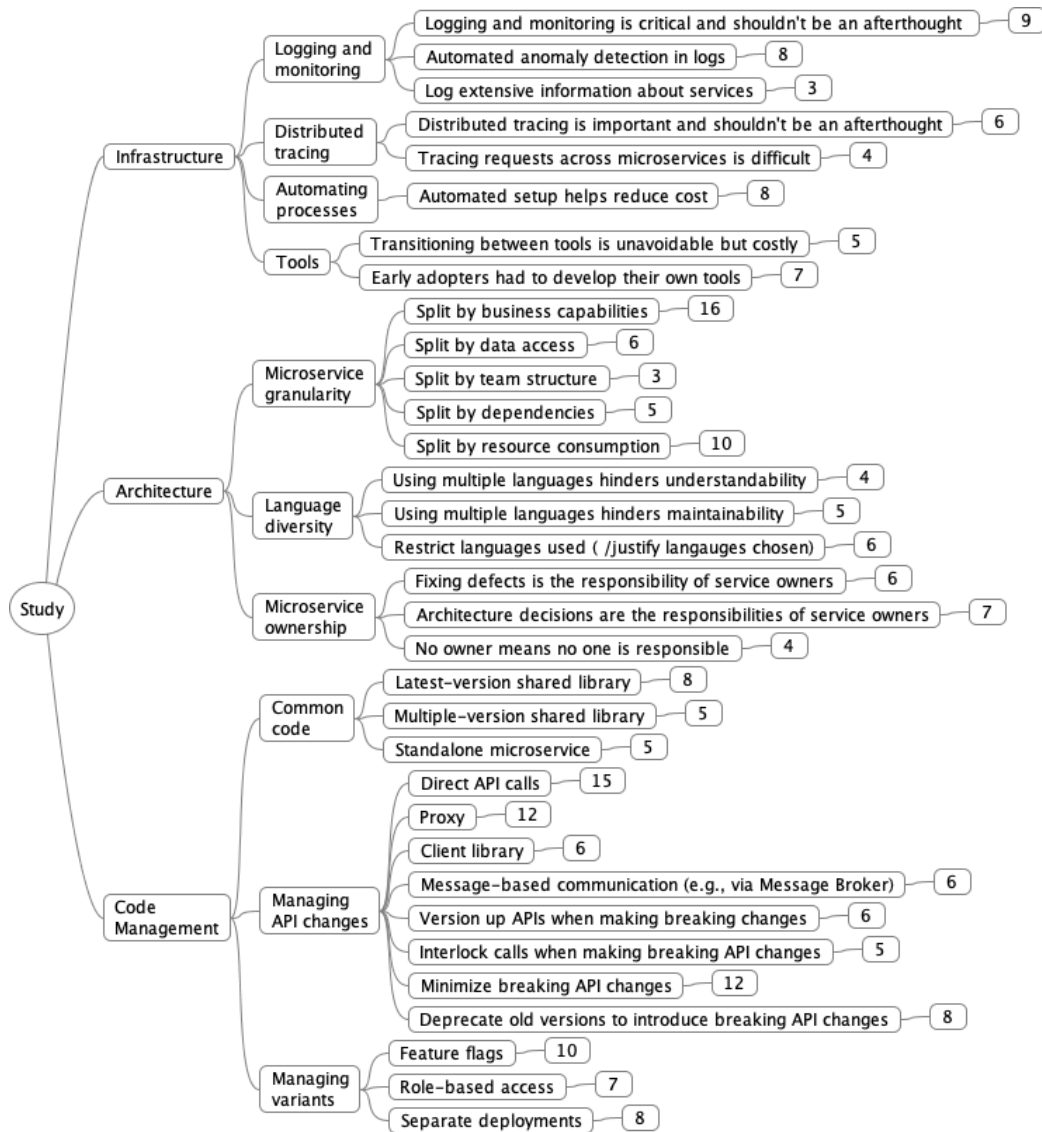
Our study team consists of three investigators in total: the author of this thesis, the author's supervisor, and another MASc student with microservices and qualitative studies background.

The interviews were conducted in English by at least two investigators of this work. Four interviews were in person and the remaining ones – over the phone or using telecommunication software, such as Skype. All but three interviewees agreed to be recorded and the collected data was further transcribed word-by-word, only removing colloquialism such as “um”, “so”, and “you know”, and breaking long sentences into shorter ones. The additional three interviews were summarized during the conversation.

We shared the transcripts with each corresponding interviewee for his or her approval or corrections. In total, we received five corrections; most of them were minor and related to the names of companies and tools, confidentiality-related issues, and clarifications on the discussed topics. We applied all corrections to the



**Figure 3.2: Mapping of Categories to Concepts**



transcripts.

### 3.3 Data Analysis

To analyze the data, we used *open coding* – a technique from grounded theory for deriving theoretical constructs from qualitative analysis [60]. More specifically,

**Table 3.2:** Concept Frequency

Category	# Participants
<b>Architecture</b>	
Microservice granularity	15
Microservice ownership	9
Language diversity	7
<b>Infrastructure</b>	
Logging and monitoring	11
Distributed tracing	6
Automating processes	5
Tools	6
<b>Code Management</b>	
Common code	14
Managing variants	12
Managing API changes	20

two of the investigators independently read the transcripts line by line and identified *concepts* – key ideas contained in data. When looking for concepts, we searched for the best phrase that describes conceptually what we believe is indicated by the raw data.

On a weekly basis, all investigators met to discuss the identified concepts and to refine and merge them if needed. As mentioned earlier, we proceeded with more interviews as long as new concepts were detected, i.e., until we reached data saturation and no new concepts emerged in the last five consecutive interviews. Such interrelated data collection and analysis process is inspired by grounded theory and is typical for interview studies.

