

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Designing Microservice Systems Using Patterns: An Empirical Study On Architectural Trade-offs

Guilherme Vale



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Filipe Correia

Co-supervisor: Eduardo Guerra

July 26, 2021

Designing Microservice Systems Using Patterns: An Empirical Study On Architectural Trade-offs

Guilherme Vale

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Pedro Souto

External Examiner: Prof. Nour Ali

Supervisor: Prof. Filipe Figueiredo Correia

July 26, 2021

Abstract

Organisations are increasingly looking to most effectively and continuously deliver business value by transitioning to microservices. The promises of increased agility, autonomy, scalability, and reusability has made the architectural style a de facto standard for the development of large-scale and cloud-native commercial applications. Existing research tends to focus on technical problems and decisions such as how to split services, while equally important organisational or business issues and their relationship with the technical aspects usually remain out of scope.

Design patterns are an important factor for assessing the quality of software system, but the selection and combination of patterns to optimise quality attributes is often a difficult task due to the lack of knowledge about them among software architects. Additionally, most of them have not been validated against industry practice. This problem is exacerbated in the context of microservices, an architectural style still in its growing stages.

We investigate the impact 14 design patterns have on 7 quality attributes by conducting a semi-structured interview study with 9 practitioners, identifying tendencies in industry regarding (1) knowledge and adoption of design patterns, (2) the perceived architectural trade-offs of patterns, and (3) metrics professionals use to measure quality attributes. Our study approaches the matter with a strong emphasis on the human aspect of applying design patterns in microservices through stakeholders, their rationales before and perceptions after the adoption of these patterns.

Our findings reveal three significant insights. Firstly, practitioners seem uninterested in consciously using design patterns to explore solutions for hurdles seen in microservices, despite many of them recognising the solutions embodied by the patterns presented. Secondly, practitioners seem also uninterested in expressing quality concerns through quality attributes, i.e. quality attributes are not part of the day-to-day vocabulary of developers, with the exception of well-known terms like scalability, performance, and maintainability. Thirdly, practitioners nevertheless show serious concerns regarding tracking and improving the manifold facets of their systems through tools and techniques which they perceive as improvements directly related to addressing quality attributes.

We found that the trade-offs reported by our study participants largely matched the documentation of each respective pattern, with new gains and pains having been identified as a result of this study, representing novel insight about these patterns.

Keywords: Software architectures, Design patterns, Software design trade-offs

Resumo

As empresas estão cada vez mais a procurar fornecer valor de negócio de uma forma eficaz e contínua através da transição para microserviços. As promessas de aumentos na agilidade, autonomia, escalabilidade e reusabilidade tornaram este estilo arquitectural o modelo de facto para o desenvolvimento de aplicações comerciais de larga escala e nativas à cloud. A pesquisa existente tende a se focar em problemas técnicos e decisões tais como a separação em serviços, deixando de fora a relação que feições organizacionais e problemas de negócio podem ter com esses aspectos técnicos.

Os padrões de design são um factor importante na avaliação de qualidade de um sistema de software, mas o processo de selecção e combinação de padrões de modo a otimizar atributos de qualidade é, muitas vezes, uma tarefa difícil devido à falta de conhecimento sobre eles entre os arquitectos de software. Adicionalmente, muitos desses padrões não foram validados na prática da indústria. Este problema é exacerbado no contexto de microserviços, um estilo arquitectural ainda com as suas dores de crescimento.

Investigamos o impacto que 14 padrões de design têm em 7 atributos de qualidade através da realização de um estudo de entrevistas semi-estruturadas com 9 participantes, identificando tendência na indústria relacionadas com (1) o conhecimento e adopção de padrões de design, (2) os *trade-offs* arquitecturais desses padrões, e (3) métricas que os profissionais utilizam para medir atributos de qualidade. O nosso estudo aborda a matéria com um forte ênfase no aspecto humano de aplicar padrões de design em microserviços através de pessoas envolvidas no processo, os seus raciocínios antes e as suas percepções depois da adopção destes padrões.

