Promises and challenges of microservices: an exploratory study

Yingying Wang[1] · Harshavardhan Kadiyala[1] · Julia Rubin[1]

Abstract

Microservice-based architecture is a SOA-inspired principle of 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 work, we aim at collecting and categorizing best practices, challenges, and some existing solutions for these challenges employed by practitioners successfully developing microservice-based applications for commercial use. Specifically, we focus our study on "mature" teams developing microservice-based applications for at least two years, explicitly excluding "newcomers" to the field. We conduct a broad, mixed-method study that includes in-depth interviews with 21 practitioners and a follow-up online survey with 37 respondents, covering 37 companies in total. Our study shows that, in several cases, practitioners opt to deviate from the "standard" advice, e.g., instead of splitting microservices by business capabilities, they focus on resource consumption and intended deployment infrastructure. Some also choose to refrain from using multiple programming languages for implementing their microservices, as that practice hinders reuse opportunities. In fact, our study participants identified robust and shared infrastructural support established early on in the development process as one of the main factors contributing to their success. They also identified several pressing challenges related to the efficient managing of common code across services and the support of product variants. The results of our study can benefit practitioners who are interested to learn from each other, borrow successful ideas, and avoid common mistakes. It can also inform researchers and inspire novel solutions to some of the identified challenges.

Keywords Microservices · Cloud-native applications · Development practices · Empirical study

✉ Yingying Wang
  wyingying@ece.ubc.ca

Extended author information available on the last page of the article.

1 Introduction

Microservice-based architectures became popular around 2014, when Lewis and Fowler published their blog on the topic (Lewis and Fowler 2014) and Netflix shared their expertise and lessons learned from a successful transition to microservices at a Silicon Valley Microservices Meetup (Cockroft 2014). 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. This architectural style is particularly popular in cloud-based software-as-a-service (SaaS) offerings; its advantages include greater software development agility and improved scalability of deployed applications (Lewis and Fowler 2014; Richardson 2014). Splitting large applications into individual microservices provides the next degree of independence for agile teams (Beck et al. 2001), further reducing synchronization effort and allowing each team to independently deploy changes to production at a time appropriate for the team. It also saves deployment costs by allowing to spin up only instances of individual "busy" microservices, without the need to scale up the entire application. Finally, for many organizations, the availability of open-source tools, such as Docker[1], Kubernetes[2], Prometheus[3], Grafana[4] for deploying, managing, and monitoring microservices substantially simplifies the transition. Nowadays, microservice-based architectures are adopted by many successful companies, including Amazon, IBM, Uber, LinkedIn, Groupon, and eBay (Richardson 2018c).

Yet, just "jumping on the microservices trend", expecting that the use of the new tools will allow a company to achieve improvements similar to those reported by Amazon and Netflix, is a false belief (Nadareishvili et al. 2016). Proper adoption of microservices requires several technical and organizational changes companies need to consider. In the past few years, several studies focused on the process of transitioning to microservices (Taibi et al. 2017; Di Francesco et al. 2018; Gouigoux and Tamzalit 2017; Bucchiarone et al. 2018; Balalaie et al. 2016, 2018; Ghofrani and Lübke, 2018; Vural et al. 2017; Carvalho et al. 2019; Knoche and Hasselbring 2019; Fritzsch et al. 2019; Bogner et al. 2019). 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. That is because, based on our own experience and our interactions with practitioners, we observed that while there is much emphasis on the initial microservice build, little information is available about the pitfalls and support organizations need in the long run. We thus focus our study on experiences of teams developing microservices for commercial use for more than two years.

Our investigation is driven by the following research questions:

RQ1: What are best practices learned by practitioners successfully adopting microservices for commercial use?

---

[1]https://www.docker.com

[2]https://kubernetes.io

[3]https://prometheus.io

[4]https://grafana.com

RQ2: What are challenges practitioners face?

RQ3: Which solutions to the identified challenges exist?

We report on the best practices and lessons learned we collected by conducting 21 semi-structured interviews and 37 follow-up online surveys with practitioners employed by 37 different companies. Our semi-structured interviews were designed to elicit the main concepts considered by organizations that apply microservice-based architecture. The online survey was used to validate and extend our findings by reaching out to additional practitioners. All of our study participants are experienced software developers, with 12.4 years of experience on average for interviewees and 13.4 years for survey participants. All have at least two years of experience developing microservice-based applications (see Tables 1 and 2).

We used qualitative data analysis techniques borrowed from grounded theory (Strauss and Corbin 1998), namely *open and axial coding*, for identifying and categorizing the main concepts reported by our study participants. In a nutshell, for **RQ1**, 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, is one 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 it, etc.

Answering **RQ2**, our study also identified 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. For **RQ3**, our study lists different possible solutions applied by the studied companies and discusses the trade-offs related to each solution.

Contributions. To summarize, this paper makes the following contributions:

(1) It describes the main characteristics of microservice-based applications and outlines the rationale for using microservices (Section 2).

(2) It identifies best practices, challenges, and lessons learned in developing microservices, as viewed by our study participants, grouping them into three high-level themes: architectural considerations, infrastructure support, and code management (Section 4).

(3) It outlines possible directions for researchers and tool builders who want to provide better support for practitioners developing microservice-based applications (Section 5).

Target Audience. Our study collects best practices and lessons learned from a large number of companies successfully developing microservice-based applications for commercial use. We thus believe it can benefit industrial practitioners who develop such applications and are interested to learn from each other, borrow successful ideas, and 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.

Paper Structure. The remainder of the paper is structured as follows. Section 2 describes microservices and outlines their advantages. In Section 3, we describe our study methodology, including the selection of interview and survey participants. Section 4 outlines best practices, lessons learned, and challenges identified by the study participants. We discuss the implications of our findings in Section 5. In Section 6, we outline possible threats to the validity of our results. Related work is discussed in Section 7. Finally, Section 8 summarizes and concludes the paper.

2 Background

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 are stitched together to create applications (Newcomer and Lomow 2005).

Service-orientation has received a lot of attention in the early 2000s (Fenn and Linden 2005), but lost momentum around 2009. Since 2014, the interest in service-oriented paradigms was renewed under the *microservices* moniker (Zimmermann 2017). Renewed interest in service-orientated development can be explained by a combination of multiple factors, the most prominent of which are the popularity of cloud-based offerings, the pay-as-you-go infrastructure costs, and the constantly increasing demand for development agility.

In such reality, a microservice-based approach promotes building a single application as a suite of (micro-)services which run as separate processes and can be deployed and scaled independently (Lewis and Fowler 2014). Microservices communicate with each other—synchronously or asynchronously—via lightweight language-agnostic protocols, such as HTTP REST (Fielding 2000). Synchronous communication can rely on various API gateways, such as Amazon API Gateway.[5] Asynchronous communication is facilitated by distributed stream-processing software, such as Apache Kafka[6], and message brokers, such as RabbitMQ.[7]

As any service call can fail due to the unavailability of the supplier, services must be designed for failures and must respond to failures gracefully. This includes dealing with partial consistency of decentralized data and gracefully degrading part of the application functionality, if needed. Moreover, detecting failures quickly for automatically restoring services is critical and is typically achieved using logging and monitoring tools, such as Grafana[8] and Kibana.[9]

---

[5]https://aws.amazon.com/api-gateway/

[6]https://kafka.apache.org

[7]https://www.rabbitmq.com

[8]https://grafana.com

[9]https://www.elastic.co/products/kibana

Microservice-based applications promote autonomous teams that are organized around business capabilities and assume end-to-end responsibility for these capabilities: from development to production. The teams have the freedom to choose technology platforms, languages, and tools that work best for the functionality they want to implement (Lewis and Fowler 2014).

Advantages of Using Microservices. When implemented properly, microservices aim at shortening the development lifecycle while improving the quality, availability, and scalability of applications at runtime (Chen 2018; Fritzsch et al. 2019; Ghofrani and Lübke 2018; Knoche and Hasselbring 2019; Luz et al. 2018; Taibi et al. 2017). 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 (Fowler 2015). One cannot just "take shortcuts" and access code at a wrong layer (Chen 2018); APIs specified in the service contract are the only channel for accessing the service.

Developers can also focus on small parts of an application, without the need to reason about complex dependencies and large code bases; this is especially beneficial for junior developers and for onboarding new team members. Moreover, as the failure domains of microservices are rather independent, only the specific teams responsible for the failing microservice need to be notified. Dealing with legacy software also becomes easier: as the application is broken down into smaller, replaceable pieces, they become easier to understand and upgrade (Ghofrani and Lübke 2018; Gouigoux and Tamzalit 2017).

A major advantage of microservice-based architectures is independent deployment (Chen 2018; Luz et al. 2018; Soldani et al. 2018; Viggiato et al. 2018), which reduces the coordination effort needed to align on common delivery cycles practiced for monolithic applications. Independent deployment also leads to independent scaling at runtime, when instances of only those microservices that experience increasing traffic are spun off and later stopped, without affecting other parts of the application (Bucchiarone et al. 2018; Fritzsch et al. 2019; Ghofrani and Lübke 2018; Knoche and Hasselbring 2019; Soldani et al. 2018; Taibi et al. 2017; Viggiato et al. 2018). Finally, when implemented for fault-tolerance, defective microservices do not crash the entire application, contributing to application stability and availability (Bucchiarone et al. 2018; Chen 2018; Fritzsch et al. 2019; Luz et al. 2018; Taibi et al. 2017).

Together with advantages, microservice-based systems introduce additional challenges and call for software development tools and practices suitable to support microservice-based engineering. We investigate these in our work.

3 Methodology

Figure 1 provides a high-level overview of our study methodology. Given the exploratory nature of our study, we followed the recommendation of Bratthall and Jørgensen (2002) and applied a mixed-method approach: first, we conducted a qualitative semi-structured interview study to identify challenges and lessons learned in microservice-based development. We then distributed an online survey to enrich and validate (or not) the interview findings using collected quantitative data. We synthesized the findings from both the interview and survey studies, forming the final results reported in this paper.
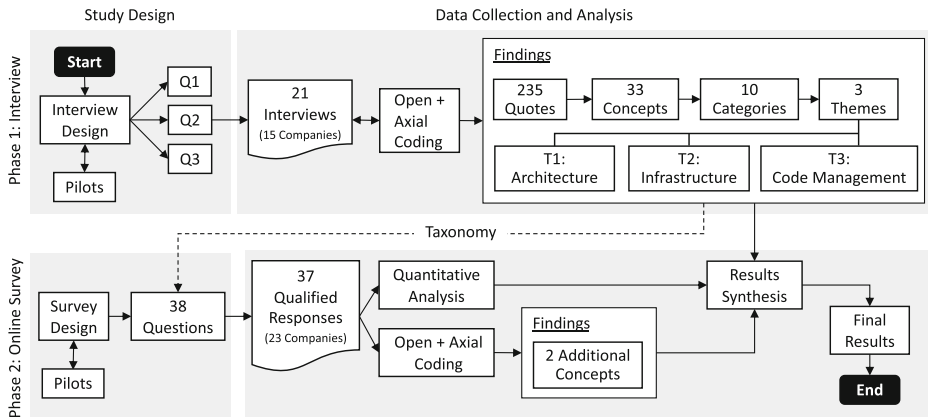
Fig. 1 Data collection and analysis process

We now describe this process in detail, including our selection of subjects and our approach to data collection and analysis, for both the interview study and the follow-up survey.

