Designing Microservice Systems Using Patterns:
An Empirical Study on Quality Trade-Offs

Guilherme Vale
Faculty of Engineering
University of Porto
Porto, Portugal
up201709049@edu.fe.up.pt

Filipe Figueiredo Correia
Faculty of Engineering
University of Porto
INESCTEC
Porto, Portugal
filipe.correia@fe.up.pt

Eduardo Martins Guerra
Computer Science Faculty
Free University of Bozen-Bolzano
Bolzano, Alto Adige, Italy
eduardo.guerra@unibz.it

Thatiane de Oliveira Rosa
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, SP, Brazil
thatiane@ime.usp.br

Jonas Fritzsch
Institute of Software Engineering
University of Stuttgart
Stuttgart, Germany
jonas.fritzsch@iste.uni-stuttgart.de

Justus Bogner
Institute of Software Engineering
University of Stuttgart
Stuttgart, Germany
justus.bogner@iste.uni-stuttgart.de

Abstract—The promise of increased agility, autonomy, scalability, and reusability has made the microservices architecture a *de facto* standard for the development of large-scale and cloud-native commercial applications. Software patterns are an important design tool, and often they are selected and combined with the goal of obtaining a set of desired quality attributes. However, from a research standpoint, many patterns have not been widely validated against industry practice, making them not much more than interesting theories. To address this, we investigated how practitioners perceive the impact of 14 patterns on 7 quality attributes. Hence, we conducted 9 semi-structured interviews to collect industry expertise regarding (1) knowledge and adoption of software patterns, (2) the perceived architectural trade-offs of patterns, and (3) metrics professionals use to measure quality attributes. We found that many of the trade-offs reported in our study matched the documentation of each respective pattern, and identified several gains and pains which have not yet been reported, leading to novel insight about microservice patterns.

Index Terms—software architecture, microservices, design patterns, software design trade-offs

I. INTRODUCTION

Designing successful microservices-based applications requires mastering a set of new architectural insights, principles, and practices. These aspects are, however, still far from being fully understood. One such aspect is the use of patterns; as professionals encounter and overcome challenges that arise from this architectural approach, shared solutions to specific problems begin to emerge. The documentation of these patterns gives software architects an array of techniques to common hurdles, thus reducing the technical risk to their projects by not having to employ new and untested designs.

On the other hand, patterns are not *inherently* tried-and-true [1]. Although they are usually documented when they have been observed in practice at least three times, such a *convenience sample* [2] often does not provide enough evidence of their validity. Therefore, there is value in subjecting patterns to further evaluation, like with any other type of theory [3].

The promises of microservices—*e.g.*, of increased agility, scalability, maintainability, performance—can only be leveraged with a careful understanding of its underlying principles. Amid the hype created in recent years, as many big industry players pushed this architecture [4], the approaches that work best for adopting it are still an object of study. For this reason, it is not uncommon to find contexts wherein microservices would be a good fit, but where teams fail to implement them successfully [5]. Discovering how industry experts currently perceive the influence of design patterns on quality attributes (QAs) will thus help to judge their benefits and relevance, which forms the motivation for this work.

II. RESEARCH APPROACH

We seek a more in-depth understanding of microservice design patterns by way of an empirical study of their trade-offs. Specifically, we want to obtain insights on the relevance of design patterns in industry, how practitioners perceive their influence on software qualities as a consequence of their usage, and what metrics practitioners use, if any, to determine these derived effects, reflected as software qualities. Based on these objectives, we defined the following research questions (RQs).

- **RQ1:** *What is the rationale for the adoption of patterns in microservices systems?* — Considerations: how relevant are microservice design patterns in industry? What is their utility? Are developers aware of the problems that benefit from the recommended usage of these patterns?
- **RQ2:** *How are QAs influenced as a result of applying microservice patterns?* — Considerations: what are the trade-offs, in terms of advantages and disadvantages, of current microservice patterns as perceived by professionals?
- **RQ3:** *How are QAs measured in microservices?* — Considerations: do practitioners evaluate their non-functional requirements? If so, how? With which techniques and resources? If not, why?

To answer these RQs, we opted for a semi-structured interview study with microservices experts, as face-to-face interviews can help us establish a more effective communication channel with participants.

A. Research scope

Several patterns have been described in the scope of the microservices architecture and cloud-native development. We sought to select a set of patterns as a basis for our study that would be broad, consistent, detailed, and regarded as a reference. Some works, such as Richardson's [6] do not describe the patterns themselves, or when they do, the descriptions vary widely in depth, often not identifying the consequences inherent to applying the patterns[1]. Other pattern languages did not seem to be sufficiently broad [7]–[12] or were in very early and active development [13].

Considering our focus on industry-relevance, we selected a set of patterns from the Azure Architecture Center (AAC), a repository of software architecture documentation hosted by Microsoft. The AAC has an extensive language of patterns termed the *Cloud Design Patterns* [14], divided into 7 categories, with most patterns belonging to more than one category. In particular, we selected the patterns of the *design and implementation* category, as they address the most general aspects of designing a microservices architecture.

The categorization and taxonomy of architectural patterns for microservices varies significantly in literature: Márquez and Astudillo [15] assigned 17 architectural patterns used in open-source projects to 11 groups while Valdivia et al. [16] identified 54 patterns from white and grey literature and divided them into 6 groups. The taxonomy suggested by Osses et al. [17] who reported 124 architectural patterns in the academic and industrial field has the greatest overlap with Microsoft's AAC. 8 of the 14 patterns in the AAC *design and implementation* category were also tagged as design patterns by Osses et al. while another one occurs as a migration pattern.

The patterns in the AAC are available in a public GitHub repository[2] that is actively maintained and has frequent community contributions. We have analyzed 14 specific patterns (*cf.* Section III) in the version corresponding to the commit with hash `eff408a`, from June 11th 2021.

As for QAs, we turned to the work by Li *et al.* [18], which dealt with questions of assessing QAs in microservice systems, as it matched our desired level of architectural perspective. The authors produce a list of six QAs most relevant to the microservices architecture. For our part, we extend this set of QAs to include a seventh element, *maintainability*. The list that follows contains the full set of QAs of our study, a brief exploration of each, and a description of their relevance to the microservices architecture.

Scalability. A measure of a system's ability to handle a varying number of requests [19]. A microservices architecture is often adopted to maximize aspects of scalability,

given its proximity to cloud-native technologies and ability to simplify both vertical and horizontal scaling techniques.

Performance. A measure of a system's ability to meet time requirements when responding to inputs, requests, or events [19]. As a distributed architecture, the latency in network requests involved in inter-service communication is an obvious source of performance hindrance. Established lightweight and REST-based communication mechanisms are commonly used to mitigate this threat.