Os nossos resultados revelam três discernimentos importantes. Primeiramente, os profissionais mostram-se desinteressados em utilizar, conscientemente, os padrões de design para explorar soluções para os problemas encontrados em microserviços, apesar de muitos deles reconhecerem as soluções encorporadas nos padrões apresentados. Em segundo lugar, os profissionais mostram-se também desinteressados em expressar preocupações de qualidade do seu software através de atributos de qualidade, i.e. os atributos de qualidade não fazem parte do vocabulário do dia-a-dia dos profissionais, à exceção de termos mais conhecidos como *scalability*, *performance* e *maintainability*. Em terceiro lugar, os profissionais todavia mostram preocupações sérias com acompanhar e melhorar as diversas facetas que compõem os seus sistemas, através de ferramentas e técnicas que entendem como melhorias relacionadas diretamente com a resolução de atributos de qualidade.

Nós concluímos que os *trade-offs* relatados pelos participantes do nosso estudo condiziam, em larga medida, com a documentação do padrão respetivo, com novos ganhos e dores identificados como resultado deste trabalho, o que representa uma visão nova sobre estes padrões.

Keywords: Arquiteturas de software, padrões de design, trade-offs em design de software

Acknowledgements

I'm deeply indebted to the counsel and guidance of my supervisor, Filipe Correia, who has been a reliable and invaluable help throughout this dissertation. I am also extremely grateful to Eduardo Guerra and Thatiane Rosa for their wonderful advice and resources selflessly shared with me which have greatly bolstered the quality of this thesis. I would also like to extend my deepest thanks to Justus Bogner and Jonas Fritzsich for their superb insights and feedback which have assisted the direction of this project. Additionally, I would also like to thank João Francisco for giving me his insight and for helping me conduct a pilot interview which proved to be a wonderful boon.

It is difficult to overstate the gratitude I feel towards the friends and colleagues I have made in the University of Porto, for the many meals, study sessions and conversations we have had as well as for keeping me sane throughout the four years of this journey.

This dissertation would not have happened without the support of my mother and my brother, who, along with my late father, have shaped me into the person I am today.

And for my lovely girlfriend Mariana, who has been my rock, an endless source of support, love and joy, giving me the confidence and courage I needed to write the best thesis I could write. This would have turned out much worse without her.

Guilherme Vale Martins

*“You have to accept being finite:
being here and nowhere else,
doing this and not something else,
now and not always or never
... having only this life.”*

André Gorz

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Contributions	2
1.4	How to read this document	2
2	Background	5
2.1	Software architecture	5
2.1.1	Monolithic architecture	6
2.1.2	Microservice architecture	7
2.2	Software qualities	10
2.3	Design patterns	11
2.4	Summary	13
3	State of the Art	15
3.1	Software quality in microservices	16
3.1.1	Discussion	21
3.2	Empirical research in microservices	21
3.2.1	Case studies	22
3.2.2	Controlled experiments	25
3.2.3	Surveys	25
3.2.4	Discussion	30
3.3	Summary	32
4	Problem Statement	33
4.1	Thesis statement	33
4.2	Research questions	34
4.3	Research strategy and scope	34
4.3.1	Design patterns	34
4.3.2	Quality attributes	42
4.4	Summary	43
5	Empirical Study	45
5.1	Design	45
5.1.1	Preliminary analysis	46
5.1.2	Population sampling	46
5.1.3	Interview structure	48
5.1.4	Quality assurance	50

5.2	Execution	50
5.2.1	Pilot interview	53
5.2.2	Interviews	53
5.2.3	Iterations	55
5.3	Findings	56
5.3.1	Rationale for pattern adoption	57
5.3.2	Pattern trade-offs	60
5.3.3	Measuring and addressing quality attributes	71
5.3.4	Other findings	80
5.3.5	Threats to validity	81
5.4	Summary	82
6	Conclusions and Future Work	85
6.1	Overview	85
6.2	Research answers	86
6.3	Future work	87
A	Interview resources	89
A.1	Sign-up form	90
A.2	Interview guide	94
A.3	Interview helper	97
	References	103

List of Figures

2.1	Example of a monolithic system	6
2.2	Example of a microservice system	9
2.3	High-level overview of a microservice pattern language	12
3.1	A proposed taxonomy of design patterns	18
3.2	Example of a microservice design pattern trade-off diagram	19
3.3	Known tactics of addressing essential microservice quality attributes	20
3.4	Microservice API management design spaces and process	23
3.5	Areas of microservice design	26
3.6	Importance of identified design areas in microservices	27
3.7	Microservice advantages and challenges felt by professionals	28
5.1	Interview study procedure	50
5.2	Coding process	56
5.3	Number of patterns adopted by each participant	58
5.4	Number of participants that adopted each pattern	58
5.5	Reported third-party monitoring tools	76