3.1 Interview Study

Subjects. For the interview study, we recruited software developers with solid experience developing microservice-based applications. More precisely, our selection criteria were for participants to (1) have more than two years of hands-on experience using microservice-based architecture in industry, (2) be a member of a team that designs, develops, and deploys microservices for commercial use for at least two years, and (3) be 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 interviewees, we initially approached our network of collaborators and colleagues. We also reached out to developers who actively participate in microservice-related events and Meetup[10] groups. In addition, we used the LinkedIn[11] web platform to recruit developers listing microservices as their core skills and holding active software development positions. Finally, we applied *snowballing* (Goodman 1961), 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* (Morse 1995), i.e., we did not identify new concepts in the last five interviews. Interleaving data collection and analysis processes is inspired by grounded theory and is typical for interview studies. We further elaborate on our interview stopping criteria when describing the data analysis process.

Overall, we interviewed 21 practitioners from 15 companies. For four large enterprises we selected for our study, we interviewed more than one employee, as these employees work in different, non-overlapping areas. Moreover, some of these companies are among

---

[10]https://www.meetup.com/

[11]https://www.linkedin.com/

Table 1 Interview participants

ID	Position	Ind. Exp. [yrs][a]	MS Exp. [yrs][b]	Team MS Exp. [yrs][c]	Team size	#MS[d]	Co. size[e]	Age	Education
P1	Software Engr.	5	3	3	15	25	L	33	Undergrad
P2	Application Architect	8	4	4	10	7	S	30	Undergrad
P3	Director of R&D	19	3	3	4	20	L	–	Undergrad
P4	Manager	–	2	2	4	15	L	–	Master's
P5	Architect	9	2	2	7	5	L	34	Master's
P6	Lead Front End Dev.	18	4	4	10	20	L	42	Master's
P7	Front End Dev.	3	2.5	4	9	7	L	25	Undergrad
P8	Software Dev.	20	2	4	5	30	L	42	Undergrad
P9	Senior Software Engr.	3	2	15	4	4	L	25	Undergrad
P10	Principal Software Engr.	20	10	4	20	20	M	36	No Degree
P11	Full Stack Dev.	20	4	4	–	40	L	45	Undergrad
P12	Senior Software Dev.	6	4	4	4	4	L	39	Undergrad
P13	Software Architect	22	4	–	–	–	L	–	Undergrad
P14	Senior Software Dev.	5	4	4	8	7	L	29	Undergrad
P15	Principal Software Engr.	20	7	2.5	–	6	M	47	PhD
P16	Platform Engineering Lead	14	4	2.5	12	25	S	37	Undergrad
P17	Senior Site Reliability Engr.	10	3	2	14	7	L	32	Undergrad
P18	Cloud Dev.	2.5	2.5	3	15	8	L	27	Master's
P19	Distinguished Systems Engr.	21	3	5	6	–	L	46	Undergrad
P20	Software Dev. Engr.	16	4	15	11	50	L	37	Undergrad
P21	Software Dev.	6	3	–	7	15	M	30	Master's
Mean		12.4	3.7	4.6	9.2	16.6	–	35.3	–

[a]Industry Experience [years]

[b]Microservice Experience [years]

[c]Team Microservice Experience [years]

[d]Number of Microservices the Team is Responsible For

[e]Company Size. S: <100 Employees, M: 100-499 Employees, L: >499 Employees

the pioneers of microservice-based software development; we thus believe their experience is highly beneficial for our study. The detailed info about our study practitioners, including their positions, years of industrial experience, experience with developing microservices, information about their development teams and companies, and their age and education level – for those who agreed to share this info with us—are given in Table 1.

Most interviewees 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, e.g., from a company known as one of the main pioneers of microservice-based development, developed microservice-style systems long before the term was popularized. We thus believe their experience is valuable and reliable.

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. Most work for large companies. 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. While we cannot share the names of the companies due to confidentiality reasons, we report their size in Table 1. We followed the Canadian Small to Medium Enterprise (SME) definition[12] that classifies companies into small (S), medium (M), and large (L) based on the number of company employees: a company with less than 100 employees is classified as small, a company with 100 to 499 employees is medium, and a company with more than 499 employees is considered large.

As for the demographic information, the ages of the interviewees range between 25 and 47 years, with a mean of 35.3 and a median of 35 years. Six interviewees hold a graduate-level degree (one PhD and five Master's), 14 hold an undergraduate degree, and one participant, with 20 years of experience, is entirely self-taught.

Interview Design. We performed semi-structured interviews with a set of open-ended questions. To identify an appropriate study protocol, we first conducted 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 min each (min = 26; max = 90; mean = 49.9; median = 47 min, total = 17.5 h). 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?
(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.

The interviews were conducted in English by at least two of the authors of this paper. 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

---

[12]https://www.ic.gc.ca/eic/site/061.nsf/eng/Home

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. At the time of writing, 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 transcripts.

Interview Data Analysis. We relied on qualitative data analysis techniques borrowed from the grounded theory, namely *open and axial coding*, for deriving theoretical constructs from qualitative analysis (Strauss and Corbin 1998). Note that we did not fully adopt the grounded theory methodology as our goal is to identify and categorize best practices and challenges in using microservices rather than establish any theory.

For open coding, two of the authors of the paper 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. We measured the inter-rater reliability for interview data coding with the widely used Krippendorff's alpha coefficient, typically used for such studies (Krippendorff 2011; O'Connor and Joffe 2020). Our K_α score is 0.89 (higher than the standard reference score $K_\alpha > 0.8$ score).

On a weekly basis, all the authors met to discuss the identified concepts and any disagreements, refine concept labeling, and merge related concepts if needed. We further grouped the identified concepts into *categories*, which represent high-level ideas that we found in the analyzed data. We then related categories to each other, grouping them into higher-level themes.

As qualitative analysis seeks to find significant concepts and explore their relationships (Strauss and Corbin 1998), we aimed at identifying ideas mentioned by at least three study participants. We considered an interview contributing new data for the study when it (a) identified new, previously unseen concepts or (b) contributed to concepts mentioned by less than three different participants. We continued recruiting study participants until we could not derive any new data in five consecutive interviews. This was done to avoid premature closure of data collection: a common practice in interview studies (Francis et al. 2010; Jackson et al. 2000; Jassim and Whitford 2014). That is, as mentioned earlier, we proceeded with more interviews until we reached data saturation.

In total, we identified 33 concepts, linked to 235 quotes and excluding 19 quotes that did not correspond to concepts mentioned by at least three interviewees. The concepts were further grouped into ten higher-level categories and three main themes: architectural considerations, infrastructure support, and code management, as shown in Fig. 1. As all our findings are linked to quotes extracted from the interviews, results we report are grounded on the collected data. Detailed information about our interview data analysis process, all the identified categories, concepts, and their mapping to quotes is available online (Wang et al. 2020).

3.2 Online Survey

We further validated and enhanced the findings from the interviews with an online survey. In a nutshell, we used the concepts brought up by our study participants as a starting point for the survey, with the aims to (a) validate or refute the insights collected in the interviews and (b) collect additional lessons learned and challenges that are not raised by our interviewees.

Table 2 Survey participants

Ind. Exp. [yrs][a]	MS Exp. [yrs][b]	Team MS Exp. [yrs][c]	Team size	#MS[d]	Co. Size[e]	Age	Education
Min: 2	Min: 2	Min: 0.5	<6: 10	<6: 4	Small: 10	Min: 23	Undergrad: 21
Max: 20	Max: 10	Max: 8	6–10: 13	6–10: 12	Medium: 7	Max: 51	Grad: 12
Mean: 13.4	Mean: 3.4	Mean: 2.5	11–20: 6	11–20: 8	Large: 7	Mean: 36.6	NoDegree: 2
Median: 13.5	Median: 3	Median: 2	21–50: 7	21–50: 5	Unknown: 13	Median: 35.5	Unknown: 2
			51–100: 1	51–100: 7			
			>100: 0	>100: 1			

[a]Industry Experience [years]

[b]Microservice Experience [years]

[c]Team Microservice Experience [years]

[d]Number of Microservices the Team is Responsible For

[e]Company Size. S: <100 Employees, M: 100~499 Employees, L: >499 Employees

Subjects. To mitigate potential sample bias of using one source of information, we reached out to a different group of participants for the online survey. To this end, we sent the survey via a diverse range of online channels, such as public microservice forums on Reddit,[13] Google Groups,[14] Facebook,[15] and Twitter.[16] We also explicitly searched for developers with microservice expertise on LinkedIn and sent them a targeted invitation to participate in the survey and further distribute it to their colleagues.

The survey was available online for eight months, till the end of September 2019, and we received a total of 185 responses. As we distributed the survey in open forums, we could not track the views or calculate the response rate. For the completion rate, 61 of the received responses were complete (33%). The majority of incomplete responses (96/124, 77%) were when the survey was only opened on page one, likely because the individuals only checked the description/participation criteria or accidentally pressed the link.

Out of the 61 complete responses, we only considered responses from practitioners with at least two years of microservice development experience—a selection criterion that we also applied to our interviewees. That resulted in 37 qualified responses from practitioners employed by 23 different companies. As there is only one overlap between employers of our interview and survey participants, our study collects data from 58 participants employed by 37 companies in total.

Demographic information about the survey participants is given in Table 2 and resembles that of the interviewees. Our survey participants have been working in industry for two to

---

[13]https://www.reddit.com/

[14]https://groups.google.com/

[15]https://www.facebook.com/

[16]https://twitter.com/

20 years, with a mean of 13.4 and a median of 13.5 years. They have between two and 10 years of experience developing microservice-based applications, with a mean of 3.4 and a median of 3 years. Their teams diverse in size and the number of microservices they develop. Ten of them are from small, seven from medium, and another seven are from large companies. In 13 cases, the participants preferred not to declare the size of their company, so we do not report on that data.

Survey Design. The main goal of the survey was to validate results from the interview study using a different sample and to enrich our findings. To ease participants' task and provide context to our study (Flanigan et al. 2008), we used the taxonomy derived from the interviews as a starting point for the survey. We asked participants whether they agree or disagree with statements made by the interviewees, whether they see additional solutions to the identified problems, and whether they want to report on additional best practices and challenges.

The survey consisted of 13 groups of questions related to the ten categories identified in the interview study as well as some background and demographic information. In total, there were 38 questions, most of which are multiple-choice, with an additional field to provide extra information. Several open-ended questions were designed to collect information beyond that observed in the interviews.

We also asked participants to distinguish between the solution they apply in practice (what they do) and their perception (what they think they should do). Towards this end, we focused the participants' attention on their most significant microservice-related development experience, as providing a specific frame of reference helps reducing recall bias and facilitates cognitive processing (Sinkowitz-Cochran 2013).

As with the interview study, we performed several pilots for the survey to improve the clarity of the questions and arrive at a reasonable time required to complete the survey: around 20 min. Detailed information about the survey is available online (Wang et al. 2020).

Survey Data Analysis. For multiple-choice questions, we calculated statistics on (a) how many survey participants agree or disagree with best practices identified in the interviews, (b) how many participants actually apply these best practices in their work, (c) how many participants apply each solution to a challenging microservice-related problem identified in the interviews, and (d) additional solutions to these problems the survey participants apply.

For the open-ended questions, we followed the process similar to that in the interviews and performed open and axial coding for all write-in survey replies. As the survey responses were much more structured and focused around the explicit questions we asked, there were no disagreements between the two authors looking at the data. We thus did not calculate K_α for the survey results.