Availability. A measure of a system's ability to repair faults within a defined period so as not to incur significant performance penalties [19]. As such, we can think of *availability* as interchangeable with *reliability* [18]. Since services typically have at least one dependency, downtime in one faulty service can easily have implications that extend to otherwise well-behaved services. Several patterns for addressing availability are known, most notably circuit breakers, fault monitors, and service registries.

Monitorability. A measure of a system's ability to support the operations staff in monitoring the system at runtime, *i.e.*, while it is executing [19]. Any distributed system, conceived as a microservice system or otherwise, exhibits a high level of dynamic structure and behavior, representing an added challenge for monitoring [18]. This QA is often also referred to as *observability*. We use the two terms interchangeably throughout this work.

Security. A measure of a system's ability to protect data from unauthorized access while enabling access to authorized users and systems [19]. The distribution of application logic into different processes results in complex network interaction models between services [18]. As a result, this added complexity can be more easily exploited by attackers if *security* is neglected.

Testability. A measure of a system's ability to demonstrate its faults through testing—typically execution-based testing [19]. Due to the complex relationship between services, *testability* is an important property that helps to detect and minimize impacts on other qualities like *performance*, *security* or *availability* [18].

Maintainability. It concerns the balancing act between the time and resources required to apply changes and the need to introduce new business logic or improve system functionality [19]. Poor design can easily inflate the cost of change. The coupling of different services may imply that changes to a service cannot be realized without orchestrating similar changes in services depending on it, which is a sign of poor maintainability.

B. Interview design

Semi-structured interviews help to explore both planned-ahead and unforeseen topics. We used the advice of Hove and Anda [20] and of Seaman [21] for planning and conducting the interviews. As well, we considered the guidelines by Ralph *et al.* [22], which provide checklists that help to ensure scientific rigor for empirical studies in software engineering.

---

[1]In addition to the book, which does not describe the patterns themselves, Richardson makes a few patterns available at https://microservices.io.

[2]Available at https://github.com/MicrosoftDocs/architecture-center

All 9 interviews followed the below outlined three-part structure, that was initiated by general and preparatory remarks, *i.e.*, greetings, personal presentation, study presentation and obtaining authorization for recording:

Part 1: Personal experience with microservices. We ask the participant to elaborate freely on their professional experience with microservices. We ask them to consider details like the number of services that were in development and operation in these projects, the number of lines of code that composed the system, the service communication mechanisms used, and the deployment strategies used.

The goal of this part is to understand the microservice systems the participant has worked with, helping to contextualize their answers in the rest of the interview.

Part 2: Design pattern trade-offs. We present the participant with the design patterns we selected in Section II-A, using a helper slideshow document. Each pattern was described by a different slide, with its name, brief description, and a figure. The figures are taken directly from the respective documentation of the patterns in the AAC. In any case, for each pattern, we show its corresponding slide, briefly describe it to the participant and then immediately ask if the participant has ever applied this pattern. If the answer is no, we skip to the next pattern. If the answer is yes, we ask open-ended questions about the decisions that motivated the adoption of the pattern and what they felt was improved and worsened, in a global sense: not necessarily what was impacted on a localized level (*e.g.*, per service) but on a generalized level.

The goal of this part is twofold: to understand the rates of knowledge and adoption of the patterns, and to understand the trade-offs practitioners perceive as inherent to each.

Part 3: Microservice quality metrics. We present the participant with the QAs we selected in Section II-A. We then ask the participant to go through the set of QAs, in any order they choose, and to tell us, for each attribute, how much of a concern it is for their company, how they measure it, and how often they measure it.

The goal of this part is to determine the perceived importance of software qualities, plus the metrics used to evaluate them.

C. Sampling strategy

We chose to use *purposive sampling* [2] and reached out to colleagues or associates in the industry who either served as participants or forwarded us to potential participants. We also attempted *heterogeneity sampling* by contacting candidates from a breadth of different industry domains and performing different roles. All in all, we have interviewed, over a period of two and a half months, 9 professionals from 9 different companies; all with development offices in Portugal. An overview of the participants is shown in Table I.

The interviewees work in widely different industries, with only 2 industry domains appearing more than once—*artificial intelligence* and *finance*. All participants have at least 5 years of experience with software development. A third of them (3/9) reported 5 or more years of experience with microservices, the remaining two thirds (6/9) have between 2 and 4 years of experience with microservices. Some participants are in roles that play a strong part in defining the system architecture (*i.e.*, CTO, architect, head of engineering), others require, at the very least, a sound understanding of the architecture (*i.e.*, operations, developer, delivery manager). The remaining two had professional profiles and *de facto* roles that went beyond their official role (*i.e.*, i_5 often is involved in technical leadership, and i_4 is concerned with internal quality aspects, which extend to the architecture of the system).

The transcription of the interviews was done manually using the *oTranscribe* tool. A total of $47\,322$ words were transcribed from 537 minutes and 40 seconds of recorded video.

D. Data analysis

The data analysis in this study was performed with a combination of manual and computer-assisted techniques: we used the qualitative data analysis tool NVivo for coding the large amount of text data we obtained, and we manually identified themes and other findings into a structured form. In an ongoing process parallel to data collection, we performed initial open theory-driven coding, a process of breaking data apart and defining boundaries to stand for blocks of raw data [23]. By envisioning a set of expected topics that we had foreseen *a priori* from our RQs, we developed the top-level codes *Preliminary*, *Design Patterns*, *Measuring QAs*, and *Miscellaneous* and a code for every pattern and QA.

TABLE I
PARTICIPANT DEMOGRAPHICS.

Interviewee	Role	Years of experience	Years with microservices	Industry
i_1	Architect	10 to 14	5 or more	Artificial Intelligence
i_2	Chief Technology Officer	10 to 14	2 to 4	Retail
i_3	Operations	5 to 9	2 to 4	Artificial Intelligence
i_4	QA	5 to 9	2 to 4	Consultancy
i_5	Scrum Master	5 to 9	5 or more	Automotive
i_6	Chief Technology Officer	15 to 19	2 to 4	Logistics
i_7	Developer	5 to 9	2 to 4	Gaming
i_8	Head of Engineering	10 to 14	2 to 4	Finance
i_9	Delivery Manager	10 to 14	5 or more	Finance

These form the structural areas of our analysis. Through later axial coding, a higher level of coding [24], we establish links between the codes through deductive reasoning: each pattern is a subcategory of the *Design Patterns* code, similarly for each QA being a subcategory of *Measuring QAs*.