List of Tables

3.1	Overview of related empirical studies in microservices	31
5.1	Preliminary design pattern trade-off analysis	47
5.2	Quality assurance checklist (essential)	51
5.3	Quality assurance checklist (desirable)	52
5.4	Participants of interview study	54
5.5	Statistics on interviews transcribed	55
5.6	Reported patterns adopted by interviewees	57
5.7	Reported gains and pains of the STRANGLER pattern	60
5.8	Reported gains and pains of the ANTI-CORRUPTION LAYER pattern	61
5.9	Reported gains and pains of the SIDECAR pattern	62
5.10	Reported gains and pains of the AMBASSADOR pattern	63
5.11	Reported gains and pains of the COMPUTE RESOURCE CONSOLIDATION pattern	64
5.12	Reported gains and pains of the COMMAND QUERY RESPONSIBILITY SEGREGATION pattern	64
5.13	Reported gains and pains of the EXTERNAL CONFIGURATION STORE pattern	65
5.14	Reported gains and pains of the GATEWAY ROUTING pattern	66
5.15	Reported gains and pains of the GATEWAY AGGREGATION pattern	67
5.16	Reported gains and pains of the GATEWAY OFFLOADING pattern	67
5.17	Reported gains and pains of the BACKENDS FOR FRONTENDS pattern	68
5.18	Reported gains and pains of the LEADER ELECTION pattern	69
5.19	Reported gains and pains of the PIPES AND FILTERS pattern	70
5.20	Reported gains and pains of the STATIC CONTENT HOSTING pattern	70
5.21	Reported indicators for measuring and techniques for addressing scalability	72
5.22	Reported indicators for measuring and techniques for addressing performance	73
5.23	Reported indicators for measuring and techniques for addressing availability	74
5.24	Reported indicators for measuring and techniques for addressing monitorability	75
5.25	Reported indicators for measuring and techniques for addressing security	76
5.26	Reported indicators for measuring and techniques for addressing testability	77
5.27	Reported indicators for measuring and techniques for addressing maintainability	78

Abbreviations

SOA	Service-Oriented Architecture
MSA	Microservice Architecture
DDD	Domain-Driven Design
IDE	Integrated Development Environment
NFR	Non-Functional Requirement
LOC	Lines Of Code
API	Application Programming Interface
AAC	Azure Architecture Centre
QA	Quality Attribute

Chapter 1

Introduction

In this chapter we present an introduction to the present work. In Section 1.1 we produce a synthesis of the context that forms the background of our research. In Section 1.2 we note the gaps in the research literature that we hope to fill with our work, stating our motivation and goals. In Section 1.3 we briefly describe the most significant contributions to the literature that have been made as a result of this dissertation. In Section 1.4 we briefly summarise the contents of each of the chapters that follow, providing readers with a clearer sense of the structure of this document.

1.1 Context

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, especially in academia. One such aspect is the knowledge and usage of design patterns; as professionals encounter and overcome challenges that arise from this novel approach to building software systems, shared solutions to certain problems begin to emerge. The documentation of these design patterns gives software architects an array of tried and tested techniques to common hurdles, thus reducing the technical risk to their projects by not having to employ new and untested designs.

The promises of microservices – of increased agility, scalability, maintainability, performance, etc. – can only be enjoyed with a careful and solid understanding of its underlying principles. In the midst of a hype created in the past several years due to the extolling of this novel architecture by many big industry players, e.g. Amazon, Netflix, Uber, Soundcloud, etc. [11], the dust has yet to settle on the best ways to go about adopting this model. For this reason, it is not uncommon to find situations wherein microservices would be a good fit, but teams fail to implement them successfully [31].

To help ease the paradigm shift for professionals coming from more traditional systems, software architects and developers can resort to a pattern language. However, this requires knowledge

of the patterns, i.e. the context in which it makes sense to apply them, their strengths and their weaknesses [43].

1.2 Motivation

Design patterns are one of the many factors that influence the quality of software systems [38]. Yet, in the context of a microservice architecture (MSA) – an architectural style still in its infancy – choosing between different design patterns is a difficult task, as there is a wide range of patterns to choose from, but little in the way of information about the relationship between patterns and, in particular, the impact these patterns have on quality attributes [14].