We unified the concepts identified in the survey with those from the interviews and included in the final results all concepts that were mentioned by at least three interview or survey participants. As a result, we identified two additional concepts, arriving at a total of 35 concepts belonging to ten categories and three themes. We discuss these in Section 4. In Section 5, we also outline some of the ideas mentioned by less than three participants.

Quality Considerations. To make sure our results are reliable, we sent the final version of this report summarizing our findings to all study participants, asking them to comment on any misinterpretations that might have occurred. That was done in addition to sharing the transcripts with the corresponding interviewees, as described above. The feedback we

received shows that the study is accurate and representative: P13: «Overall, I think this is a very strong study that accurately reflects my experience in the industry».[17]

Ethical Considerations. Our study was also approved by the Research Ethics Board of the University of British Columbia. In the ethics board application, we summarized the intended purpose of our study, potential participants, recruitment strategy, informed consent process, possible conflicts of interests, data collection strategies, and confidential data handling. Before each interview or survey session, the participant was required to give us an informed consent to conduct the study and record the interview.

4 Results

We first present a high-level overview of our results (Section 4.1) and then discuss these results in detail (Section 4.2). Following the detailed discussion, we summarize our findings and explicitly answer research questions RQ1-RQ3 outlined in the Introduction section (Section 4.3).

4.1 Results: Overview

Table 3 lists the ten categories and three themes identified in the interviews, together with the number of interview participants mentioning concepts in each category. The majority of the interviewees (71%) brought up topics related to identifying the right granularity for microservices, beyond the typical considerations of splitting microservices by business capabilities. Some interviewees also discussed the need for identifying and enforcing strict service ownership (43%). Considerations related to programming languages used for implementing microservices concerned one-third of our interview participants (33%); most of them indicated that they moved away from the initially attractive idea of letting each team decide which language to use for their microservices towards more standard organization-wide decisions promoting reuse and common support. We grouped these three categories of concerns into the Architecture theme and discuss them in detail in Section 4.2.1.

For the Infrastructure theme, around half of the interviewees (52%) considered logging and monitoring (or lack of support for logging and monitoring) as a concern for their organization. They mostly agree that setting up such infrastructural support early in the development process is essential for the success of their businesses. Similar concerns about an early setup of distributed tracing and automated development processes were mentioned by 29% and 24% of the interviewees, respectively. Moreover, 29% of the interviewees raised consideration about development and deployment tools used in their organizations. We discuss all these topics in detail in Section 4.2.2.

Finally, Code Management issues, especially around dealing with and preventing breaking API changes concerned almost all our interview participants (95%). Issues related to the management of code used in multiple microservices and the management of product variants targeting different types of customers concerned more than half of the interviewees (67% and 57%, respectively). We discuss these in detail in Section 4.2.3.

---

[17]The paper presents all participants' quotes in this style: PID: «quote».

Table 3 Categories identified in the interviews

Theme	Category	#Interview participants (out of 21)
Architecture	Microservice granularity	15 (71%)
	Microservice ownership	9 (43%)
	Language diversity	7 (33%)
Infrastructure	Logging and monitoring	11 (52%)
	Distributed tracing	6 (29%)
	Automating processes	5 (24%)
	Tools	6 (29%)
Code Management	Managing common code	14 (67%)
	Managing API changes	20 (95%)
	Managing variants	12 (57%)

Due to the qualitative nature of our study, the list of concepts, categories, and themes we report on is not intended to be complete. These constructs rather emerged from performing open and axial coding on the data we collected and represent topics raised by our study participants. As we only asked open-ended questions during the interviews and did not steer the participants in any particular direction, we believe these represent major concerns practitioners deal with on a daily basis.

Interestingly, we did not find any strong correlation between the demographics of the interviewees and the topics that they raised, indicating that the topics described in this paper were raised by engineers of various ages, professional and educational background, educations, etc. For example, Fig. 2a shows the distribution of topics brought up by the study participants with different development experience. In this figure, we show the percentage of participants of each type mentioning a particular topic. E.g., two out of five participants with 2–5 years of experience (40%) discussed topics related to the microservice granularity.

Figure 2b shows similar information when participants are distributed according to the size of their team. Here, engineers working on larger teams (>10 team members) typically have more code-management-related challenges, consistent with our expectations. Members of smaller teams (4–5 members) raise more issues in the tool category compared with other study practitioners; we conjecture that is because tool management is more difficult for smaller teams that cannot afford to split the workload between multiple team members. However, we note that as with any qualitative studies, all numbers reported in this paper have informative rather than statistical nature. We do not report on by-company-size distributions, as most companies considered in our interview study are large enterprises.

To further validate or refute information identified during the interview study, we reach out to a broader audience using a survey. Table 4 shows the agreement of survey participants with the importance of the topics identified by the interviewees in their current project (i.e., what they are currently *doing*). We also asked the survey participants about their conceptual perception of the importance of the topics (e.g., what they *think* they should do). We report on these results with dashed bars in Table 4. Interestingly, for the service ownership and language diversity categories (which deviate from the standard advice in the microservice community), there are more practitioners actually applying the practice than those that believe it is important. However, practitioners believe in the importance of the infrastructure support more frequently than actually being able to implement such support in practice.
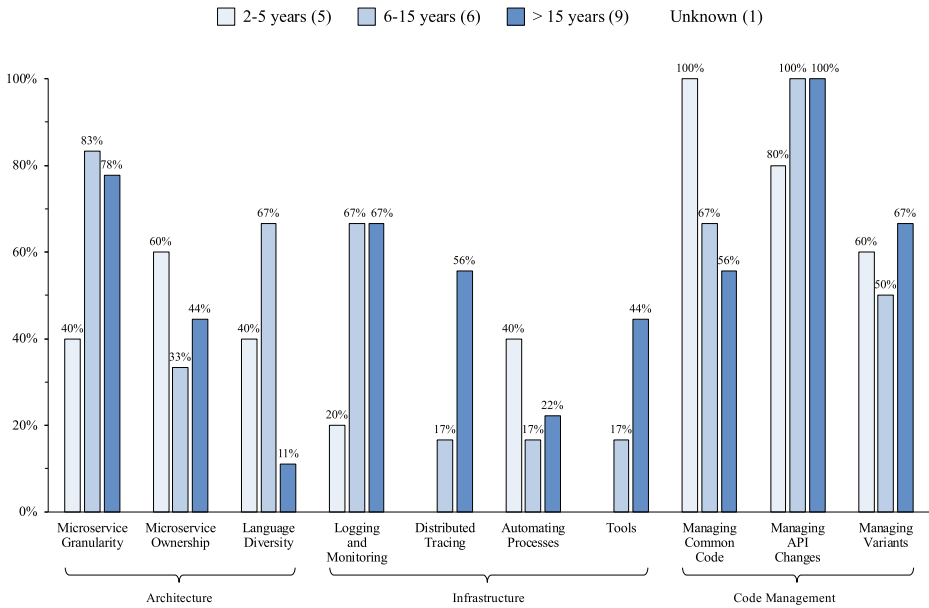
(a) By Industrial Experience
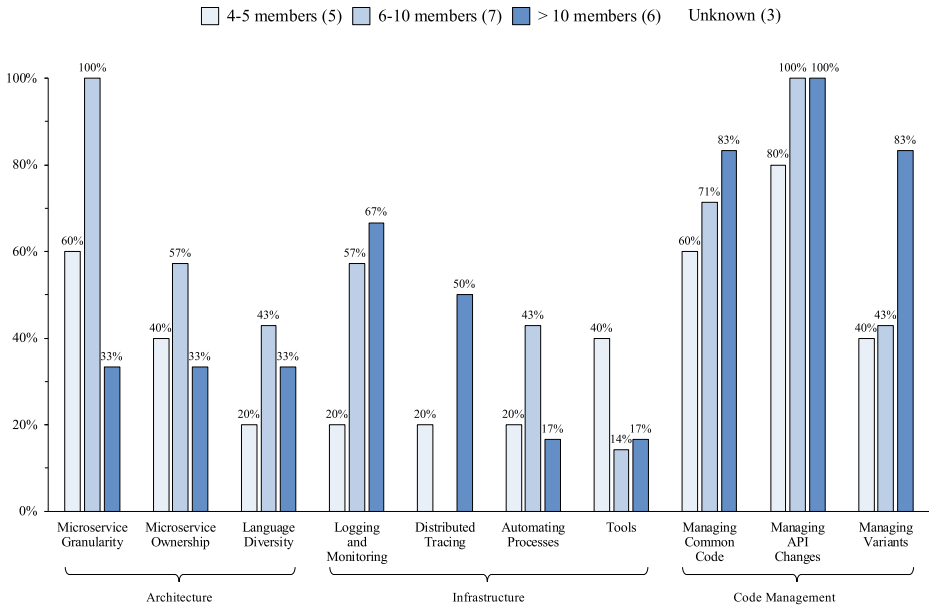
(b) By Team Size

Fig. 2 Interviewees distribution by demographic info

Table 4 Agreement of the survey participants with the interview results

Theme	Category	#Survey Participants (out of 37)
		Practice ■ Perception ⌐⌐
Architecture	Microservice granularity	37 (100%)
	Microservice ownership	24 (65%), 21 (57%)
	Language diversity	27 (73%), 21 (57%)
Infrastructure	Logging and monitoring	36 (97%), 37 (100%)
	Distributed tracing	26 (70%), 31 (84%)
	Automating processes	25 (68%), 26 (70%)
	Tools	22 (59%), 27 (73%)
Code management	Managing common code	35 (95%)
	Managing API change	37 (100%)
	Managing variants	31 (84%)

Similar to interview results, we did not find a strong correlation between the demographics of the survey participants and the topics they deem important. Figure 3 shows information similar to Fig. 2, only for the survey participants. It is interesting to note that infrastructure-related topics generally receive less attention than topics related to architecture and code management. Infrastructure-related topics also concern experienced engineers slightly more than their junior counterparts. Service ownership is less of a concern for smaller teams, as expected. We discuss these results in detail next.

4.2 Results: Detailed Discussion

We organize the detailed discussion of the interview and survey results by three main themes: architectural considerations, infrastructure support, and code management. We outline best practices and lessons learned, list possible solutions to challenges related to microservice-based development, and report the numbers of survey participants employing each solution. We do not report on the number of interview participants employing the solutions as our interviews had a more exploratory nature and we did not explicitly ask interviewees to enumerate all solutions that they use.

4.2.1 Architecture

Architectural considerations include concepts related to microservice granularity, ownership, and languages used.