With our raw data extracted, we assigned these codes to reduce and simplify our data. When suitable, we expanded the data (forming new connections between concepts) or reconceptualized it (rethinking our previous theoretical assumptions). At the same time, we examined how the theory guiding our research was supported or contradicted by the data, as well as the impact of the data on the current research literature. An overview of this process is shown in Figure 1. We make available the materials used by the researchers to conduct the study, as well as aggregated and anonymized data produced in this context [25].
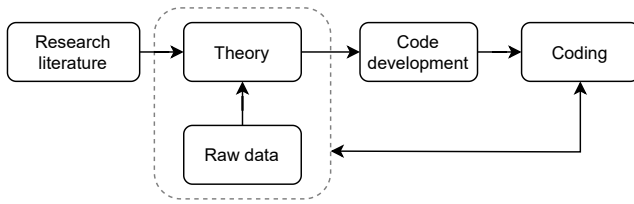

Fig. 1. Coding process.

III. RESULTS

The following sections describe our results and explore their implications and limitations. We map the first three sections to our RQs, in Sections III-A, III-B and III-C, respectively. In Section IV-C we discuss the threats to the validity of our results. We present these findings almost always in a table format, with a column "#" representing the number of unique respondents that mentioned a particular theme, *i.e.*, the number of occurrences of a given idea.

A. Rationale for pattern adoption

Table II presents the reported adoption of patterns by our participants. The average number of patterns adopted by a participant is 9. Only 1 of the 9 participants (i_2) reported having had experience with all 14 patterns. Only 2 patterns were applied by all, STRANGLER and GATEWAY ROUTING, with the former receiving particular praise from most respondents.

Many participants recognized many of the patterns but had not implemented them themselves. The LEADER ELECTION pattern is probably the best to illustrate this point: all but one participant were aware that leader election is a strategy used in distributed systems, *e.g.*, in database management systems and container orchestration mechanisms, but only three reported having actually implemented it in their products.

In some instances, participants responded with having adopted a pattern but, upon further discussion regarding their application, we concluded that it was not an application of the pattern at hand. Two participants erroneously said they applied the AMBASSADOR pattern, confusing it with a gateway.

Additionally, one participant interpreted the CQRS pattern as referring simply to the usage of read-only database replicas to improve performance. While using read replicas is one way to set up a read store with this pattern, a crucial component is the separation of a data model in two, one for updates (commands) and one for reads (queries). We assign these instances a ◐ value. These most likely reflect miscommunication or misunderstandings; instances of our inability to convey the essence of a pattern to respondents. As a result, in these cases, we ignore the participants' perceptions of trade-offs to not jeopardize or corrupt our findings in the next section.

B. Pattern trade-offs

In the following sections, we explore the findings related to the professional perception of the software qualities affected by each pattern. We express the trade-offs of the patterns through the form of *gains* and *pains*. Table III summarizes the trade-offs as reported by participants for each pattern. We attach one of two symbols to each concern raised by respondents. The ✎ symbol denotes remarks which are already identified in the AAC documentation, and the ✳ symbol denotes novel remarks which are not present in the AAC.

We note here that, when asked for advantages and disadvantages, participants rarely invoked QAs directly, referring instead to benefits and liabilities in terms of concerns raised at design, development or operation times. As such, the findings mix QAs, when they were mentioned explicitly in responses, and the aforementioned procedures or properties that were relayed to us.

1) STRANGLER: Applied by all nine participants. Over half of them mention **scalability** as a gain received from applying this pattern, with **maintainability** trailing close by. Additionally, the fact that the strangler façade allows developers to keep working on migrating to microservices without having to demand adjustments from clients—what we term, for our purposes, as **consumer transparency**—was a major advantage that practitioners mentioned. As for downsides, the **costs to maintain** the arrangement this pattern demands and the careful **planning of transition periods** required to ensure successful migrations received most mentions. One participant reported difficulties **managing data consistency** between subsystems, and another reported **initial resistance by the development team** when presenting the idea for this pattern. Finally, another participant noted **backwards compatibility** as a particular challenge introduced by this pattern.

2) ANTI-CORRUPTION LAYER: Applied by four participants. All of them note the same advantage with this pattern: the **decoupling between systems**. On the other hand, the **complexity of the layer** was mentioned by three participants as a serious drawback to be considered. Additionally, a negative impact on **performance** was noted by one participant.

3) SIDECAR: Applied by five participants. All of them noted **monitorability** as the major gain of this pattern. **Maintainability** also followed in the set of benefits of sidecars. One participant additionally suggested **scalability** and **security** as further gains from this pattern. Additionally, one participant

TABLE II
Reported patterns adopted by interviewees, ordered by frequency of adoption (#).

Pattern	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	#
GATEWAY ROUTING	●	●	●	●	●	●	●	●	●	9
STRANGLER	●	●	●	●	●	●	●×	●×	●×	9
EXTERNAL CONFIGURATION STORE	●	●	●	○	●	●	●×	●	●	8
STATIC CONTENT HOSTING	●	●	●	○	●	●	●	●	●	8
PIPES AND FILTERS	●	●	●	●×	●	●	●	○	●	8
COMMAND QUERY RESPONSIBILITY SEGREGATION	●	●	○	○	◐	●	●×	●×	●	6
GATEWAY OFFLOADING	●	●	●	○	●	●	○	●	○	6
BACKENDS FOR FRONTENDS	●	●	○	○	●	●	○	○	●	5
COMPUTE RESOURCE CONSOLIDATION	●×	●×	○	○	●×	●	○×	○	●×	5
SIDECAR	●	●	○	●	●	●	○	○	○	5
ANTI-CORRUPTION LAYER	○	●	○×	●	○	○×	○	●×	○	4
AMBASSADOR	●	●	◐	○	●	○	○	◐	○	3
GATEWAY AGGREGATION	●	●	○	○	●	○	○	○	○	3
LEADER ELECTION	○	●	○	○	●	○	●	○×	○	3
Number of patterns adopted by participant	12	14	6	5	12	10	7	7	9	

Legend:

- ●   The pattern was said to have been applied by the participant;
- ●×   The pattern was said to have been applied, but the participant was not familiar with the given pattern's name;
- ○   The pattern was said to not have been applied by the participant;
- ○×   The pattern was said to have neither applied nor even known to the participant
- ◐   The pattern was said to have been applied but the participant's description of their usage of the pattern does not match the intent originally documented.

mentioned **testability** as a facet enhanced by sidecars, with an interesting use case to illustrate. Participants seemed happy with this pattern, with only one pointing out **operational complexity** as a pain brought about by its adoption.