Since these design decisions have a long-lasting and reverberating influence throughout the life cycle of a development project, analysing and determining the architectural trade-offs of patterns will help better inform software architects and hence assist them in their ambitions to provide better products. Discovering how industry experts currently perceive the influence design patterns have on quality attributes becomes an imperative, and forms the impetus for this work.

As such, the two main objectives of this dissertation are the following:

- Identify and understand the relationships between microservice design patterns and the quality attributes that they may support or hinder;
- Conduct an empirical study to gain industry insight into the trade-offs of design patterns;

1.3 Contributions

Three significant contributions were achieved as a result of our research:

- ✧ A **review of the state of the art** on software engineering aspects of the microservice architecture, with heavy emphasis on research related to software qualities and design patterns as well as empirical studies performed in the context of microservices.
- ✧ A **design of an interview study** aimed at identifying pains and gains of design patterns and metrics used in industry to judge quality attributes.
- ✧ A **report on the application of the aforementioned survey design** which offers an extensive account of lessons learned and a prolonged discussion of the findings and their implications for practitioners and academics.

1.4 How to read this document

This document is structured as follows:

- Chapter 2 presents an overview of the theoretical concepts related to the goals of this dissertation required in order to allow inexperienced readers to appreciate the purpose of this work.
- Chapter 3 describes a review of the state of the art in microservice architectures; we present and discuss the literature related to the validation of patterns and analysis of trade-offs in software design as well as published empirical studies, all in the context of microservices.
- Chapter 4 formally defines the problem at hand; we describe our problem statement, the research questions that determine the contours of our work, an overview of our methodology as well as the constraints in scope we defined in order to carry out a feasible study.
- Chapter 5 describes the entire process related to our empirical study, extensively covering its design, execution and findings.
- Finally, Chapter 6 presents final remarks which summarise the results of this interview, implications for professionals and researchers, and possible avenues for future work.

Chapter 2

Background

In this chapter we present an introduction to the several essential concepts of software engineering that form the vocabulary and contours of this dissertation. Experts in these concepts can safely skip this chapter. In Section 2.1 we give a basic history of software architecture and present the three architectural styles most relevant to our purposes: the monolithic architecture, the service-oriented architecture and the microservices architecture. In Section 2.2 we present the notion of software qualities and quality attributes, the terms used to express software qualities as pillars for defining the features of a software architecture. In Section 2.3 we go over the concept of design pattern, its utility and structure and importance to the field of software engineering.

2.1 Software architecture

Software architecture refers to a holistic conception of a system’s architectural design as the conjunction of the design rationale, design decisions, assumptions, context, and other elements that define the nature of the architecture [13]. Knowledge of software architecture consists, primarily, of being able to identify these factors. Decision-making on the architectural level defines the structural properties of the software and, hence, determines its qualities.

These fundamental decisions are made in the very early stages of software development and condition the way we analyse a system and express its design. Software systems are expected to evolve and adapt to market needs, but mistaken decisions in the architectural design phase of a project’s life-cycle can spell the demise of a project if the cost to adapt is too great to bear. Whereas traditional development approaches defined an up-front architecture for the entirety of a software system’s life-cycle, modern software development is much more concerned with continuous design, wherein the architecture is expected to evolve over time [13], to respond to these needs.

It is in the endless movement of market relations that we see new techniques and tools emerge, in efforts to out-compete and win over others. In the following sections we describe two popular

architectural styles in large-scale and enterprise software development: the monolithic architecture and the microservice architecture.

2.1.1 Monolithic architecture

Commercial server-side applications have typically been built with mainstream, object-oriented languages like C/C++, Java, Python, etc.. These languages provide abstractions that break down a program into functional modules, helping developers reason about larger systems. However, these languages are designed for the creation of single executable artefacts, also known as monoliths [19], and these abstractions depend on local resource sharing, i.e., the sharing of memory, databases and files within the same machine. Due to this limitation, modules in a monolith are not independently executable. This is the key characteristic of the monolithic architecture. An illustration of this type of system can be seen in Figure 2.1. Client applications connect to a single endpoint which is the web server that manages all of the different modules of the system.