In total, we identified 47 concepts, linked to 254 quotes. As qualitative analysis seeks to find significant concepts and explore their relationships [60], we only considered concepts that capture ideas mentioned by at least three study participants, excluding from the final results 19 quotes belonging to 14 concepts. We ended up with 235 quotes and 33 concepts, as shown in Figure 3.1. We further abstracted concepts into ten higher-level subcategories, which we grouped into

three main categories: architectural considerations, infrastructure support, and code management.

Figure 3.2 further elaborates on these numbers, giving the detailed mapping of categories to concepts. Due to confidentiality issues, we cannot list the full quotes; we thus give the number of quotes for each concept.

Table 3.2 lists the identified categories/subcategories and the total number of participants whose quotes have contributed to these categories. As our study has a qualitative nature, all the reported numbers have informative rather than statistical nature. As all our findings are linked to quotes extracted from the interviews, results we report are grounded on the collected data. In the rest of the manuscript, the quotes are presented in «this style».

For quality control, we sent the final report summarizing our findings to all interviewees, asking them to comment on any misinterpretations that might have occurred. The feedback we received shows that the study is accurate and representative: “Overall, I think this is a very strong study that accurately reflects my experience in the industry.”

### **3.4 Ethical Considerations**

The University of British Columbia (UBC) requires that research involving human participants must be reviewed and approved by the UBC-affiliated Research Ethics Board (REB) prior to data collection. Prior to the start of our data collection, we obtained an ethics approval from UBC’s REB. In the ethics application, we summarize the intended purpose of our study, potential participants and recruitment strategy, informed consent process, possible conflicts of interests, data collection strategies, confidential data handling, etc.

Before each interview session started, the participant was required to review our REB approved consent form and give us an informed consent to conduct the study; we also ask for consent for recording the interview in the consent form.

## **3.5 Threats to Validity**

Following the validity classification scheme described by Wohlin et al. [66], we discuss the threats to the validity of this study from the following aspects: external validity, internal validity, and construct validity.

### **3.5.1 External Validity**

External validity is concerned with the conditions that limit the generalization of our findings. As in many other exploratory studies in software engineering, our research is inductive in nature and thus might not generalize beyond the subjects that we studied. Yet, our sample is large enough and diverse enough to give us confidence that it represents central and significant views. We intentionally included in the study software developers from companies of different type and size. We also interviewed practitioners in different roles – from software developers to team leads and managers. We believe that these measures helped to mitigate this threat.

### **3.5.2 Internal Validity**

Threats to internal validity can affect the causality of the results due to the influences on the independent variable that are not acknowledged by the researchers. In our case, we might have misinterpreted participants’ answers or misidentified concept and categories, introducing researcher bias to the analysis. We attempted to mitigate this threat in two ways. First, all data analysis steps were performed independently by at least two investigators of the work; furthermore, all disagreements were discussed and resolved by all investigators. Second, we shared both the “raw” data collected during the interviews and the resulting report with the participants of the study for their validation. We thus believe our analysis is solid and reliable.

### **3.5.3 Construct Validity**

Construct validity concerns whether the study builds up the correct operational measures for the concept being studied. Since our goal was to understand the best practices, lessons learned, and challenges in using microservices, we have to approach industrial practitioners from different companies, thus we chose to conduct an interview study as it serves our goal best.

# Chapter 4

## Architecture






In this chapter, we discuss architectural considerations of microservice-based development, including concepts related to microservice granularity, ownership, and languages used.

### 4.1 Microservice Granularity

Identifying the right granularity for a microservice is probably the most frequent question raised by both microservice newcomers and experienced microservice developers. Table 4.1 shows different, not necessarily mutually exclusive strategies organizations apply, according to our study.

The majority of the study participants follow the common guidelines stating that business capabilities are the primary consideration for defining a microservice:

**Table 4.1:** Microservice Granularity

Alternatives	# Participants
Business capabilities	 14
Data access	 5
Dependencies	 4
Team structure	 3
Resource consumption	 3

«It just does one thing and does it really well.» The size of the microservice in terms of lines of code is a less important consideration: «Having a big service is not that bad, as long as the service is cohesive and it deals with one thing.»

Looking at service dependencies and grouping together functions that talk to the same APIs leads to minimizing service-to-service communication – another consideration expressed by the study participants: «The key is to have correct separation of concerns, such that [...] the service-to-service communication is minimized. Otherwise, you just pulled in a monolith again with network traffic between the various systems.»


Grouping together functions that need access to the same data is another well-known guideline that enforces separation of concerns. In addition, it ensures that only developers with the required permissions can access the appropriate data: «Machine learning has access to user data, so we want to separate it out from [other] microservices. [...] We have a couple of people who have the responsibility and the ownership of that code.» In fact, several practitioners mentioned that team structure and capabilities help them decide on the microservice boundaries: «If [code] is very frontend-centric, then we will probably have teams of people who think about web development and we will say that is a separate service. With the data-centric thing, we might have a different group of people with different set of skills that they work on that thing.» Another participant noted: «The size of a microservice matters only as it relates to the size of the team that can support it. [...] that matters more than just saying, ‘Well, this microservice needs to handle these six methods only’ or something like that. It’s more about team size.»

Yet, we observed multiple cases where companies that initially split their microservices based on these commonly accepted guidelines had to later revisit this decision. The most prominent is the concern related to computing resource consumption, i.e., CPU, memory, and disk space. Consumption of these resources might increase substantially when common libraries are duplicated in many individual containers, e.g., for executing common operations, such as authentication and database access, in each individual microservice. Excessive resource consumption

increases costs for companies that deploy their solutions in a pay-as-you-go cloud environment.