4) AMBASSADOR: Applied by three participants. **Availability** was praised by all three participants as a major gain from this pattern, as it enables easy implementation of resilience mechanisms. Additionally, the proxy nature of the ambassador layer allows for greater **security**, as reported by two participants. Similar to SIDECAR, **monitorability**, **maintainability** and **scalability** were also cited as benefits of this pattern. All three participants reported implementing this pattern as a sidecar. **Operational complexity** was mentioned by all three participants as the only downside to this pattern.

5) COMPUTE RESOURCE CONSOLIDATION: Applied by five participants. All of them noted **scalability** as the QA most helped by the application of this pattern, praising the autoscaling solutions which provide an easy interface for sophisticated scaling strategies, calculating the best consolidation of instances automatically. One participant noted that the scalability enabled by this pattern adds greatly to the serverless functions they use, as the serverless paradigm encourages processing on-demand rather than always running and waiting for requests. Additionally, with the conjunction of sophisticated techniques like autoscaling, **cost savings** were noted by two participants as a welcome addition. No participant reported pains with this pattern. This suggests that participants consider this pattern to be a boon with minimal to no downsides.

6) COMMAND QUERY RESPONSIBILITY SEGREGATION: Applied by six participants. They note gains in **scalability**, **performance** and **availability** as a result of this pattern. As for drawbacks, the **complexity** introduced was mentioned by all but one participant. Finally, **ensuring data consistency** was another problem felt by 2 participants.

7) EXTERNAL CONFIGURATION STORE: Applied by eight participants. Our results suggest a consensus among our participants: the **decoupling between development and operational concerns** and **maintainability** are the primary gains of this pattern. Additional benefits reportedly include **security**, **scalability** and **availability**. On the other hand, the **complexity** of this setup should be taken into account, according to three participants. Additionally, the external store represents a **single point of failure**, as mentioned by two participants. Finally, one participant mentioned **difficulties introducing breaking changes** with this pattern present.

8) GATEWAY ROUTING: Applied by all nine participants. Participants see this pattern straightforwardly. The fact that the gateway functions provides a **decoupling between the client and the services** through a single access layer was a gain reported by seven participants. However, a gateway represents a **single point of failure**, a concern raised by four participants.

9) GATEWAY AGGREGATION: Applied by three participants. The three have noted that **performance** was the main reason behind choosing this pattern, as it allows for client applications to spend less time communicating with the API gateway. However, none were particularly pleased with this solution. Similar to the previous pattern, the gateway remains a **single point of failure**. Furthermore, negative impacts in the forms of difficulties **scalability** of the gateway and possibilities of **coupling** between the gateway and the business microservices were noted by participants.

10) GATEWAY OFFLOADING: Applied by six participants. Widely praised by all who adopted it, this pattern gives practitioners ways of tackling **security** foremost, followed by **maintainability** and **monitorability**. One participant noted **performance** as an extra gain, which logically followed from the security benefits gained. As for drawbacks, only one participant mentioned **difficulties defining which cross-cutting concerns to share** in the gateway.

11) BACKENDS FOR FRONTENDS: Applied by five participants. All these participants note **performance** as a QA positively affected, with one participant noting that **maintainability** was also improved. However, three participants mention costs in the form of **complexity**.

12) LEADER ELECTION: Applied by three participants. All three note **availability** as the major gain from this pattern.

73

TABLE III
REPORTED GAINS AND PAINS OF THE PATTERNS CONSIDERED IN THE STUDY.
THE SYMBOLS SHOW WHICH ITEMS WERE ALREADY PRESENT IN THE AAC DOCUMENTATION (🔖) OR WERE NOT PRESENT (✷).

Pattern	Gains	#	Pains	#
Strangler	✷ Scalability	5	✷ Costly to maintain	2
	✷ Maintainability	4	🔖 Planning transitions	2
	🔖 Consumer transparency	4	🔖 Managing data consistency	1
			✷ Initial developers resistance	1
			🔖 Backwards compatibility	1
Anti-Corruption Layer	🔖 Decoupling systems	4	🔖 Layer complexity	3
			🔖 Performance	1
Sidecar	🔖 Monitorability	5	🔖 Operational complexity	1
	🔖 Maintainability	4		
	✷ Scalability	1		
	✷ Security	1		
	✷ Testability	1		
Ambassador	🔖 Availability	3	🔖 Operational complexity	3
	🔖 Monitorability	2		
	🔖 Security	2		
	✷ Maintainability	1		
	✷ Scalability	1		
Compute Resource Consolidation	🔖 Scalability	5	—	
	🔖 Cost savings	2		
CQRS	🔖 Scalability	3	🔖 Complexity	5
	🔖 Performance	2	🔖 Ensuring data consistency	2
	✷ Availability	2		
External Configuration Store	🔖 Decoupling	6	🔖 Complexity	3
	🔖 Maintainability	6	🔖 Single point of failure	2
	✷ Security	2	✷ Making breaking changes	1
	✷ Scalability	1		
	✷ Availability	1		

Pattern	Gains	#	Pains	#
Gateway Routing	🔖 Decoupling	7	🔖 Single point of failure	4
Gateway Aggregation	🔖 Performance	3	🔖 Single point of failure	2
			🔖 Scalability	2
			🔖 Coupling	1
Gateway Offloading	🔖 Security	5	🔖 Defining shared functionality	1
	🔖 Maintainability	3		
	🔖 Monitorability	2		
	✷ Performance	1		
Backends for Frontends	🔖 Performance	5	🔖 Complexity	3
	🔖 Maintainability	1		
Leader Election	✷ Availability	3	✷ Managing leader resources	1
Pipes and Filters	🔖 Performance	5	✷ Monitorability	2
	✷ Maintainability	3	🔖 Complexity	2
	🔖 Scalability	1		
	🔖 Reusability	1		
Static Content Hosting	🔖 Performance	5	✷ Eventual consistency	1
	🔖 Availability	3	🔖 Complexity	1
	🔖 Maintainability	1		
	🔖 Security	1		

As for downsides, one participant mentioned that **managing leader resources** posed a significant challenge.

13) PIPES AND FILTERS*:* Applied by eight participants. **Performance** is cited by five of the participants as a gain, followed by **maintainability**, with three respondents claiming it is enhanced thanks to this pattern. **Scalability** and **reusability** were mentioned by one participant each as additional gains. As for drawbacks, two participants reported **difficulties monitoring** their systems with this programming model. Finally, **complexity** was noted by two participants as a challenge.

14) STATIC CONTENT HOSTING*:* Applied by eight participants. Over half of them declared **performance** as a benefit gained. Following that, participants also mentioned **availability**, **maintainability** and **security** as perceived gains. On the other hand, **eventual consistency** and **complexity** were cited as problems by one participant each.