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 5 shows different, not necessarily mutually exclusive strategies that 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: P15: «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: P14: «Having a big service is not that bad, as long as the service is cohesive and it deals with one thing.»
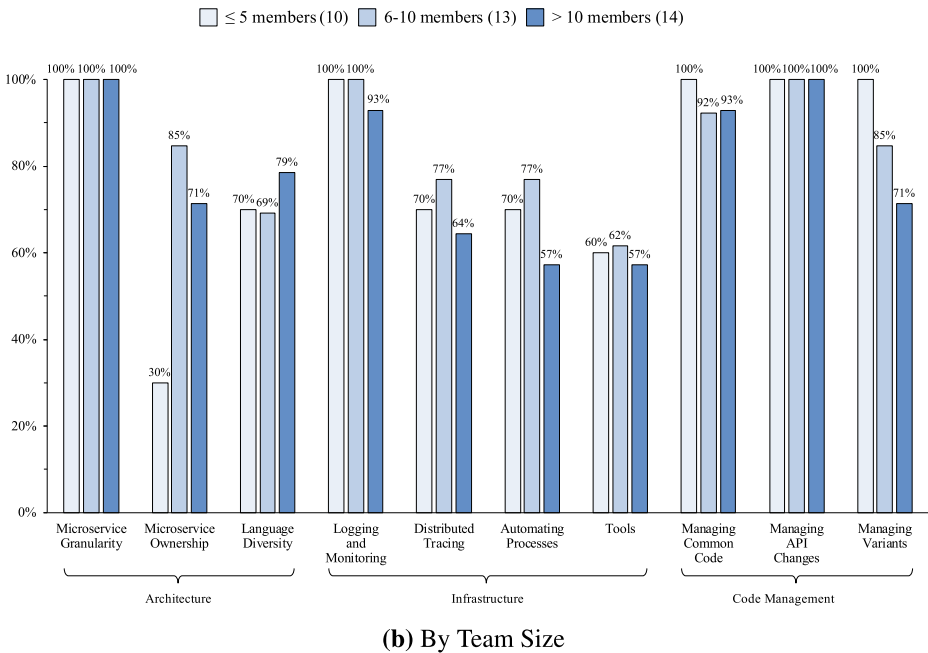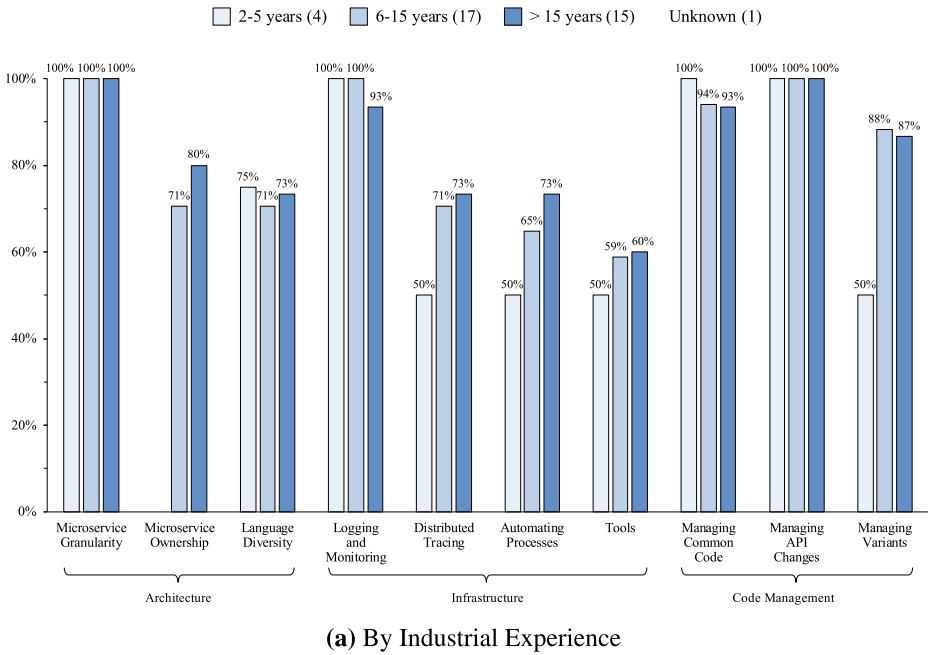
(a) By Industrial Experience

(b) By Team Size

Fig. 3 Survey participants distribution by demographic info

Table 5 Microservice splitting criteria

Criterion	#Survey participants (out of 37)
Business capabilities	30 (81%)
Data access	16 (43%)
Team structure	9 (24%)
Dependencies	7 (19%)
Resource consumption	7 (19%)
Delivery cycles	5 (14%)

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: P2: «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: P15: «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: P13: «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.»

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: P11: «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.»

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. P1: «That has been fairly expensive because we have to [...] figure out how exactly their server is set up, so we can deploy this piece there with the dependencies.» When applications exceeded the hardware capacity allocated by the customer, companies had to merge microservices: P6: «[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: P6: «We do not want too little traffic to a microservice. If we just serve for one

Table 6 Microservice owner responsibilities

Responsibility	#Survey participants (out of 24)
Design microservices	20 (83%)
Identify bugs	22 (92%)
Fix bugs	19 (79%)
Implement new features	18 (75%)
Train newcomers	1 (4%)

particular page that is not visited frequently, we are wasting our resources starting up that microservice doing nothing for most of the time.»

Splitting applications into too granular microservices also induces additional overhead on the development process: P1: «The more you split it up, the more overhead you have. If you want to release, for example, a security patch for Java, [...] you have to re-release each of your services. That is overhead.» To mitigate that overhead, some study participants consider the frequency of changes and anticipated delivery cycles as another criterion for defining microservices.

> 💡 *Lesson learned #1: Apart from the common practice of considering business capabilities and data access when deciding on microservice granularity, developers should apply product-specific considerations, such as the anticipated resource consumption, team structure, and delivery cycles of microservices.*

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. More than half of the survey respondents (57%) also agreed with the need to define a service owner and even more than that (65%) do have an owner defined in practice (see Table 4).

A service owner is a person or a team, who is the primary point-of-contact when the microservice malfunctions: P12: «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.» The survey respondents that employ this practice (24 people, see Table 6) report that, apart from being responsible for the design and architecture of microservices (83%), owners identify and fix bugs (92% and 79%, respectively), implement new features (75%), and train trusted committers for the service (4%).

Having service owners is important for defining a clear architecture, with well-defined service responsibilities and boundaries, and reducing architectural debt (Kruchten et al. 2012). 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: P20: «Because there wasn't that sense of ownership, there wasn't that sense of continuum 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 development and architectural integrity.*

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: P15: «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: P17: «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.»

Our survey confirms this finding: 57% of respondents believe that companies should prescribe a specific number or set of languages to be used (see Table 4). Even a larger number, 73% of respondents, report that their organizations already regulate language diversity. One of the interviewees described the policy their company follows: P15: «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: P15: «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.'»

Our participants emphasized the need for standards on coding styles and development workflows for all developers, which promote shared linting tools, testing and deployment practices, etc. Such standards unify the development workflows for hundreds or thousands of microservices, making knowledge and ownership transfer more efficient.

> 💡 **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 development "type".*

4.2.2 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.

Logging and Monitoring. One of the main criteria for a mature microservice-based development process is the robustness of the logging and monitoring framework. All the survey participants agree that the logging and monitoring framework is essential and more than 90% believe it should be set up as early as the project starts. In practice, practitioners

Table 7 Logging and monitoring: what is logged?

Information	#Survey participants (out of 36)
Request and response time	29 (81%)
Service availability	28 (77%)
Number of requests	24 (67%)
Resource utilization	21 (58%)
Input, output, error data	20 (56%)
Transaction id	2 (6%)
Configuration values	1 (3%)

Table 8 Logging and monitoring: how used?

Purpose	#Survey participants (out of 36)
Troubleshoot	36 (100%)
Customer reports	9 (25%)
Compliance and auditing reports	1 (3%)

Table 9 Logging and monitoring: how captured?

Solution	#Survey participants (out of 36)
Visualization	22 (61%)
Statistical data	19 (53%)

report that this step can be neglected or postponed to a later stage, when their project is experiencing failures and delays.

Practitioners often regret the decision not to set up a logging and monitoring framework early as the project starts: P16: «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.»

Our participants report that, apart from request and response time and service availability, they log the number of requests, resource utilization, input, output, and error data, request transaction id, and configuration values (see Table 7). 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. P20: «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.»

As shown in Table 8, practitioners use logged info for troubleshooting (100%), for creating customers' reports, e.g., on the delivered quality of service (25%), and for creating reports for auditing purposes (3%). They mostly rely on monitoring and visualization of the overall health of the system (61% in Table 9). Moreover, more than half of the respondents mentioned that they run statistical analysis on the logged info, to automatically identify failures, even before customers notice them, and notify the corresponding team: P20: «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.»

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

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* (Richardson 2018b), which track services participating in fulfilling a request, are applied: P23: «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 with logging and monitoring, setting up a distributed tracing framework is an important task to do early as the project starts: P17: «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.» Around 65% of the survey participants agree that distributed tracing should be established right as the project begins, but less than half actually did that in their own project. Overall, 84% of the survey participants think that a distributed tracing system should be set at some point during the lifecycle of the project but, again, only 70% have such a system established in practice, as shown in Table 4.

> 💡 ***Lesson learned #5:*** *To efficiently troubleshoot a microservice-based application, distributed tracing should be set up at the onset of the project.*

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: P14: «When we first started doing microservices, it actually took about a month or two to get all the infrastructure ready to create a new microservice. Now we have the ability to create a microservice in 5 min.» P12: «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: P6: «Today we are working on our starter-kit for other teams to easily get on board with the microservice-based architecture. Back at 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 (e.g., P6, P12, P14), mentioned that an important lesson they learned is not treating automation as an afterthought: P12: «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.»

Interestingly, while 59% of the survey participants agree with this statement and 57% indeed set up automated processes as early as possible, the rest either prefer to postpone investing in an automated setup (11%) or do not believe such investment is necessary at all (30%). These practitioners typically work on smaller projects which rarely need to create new services; thus automation is deprioritized. In Table 4, we show the total number of survey participants that believe the automation has to be set at some point of the project lifecycle: 70%.

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

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 developed by Docker Inc., Kubernetes and Istio[18] by Google, and Envoy[19] by Lyft. By now, there are more than 370 tools out there to support development and deployment of microservices, service discovery and API management at runtime, and more.[20] For smaller companies, building proprietary tools no longer makes economic sense: P13: «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 solution, out of all available options, takes time; they usually lack time and incentive to do that, as the development is driven by business needs: P10: «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.» Around 73% of the survey respondents believe that they should actively explore and evaluate new tools, but only 59% had time to do that in practice.

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[21] or Jaeger,[22] using the vendor-neutral distributed tracing framework OpenTracing[23] facilitates the transition between vendors.

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

---

[18]https://istio.io

[19]https://www.envoyproxy.io

[20]https://landscape.cncf.io/license=open-source

[21]https://zipkin.io

[22]https://www.jaegertracing.io

[23]https://opentracing.io

Table 10 Common code: how managed?

Solution	#Survey participants (out of 35)
Duplicate code	▮ 1 (3%)
Latest-version shared library	▬▬▬▬ 15 (43%)
Multiple-version shared library	▬▬▬▬ 15 (43%)
Standalone microservice	▮ 1 (3%)
Sidecar	▬ 3 (9%)

4.2.3 Code Management

In this section, 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.

Managing Common Code. Splitting an entire system into a set of fully independent microservices is not realistic in practice. Virtually all study participants stated that some functionalities and code need to be shared among 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 (Hunt and Thomas 1999) and will make code unmanageable in the long run. Only one survey participant follows this practice (see Table 10).

The most common practice other participants employ to minimize code duplication observed is to package the common code in a shared library, which is imported at build time (Mitra 2018). In the simplest case, all microservices use the same latest version of a library; if a microservice requires a new feature, it has to upgrade to the newest version of the library. Around 43% of the survey participant apply this strategy. Its downside is that a microservice cannot make independent changes to that common code: P15: «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, another 43% of the survey participants carefully version libraries and allow services to rely on multiple versions of a library. Such an approach could introduce another challenge—inconsistencies and application instability: P12: «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.»