Furthermore, some companies need to deploy their solutions on proprietary hardware, their own or of their customers. When applications exceeded the hardware capacity allocated by the customer, companies had to merge microservices: «[Our] customers have more restricted resources. Splitting up to many microservices scared them away because each microservice takes a certain amount of resources. [...] We were able to trim down more than half of the size of that original module; we are still doing that today.»

Closely related to resource consumption is the anticipated microservice utilization criterion: «We do not want too little traffic to a microservice. If we just serve for one particular page that is not visited frequently, we are wasting our resources starting up that microservice doing nothing for most of the time.»

 **Lesson learned #1:** *Apart from the common practice of considering business capabilities and data access when deciding on microservice granularity, developers should also consider team structure, resource consumption, and communication patterns between microservices.*

## 4.2 Microservice Ownership

Some of the interviewees strongly believed that having a clear owner for each microservice is a necessity for functional organizations. That is in contrast to the currently becoming popular practice of feature-based teams that can “touch” any microservice when implementing a new feature.

A service owner is a person or a team, who is the primary point-of-contact when the microservice malfunctions: «Generally what happens is that if a service isn’t working, that service’s owners or maintainers are informed and they are asked to look into it.» Our participants report that, apart from troubleshooting, fixing bugs, and deciding on the design and architecture of the microservice, service owners also need to implement new features and train trusted committers for the



service.

Having service owners is important for defining clear architectures, with well-defined service responsibilities and boundaries, and reducing architectural debt [35]. When changes to a microservice are proposed, the owner is responsible for assessing how appropriate the change is and, if not appropriate, deciding on an alternative way to satisfy the demand: «Because there wasn't that sense of ownership, there wasn't continual like 'this is the purpose of this service and it serves only that purpose'. It started being '[this service] gonna do this, and this, and this, and that'. And it did not do either of them particularly well as opposed to doing one thing really well.»

💡 **Lesson learned #2:** *Assigning owners to microservices facilitates efficient troubleshooting and microservice architectural integrity.*

## 4.3 Language Diversity

Being polyglot is one of the most advertised advantages of microservices, as it gives developers the flexibility to choose the technology stack and languages that work best for their needs, as well as the ability to try new languages on small components without affecting the whole system.

However, introducing many languages and frameworks may actually decrease the overall understandability and maintainability of the system: «We said, 'Hey, why not try using Golang? Why not try using Elixir?' [...] so we wrote a service in that language. But what happens then is you end up with one, maybe two people who will understand it and nobody else can read the code or they struggle to read it, so it makes the service a little bit less maintainable. And nobody wants to touch it, and that is a big problem.»

Introducing many languages also makes the system harder to test: «We've got some ColdFusion parts. We've got JS. We've got some Golang, we've got some Python. So now you've got all these different microservices [...] they're using multiple different tools for testing and at the same time to test the entire flow from

front to the end, it's quite difficult.»

As a result, several companies we interviewed now decided to standardize their processes and restrict the development for one language for each “purpose”: «We are trying to converge on three main languages, JavaScript in the front-end, Scala for data, and Ruby on Rails for business logic. When we are building a new service, we pick one of those three languages depending on what seems the most logical for the problem that we are solving as well as the team that is gonna be implementing it.»

Another justification for steering away from multiple languages and technologies is code and knowledge sharing: «We ended up saying, ‘Yeah, we kind of do [standardize languages] just because of the shared knowledge’. You can reuse ideas from one microservice to another even though all the documents say that can be a really bad idea because it limits creativity and it limits your ability to try out new ideas and stuff. And we are like, ‘You know what? We don’t wanna do that. We just want to have a small number of different technologies and live with it.’»

💡 **Lesson learned #3:** *For practical purposes, organizations should restrict the number of programming languages used in a microservice-based system to a few core languages: one for each high-level “purpose”.*

# Chapter 5

## Infrastructure

As microservice-based applications consist of numerous independent components, tracking their availability and performance, debugging the entire system, and using an appropriate setup and maintenance infrastructure are critical activities, according to our study participants.

### 5.1 Logging and Monitoring

One of the main criteria for a mature microservice-based development process is the robustness of the logging and monitoring framework. Our participants report that, apart from service availability and the number of requests, they log input, output, and error data, response time, resource utilization, and more. Identifying the right granularity level for logging is not a trivial task. Not logging enough will cause problems when troubleshooting failures; logging too much might harm the application performance. Several participants also stated that, when logging, particular care is needed to preserve client privacy, e.g., by obfuscating logged data. «Having metrics in place and reviewing them, making sure that they are the right metrics, those are all things that get reviewed by the team ahead of time before you actually launch.»

Practitioners use logged info for troubleshooting, monitoring and visualizing

the overall health of the system, creating reports for customers, e.g., on the delivered quality of service, and compliance and auditing reports. Moreover, some run statistical analysis on the logged info, to automatically identify failures, even before customers notice them, and notify the corresponding team: «Any time that the logs record a failure, we have tools to find out where there were a lot of failures in this particular period of time, what percentage of failures there were, was the latency for that call abnormal or something, did it exceed a certain threshold over a certain period of time, etc. We have a tool that would monitor those logs and send us alarms when those thresholds were exceeded, for example, latency thresholds or success thresholds.»

Practitioners often regret the decision not to set up a logging and monitoring framework early as the project starts: «Probably I will focus more on logging and monitoring, right off the bat. Because trying to retrofit monitoring and logging once we have all the services, is quite a bit of work.»