C. Measuring and addressing quality attributes

Architectural trade-offs are often expressed as *quality attributes* that are affected in some way by the application of a given strategy, *e.g.*, the usage of a new deployment infrastructure or the application of a pattern. Obtaining a concrete understanding of changes in software quality should ideally go beyond a developer's subjective perception, and be informed by metrics one can point to and track over time.

However, we noticed recurring participant behavior in this area: when explicitly asked for metrics for each QA, practitioners often jumped directly to techniques for addressing and improving those attributes, and we typically had to reiterate and ask them again for particular markers or indicators that can be used to measure a given attribute. Because of this, we were able to collect additional information that we did not originally set out to find. In the following sections, and through Table IV, for each QA we describe reported quality indicators for measuring and techniques for improving it.

1) Scalability: When it comes to getting a feel for scalability, almost half of our participants noted a close and **direct relationship to performance** (see Section III-C2). This relationship is also commonly found in the literature, *i.e.*, a scalable system is understood as showing increased

performance in a manner proportional to the resources added. Two participants stated that measuring scalability is a matter of **subjective interpretation**. One participant explicitly stated that there was no interest in measuring scalability in their company, noting that a single instance of each service was enough for their needs, even in peaks of 90 000 users. Participants reported a handful of techniques for addressing scalability, most often mentioning **autoscaling**, *i.e.*, using infrastructure (such as Kubernetes or AWS) that handles horizontal or vertical scaling depending on specified criteria. Additionally, **load testing** was reported by three participants. Practitioners turn to this form of execution-based testing to evaluate how their applications handle increases in requests. **Dashboards** were mentioned by two participants as tools that visualize statistics and metrics about their applications which they turn to for guiding their business decisions (see Section III-C4). **External configurations** were also noted as a form of addressing scalability, as it allows teams to autonomously reconfigure application container properties or other run-time settings. The **segregation of responsibilities**, encouraged naturally by microservices, was raised as an important factor for addressing scalability: decoupling and isolating business concerns allows teams to scale independently and with relative ease. Finally, **FinOps**, *i.e.*, Financial Operations, was cited by one participant as a strategy to use when addressing scalability.

2) Performance: With performance, participants were aware of lower-level metrics for assessing the current level, namely **hardware or infrastructure properties** (*e.g.*, CPU and RAM usage) and **server properties** (*e.g.*, number of active requests or time taken handling requests, or database queries). Besides technical metrics, one participant referred to **business performance metrics**, also known as key performance indicators, as an additional instrument for evaluating performance. One participant stated explicitly that his company has no interest in run-time performance, as it poses little cause for concern for their particular business. For addressing performance, participants mention **dashboards** the most, with 3 mentioning these tools as the go-to for assessing performance at a glance—for more details on dashboards, see Section III-C4. Dashboards effectively synthesize a range of indicators into one, sitting at a higher level of abstraction. Furthermore, **performance testing** was reported by two participants as a strategy for measuring performance, integrated into continuous deployment pipelines.

3) Availability: For measuring availability, two participants noted run-time statistics like **latency, traffic, errors and saturation** as ways to evaluate availability. Additionally, two participants mention tracking Service Level Agreements, Service Level Objectives, and Service Level Indicators for judging availability (**SLAs, SLOs, and SLIs**, respectively). One participant reported explicitly no interest in availability due to not having concerns for critical or otherwise real-time processes in their business. As for tackling availability, performing **health checks** received most mentions from three respondents, who all noted having automatic alerts set up to notify them when their systems are down. Furthermore, **service replication** and the usage of a **load balancer** were

found, as expected.

4) Monitorability: While most participants ignored the question of *how they measure monitorability*—jumping straight to the tools they use which aim to address it—one participant told us that, while an essential aspect, getting a feel for monitorability is a matter of professional intuition and experience. As for addressing monitorability, most respondents cited using **third-party monitoring** for assisting their observing needs. Some tools were mentioned several times. Grafana and the ELK stack (Elasticsearch, Logstash and Kibana) were the more popular ones (each with 4 mentions), followed by Prometheus (3 mentions) and Google Analytics (2 mentions).

Other third-party tools were mentioned by only one participant each: VictorOps, Opsgenie, Datadog, Dynatrace, Microsoft Power BI, and Google Data Studio. In addition, three respondents explicitly mentioned developing **first-party monitoring** solutions. One main driver cited for developing an in-house monitoring infrastructure is to avoid vendor lock-in.

5) Security: Security seemed a puzzling question for our participants, with only one pointing out any concrete metrics, in the form of **code security**. As a key part of a business, practitioners are, however, very interested in addressing security. They do this via **internal audits**, which received most mentions from participants, followed by **penetration testing**, noted by three participants. Our two participants in the finance industry additionally mention **external audits** from entities that issue PCI certificates, required for these companies to stay in business. Finally, other techniques like **security information and event management (SIEM)**, **bug bounties** and **multi-tenancy** were mentioned by one participant each as approaches to ensure better security in their business.

6) Testability: For measuring testability, having a **number of automated tests** within DevOps pipelines and **code coverage** are the go-to indicators for practitioners to judge this QA. When describing how they addressing it, **unit testing** receives most mentions from participants, followed by **integration testing**, **performance testing**, **penetration testing** and **functional testing**. One participant specified, in addition, the usage of **smoke testing** and **non-regression testing**. The same interviewee commented that he felt it was the most neglected QA out of the set. One participant mentioned their company having a **QA Manifesto**, an internal document containing their definition of quality revolving around tests.

7) Maintainability: Regarding metrics for maintainability, participants were more divided. Similar to testability, **code coverage** was mentioned by two participants as an indicator of maintainability. Beyond that, one participant mentioned tracking, through Slack and Jira, bug reports and the **time taken to resolve issues** as a recent effort started in the company aimed at upholding a desired level of maintainability. Finally, one other participant reported an analysis of **architectural debt** as a measure of levels of maintainability. As for increasing maintainability, **static code analysis** seems to be the preferred strategy, with three participants mentioning using SonarQube and one using Codacy for this effect. Following that, **automated pipelines** and **code reviews** were also cited

TABLE IV
REPORTED INDICATORS FOR MEASURING AND TECHNIQUES FOR ADDRESSING THE SEVEN QAS.