To avoid the library management hassle, one survey participant opted for wrapping common code in a standalone microservice and several interviewees suggested such a practice as well. This approach simplifies procedures as no redeployment of all microservices that use the changed library is required: P18: «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.

Table 11 APIs: how managed

Solution	#Survey participants (out of 37)
Direct API calls	15 (41%)
Message-based communication	10 (27%)
A proxy	25 (68%)
A client library	8 (22%)

A more systematic solution that addresses the disadvantages of a standalone microservice while keeping its advantages is called sidecar. This solution has recently emerged from industry and did not yet experience wide adoption: only 9% of our survey respondents started to use this strategy. 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 sidecars helps 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: P13: «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 an approach require careful planning and management. Practitioners should also experiment with the newly emerged sidecar solution.*

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 (Postel 1980): 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: P21: «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: P13: «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.»

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: P6: «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. Around 41% of our survey participants (see Table 11) use direct API calls for synchronous communication between microservices. Another 27% use asynchronous message-based communication within their product or across product

Table 12 Variants: how managed?

Solution	#Survey participants (out of 31)	
Feature flags	▬▬▬▬	10 (32%)
Role-based access	▬▬▬▬▬▬	16 (52%)
Separate deployments	▬▬	5 (16%)

boundaries. They mostly rely on distributed stream-processing software, such as Apache Kafka or message brokers, such as RabbitMQ.

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: P20: «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 (Richardson 2018a), push the complexity to the server side. Proxies are employed by 68% of our survey respondents.

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 utilized by 22% of the survey respondents. They are typically automatically generated, e.g., using tools such as Swagger.[24] Using libraries has numerous benefits over clients making direct HTTP calls (Google Cloud 2019). 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. API gateways and client libraries help to mitigate the burden of upgrades for the client.*

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 80% of our survey participants (31) need to manage variants of their products. Out of those, 32% rely on feature flags (see Table 12), which are basically conditions that are passed with requests and that control different paths in code. In the simplest form, these flags can

---

[24]https://swagger.io/

identify a certain group of users: P12: «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 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: P21: «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 (Rossi et al. 2016) and to roll-out changes to a certain group of users: P12: «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 features 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, which is employed by the majority of our survey respondents (52%, see Table 12), 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 (Richardson 2018a). Each incoming request has an API key tied back to a particular API plan, and an API gateway routes the requests to 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: P20: «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, 16% of the survey 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: P7: «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: P6: «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: P5: «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 issues. 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.*

4.3 Results: Summary

We now present the main best-practices, challenges, and lessons-learned identified by our study participants, answering the research questions presented in Section 1.

RQ1: What are best practices learned by practitioners successfully adopting microservices for commercial use?

- When deciding on microservice granularity, apart from splitting by business capabilities, developers should also consider product- and deployment-specific constraints, e.g., those related to the anticipated resource consumption, team structure, delivery cycles, and more.
- Assigning an explicit owner to each microservice (vs. shared ownership model) helps improve architectural integrity, productivity, and the resulting product quality.
- Restricting the number of programming languages used within an organization helps facilitate maintenance and reuse.
- Setting up extensive and comprehensive logging and monitoring support early in the development lifecycle, as well as investing in distributed tracing early on, help increase product quality.
- Automated tool support, the use of standardized tools, and the use of open interfaces help reduce operational costs.

RQ2: What are challenges practitioners face?

- Managing common code shared by multiple microservices, e.g., for authentication and logging.
- Preventing and dealing with breaking API changes.
- Supporting products consisting of multiple different customer offerings using a single code base.

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.

RQ3: Which solutions to the identified challenges exist?

- For managing common code, our study enumerated several options employed by the participants; they are shown in Table 10. While most participants rely on shared libraries, the emerging sidecar technology mentioned by some could be a promising and elegant microservice-native solution to this problem.
- Numerous options for managing APIs are listed in Table 11. API gateways and client libraries, although not adopted by the majority of the practitioners involved in our study, have a high potential to mitigate the burden related to managing API changes.
- Options for supporting product variants are listed in Table 12. Each of these solutions has advantages and disadvantages, with most practitioners applying a role-based access model to support different product offerings within the same code base.

5 Discussion and Future Research Directions

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. In this section, we discuss the implications of our findings in detail and outline possible future research directions that emerged from our study (Section 5.1). We then summarize the main discussion points and implications (Section 5.2).

The discussed topics were raised by study participants when we asked them an open-ended question about emerging challenges they envision. Even though some of these topics were raised by less than three participants (and thus we did not include them in Section 4, per our study methodology), we believe it is valuable to report on these results: these topics were raised by the most experienced participants of our study (mostly Principle Software Engineers and Architects with more than 10 years of experience) and thus could be of interest to the reader.

As with Section 4.2, we structure the discussion around architecture, infrastructure, and code management themes.

5.1 Detailed Discussion

5.1.1 Architecture

Identifying proper service granularity is one of the main concerns of multiple practitioners. Splitting and merging microservices is a process that is performed continuously, long after the initial microservice topology is identified: P7: «A major reason for creating a new microservice is to spin off one microservice that started to get large.» This process should be informed not only by architectural considerations, such as functional decomposition, coupling and cohesion, and fan-in and fan-out metrics, but also by additional concerns such as performance, reliability, and security. In particular, breaking the system into services that communicate over the network increases the attack surface, security of data transferred within and between applications is to be considered and minimizing transfer of sensitive data over the network could be beneficial: P13: «There needs to be a lot more thought putting into security as you are building application components.» P19: «How do you create security between microservices? [...] you might have encryption to an application [...] but once you are inside, it is not encrypted anymore. [...] What if there was a hacker that was able to get inside the overall system through some means and they were able to do a man-in-the-middle [attack]?»

Another consideration related to the fact that all data between microservices is transferred over the network is the volume of the transferred data, which could add an additional decomposition consideration: P21: «Sometimes even choosing microservices may not be the best option because you have a lot of communication overhead then.»

As with any other systems, the quality of microservice-based architecture degrades over time and the abstractions introduced at the design stage get lost. The systems accumulate, so to say, architectural debt: P2: «There will be technical debt that you would accumulate even in terms of integration, in terms of source control and versioning.» Metrics and tools that help practitioners continuously assess and provide suggestions for refactoring their microservice-based architectures are thus needed.

> ♀   ***Future work #1:*** *Guidelines and metrics for assessing the quality of a microservice-based system, which go beyond the "classical" architectural patterns, e.g., coupling and cohesion, and also consider performance-, reliability-, and security-related concerns for microservices.*

More generally, several study participants noted that microservices is not a magic "pixie dust" and transitioning to microservices cannot solve all development problems. Moreover, microservices are not the solution to all applications. Organizations should avoid being "deceived" by a few promises of microservices; more structured guidelines on what types of applications would benefit from transitioning to microservices and which should be more cautious about migrating would be helpful: P10: «At some point, it just does not make sense to make everything into a discrete unit. The functionality is not important enough to spend the time on doing it or it would just not have enough value to spend the time doing it.» In addition, guidelines on co-existence of monolithic and microservice-based parts of an application are needed. Our participants believed that at present, there is too much emphasis on the initial microservice build, with not much support for assessing the necessity of this decision and its implications in the longer run: P13: «It is an architectural style that is optimized for post-implementation change as opposed to implementing the first go-around.»

> ♀   ***Future work #2:*** *Guidelines on types of applications that would benefit from transitioning to microservices and support for co-existence of monolithic and microservice-based parts of an application.*

5.1.2 Infrastructure

Several study participants believed that breaking a system into small components makes it easier to understand: P7: «For someone who joins our team about one or two weeks ago, instead of giving them all 2.5 million lines of code that make up our entire product, we can give them 10,000 lines of code and say 'hey, you do not have to worry about anything but go learn this 10,000 lines of code, the changes you are going to need to make are within this area.' It is a lot less overwhelming. All that code is consistent too, we just find it is a lot cleaner.» Others mentioned that while a limited scope indeed decreases the entrance barrier, it weakens the developers' understanding of the system as a whole: P12: «If you are a principal developer, [...], someone who is making more wide-reaching decisions, definitely understanding the whole system is important.»

Under such a model, developers, in particular the junior ones, are trained to believe that they should only care about a few microservices that they directly work on. They develop the impression that it is easy (or easier) to debug microservices: P-: «if something is working incorrectly, you do not have to debug the whole application. You just debug this tiny little part from the test.»[25] Yet, understanding and debugging a distributed system composed of multiple, independently managed and involving components, is in fact, a challenging task (Liu et al. 2008; Beschastnikh et al. 2016): P15: «I think you cannot find everything when you are just doing service-level testing because you do not see the entire picture. The problems tend to be different [...].»

---

[25]The quote is anonymized, to preserve the confidentiality of the participant.

Performance debugging for microservice is especially difficult and is now mostly based on metrics and logs, and is mostly done ad-hoc. Current tools help practitioners understand the health and performance of a single microservice, a container, or the application as a whole. However, the interpretation of how all these components work together to identify the root cause of problems is mostly done manually: P7: «You also need to understand the overall performance characteristics of the service as a whole [...]. I am trying to pull out it together, to get a good overview of the performance and the various bottlenecks, which is significantly more challenging than just a single Java program.» Application performance management software for microservices has not gained sufficient attention in the relevant communities (Seifermann 2017; Heinrich et al. 2017).

Moreover, since the data and traffic load in the testing and production environments are different, defects are not readily apparent until going to the production environment: P17: «It is a lot more difficult to try and test some product end-to-end just because it is very difficult in the microservices world to actually replicate production-like data in testing. That is always been a challenge ever since microservices have been introduced because you cannot really test the full breadth and scope of a product in testing simply because of the data inconsistencies and the scaling inconsistencies.»

> ♀ **Future work #3:** *Tools for understanding, debugging, and assessing performance of a microservice-based system.*

Microservices are commonly thought of as an architectural style that emerged from the Agile and DevOps movements (McLarty 2016; MuleSoft 2018), 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, which are bound by privacy laws that require careful handling of confidential and personal data, implementing DevOps is not always straightforward: such companies often rely on contractors that develop the software, who do not have access to the company's infrastructure which is managed by the operations team: S32: «Data privacy concerns require data to be kept in-house.»

> ♀ **Future work #4:** *Development practices for microservice-based products bound by privacy laws and/or requiring regulatory compliance.*

5.1.3 Code Management

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 (Weiss and Lai 1999; Clements and Northrop 2002; Pohl et al. 2005). Adapting the techniques developed by this community to the context of microservices could be a fruitful research direction.

> ♀ **Future work #5:** *Tools and techniques for efficient development, maintaining, testing, and debugging microservice-based system variants.*

In the realm of managing API changes, several of the study participants indicated that this process involves constant synchronization effort between the teams to propose and discuss changes and even synchronization on deploying changes simultaneously. Our participants mentioned having meetings, emails, and Slack channel conversations, to propose and discuss API changes: P15: «We talk to each other, that is usually how it works. We basically get in a room and draw on a whiteboard, say, 'Okay, you do this. You change this. Okay, and then we deploy that. And then you go and change this and you deploy that,' and so on. [...] We have a meeting or we have just email discussion [...] on what the change is and we discuss it. We do this at least once a week.»

In fact, communication and coordination within and between teams still remain a challenge. Even though most companies adopted microservice-based architectures because of the promise to decouple the teams, synchronization and collaborative planning still takes a substantial portion of the teams' time: P17: «What ends up happening is that a lot of different teams logistically are working on different microservices. They are speaking on a similar type of protocol, so it is very easy to deploy the product. But when it comes to making changes across all of the microservices in one go or change the entire flow, it is difficult logistically because you are usually dealing with different teams. So there is that challenge, as well.»