💡 **Lesson learned #4:** *Logging and monitoring frameworks should be set up at the onset of the project. Developers should carefully review the logged information, to evaluate the effect of logging on the troubleshooting, monitoring, and reporting abilities, performance of the application, and client privacy.*

## 5.2 Distributed Tracing

When locating failures, most companies follow the chain of ownership, i.e., start from the failing microservice and gradually track where problems are coming from.

As requests often span multiple services and the call relationships between microservices get very complex, solutions like *distributed tracing* [55], which track services participating in fulfilling a request, are applied: «There have been issues that I have worked on which were chained through five different teams that are completely unrelated. And it was like, ‘okay, this is where the initial source of this thing came from’.»

Like logging and monitoring, setting up a distributed tracing framework is an

important task to do early as the project starts: «A lot of companies that start out, do not think about distributed tracing right from the get-go. They think about it as an afterthought. And distributed tracing is a lot harder to implement after the fact than it is when you start a project.»

💡 **Lesson learned #5:** *To efficiently troubleshoot a microservice-based application, distributed tracing should be set up at the onset of the project. It is harder to implement after the fact.*

## 5.3 Automating Processes

Building infrastructure to automatically create new microservice stubs and to add newly created microservices to the build and deployment pipeline substantially reduces the operation costs: «When we first started doing microservices, it actually took about a month or two to get all the infrastructures ready to create a new microservice. Now we have the ability to create a microservice in five minutes.» «The tooling is very important. There is kind of one way to create, at least, the structure of projects for different platforms. So like, Scala microservices, they will all look about the same. They will have the same structure. They will have the same Jenkinsfile. They will have the same format in Kubernetes, the same way to configure them, roughly. That is the core.»

Automated pipelines also help newcomers to start using microservices: «Today we are working on our starter-kit for other teams to easily get on board with the microservice-based architecture. Back to that time, we did not have a good way to easily start up a new microservice, it involved a lot of learning curves.»

Like the aforementioned infrastructures, several interviewees, especially those from large companies, mentioned that an important lesson they learned is not treating automation as an afterthought: «If we had to start microservices again, I think we would try to get the tooling for creating new services ready and begin standardizing earlier. It is easy to say this in hindsight.»

💡 **Lesson learned #6:** *For large projects, automating microservices setup help reduce operation costs.*

## 5.4 Tools

Early adopters of microservice-based architectures had to develop numerous proprietary frameworks and tools, to support development, maintenance, and deployment of microservices. Some of these tools were later contributed to open source, e.g., Docker [31] developed by Docker Inc., Kubernetes [20] and Istio [4] by Google, and Envoy [2] by Lyft. By now, there are more than 250 tools out there to support development and deployment of microservices, service discovery and API management at runtime, and more [19]. For smaller companies, building proprietary tools no longer makes economic sense: «In the time that they were trying to build their own, Kubernetes came about, and I think they realized now that they have wasted a lot of time building the toolset.»

However, our study participants noticed that identifying an appropriate existing solution takes time, which they usually lack as the development is driven by business needs: «There are tools that are out or coming out that are solving a lot of problems that we have. Things like gRPC, GraphQL, code generation and documentation, service meshes, [...], they are great at solving a lot of problems. We just do not have the time to actually move over to them.»

An advisable strategy is to stick to vendor-neutral interfaces, when possible, which decouples the implementation from vendors. For example, instead of instrumenting the code to call a particular distributed tracing vendor, such as Zipkin [49] or Jaeger [5], using the vendor-neutral distributed tracing framework OpenTracing [6] facilitates the transition between vendors.

💡 **Lesson learned #7:** *When available, choose vendor-neutral interfaces to avoid vendor lock-in.*

# Chapter 6

## Code Management




In this chapter, we discuss considerations related to managing code of microservice-based applications. In particular, we present alternative strategies our participants apply for dealing with code shared by multiple microservices, managing evolving APIs, and developing applications consisting of multiple variants serving different types of customers.

### 6.1 Common Code

Splitting an entire system into a set fully independent microservices is not realistic in practice. Virtually all study participants stated that some functionalities and code needs to be shared between microservices. These include cross-cutting concerns such as logging and authentication, database access, common utilities, and more. Simply duplicating code in multiple microservices is against the “do not repeat yourself” (DRY) principle of software development [30] and will make code unmanageable in the long run. We observed three alternative solutions for managing common code (see Table 6.1).

The most common practice participants employ to minimize code duplication observed is to package the common code in a shared library, which is imported at build time. In the simplest case, all microservices use the same latest version of the

**Table 6.1:** Common Code

Alternatives	# Participants
Latest-version shared library	 6
Multiple-version shared library	 4
Standalone microservice	 4

library. Its downside is that a microservice cannot make independent changes to that common code: «That is a [...] hassle because you change the common library and then all the services that depend on this library need to change. [...] you have to coordinate carefully and make sure that you do not do any breaking changes.»

To address this problem, several participants carefully version libraries and allow services to rely on different versions of these libraries. Such an approach could introduce another challenge – inconsistencies and application instability: «Hopefully, we pick the right versions so that everything works because the worst thing is when you have had transitive dependencies, and their version clashes with something else that another library brings in. So dependency management can be painful.»

As a compromise, participants agree that new changes should always be introduced only in the latest version of the library. If a microservice requires a feature, it has to import the newest version of the library: «We would not support changes in the older version. We say, ‘If you are using version two and you want the new features, you have to upgrade to version three first before you get new features.’ Everybody has to be converged on the same mainline version.»