	Indicator	#	Technique	#		Indicator	#	Technique	#
Scalability	Through performance	4	Autoscaling	4	Security	Code security	1	Internal audits	5
	Subjective interpretation	2	Load testing	3				Penetration testing	3
		1	External configs	2				External audits	2
			Dashboards	2				Bug bounties	1
			Segregation of responsibility	1				Multi-tenancy	1
			FinOps	1				SIEM	1
Performance	Infrastructure (*e.g.*, CPU, RAM)	2	Dashboards	3	Testability	Number of automated tests	3	Unit testing	8
	Server (*e.g.*, requests, queries)	2	Performance testing	2		Code coverage	3	Integration testing	4
	Business performance metrics	1						Performance testing	3
		1						Functional testing	2
								Penetration testing	1
								Smoke testing	1
Availability	Latency, traffic, errors, saturation	2	Health checks	3				Non-regression testing	1
	SLAs, SLIs, SLOs	2	Service replication	2				QA Manifesto	1
		1	Load balancer	1	Maintainability	Code coverage	2	Static code analysis	4
Monitorability	Subjective interpretation	1	Third-party monitoring	5		Time to fix bugs	1	Automated pipelines	3
			First-party monitoring	3		Architectural debt	1	Code review	3
								Domain-driven design	2

three times as fundamental aspects of higher maintainability. Finally, **domain-driven design**, which affirms a strict logical match between business concerns and the structure of software systems, was mentioned by two participants as one of the ways they can ensure a better level of maintainability.

IV. DISCUSSION

The next sections present our interpretation and discussion of the results described in Section III and identify the main threats to validity and strategies to their mitigation.

A. Pattern trade-offs

As mentioned in Section III-B, our findings directly mix QAs and more concrete concerns such as, for example, "*Managing data consistency*". We refrained from approximating these instances to QAs which *best matched—e.g.*, using "*Maintainability*" in place of "*Complexity*"—for one main reason: such a mapping would invariably produce ambiguity and biases which could negatively impact our (and the reader's) interpretation of the results.

All in all, we were able to gather many insights into each of these particular patterns, some of which differ quite substantially from their original intention documented in the AAC. Some concerns were raised in the AAC but not by our participants. We did not raise these concerns during the interview, so the professional perception of their severity is left unclear. However, their lack of mention suggests they are minor or secondary issues.

In several cases, we unfortunately could not discuss some concepts in-depth due to a mix of time constraints, differences in how the unstructured segments of the interviews unfolded, and participants not having had experience with some of the patterns, impairing our ability to extract more findings.

In any case, **we identified 14 new gains and 6 new pains** for these patterns that were not present in the AAC

documentation, out of a total of 35 gains and 25 pains reported by participants. This suggests that the knowledge in the AAC may be incomplete. COMPUTE RESOURCE CONSOLIDATION was the only pattern without any pains reported.

In a scenario where all 14 patterns are applied, we gain benefits in the form of: *maintainability* from 7 patterns; *performance* from 6 patterns; *scalability* from 6 patterns; *availability* from 5 patterns; *security* from 5 patterns; *monitorability* from 3 patterns; and other gains from 1 pattern each. On the other hand, we are also afflicted with: *complexity* from 5 patterns; *single point of failure* from 3 patterns, *operational complexity* from 2 patterns; and other pains from 1 pattern each.

B. Measuring and addressing quality attributes

The results reported in Section III-C, concerning how QAs are measured and addressed, were not possible to obtain for all participants. We are unclear as to the reasons for this, but we do not think it is due to these practitioners not actively measuring a given QA. In those cases, the interviewees briefly described, instead, the importance they gave to that attribute, from their personal or company's perspective. We omit these remarks from our findings due to their general irrelevance to our RQs and lack of depth. These hurdles give us less confidence in asserting saturation of our findings, though we do feel we provide a relevant account of industrial perception of software QAs.

Furthermore, it is possible that metrics for measuring certain QAs went unreported simply due to being obvious to the point of second nature. For example, response time is an immediately obvious metric for assessing scalability or performance (to an extent) yet it was not mentioned explicitly by any participant. This is likely due to already being accounted for by measuring tools like dashboards, and, as such, does not warrant special mention from respondents.

76

Overall, we found that participants do not measure software qualities directly. They correctly conceive of each individual quality as a sum of lower level attributes, which they can then project onto a whole, when needed. But, due to the nature of commercial software development, software engineers want measures for QAs that are easy to check, and they want simple procedures to confirm that the readings are consistent. As a result, we were left with the impression that QAs are not part of the daily vocabulary of the software developer, likely due to an insipid and highly personal conceptual understanding of each attribute, hampering interpersonal agreements on a semantic basis. All in all, these findings seem to support the arguments presented by Moses [26], particularly the author's note of skepticism regarding industry practice: "*software developers and researchers may remain content to measure attributes or processes considered to be related to quality attributes*".

Furthermore, when we compare our findings with the work by Li [18], we see several techniques practitioners resort to for addressing these QAs that were not covered by their systematic literature review.

C. Threats to validity

The credibility of results obtained through empirical studies is always hampered by threats to validity [27]. We can think of validity through four different aspects [28]: construct validity; internal validity; external validity; and reliability.

Construct validity. An interview study is susceptible to social desirability bias, which refers to the tendency for participants to present themselves in the most flattering light [27]. To mitigate this threat, we assured participants that no identifying data would be shared.

Internal validity. Through our previous interest in the field, we were pre-exposed to existing research. This might have biased our interview design and may have led to some elements going unnoticed. For instance, we targeted our interview questions towards the impact patterns have on explicitly mentioned or obviously implicit QAs, which may have left some attributes unaddressed. Further, this study demanded extensive knowledge of all design patterns and QAs that were analyzed, as well as a proficiency for conveying this knowledge to participants to prevent miscommunication or confusion. While we sought a serious depth of knowledge of the theoretical elements involved in this study, all interviews were performed by one author, who cannot claim extensive expertise of the domain of software architecture. As a result, it is possible that degradation of communication occurred with some respondents or misinterpretation on our part, negatively affecting the efficiency of the interviews. Finally, we cannot claim completeness of either the set of identified architectural trade-offs of design patterns or the set of indicators for measuring and techniques for addressing QAs.

External validity. We cannot claim saturation of our findings for the patterns with a low adoption rate. Especially for these patterns, nine interviews is clearly not enough to draw conclusions that can be confidently generalized to the software industry in general. Furthermore, voluntary surveys are inherently susceptible to self-selection bias [29] as uninterested respondents are unlikely to participate. Finally, all interviewed participants are currently residing and working in Portugal. As such, it is theoretically possible that a regional technology culture might have rendered a biased lens into the industrial reality of the microservices architecture. To mitigate this threat, we selected participants from diverse backgrounds to cover companies of different sizes and business domains.

Reliability. Many of steps described in this study were performed by only one of the researchers. It is possible that different people might have drawn different answers during the interviews, or might have made different options during coding. We may still improve the latter in subsequent steps of our work, by having coding done by a second researcher, and bring in a third researcher to solve disagreements.