> 💡 **Future work #6:** *Tools for assessing the impact of the proposed changes and facilitating communication of these changes between the teams.*

5.2 Summary

The main directions for possible future work, which emerged from our study, are summarized below.

Architecture. The microservice-based architecture style is appropriate for some but not all applications. Practitioners need structured guidelines to determine whether the transition to microservices is beneficial, which parts of an application will benefit from transitioning to microservices, and how to efficiently manage microservice-based and monolithic components which are parts of the same system. Non-functional considerations, such as performance, reliability, and security, need particular attention.

Infrastructure. Practitioners need better tools that support performance testing and debugging for microservices. Moreover, companies that cannot implement the "classical" DevOps pipelines, e.g., because the private customer data cannot be shared with third-party developers, need solutions replacing a full-ownership model and allowing smooth collaboration between developer, infrastructure, and support teams.

Code Management. Change management is challenging in monolithic applications and becomes even more challenging in microservice-based applications. Tools for assessing and communicating the impact of a change in one microservice on other microservices is needed. Solutions for developing, managing, and debugging microservice-based application variants targeting different related customers or market systems are in demand as well.

6 Threats to Validity

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 types and sizes. We also interviewed practitioners in different roles – from software developers to team leads and managers, and followed up on the interview study with an online survey reaching a different industrial segment. We believe that these measures helped to mitigate this threat.

Internal Validity. For internal validity, we might have misinterpreted participants' answers or misidentified concepts 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 authors of the paper; furthermore, all disagreements were discussed and resolved by all the authors. Second, we shared the "raw" data collected during the interviews with each corresponding interviewee and the resulting report (this paper) with all study participants for their validation. We thus believe our analysis is solid and reliable.

7 Related Work

We divide the discussion of related work into three main areas: studies discussing migration to microservices, studies focusing on post-migration practices and challenges, and meta-studies on microservice-based development.

Table 13 lists papers in the first two areas and relates their findings to the finding identified in our study. With a ● symbol we indicate that the related work is fully aligned with our findings on a particular topic. A ◑ symbol indicates that the related paper mentions some but not all concepts we identified on a topic. Finally, a ○ symbol indicates that the related work does not cover the topic at all. With a + sign we indicate that the related work mentions additional concepts not covered in our study; a * sign indicates that the related work mentions views contradicting those identified in our study.

As a high-level comparison, the table shows that while some of the papers cover part of the topics we identified, none of them covers all our findings. Moreover, none of the papers identifies the need for managing variants in microservices, the importance of service ownership, and the responsibility of the owners.

Furthermore, while some of the papers discuss topics similar to those identified in our work, most touch them at a high-level, without explicitly synthesizing a list of possible solutions to the challenges faced by the practitioners. For example, Di Francesco et al. (2018), Balalaie et al. (2018), Fritzsch et al. (2019), Luz et al. (2018), Gouigoux and Tamzalit (2017), Bogner et al. (2019), and others discuss the challenge of identifying proper service granularity but do not provide a practical set of options collected from practitioners (thus the ◑ in Table 13). Likewise, managing API changes is outlined as a challenge by Balalaie et al. (2016), Zhang et al. (2019), Taibi and Lenarduzzi (2018), Taibi et al. (2020), Bogner et al. (2019), Chen (2018), Soldani et al. (2018) but these papers do not provide alternative solutions and discuss their advantages and disadvantages. Di Francesco et al. (2018), Balalaie et al. (2018), Knoche and Hasselbring (2019), Bucchiarone et al. (2018), Taibi et al. (2020), Bogner et al. (2019), and Chen (2018) discuss the importance of logging and monitoring, but do not provide suggestions on what to log and how to use the logged info. As our study is larger and involves more companies than most of previous work (with exceptions being only Knoche and Hasselbring (2019), Taibi and Lenarduzzi (2018), and Taibi et al. (2020)), we were able to consolidate numerous concerns scattered in multiple papers in related literature.

Table 13 Comparison with related work

Study	Microservice granularity	Microservice ownership	Language diversity	Logging and monitoring	Distributed tracing	Automating processes	Tools	Managing common code	Managing API changes	Managing variants
Post-migration Studies										
Viggiato et al. (2018)	○	○	◐*	○	○	○	○	○	○	○
Soldani et al. (2018)	◐	○	○	◐	◐	○	○	○	◐	○
Bandeira et al. (2019)	○	○	○	○	○	○	○	○	○	○
Alshuqayran et al. (2016)	○	○	○	◐	●	○	○	○	○	○
Zhou et al. (2018)	○	○	○	◐	●	○	○	○	○	○
Chen (2018)	○	○	●	◐	○	○	○	○	◐	○
Bogner et al. (2019)	◐	○	●	◐	○	◐+	●	●	◐	○
Taibi et al. (2020)	◐+	●	●	◐	○	◐+	●	◐*	●	○
Taibi and Lenarduzzi (2018)	◐+	○	●	○	○	○	●	◐*	●	○
Zhang et al. (2019)	◐+	○	●	◐+	●	●+	◐+	○	◐	○
Migration Studies										
Balalaie et al. (2016)	○	○	○	○	○	◐+	○	○	◐	○
Bucchiarone et al. (2018)	○	○	○	◐	○	○	○	○	○	○
Gouigoux and Tamzalit (2017)	◐	○	○	○	○	○	○	○	○	○
Luz et al. (2018)	◐	○	○	○	○	●	○	○	○	○
Fritzsch et al. (2019)	◐	○	◐*	○	○	◐+	○	○	○	○
Knoche and Hasselbring (2019)	○	○	●	◐	○	◐+	○	○	○	○
Taibi et al. (2017)	○	○	○	○	○	●+	○	○	○	○
Ghofrani and Lübke (2018)	◐+	○	○	○	○	○	○	○	○	○
Carvalho et al. (2019)	◐+	○	○	○	○	○	○	○	○	○
Balalaie et al. (2018)	◐	○	○	◐	○	◐+	○	○	○	○
Di Francesco et al. (2018)	◐	○	●	◐	○	◐+	○	○	○	○

Topics — Architecture: Microservice granularity, Microservice ownership, Language diversity. Infrastructure: Logging and monitoring, Distributed tracing, Automating processes. Tools. Code management: Managing common code, Managing API changes, Managing variants.

The ● symbol indicates that the related work is fully aligned with our findings on a topic.

The ◐ symbol indicates that the related work mentions some but not all concepts we identified on a topic.

The ○ symbol indicates that the related work does not cover the topic at all.

The + symbol indicates that the related work mentions additional concepts not covered in our study.

The * symbol indicates that the related work mentions view contradicting those identified in our study.

Table 14 Additional topics mentioned in related work

Group	Study	Managing repositories*	Testing and debugging*	Security*	Deployment*	Data management	Communication protocols	API design	Quality metrics selection	Human and organizational
Post-migration Studies	Viggiato et al. (2018)	○	⊕	○	○	⊕	○	○	○	○
	Soldani et al. (2018)	○	⊕	⊕	○	⊕	○	○	○	○
	Bandeira et al. (2019)	○	○	⊕	○	⊕	⊕	○	○	○
	Alshuqayran et al. (2016)	○	⊕	⊕	⊕	⊕	⊕	⊕	○	○
	Zhou et al. (2018)	○	⊕	○	○	○	○	○	○	○
	Chen (2018)	○	⊕	○	⊕	○	○	○	○	○
	Bogner et al. (2019)	⊕	⊕	○	○	○	○	○	⊕	⊕
	Taibi et al. (2020)	○	⊕	○	○	⊕	⊕	⊕	○	⊕
	Taibi and Lenarduzzi (2018)	○	⊕	○	○	⊕	⊕	⊕	○	○
	Zhang et al. (2019)	○	⊕	○	○	⊕	⊕	○	○	⊕
Migration Studies	Balalaie et al. (2016)	○	○	○	⊕	○	○	○	○	⊕
	Bucchiarone et al. (2018)	○	○	○	○	○	○	○	○	○
	Gouigoux and Tamzalit (2017)	○	○	○	⊕	○	⊕	○	○	○
	Luz et al. (2018)	○	○	⊕	○	⊕	○	○	⊕	⊕
	Fritzsch et al. (2019)	○	○	⊕	○	○	○	○	○	⊕
	Knoche and Hasselbring (2019)	○	○	○	⊕	⊕	○	○	○	⊕
	Taibi et al. (2017)	○	○	○	○	○	⊕	○	○	⊕
	Ghofrani and Lübke (2018)	⊕	⊕	○	○	⊕	○	○	○	⊕
	Carvalho et al. (2019)	○	○	○	○	○	○	○	○	○
	Balalaie et al. (2018)	○	○	○	○	○	○	○	○	○
	Di Francesco et al. (2018)	○	⊕	○	○	○	○	○	○	⊕

The ⊕ symbol indicates that the related work mentions the topic

The ○ symbol indicates that the related work does not cover the topic

The * symbol indicates that the topic was covered by less than three participants of our study

As with any qualitative exploratory study, the list of topics we report on is not intended to be complete but rather represents topics raised by our study participants. Table 14 summarizes the main topics reported by related work, which were not covered by our study. Topics related to managing multiple repositories, testing and debugging, security, and deployment were mentioned by less than three of our study participants; we did not report on those in our results as our goal was to focus on topics of importance that concern a substantial number of participants. We discuss some of these topics in Section 5 and mark them with a * in Table 14. Data management considerations, as well as challenges related to communication protocol selection, API design, and selection of appropriate quality metrics did not come up explicitly in our study. Human and organizational challenges, including developer mindset change and recruitment of skilled personnel, did not come up in our study either; these topics are often important when migrating to microservices. We believe our study participants did not mention these topics to us as we focused our study on teams that develop microservice-based applications for several years, who often have migration-related issues already resolved. We now discuss each of the related papers in detail.

Migration to Microservices. Overall, the work in this area 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. Di Francesco et al. (2018) investigated the practices and challenges in migration to microservices by conducting five interviews and following up with a questionnaire involving 18 practitioners. They observed that finding a proper service granularity and setting up an initial infrastructure for microservices are some of the migration challenges developers face. They also discussed challenges related to the automated support for testing (hence the ◑+ in the "Automating processes" row in Table 13).

Balalaie et al. (2018) reported on a set of migration design patterns that the authors collected from industrial-scale software migration projects, showing how to decompose monolith based on data ownership, transform code-level dependency to service-level dependency, monitor the running microservices, and more. The authors also presented three case studies showing the applicability of their patterns. They highlighted the value of using automated DevOps practices, which did not come up in our study, hence the ◑+ in the "Automating processes" row in Table 13 again.

Carvalho et al. (2019) investigated the usefulness of microservice extraction criteria proposed in academia by surveying 15 practitioners. The results suggested that practitioners often consider multiple criteria, such as coupling, cohesion, communication overhead, and reuse potential simultaneously when extracting microservices from monolith. Yet, academic solutions do not satisfy the industrial needs and existing tools are insufficient for making well-formed decisions on microservice extraction. This paper identified reuse as an additional consideration when splitting a monolithic application to microservices. We thus put a ◑+ in the "Microservice granularity" row in Table 13.