A number of interviewees opted for wrapping common code in a standalone microservice. This approach simplifies procedures as no redeployment of all microservices that use the changed library is required: «If you put this common code in a JAR and then [...] there’s a bug, you have to redeploy those [microservices that use the JAR]. But if you have it [common code] outside [as a service], you just deploy it once, that would be enough.» Yet, the clear disadvantage of this approach is the introduction of network delays when executing the common code, which affects application performance.



The other solution to the common code issue is sidecar, which has recently emerged from the industry. Since such a strategy was mentioned by one participant only, we did not include it in Table 6.1. Yet, we believe that this solution is worth highlighting. The main idea behind sidecar resembles a sidecar attached to a motorcycle: it co-locates common code implemented as a standalone microservice with the primary microservice, dynamically attaching it to the main container.





Using sidecar helps to avoid network overhead (as it runs on the same host as the primary microservice) and solve the deployment problem (as sidecar can be updated independently from the primary microservice). Moreover, it provides a homogeneous interface for services across languages: «Applications do not have to embed libraries and then encode anymore, they can use a container-based solution that is going to provide common functionality. And that way, if one developer makes a change to that common component, people can just pull down the latest container image and run it as a sidecar. They do not need to go and get a new version of the code to compile under their applications.»

💡 **Lesson learned #8:** *Sharing common code in the form of software libraries violates microservice independence. Such approach should be carefully managed. Participants should also experiment with the newly emerged sidecar solution.*

## 6.2 Managing API Changes

With services, the contract with downstream customers, external or internal, is done at the API level. Best practices for defining APIs are well captured by Postel's law [52]: being liberal in what you accept and conservative in what you send. Our study participants make an extra effort to avoid breaking API changes, unless those are security-related. Breaking changes that face external customers are especially discouraged: «If you break those, that's a huge problem.» To reduce the chance of breaking customer code, our participants encourage customers to ignore data they do not use: «If [the clients] receive some data that they do not need, do not break on that, just drop it because that may be a way of introducing new changes.»

**Table 6.2:** Managing API Changes

Alternatives	# Participants
Direct API calls	 14
Proxy	 7
Client library	 5
Message-based communication	 4

When a company must change an API in a way that it is no longer compatible with the original version, e.g., to support security features, they version the APIs up, eventually deprecating the old version without breaking it: «If we are going to delete something from the payload or we completely change the signature, we will have to bump up the major version and create another version of the API and ask people to move over.»

API changes are typically discussed in internal or external interlocks, documented, and systematically deployed. As shown in Table 6.2, a number of interviewed teams use direct API calls for synchronous communication between microservices. Some participants mentioned they use asynchronous message-based communication within their product or across product boundaries. They mostly rely on distributed stream-processing software, such as Apache Kafka [34] or message brokers, such as RabbitMQ [51]).

With both direct calls and messaging, when a change is introduced, it is implemented by an API with a new version number. Both the old and the new versions are deployed alongside to allow a smooth transition of the clients, and the old API is removed when clients stop using it: «You do your first deployment on the API. You do the second deployment on the calling service. And then you do another deployment on the API again to remove the old stuff.»

This strategy makes the transition simple and straightforward, but requires changes in all clients using the API. Identifying and notifying such clients is a challenging task. Various proxy solutions, e.g., API gateways [54], push the complexity to the server side.

As breaking API changes often involve simple modifications, e.g., adding a

new input parameter to the API, a proxy can be programmed to detect old API calls and transform them to a new version, e.g., by adding default values to fields not passed by the client, making them backward compatible before forwarding the requests to the new API version. In this way, clients stay oblivious to the actual version they are using, as every request goes through the gateway and the gateway just routes the request as needed.




Another solution for simplifying clients' transition to new API versions is to provide clients with a library that wraps calls to server APIs. Client libraries are typically automatically generated, e.g., using tools such as Swagger [59]. Using libraries has numerous benefits over clients making direct HTTP calls [14]. First, developers can push upgrades to the clients more easily by using package management tools. The upgrade process is also simplified for the customer as they do not need to get acquainted with the details of the changes. Finally, client libraries can handle low-level communication issues such as authentication with the API server, again, reducing the burden on the customer side.

💡 **Lesson learned #9:** *API breaking changes, especially those facing external clients, are discouraged. When needed, e.g., for security upgrades, they should be introduced via deprecation. APIs gateways and client libraries help to mitigate the burden of upgrades for the client.*

## 6.3 Managing Variants

Managing different customer offerings (a.k.a variants), e.g., for “free” vs. “premium” customers, is yet another challenge in developing microservices. More than half of our study participants need to manage variants of their products and they reported on three common strategies for managing product variants (see Table 6.3). Feature flags are basically conditions that are passed with requests and that control different paths in code. In the simplest form, these flags can identify a certain group of users: «It can say, ‘Oh, if you are User X, do this code’, or ‘if you are in this cohort, do this code’. We use feature flags very heavily, so we can turn on

**Table 6.3:** Managing Variants

Alternatives	# Participants
Separate deployments	 5
Feature flags	 4
Role-based access	 3

some code path for different people.»

More robust feature flags do not condition over particular users, which is hard to scale, but rather specify high-level features: «When the feature is toggled for a customer, it is more like a temporary thing where it is like a hack. [...] Typically, when it is a specific feature, that is something that is determined at the API level and passed in. It is a call to the service with feature X enabled.»

The biggest benefit of using feature flags is to be able to make changes on the fly at run-time. Companies also use feature flags to support blue-green deployments [57] and to roll-out changes to a certain group of users: «We can say, ‘Let’s push this new feature out to 1% of our user groups, 10%, and slowly see what the response is over time’.»