V. RELATED WORK

Our work combines two relevant aspects in microservices design: the use of patterns and software quality. While both topics have already been addressed in scientific literature, studies examining their interplay are still rare. We therefore discuss the studies that make a connection between both aspects and highlight important empirical investigations that reflect the current industry practices in this regard.

Osses *et al.* [30] extracted 44 patterns and tactics used for microservices from 1,067 studies in a systematic review. In a related study [17], the authors focus on the intersection of patterns and tactics that have been described in academia as well as in industry. The authors found that most architectural patterns are associated with five quality attributes: scalability, flexibility, testability, performance, and elasticity. As another result, they propose a pattern taxonomy for microservices. Together with the review by Li *et al.* [18], who sought a comprehensive understanding of QAs in the context of the microservices architecture, these contributions build a solid foundation for our work. The authors of the latter study also affirm a need for further empirical research addressing particular QAs like *maintainability*.

Valdivia *et al.* [31] note the unclear mapping between QAs and patterns in microservices and sought to address this issue by a systematic literature review. While the authors present a comprehensive compilation of patterns with their related QAs, the impact of the QAs is not investigated. In this regard, the authors state that selecting certain patterns requires an analysis of the trade-offs specifically for the system in question, as their impact could not be generalized. This inherent difficulty is confirmed by Rosa *et al.* [32], who identified a lack of knowledge about patterns among software architects and, as mitigation, developed a method for an architectural trade-off analysis based on patterns in microservices. The authors, however, only investigate the relationship between patterns and structural attributes, and do not target QAs in particular.

Besides these works, several empirical studies have been conducted, including a controlled experiment on the impact of service-oriented patterns on software evolvability [33] and a survey on the adoption of patterns for engineering cloud

77

software [34]. Taibi *et al.* [35] surveyed 23 publications on industry case studies and thereby identified widely adopted architectural principles and patterns for microservices, including their trade-offs.

The presented studies cover the analysis of architectural properties and design trade-offs for microservices reasonably well. An adherence to patterns is commonly desired in professional software development and can be an effective facilitator for creating systems with a set of desired software qualities. However, empirical evidence is needed to confirm this hypothesis. The discussed studies have rarely attempted to evaluate the consequences that patterns have on particular QAs, and the few that did have generally resorted to reviewing academic literature or analyzed a small set of patterns and QAs. As the existing studies did not sufficiently address this topic, this gap in scientific research sets the motivation for our study. The work by Valdivia *et al.* [31] took the first steps in this direction but solely relied on existing literature. To the best of our knowledge, there exists no empirical study reflecting the perception by industry on design patterns in the microservices architecture and their quality impacts—positive and negative.

VI. CONCLUSION

We began with the observation that the microservices architecture is an increasingly accepted and adopted architectural style, as it helps overcome the limitations of traditional monolithic systems, namely when it concerns issues of scalability, crucial when moving to a cloud-first paradigm. Designing these systems, however, is no easy task, and the careful consideration of which design patterns to apply in the context of large-scale distributed systems has been subject to little rigorous empirical research. Our RQs, initially posited in Section II, can now be answered the following way:

- **RQ1:** *What is the rationale for the adoption of patterns in microservices systems?* — Practitioners are aware of the problems presented, but they do not think of their solutions in terms of patterns. Oftentimes, participants would state they are familiar with the solution described in a pattern but unaware that there was such a name for that solution. This suggests that some professionals may currently lack the language to explore solutions for hurdles seen in microservices, which might lead to risky and untested paths being taken unnecessarily. Queried for the expertise regarding design patterns, each practitioner had experience with, on average, 9 of the 14 patterns we showed them, with only one participant having had experience with all. STRANGLER and GATEWAY ROUTING were the only patterns adopted by all respondents.

- **RQ2:** *How are QAs influenced as a result of applying microservice patterns?* — Practitioners' perceptions of the architectural trade-offs inherent to the design patterns we asked about largely matched the original documentation, implying that these patterns are a reliable resource. However, new gains and pains were identified in some patterns as a result of this study, suggesting that they may be incomplete or not completely accurate. For example, GATEWAY

AGGREGATION was viewed very poorly by all participants who applied it, leading us to hypothesize that the gains this pattern may provide are not worth the costs. On the other hand, COMPUTE RESOURCE CONSOLIDATION was viewed very positively, with no participants reporting any considerable pains related to it.

- **RQ3:** *How are QAs measured in microservices?* — Practitioners have shown interest in tracking key facets of software quality, though they do not often turn to QAs as the terminology to express and measure characteristics of their systems. Only a few well-known terms emerged in our results, like *scalability*, *performance*, and *maintainability*. Similarly, practitioners have shown little interest in discussing concrete metrics. Instead, they directly think in terms of techniques and tools.

The results and findings from this article offer implications for both professionals and researchers. For professionals, this work provides further understanding of QAs impacted by patterns, which can serve as a useful guide for strategic engineering decision-making on moving forward with adopting microservices. For researchers, we provide a report of an interview study's design and execution, with findings that support new hypothesis of how the existing body of knowledge may be incomplete or inaccurate.

VII. FUTURE WORK

Further empirical research may provide more insights regarding the methods for designing microservice systems. Namely, applying this study's methodology using a larger sample size will yield more reliable results, and studies specifically targeting quality attributes in microservices are also sorely needed. Additionally, a systematic review of the empirical literature on microservices that aggregates knowledge on pains and gains would further assist researchers in obtaining an actualized perspective on the design trade-offs inherent to this architectural style. In this article, we restrict our analysis of related works (*cf.* Section V) to those addressing simultaneously patterns and software quality aspects in the scope of microservices design, but such a systematic review could also look in more detail into works as those done by Li *et al.* [36], Reis *et al.* [37], Bogner *et al.* [38], [39], Soldani *et al.* [40] and Waseem *et al.* [41], which study practices and quality aspects in the context of microservices. Even if not necessarily leaning on pattern literature and on how patterns are perceived by professionals, we believe that some of the results of these studies could be used to improve existing pattern knowledge.

References

[1] C. Kohls and S. Panke, "Is that true...? thoughts on the epistemology of patterns," in *Proceedings of the 16th Conference on Pattern Languages of Programs*, 2009, pp. 1–14.

[2] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," *CoRR*, vol. abs/2002.07764, 2020. [Online]. Available: https://arxiv.org/abs/2002.07764

[3] D. Riehle, N. Harutyunyan, and A. Barcomb, "Pattern discovery and validation using scientific research methods," *arXiv preprint arXiv:2107.06065*, 2021.