Ghofrani and Lübke (2018) conducted an online survey with 25 industrial and academic participants who answered questions related to goals and challenges in devising a microservice-based architecture, notations used to describe it, and solutions for improving a microservice-based architecture. The authors found that identifying proper service cuts is challenging and that optimization in security, response time, and performance are the major concerns in microservice-based architecture. Our study did not consider security as a criterion when splitting to microservices, but enumerated other important criteria, such as team structure and delivery cycles, hence the ◑+ in Table 13.

Taibi et al. (2017) surveyed 21 practitioners from industry who adopted a microservice-based architecture style for at least two years. Unlike our work, 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 motivation, 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. Similar to our work, the authors also noticed that an investment in setting up automated infrastructure early on pays off in the long term; they also discussed the importance of setting DevOps processes and practices, hence the ●+ in Table 13.

More recently, Knoche and Hasselbring (2019) and Fritzsch et al. (2019) investigated the drivers and barriers of migration to microservices among practitioners in Germany and confirmed findings of Taibi et al. (2017) that maintainability and scalability are the primary motivations for adopting microservices. Major migration barriers include identifying appropriate service boundaries and human skill- and mindset-related challenges. Knoche and Hasselbring (2019) also discussed topics related to logging, monitoring, automating testing infrastructure, and runtime deployment environment, but did not provide any concrete suggestions on, say what to log and how to used the logs, hence the ◑ in Table 13. Interestingly, Fritzsch et al. (2019) suggested that prescribed technological and architectural decisions can be troublesome while our participants suggested the necessity to restrict programming language and technology stack diversity (indicated by a * in Table 13). This divergence likely stems from the fact that most participants in the study by Fritzsch et al. (2019) are still migrating their application to microservices, whereas our participants shared post-transition perspectives after several years of experience with technological heterogeneity. Our findings are also in agreement with other post-migration studies, e.g., Chen (2018), Bogner et al. (2019), Zhang et al. (2019), Taibi and Lenarduzzi (2018), and Taibi et al. (2020).

Luz et al. (2018) shared observations on the transition process of three Brazilian Government Institutions and cross-validated the results by surveying 13 practitioners in the studied institutions. The authors found that the lack of guidance on criteria for decomposing a monolithic system into a set of microservices, infrastructural support, and metrics for evaluating the quality of a microservice-based system are commonly perceived as challenges in the transition process. Gouigoux and Tamzalit (2017) 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. Likewise, Bucchiarone et al. (2018) reported on experience transitioning from monolith to microservices in Danske Bank, showing that such transition improved scalability. Balalaie et al. (2016) 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 (hence the ◑+ in the "Automating processes" row in Table 13). The authors learned that even a small API change can break the system and suggested to use customer-specific contracts to make the system more error-prone. Again, our study does not focus on the migration phase. Moreover, rather than a single case study, we collected practices and challenges from multiple companies successfully completing the migration, helping researchers and practitioners get a better overview of the current state-of-practice in microservice development.

Post-migration Practices and Challenges. Zhang et al. (2019) interviewed practitioners from 13 companies to (a) identify the gaps between the ideal and practical versions of microservice-based applications and (b) to understand the tradeoffs between the benefits

and pains of microservice-based application development. The authors found that missing guidelines on decomposition to microservices, inadequate automation around logging, monitoring, and other types of infrastructural support, as well as excessive technology diversity are some of the pains practitioners face. The also discussed considerations for setting up DevOps and an infrastructural tool topology, designing microservices so that they can be reused in the future, testing and debugging considerations, and topics related to management of data and communication protocols. While some of the authors findings are similar to ours, they did not discuss all considerations for splitting to microservices mentioned in our work, such as those related to resource consumption, what to log and how to utilize the logged info, and considerations related to microservice ownership, management of common code, and management of product variants, as indicated in Table 13.

Taibi and Lenarduzzi (2018) and Taibi et al. (2020) focused on collecting code smells and anti-patterns in microservice-based development. In the former study, the authors interviewed 72 developers from 61 different organizations; they latter extended the study by collecting opinions from another 27 developers from 27 different organizations, all with at least two years of microservice development experience. The developers were asked to rank bad practices in microservice-based development that the authors collected from the literature, report on additional bad practices participants faced, and on solutions they developed. As the result of these studies, the authors identified a taxonomy of 20 anti-patterns, covering both organizational and technical aspects of microservice-based development. The identified anti-patterns include excessive language diversity, usage of local vs. distributed logging infrastructure, lack of proper version for APIs, and management of common code. The authors also identified anti-patterns related to testing and debugging, design of APIs, communication protocols, and data management, as indicated in Table 14. Our study confirms some of these findings, while identifying additional considerations related to management of product variants and robust infrastructural support, e.g., the use of distributed tracing. Our study also explicitly mentions responsibilities of service owners, what to log and how to use the logged info, and more. In the realm of common code management, we are the first to report on using a sidecar as a practical solution to the problem. Furthermore, while the studies by Taibi et al. identify shared library as an anti-pattern in microservice-based development, our study shows that several practitioners report on advantages of using such a solution in practice. We mark this divergence with a * in Table 13.

Bogner et al. (2019) interviewed 17 engineers from ten companies, collecting processes, tools, metrics, patterns applied, and challenges faced by practitioners when ensuring sustainable evolution of microservices. Like in our study, the authors also reported that several participants believe technological heterogeneity beneficial while others prefer a more sensible handling of technology diversities. The authors also reported on the need for test automation, clear guidelines about establishing the right service granularity, logging and monitoring infrastructure, and managing breaking API changes, but did not provide concrete solutions (hence the ◑ in Table 13). Performing in-depth interviews with a substantially higher number of practitioners led us to discover additional challenges not mentioned in this work, e.g., the need to redefine service granularity in resource-constrained environments, the importance of having clear service ownership, and issues related to management of variants.

Chen (2018) shared the post-migration challenges and the corresponding solutions in an Irish sports betting company called Paddy Power. The author highlighted challenges related to technology and language diversity, management of service contracts, testing, and having

a large number of services. While our study confirms some of the findings, we also identified potential solutions to some of these challenges. Again, reaching out to a larger set of companies also allowed us to identify additional challenges and solutions related to managing infrastructure, microservice variants, and more. Zhou et al. (2018) focused specifically on fault analysis and debugging of microservices. The authors interviewed 16 practitioners from 12 companies, investigating the fault cases these practitioners encounter, debugging practices they use, and practical challenges the practitioners face when debugging microservices. The authors reported that only around one third of participants in their study employ distributed tracing while the remaining ones rely on localized logging only, indicating the immature debugging practice for microservices among practitioners. Our study also emphasize the importance of distributed tracing but does not exclusively focus on fault analysis and debugging processes.

Alshuqayran et al. (2016) conducted a mapping study for identifying architectural challenges, popular microservice architectural diagram types, and quality measurement in microservice-based architectures mentioned in the literature. By studying 33 papers on microservice-based development, the authors identified and classified keywords associated with challenges of building microservice-based systems; they identified challenges related to communication and integration of services, deployment operations, performance, and more. Like in our study, the authors also mentioned challenges related to distributed tracing and setting up a logging and monitoring infrastructure, but did not provide any information on how and what to log, hence the ◑ in Table 13.

Bandeira et al. (2019) analyzed 781 microservice-related posts on StackOverflow to understand what topics are being discussed the most by the microservice community. The authors found that developers discuss topics related to communication, security, data handling, integration testing, and versioning of microservices. Soldani et al. (2018) reviewed grey literature to identify benefits and challenges of microservices. The authors found that determining the right granularity of microservices, devising security policies, managing distributed storage, application testing, and resource management are among the most reported challenges practitioners face. Again, the authors did not identify any practical solutions to the reported challenges.

Viggiato et al. (2018) collected the general characteristics, advantages, and challenges of microservice-based development from literature and online posts. The authors found that the advantages of microservices reported in the literature include independent deployment, technology diversity, scalability, and maintainability (easy to replace), while top challenges are distributed transactions, integration tests, service faults, and inter-service communication. The authors then surveyed 122 online participants who agreed with most benefits and challenges reported in the literature. While our study suggests that mature organizations prefer to limit technological diversity, the study by Viggiato et al. reported that to be an advantage of microservice-based development, hence the ◑* in Table 13.

Leitner et al. (2019) investigated Function-as-a-Service (FaaS) applications by conducting a mix-method study that combined semi-structured interviews, the analysis of online articles related to FaaS, and the analysis of responses from a web-based survey. Our work is orthogonal to that as we did not include FaaS applications in our study. For that reason, we do not include this paper in Table 13.

Meta-studies on Microservice-based Development. Pahl and Jamshidi (2016) conducted a systematic mapping study on 21 reports that were published by the end of 2015, to identify, classify, and compare the existing research body on microservices and their application in the cloud. The authors concluded that microservice research is still in an early stage and

there is a lack of tool support to facilitate microservice-based developments. Di Francesco et al. (2017) performed a systematic mapping study that included 71 papers, classifying and evaluating publication trends, research focus, and industrial adoption potential for microservices. Di Francesco et al. (2019) further extended this work to include 32 additional papers. The authors found that existing research has a strong focus on cloud and mobile paradigms, legacy migration, and tradeoffs between complexity and flexibility of microservices. Yet, the technology readiness levels of most studies suggest that industrial adoption is still far away. Vural et al. (2017) also performed a systematic mapping study that included 37 papers, aiming to find current trends, motivation behind microservices research, emerging standards, and the possible research gaps. Like Di Francesco et al., the authors also concluded that microservice research is still immature. Finally, Dragoni et al. (2017) reviewed the history of software architecture, discussed characteristics of microservices, and outlined future challenges. The authors state that security and dependability will become key challenges of microservices. Our work is orthogonal to these mapping studies as we focus on identifying challenges related to the adoption of microservices in industry rather than surveying the state-of-the-art in literature.

8 Conclusion

This paper reports on best practices, lessons learned, and challenges collected using a mixed-method study with practitioners that successfully adopted microservice-based architectures. The study included in-depth interviews with 21 practitioners and a follow-up survey that received 37 replies satisfying our selection criteria.

Our participants identified a clear sense of ownership, strict API management, automated processes, and investment in robust logging and monitoring infrastructure as some of the best practices that contribute 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 managing code shared between microservices and managing multiple product variants. Based on the study, we reported on alternative solutions practitioners employ to address these challenges and identified lessons learned from their experience. We also identified potential next steps the community can take to further facilitate efficient software engineering practices in developing microservice-based applications. These include efficient solutions for managing versions and variants for microservices, assessing the impact of API changes, tools for performance debugging, and more.

We believe our study 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.