The disadvantages of feature flags are that it requires extensive management of feature themselves and the correspondence between features and code that implement these features. Testing different combinations of features also becomes challenging. Another approach for managing variants is role-based access which provides access to APIs based on the user role. Such an approach is typically supported by the API management tools and API gateways [54]. Each incoming request has an API key tied back to a particular API plan, and an API gateway routes the requests to a certain code, based on the developer-specified configuration.

The main advantage of the role-based access approach compared to feature flags is that this approach makes the primary functionality selection in the infrastructure layer, making it possible to deny access to certain clients, instead of performing checks in the code: «Based on the customer ID, they are allowed to use a subset of the service, when they try to call these other APIs that they are not supposed to, they will just get an error back.»

Finally, a few participants reported that they use separate deployments for different customers, on the company's or customers' sides. In detailed interviews, the practitioners explained that their projects started from using feature flags and other code-level differentiation, but that complicated code management: «Previously we were doing everything within the one microservice. [...] That became very messy very quickly, so we ended up splitting an instance out into its own branch.»

Such projects maintain different code bases for different customers: «Dedicated and local [offerings] are different because they are for only certain customers. [...] We have a branch for the dedicated and local system; when we roll out a new feature to the public system, they won't get it automatically unless we pour that feature to the branch.»

While separate branches induce the overhead of synchronizing the code bases, some practitioners opt to do that to ease the maintenance effort: «Let's say we have product v6.0, v6.1, and v6.2. We found a bug in v6.0, and need to fix the bug in all three versions. But the bug fix that is needed in the three versions can be different due to compatibility issue. When fixing the bug, we need to independently fix all versions.» Such a solution might be appropriate for cases when customer offers diverge substantially.

💡 **Lesson learned #10:** *Feature flags and role-based access for managing product variants make it possible to maintain a single code base but complicate its maintenance. Separate deployments can be considered when variants are expected to substantially diverge.*

# Chapter 7

## Discussion and Future Work

Microservices are advertised as a way to speed up development, reduce communication effort and dependencies, and increase performance and scalability of an application at runtime. Together with these benefits, there are several pitfalls and challenges related to the adoption of microservices that warrant a discussion. Similar to Chapter 2, here we discuss these challenges from two perspectives: technical and organizational.

**Technical.** Identifying proper service granularity is one of the concerns of multiple practitioners. Splitting and merging microservices is a process that is performed continuously, long after the initial microservice topology is identified. This process should be informed not only by architectural considerations, e.g., functional decomposition and fan-in and fan-out metrics, but also by additional concerns such as performance and security. Metrics and tools that help practitioners continuously assess and refactor their microservice-based architectures are needed.

Likewise, many of the microservice technologies are built with the idea of an open Web; introducing strict security guidelines, i.e., ensuring that network boundaries for calls within and between applications are properly guarded, requires more efforts than in monolithic applications. It appears that innovation in this area is mostly driven by industry, e.g., the concept of a *service mesh*, with built-in access control for service-to-service communication [4, 45]. Given the increased

attack surface and distributed ownership, research on usable and robust security models for microservice-based architectures will be fruitful.

Performance debugging for microservice is now mostly based on metrics and logs, and is mostly done ad hoc. Application performance management software for microservices also has not gained sufficient attention in the relevant communities [58, 29]. More research is needed to identify strategies for performance monitoring of a microservice-based system under continuous software change.

Maintaining different variants is another important concern for microservice organizations. While techniques like feature flags and segregation of functionality by APIs are useful, they add complexity to the development processes and make testing different combinations of features harder. Efficient management of product variants is a research topic extensively studied by the software product line research community [36]. Adapting the techniques developed by this community to the context of microservices is a fruitful research direction.

**Organizational.** Interestingly, many of our study participants believed that breaking a system into small components makes it easier to understand. While a limited scope indeed decreases the entrance barrier, it weakens the developers' understanding of the system as a whole. Under such a model, developers, in particular, the junior ones, are trained to believe that they should only care about a few microservices they directly work on. They develop the impression that it is easy(er) to debug microservices, whereas debugging a distributed system composed of multiple, independently managed and involving components, is in fact, a challenging task [39, 11].

In the realm of managing API changes, several of the study participants indicated that this process involves constant synchronization effort, e.g., daily interlocks between the teams to propose and discuss changes, and even synchronization on deploying changes simultaneously. Even though most companies adopt microservice-based architectures because of the promise to decouple the teams, this synchronous communication still takes a substantial portion of the teams' time. Tools for assessing the impact of the proposed changes as well as practices that

focus on helping teams work together better in these complex, rapidly evolving ecosystems are needed.

Our study also indicates the need for a systematic analysis of the different available tools for supporting microservice-based development. Several participants indicated that they do not have a clear strategy in selecting an appropriate tool, out of available alternatives. More systematic criteria and a set of metrics for describing, evaluating, and comparing tools to each other are needed.

Microservices are commonly thought of as an architectural style that emerged from the Agile and DevOps movements [41, 46], to solve the bottleneck of centralized delivery, to reduce the communication effort, and to shorten build-test-deploy cycles. DevOps promotes full ownership from development to production and is one of the main backbones of microservice-based architectures. Yet, for government and healthcare-related organizations that are bound by privacy laws and need special treatment in handling confidential data and personal data, implementing DevOps is not always straightforward. They often rely on contractors that develop the software, who do not have access to the company's infrastructure. Regulatory compliance products, where it is essential to produce audit trails before deployment, also complicate matters. When such organizations want to adopt microservices, they need to rethink DevOps and continuous delivery practices. We believe a more focused study on development practices and needs of such organization is required.



# Chapter 8

## Conclusion

This thesis reports on best practices, lessons learned, and challenging design trade-offs collected by interviewing 21 participants from companies that successfully adopted microservice-based architectures.