[4] L. Carvalho, A. Garcia, W. K. G. Assunção, R. d. Mello, and M. J. d. Lima, "Analysis of the Criteria Adopted in Industry to Extract Microservices," in *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, May 2019, pp. 22–29, issn: 2575-4793.

[5] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[6] C. Richardson, *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019, oCLC: on1002834182.

[7] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," *Software: Practice and Experience*, vol. 48, no. 11, pp. 2019–2042, 2018.

[8] G. Márquez, M. M. Villegas, and H. Astudillo, "A pattern language for scalable microservices-based systems," in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, 2018, pp. 1–7.

[9] T. B. Sousa, H. S. Ferreira, F. F. Correia, and A. Aguiar, "Engineering software for the cloud: Automated recovery and scheduler," in *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, 2018, pp. 1–8.

[10] ——, "Engineering software for the cloud: External monitoring and failure injection," in *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, 2018, pp. 1–8.

[11] ——, "Engineering software for the cloud: Messaging systems and logging," in *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, 2017, pp. 1–14.

[12] T. B. Sousa, F. F. Correia, and H. S. Ferreira, "Patterns for software orchestration on the cloud," in *Proceedings of the 22nd Conference on Pattern Languages of Programs*, 2015, pp. 1–12.

[13] K. Brown, B. Woolf, C. D. Groot, C. Hay, and J. Yoder. (2021) Patterns for developers and architects building for the cloud. [Online]. Available: https://kgb1001001.github.io/cloudadoptionpatterns/

[14] (2021) Cloud design patterns - Azure Architecture Center. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/index-patterns

[15] G. Márquez and H. Astudillo, "Actual use of architectural patterns in microservices-based open source projects," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. Ieee, 2018, pp. 31–40.

[16] J. A. Valdivia, A. Lora-González, X. Limón, K. Cortes-Verdin, and J. Ocharán-Hernández, "Patterns Related to Microservice Architecture: a Multivocal Literature Review," *Programming and Computer Software*, vol. 46, no. 8, pp. 594–608, Dec. 2020.

[17] F. Osses, G. Márquez, and H. Astudillo, "Exploration of academic and industrial evidence about architectural tactics and patterns in microservices," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. New York, NY, USA: ACM, may 2018, pp. 256–257.

[18] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, "Understanding and addressing quality attributes of microservices architecture: A Systematic literature review," *Information and Software Technology*, vol. 131, p. 106449, Mar. 2021.

[19] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[20] S. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *11th IEEE International Software Metrics Symposium*, 2005, pp. 10 pp.–23.

[21] C. B. Seaman, "Qualitative Methods," in *Guide to Advanced Empirical Software Engineering*. London: Springer London, 2008, pp. 35–62.

[22] P. Ralph, N. bin Ali, S. Baltes, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri, B. B. N. de França, C. A. Furia, G. Gay, N. Gold, D. Graziotin, P. He, R. Hoda, N. Juristo, B. Kitchenham, V. Lenarduzzi, J. Martínez, J. Melegati, D. Mendez, T. Menzies, J. Molleri, D. Pfahl, R. Robbes, D. Russo, N. Saarimäki, F. Sarro, D. Taibi, J. Siegmund, D. Spinellis, M. Staron, K. Stol, M.-A. Storey, D. Taibi, D. Tamburri, M. Torchiano, C. Treude, B. Turhan, X. Wang, and S. Vegas, "Empirical standards for software engineering research," 2021.

[23] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 120–131.

[24] J. T. DeCuir-Gunby, P. L. Marshall, and A. W. McCulloch, "Developing and using a codebook for the analysis of interview data: An example from a professional development research project," *Field methods*, vol. 23, no. 2, pp. 136–155, 2011.

[25] G. Vale, F. F. Correia, E. M. Guerra, T. de Oliveira Rosa, J. Fritzsch, and J. Bogner, "Dataset for "Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs"," Nov. 2021.

[26] J. Moses, "Should we try to measure software quality attributes directly?" *Software Quality Journal*, vol. 17, no. 2, pp. 203–213, 2009.

[27] F. Wedyan and S. Abufakher, "Impact of design patterns on software quality: a systematic literature review," *IET Software*, vol. 14, no. 1, p. 1–17, 2020.

[28] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[29] H. M. Ayas, P. Leitner, and R. Hebig, "Facing the giant: a grounded theory study of decision-making in microservices migrations," *arXiv preprint arXiv:2104.00390*, 2021.

[30] F. Osses, G. Márquez, and H. Astudillo, "An Exploratory Study of Academic Architectural Tactics and Patterns in Microservices: A systematic literature review," *Avances en Ingenieria de Software a Nivel Iberoamericano, CIbSE 2018*, Feb. 2019.

[31] J. A. Valdivia, X. Limón, and K. Cortes-Verdin, "Quality attributes in patterns related to microservice architecture: a Systematic Literature Review," in *2019 7th International Conference in Software Engineering Research and Innovation (CONISOFT)*, Oct. 2019, pp. 181–190.

[32] T. de Oliveira Rosa, J. F. L. Daniel, E. M. Guerra, and A. Goldman, "A method for architectural trade-off analysis based on patterns: Evaluating microservices structural attributes," in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, 2020, pp. 1–8.

[33] J. Bogner, S. Wagner, and A. Zimmermann, "On the impact of service-oriented patterns on software evolvability: a controlled experiment and metric-based analysis," *PeerJ Computer Science*, vol. 5, p. e213, Aug. 2019, publisher: PeerJ Inc.

[34] T. Sousa, H. S. Ferreira, and F. F. Correia, "A survey on the adoption of patterns for engineering software for the cloud," *IEEE Transactions on Software Engineering*, p. 1–1, 2021.

[35] D. Taibi, V. Lenarduzzi, and C. Pahl, "Continuous Architecting with Microservices and DevOps: A Systematic Mapping Study," in *International Conference on Cloud Computing and Services Science*, 2019, no. July, pp. 126–151.

[36] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: an industrial survey of microservice tracing and analysis," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–28, 2022.

[37] D. Reis, B. Piedade, F. F. Correia, J. P. Dias, and A. Aguiar, "Developing docker and docker-compose specifications: A developers' survey," *IEEE Access*, 2021.

[38] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Assuring the evolvability of microservices: insights into industry practices and challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 546–556.

[39] ——, "Microservices in industry: insights into technologies, characteristics, and software quality," in *2019 IEEE international conference on software architecture companion (ICSA-C)*. IEEE, 2019, pp. 187–195.

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

[41] M. Waseem, P. Liang, M. Shahin, A. Di Salle, and G. Márquez, "Design, monitoring, and testing of microservices systems: The practitioners' perspective," *Journal of Systems and Software*, vol. 182, p. 111061, 2021.