References

Alshuqayran N, Ali N, Evans R (2016) A systematic mapping study in microservice architecture. In: Proceedings of IEEE international conference on service-oriented computing and applications (SOCA), pp 44–51

Balalaie A, Heydarnoori A, Jamshidi P (2016) Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw 33(3):42–52

Balalaie A, Heydarnoori A, Jamshidi P, Tamburri DA, Lynn T (2018) Microservices migration patterns. Softw: Pract Exp 48(11):2019–2042

Bandeira A, Medeiros CA, Paixao M, Maia PH (2019) We need to talk about microservices: an analysis from the discussions on stackoverflow. In: Proceedings of the 16th international conference on mining software repositories (MSR), pp 255–259

Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin RC, Mellor S, Schwaber K, Sutherland J, Thomas D (2001) Manifesto for agile software development. https://agilemanifesto.org (Last accessed: July 2020)

Beschastnikh I, Wang P, Brun Y, Ernst MD (2016) Debugging Distributed Systems. Commun ACM 59(8):32–37

Bogner J, Fritzsch J, Wagner S, Zimmermann A (2019) Assuring the evolvability of microservices: insights into industry practices and challenges. In: Proceedings of the 2019 IEEE international conference on software maintenance and evolution (ICSME), pp 546–556

Bratthall L, Jørgensen M (2002) Can you trust a single data source exploratory software engineering case study? Empir Softw Eng 7(1):9–26

Bucchiarone A, Dragoni N, Dustdar S, Larsen ST, Mazzara M (2018) From monolithic to microservices: an experience report from the banking domain. IEEE Softw 35(3):50–55

Carvalho L, Garcia A, Assunção WKG, de Mello R, de Lima MJ (2019) Analysis of the criteria adopted in industry to extract microservices. In: Proceedings of the joint 7th international workshop on conducting empirical studies in industry and 6th international workshop on software engineering research and industrial practice, pp 21–30

Chen L (2018) Microservices: architecting for continuous delivery and DevOps. In: 2018 IEEE international conference on software architecture (ICSA), pp 39–46

Clements P, Northrop L (2002) Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc

Cockroft A (2014) Migrating to microservices. https://youtu.be/1wiMLkXz26M, (Last accessed: July 2020)

Di Francesco P, Malavolta I, Lago P (2017) Research on architecting microservices: trends, focus, and potential for industrial adoption. In: Proceedings of IEEE international conference on software architecture (ICSA), pp 21–30

Di Francesco P, Lago P, Malavolta I (2018) Migrating towards microservice architectures: an industrial survey. In: Proceedings of IEEE international conference on software architecture (ICSA), pp 29–38

Di Francesco P, Lago P, Malavolta I (2019) Architecting with microservices: a systematic mapping study. J Syst Softw 150:77–97

Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L (2017) Microservices: yesterday, today, and tomorrow. Springer International Publishing, pp 195–216

Fenn J, Linden A (2005) Gartner's hype cycle special report for 2005. https://www.gartner.com/doc/484424/gartners-hype-cycle-special-report (Last accessed: July 2020)

Fielding RT (2000) Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine

Flanigan TS, McFarlane E, Cook S (2008) Conducting survey research among physicians and other medical professionals: a review of current literature. In: Proceedings of the survey research methods section. American Statistical Association, pp 4136–4147

Fowler M (2015) Microservice trade-offs. https://martinfowler.com/articles/microservice-trade-offs.html, (Last accessed: July 2020)

Francis J, Johnston M, Robertson C, Glidewell L, Entwistle V, Eccles M, Grimshaw J (2010) What is an adequate sample size?: Operationalising data saturation for theory-based interview studies. Psychol Health 25(10):1229–1245

Fritzsch J, Bogner J, Wagner S, Zimmermann A (2019) Microservices migration in industry: intentions, strategies, and challenges. In: Proceedings of the 2019 IEEE international conference on software maintenance and evolution (ICSME), pp 481–490

Ghofrani J, Lübke D (2018) 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), pp 1–8

Goodman L (1961) Snowball sampling. Ann Math Stat 32(1):148–170

Google Cloud (2019) Client libraries explained. https://cloud.google.com/apis/docs/client-libraries-explained, (Last accessed: July 2020)

Gouigoux JP, Tamzalit D (2017) 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), pp 62–65

Heinrich R, van Hoorn A, Knoche H, Li F, Lwakatare LE, Pahl C, Schulte S, Wettinger J (2017) Performance engineering for microservices: research challenges and directions. In: Companion proceedings of the 8th ACM/SPEC on international conference on performance engineering (ICPE), pp 223–226

Hunt A, Thomas D (1999) The pragmatic programmer: from journeyman to master. Addison-Wesley Longman Publishing

Jackson D, Daly J, Davidson P, Elliott D, Cameron-Traub E, Wade V, Chin C, Salamonson Y (2000) Women recovering from first-time myocardial infarction (MI): a feminist qualitative study. J Adv Nursing 32(6):1403–1411

Jassim GA, Whitford DL (2014) Understanding the experiences and quality of life issues of Bahraini women with breast cancer. Social Sci Med 107:189–195

Knoche H, Hasselbring W (2019) Drivers and barriers for microservice adoption—a survey among professionals in Germany. Int J Concept Model 14(1):1–35

Krippendorff K (2011) Agreement and information in the reliability of coding. Commun Methods Meas 5(2):93–112

Kruchten P, Nord RL, Ozkaya I (2012) Technical debt: from metaphor to theory and practice. IEEE Softw 29(6):18–21

Leitner P, Wittern E, Spillner J, Hummer W (2019) A mixed-method empirical study of function-as-a-service software development in industrial practice. J Syst Softw 149:340–359

Lewis J, Fowler M (2014) Microservices: a definition of this new architectural term. https://www.martinfowler.com/articles/microservices.html (Last accessed: July 2020)

Liu X, Guo Z, Wang X, Chen F, Lian X, Tang J, Wu M, Kaashoek MF, Zhang Z (2008) D3S: debugging deployed distributed systems. In: Proceedings of USENIX symposium on networked systems design and implementation (NSDI), pp 423–437

Luz W, Agilar E, de Oliveira MC, de Melo CER, Pinto G, Bonifácio R (2018) An experience report on the adoption of microservices in three brazilian government institutions. In: Proceedings of Brazilian Symposium on Software Engineering (SBES), pp 32–41

McLarty M (2016) Microservice architecture is agile software architecture. https://www.infoworld.com/article/3075880/microservice-architecture-is-agile-software-architecture.html (Last accessed: July 2020)

Mitra S (2018) Dilemma on utility modules: making a jar or a separate microservice?. https://dzone.com/articles/dilemma-on-utility-module-making-a-jar-or-separate-2 (Last accessed: July 2020)

Morse JM (1995) The significance of saturation. Qual Health Res 5(2):147–149

MuleSoft (2018) Microservices and DevOps: better together. https://www.mulesoft.com/resources/api/microservices-devops-better-together (Last accessed: July 2020)

Nadareishvili I, Mitra R, McLarty M, Amundsen M (2016) Microservice architecture: aligning principles, practices, and culture. O'Reilly Media

Newcomer E, Lomow G (2005) Understanding SOA with web services. Addison-Wesley

O'Connor C, Joffe H (2020) Intercoder reliability in qualitative research: debates and practical guidelines. Int J Qual Methods

Pahl C, Jamshidi P (2016) Microservices: a systematic mapping study. In: Proceedings of the international conference on cloud computing and services science (CLOSER), pp 137–146

Pohl K, Böckle G, Linden FJvd (2005) Software product line engineering: foundations, principles and techniques. Springer, Berlin

Postel J (1980) DoD standard transmission control protocol. RFC 761:1–88

Richardson C (2014) Microservice architecture. https://microservices.io/, (Last accessed: July 2020)

Richardson C (2018a) Pattern: API gateway/backends for frontends. https://microservices.io/patterns/apigateway.html, (Last accessed: July 2020)

Richardson C (2018b) Pattern: distributed tracing. https://microservices.io/patterns/observability/distributed-tracing.html, (Last accessed: July 2020)

Richardson C (2018c) Who is using microservices? https://microservices.io/articles/whoisusingmicroservices.html (Last accessed: July 2020)

Rossi C, Shibley E, Su S, Beck K, Savor T, Stumm M (2016) Continuous deployment of mobile software at Facebook (Showcase). In: Proceedings of the ACM SIGSOFT international symposium on foundations of software engineering (FSE), pp 12–23

Seifermann V (2017) Application performance monitoring in microservice-based systems Bachelor's thesis, Institute of Software Technology Reliable Software Systems, University of Stuttgart

Sinkowitz-Cochran RL (2013) Survey design: to ask or not to ask? That is the question.. Clin Infect Dis 56(8):1159–1164

Soldani J, Tamburri DA, Heuvel WJVD (2018) The pains and gains of microservices: a systematic grey literature review. J Syst Softw 146:215–232

Strauss A, Corbin J (1998) Basics of qualitative research: techniques and procedures for developing grounded theory. Sage, Thousand Oaks

Taibi D, Lenarduzzi V (2018) On the definition of microservice bad smells. IEEE Softw 35(3):56–62

Taibi D, Lenarduzzi V, Pahl C (2017) Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput 4(5):22–32

Taibi D, Lenarduzzi V, Pahl C (2020) Microservices anti-patterns: a taxonomy. Springer International Publishing, pp 111–128

Viggiato M, Terra R, Rocha H, Valente MT, Figueiredo E (2018) Microservices in practice: a survey study. In: Brazilian workshop on software visualization, evolution and maintenance (VEM), pp 1–8

Vural H, Koyuncu M, Guney S (2017) A systematic literature review on microservices. In: Proceedings of international conference on computational science and its applications (ICCSA), pp 203–217

Wang Y, Kadiyala H, Rubin J (2020) Promises and challenges of microservices: an exploratory study. https://osf.io/8mxeg/wiki/home/ (Last accessed: July 2020).

Weiss D, Lai CTR (1999) Software product-line engineering: a family-based software development process. Addison-Wesley Longman Publishing Co., Inc

Zhang H, Li S, Zhang C, Jia Z, Zhong C (2019) Microservice architecture in reality: an industrial inquiry. In: 2019 IEEE international conference on software architecture (ICSA), pp 51–60

Zhou X, Peng X, Xie T, Sun J, Ji C, Li W, Ding D (2018) Fault analysis and debugging of microservice systems: industrial survey, benchmark system, and empirical study. IEEE Trans Softw Eng 14(8):1–18

Zimmermann O (2017) Microservices tenets: agile approach to service development and deployment. Comput Sci–Res Dev 32(3):301–310

Yingying Wang is a PhD student in the Department of Electrical and Computer Engineering at the University of British Columbia, Canada (UBC). She is a member of the Reliable, Secure, and Sustainable Software Lab. Yingying received her Master's degree at UBC and her Bachelor's degree at Dalian University of Technology, China. Her research interests are in software engineering, collaborative software development, program analysis, and software security.

Harshavardhan Kadiyala is a Master's student in the Department of Electrical and Computer Engineering at the University of British Columbia, Canada (UBC). He also works in the Reliable, Secure, and Sustainable Software Lab. Before coming to UBC, Harshavardhan worked as System Engineer at Siemens Healthineers, India. His current research is focusing on improving resource utilization and performance of microservice-based applications deployed in the cloud; he has general interest in distributed, concurrent, and parallel systems.

Julia Rubin is a faculty member in the Department of Electrical and Computer Engineering at the University of British Columbia, Canada. She received her PhD in Computer Science from the University of Toronto, Canada and worked as a postdoctoral researcher in Computer Science and Artificial Intelligence Lab at MIT, USA. Earlier, Julia spent 10 years in industry, working for a startup company and then for IBM Research, where she was a research staff member and, part of the time, a research group manager. Julia's research interests are in software engineering, program analysis, software security, and in reliability of software and AI systems. Her work in these areas won numerous prestigious Distinguished and Best Paper Awards in major conferences in the field.

Affiliations

Yingying Wang[1] · **Harshavardhan Kadiyala[1]** · **Julia Rubin[1]**

Harshavardhan Kadiyala
devkhv129@ece.ubc.ca

Julia Rubin
mjulia@ece.ubc.ca

[1] Department of Electrical and Computer Engineering, The University of British Columbia, 2332 Main Mall, Vancouver, BC V6T 1Z4, Canada