Our participants indicated that a clear sense of ownership, strict API management, automated processes, and investment in robust logging and monitoring infrastructure as some of the best practices they consider contributing to the success of their development processes. They learned that using a plurality of languages and following the advice to split microservice by business functionality is not always fruitful.

Our study identified several common challenges faced by practitioners that use a microservice-based architecture, such as identifying the proper service granularity, ways to introduce API changes, managing code shared between microservices, and managing multiple product variants. We reported on alternative solutions our study participants employ and identified potential next steps the research community can take to further facilitate efficient software engineering practices in developing microservice-based applications. These include performance- and security-aware solutions for managing design trade-offs, managing versions and variants, assessing the impact of API changes, tools for performance debugging, and more.

We believe this thesis can help software engineering researchers to better focus

their agenda when devising solutions for organizations that employ a microservice-based architecture and also be useful for practitioners that can learn from each other's experience, adopt best practices, and avoid common mistakes.

# Bibliography

- [1] N. Alshuqayran, N. Ali, and R. Evans. A Systematic Mapping Study in Microservice Architecture. In *Proceedings of IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51, 2016.
- [2] E. P. Authors. Envoy Proxy. <https://www.envoyproxy.io/>, 2019. (Last accessed: April 2019).
- [3] G. Authors. Grafana. <https://grafana.com>, 2019. (Last accessed: April 2019).
- [4] I. Authors. Istio: Connect, Secure, Control, and Observe Services. <https://istio.io/>, 2019. (Last accessed: April 2019).
- [5] J. Authors. Jaeger: Open Source, End-To-End Distributed Tracing. <https://www.jaegertracing.io/>, 2018. (Last accessed: April 2019).
- [6] O. Authors. The OpenTracing Project. <http://opentracing.io/>, 2018. (Last accessed: April 2019).
- [7] P. Authors. Prometheus. <https://prometheus.io>, 2018. (Last accessed: April 2019).
- [8] E. Babbie. *The Practice of Social Research*. Nelson Education, 2015.
- [9] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices Architecture Enables Devops: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3):42–52, 2016.
- [10] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn. Microservices Migration Patterns. *Software: Practice and Experience*, pages 1–24, 2018.

- [11] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging Distributed Systems. *Communications of the ACM*, 59(8), 2016.
- [12] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara. From Monolithic to Microservices: An Experience Report from the Banking Domain. *IEEE Software*, 35(3):50–55, 2018.
- [13] Camunda. New Research Shows 63 Percent of Enterprises Are Adopting Microservices Architectures yet 50 Percent Are Unaware of the Impact on Revenue-Generating Business Processes.  
<https://camunda.com/about/press/new-research-shows-63-percent-of-enterprises-are-adopting-microservices-architectures-yet-50-percent-are-unaware-of-the-impact-on-revenue-generating-business-processes/>, 2018. (Last accessed: April 2019).
- [14] G. Cloud. Client Libraries Explained.  
<https://cloud.google.com/apis/docs/client-libraries-explained>, 2018. (Last accessed: April 2019).
- [15] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, Today, and Tomorrow. *Present and Ulterior Software Engineering*, pages 195–216, 2017.
- [16] DZone. Programming & DevOps News, Tutorials & Tools.  
<https://dzone.com>, 2018. (Last accessed: April 2019).
- [17] J. Fenn and A. Linden. Gartner’s Hype Cycle Special Report for 2005.  
<https://www.gartner.com/doc/484424/gartners-hype-cycle-special-report>, 2005. (Last accessed: April 2019).
- [18] R. T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [19] C. N. C. Foundation. CNCF Cloud Native Interactive Landscape.  
<https://landscape.cncf.io/license=open-source>, 2019. (Last accessed: April 2019).
- [20] T. L. Foundation. Kubernetes: Production-Grade Container Orchestration.  
<https://kubernetes.io>, 2019. (Last accessed: April 2019).

- [21] M. Fowler. Microservice Trade-Offs. <https://martinfowler.com/articles/microservice-trade-offs.html>, 2015. (Last accessed: April 2019).
- [22] P. D. Francesco, P. Lago, and I. Malavolta. Migrating Towards Microservice Architectures: An Industrial Survey. In *Proceedings of IEEE International Conference on Software Architecture (ICSA)*, pages 29–38, 2018.
- [23] P. D. Francesco, I. Malavolta, and P. Lago. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In *Proceedings of IEEE International Conference on Software Architecture (ICSA)*, pages 21–30, 2017.
- [24] J. Ghofrani and D. Lübke. Challenges of Microservices Architecture: A Survey on the State of the Practice. In *Proceedings of the 10th Central European Workshop on Services and Their Composition (ZEUS)*, pages 1–8, 2018.
- [25] A. M. Glen. Microservices Priorities and Trends. <https://dzone.com/articles/dzone-research-microservices-priorities-and-trends>, 2018. (Last accessed: April 2019).
- [26] L. A. Goodman. Snowball Sampling. *The Annals of Mathematical Statistics*, 32(1):148–170, 1961.
- [27] J.-P. Gouigoux and D. Tamzalit. From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. In *Proceedings of IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 62–65, 2017.
- [28] C. Green. Agile Development is Ideally Suited for Microservices. <https://www.belatrixsf.com/blog/agile-development-is-ideally-suited-for-microservices/>, 2017. (Last accessed: April 2019).
- [29] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger. Performance Engineering for Microservices: Research Challenges and Directions. In *Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*, pages 223–226, 2017.