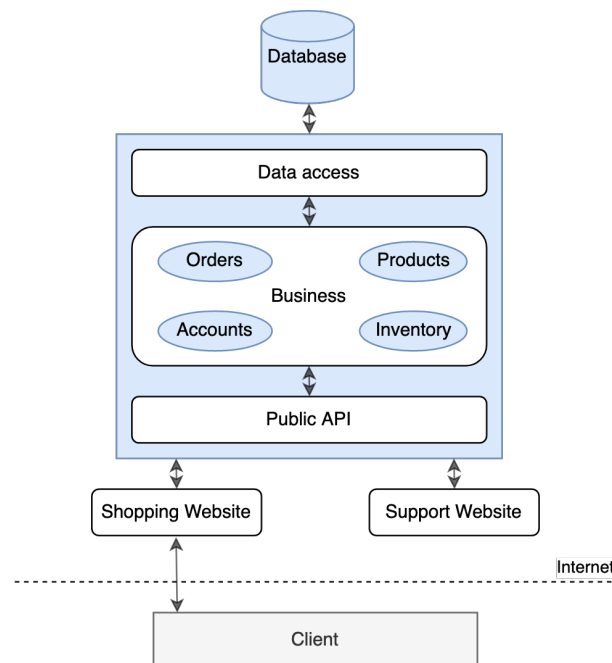


Figure 2.1: An example of a monolithic system.

When a software project is still in its infancy, the monolithic architecture can be a good fit. The ease in development, testing, deployment and scaling that this model provides makes it a very popular choice when considering the development budget and the time expected to reach the market. However, as the system's complexity increases over time, all of these factors run into bottlenecks [44].

Successful applications built in this fashion have a habit of outgrowing the monolithic architecture. The growth of a code base caused by active development is directly related to an increase in the management overhead required to maintain the project. A once small development team

quickly transforms into multiple Scrum teams, each developing a particular functional area. Development and deployment become slow and painful: day-to-day tasks like building the application take increasingly longer to complete, and multiple developers pushing new code to the same repository often leads to lengthy and tiresome merges. At scale, monoliths are faced with a similar set of problems. Different application modules may have conflicting resource requirements [44], meaning optimising the deployment machines is virtually impossible, requiring compromises in order to proceed. Depending on the business requirements, the costs of these decisions may be overbearing.

Testing larger and complex monolithic applications is another big pain, since thorough testing is notoriously difficult in large systems. The decrease in testability leads to a decrease in system reliability, so bugs can and will slip into production more often. To add insult to injury, monolithic applications typically lack fault isolation, since all modules are running within the same process [44]. A memory leak in one module, for instance, can make all instances of the application crash. Production outages become a frequent problem for systems that have reached this unfortunate state of affairs.

In addition to the difficulties that emerge with increased complexity in monolithic setups, there is a plethora of other minor issues that affect the day-to-day software developer's life when working with these systems. One such issue is the technological "lock-in" caused by decisions made early on development – developers are forced to work in an increasingly obsolete tech stack, since the monolithic architecture makes adopting new languages or frameworks an unfeasible endeavour [44]. Another problem frequent in these systems is "dependency hell" [19], a colloquial term that refers to the frustration developers face in which adding or updating libraries can result in malfunctioning or misbehaving programs, difficult to troubleshoot. Over time, the pace of software delivery tends to slow considerably as these minor issues accrue.

Overcoming these structural challenges requires a rethinking of the structure of the software system. Companies have since learned that a service-oriented approach is more suitable for their business needs, by accelerating development without compromising quality.

2.1.2 Microservice architecture

Applications that must scale to thousands or even millions of users are increasingly common due to several factors, chief among them the growth and dominance of the digital economy [56]. Companies looking to enter the digital market must be prepared for the challenges that arise at scale.

From these requirements grew service-oriented computing, a paradigm for distributed systems in which a program – a *service* – provides functionalities to other programs, usually through a message passing system [19]. It was borne out of the principle of separation of concerns, famous in object-oriented programming, which grants developers control over the design, implementation and evolution of their systems.

A core tenet of this style of computing is the idea of decoupling service interface from service implementation. This style of computing has several benefits in terms of quality attributes, among them:

- **Scalability** — Instances of a service can be quickly launched to split system load.
- **Reusability** — The notion of composing complex systems as a collection of smaller sub-systems means those same sub-systems can be used by different systems.
- **Agility** — By collectively agreeing on the interfaces of each service, different development teams can work better in parallel.