- [30] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing, 1999.
- [31] D. Inc. Docker: Build, Manage and Secure Your Apps Anywhere. Your Way. <https://www.docker.com>, 2019. (Last accessed: April 2019).
- [32] G. Inc. Gartner: the World’s Leading Research and Advisory Company. <https://www.gartner.com/>, 2018. (Last accessed: April 2019).
- [33] M. F. James Lewis. Microservices: A Definition of This New Architectural Term. <https://www.martinfowler.com/articles/microservices.html>, 2014. (Last accessed: April 2019).
- [34] A. Kafka. Apache Kafka: A Distributed Streaming Platform. <https://kafka.apache.org>, 2017. (Last accessed: April 2019).
- [35] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical Debt: from Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, 2012.
- [36] W. U. Lab. SPLC: Systems and Software Product Line Conference. <http://splc.net/>, 2018. (Last accessed: April 2019).
- [37] Lightbend. Enterprise Development Trends 2016. [https://info.lightbend.com/COLL-20XX-Enterprise-Development-Trends-2016-Report\\_RES-LP.html?lst=PR](https://info.lightbend.com/COLL-20XX-Enterprise-Development-Trends-2016-Report_RES-LP.html?lst=PR), 2016. (Last accessed: April 2019).
- [38] LinkedIn. LinkedIn. <https://www.linkedin.com/>, 2018. (Last accessed: April 2019).
- [39] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging Deployed Distributed Systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 423–437, 2008.
- [40] W. Luz, E. Agilar, M. C. de Oliveira, C. E. R. de Melo, G. Pinto, and R. Bonifácio. An Experience Report on the Adoption of Microservices in Three Brazilian Government Institutions. In *Proceedings of Brazilian Symposium on Software Engineering (SBES)*, pages 32–41, 2018.

- [41] M. McLarty. Microservice Architecture is Agile Software Architecture. <https://www.infoworld.com/article/3075880/microservice-architecture-is-agile-software-architecture.html>, 2016. (Last accessed: April 2019).
- [42] Meetup. Microservices: A Definition of This New Architectural Term. <https://www.meetup.com/microservices/events/195904072/>, 2014. (Last accessed: April 2019).
- [43] Meetup. Meetup: We Are What We Do. <https://www.meetup.com/>, 2018. (Last accessed: April 2019).
- [44] M. Milinkovich. Jakarta EE Developer Survey. <https://jakarta.ee/documents/insights/2018-jakarta-ee-developer-survey.pdf>, 2018. (Last accessed: April 2019).
- [45] W. Morgan. The History of the Service Mesh. <https://thenewstack.io/history-service-mesh/>, 2018. (Last accessed: April 2019).
- [46] MuleSoft. Microservices and DevOps: Better Together. <https://www.mulesoft.com/resources/api/microservices-devops-better-together>, 2018. (Last accessed: April 2019).
- [47] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, 2016.
- [48] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison-Wesley, 2005.
- [49] Openzipkin. Zipkin: A Distributed Tracing System. <https://zipkin.io/>, 2019. (Last accessed: April 2019).
- [50] C. Pahl and P. Jamshidi. Microservices: A Systematic Mapping Study. In *Proceedings of the International Conference on Cloud Computing and Services Science (CLOSER)*, pages 137–146, 2016.
- [51] Pivotal. RabbitMQ. <https://www.rabbitmq.com>, 2018. (Last accessed: April 2019).

- [52] J. Postel. DoD Standard Transmission Control Protocol. *RFC*, 761:1–88, 1980.
- [53] M. Razavian and P. Lago. A Survey of SOA Migration in Industry. In *Proceedings of IEEE International Conference on Service-Oriented Computing (ICSOC)*, pages 618–626, 2011.
- [54] C. Richardson. API Gateway Pattern. <https://microservices.io/patterns/apigateway.html>, 2018. (Last accessed: April 2019).
- [55] C. Richardson. Pattern: Distributed Tracing. <https://microservices.io/patterns/observability/distributed-tracing.html>, 2018. (Last accessed: April 2019).
- [56] C. Richardson. Who is Using Microservices? <https://microservices.io/articles/whoisusingmicroservices.html>, 2018. (Last accessed: April 2019).
- [57] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm. Continuous Deployment of Mobile Software at Facebook (Showcase). In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 12–23, 2016.
- [58] V. Seifermann. Application Performance Monitoring in Microservice-Based Systems. Bachelor’s thesis, Institute of Software Technology Reliable Software Systems, University of Stuttgart, 2017.
- [59] S. Software. Swagger: The Best APIs are Built with Swagger Tools. <https://swagger.io/>, 2019. (Last accessed: April 2019).
- [60] A. Strauss and J. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Thousand Oaks, CA: Sage, 1998.
- [61] D. Taibi, V. Lenarduzzi, and C. Pahl. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.
- [62] G. Trends. Google Trends. <https://trends.google.com>, 2018. (Last accessed: April 2019).



- [63] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo. Microservices in Practice: A Survey Study. In *Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM)*, pages 1–8, 2018.
- [64] H. Vural, M. Koyuncu, and S. Guney. A Systematic Literature Review on Microservices. In *Proceedings of International Conference on Computational Science and Its Applications (ICCSA)*, pages 203–217, 2017.
- [65] Wikipedia. Microservices.  
<https://en.wikipedia.org/wiki/Microservices>, 2019. (Last accessed: April 2019).
- [66] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.
- [67] V. Woods. Gartner Identifies the Top 10 Strategic Technology Trends for 2016. <https://www.gartner.com/en/newsroom/press-releases/2015-10-06-gartner-identifies-the-top-10-strategic-technology-trends-for-2016>, 2015. (Last accessed: April 2019).
- [68] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering*, 14(8):1–18, 2018.