The first "generation" of a service-oriented architecture (SOA) in the early and mid 2000s was an effort by enterprise-grade developers to define a set of principles, protocols and tools to provide a solid foundation for development.

Unfortunately, widespread adoption of SOA was hampered by the daunting and nebulous requirements for services (e.g. discoverability and service contracts [31]). SOA is typically associated with heavyweight middleware and protocols such as SOAP, WSDL and the WS-* family of standards, which the industry was reluctant to use.

The next step of service-oriented computing was the addition of a notion of *business capability* and seeing it as a primary focus of design so that the underlying basis of the architecture is determined by this element [19]. From these roots grew what we now call the microservices architecture.

The microservice architecture (MSA) is currently the most popular architecture for companies creating new applications, due to its advantages, like agility and scalability [23]. It is sometimes described as a "natural extension of SOA" [48] and even "SOA done right" [31]. It essentially reflects an approach for developing a software system as a suite of small autonomous services deployed independently, each with a single and clearly defined purpose. These services, or modules, are themselves simply computer programs. In contrast with SOA, MSA prefers lightweight middleware and protocols such as REST, for inter-service communication. Figure 2.2 shows an illustration of a microservice application. Clients connect to a gateway mechanism (itself considered a service) which exposes and is connected to all of the services that compose the application, using REST over a private network.

The growing consensus today is that one should consider the microservice architecture if one is looking to build a complex application built for the cloud [44]. In addition to the aforementioned advantages, MSA is generally described as an architectural style primarily characterised by loose coupling and high cohesion. These properties are intimately related to a series of quality attributes such as agility, autonomy, technological heterogeneity, resilience, scalability and reusability [40, 42, 48, 57].

One of the most important benefits of MSA is that it enables continuous delivery and deployment of large, complex systems [44]. This makes it suitable for high-performing DevOps organisations that seek agile, frequent and reliable delivery of their software. Services are small and

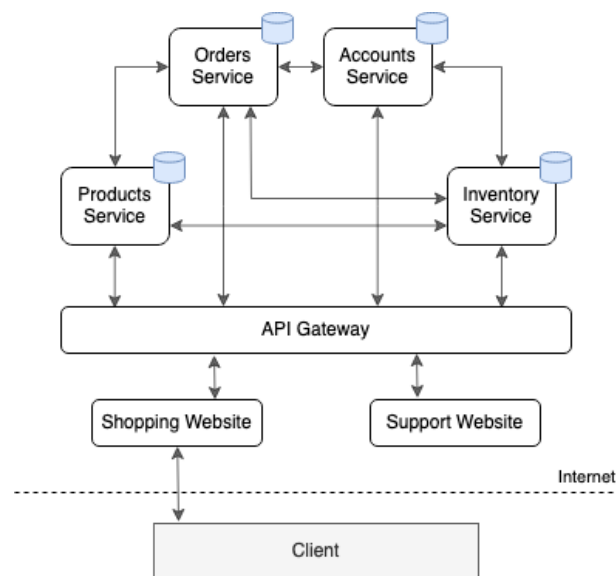


Figure 2.2: An example of a microservice system.

hence easily maintained by independent teams; each service has its own source code repository, its own automated deployment pipelines, its own database (if necessary), and is also independently scalable. Because of this modular nature, this architecture provides better fault isolation: a memory leak in one service will not compromise the runtime of other partitions of the overarching application, meaning other services will continue to handle requests normally. Additionally, software developers can experiment with and adopt new technologies when developing services – long-term commitments to any particular tech stack are no longer a concern. While it is unlikely that teams will choose wildly different programming languages for their many services, the point is that they are not constrained by past decisions.

However, it is not a silver bullet. Several major drawbacks arise when having to design systems based on microservices. Perhaps the most obvious one is that the microservice architecture is a distributed architecture. Developers must deal with the additional complexity of creating a distributed system, requiring decisions about inter-service communication mechanisms, fault tolerance, logging and monitoring, testing, database transactions, etc..

Further particularities are present. Designing a set of small services guided by business capabilities requires a different design strategy than monolithic systems. While objects are usually derived from nouns, microservices promote a "message-first" approach, derived from verbs [45]. Hence, adopting microservices in an existing product becomes a major difficulty. There is not yet a completely automated or otherwise battle-tested tool that can help software engineers decompose a system into services, turning the decomposing process, also known as *microservitisation*, i.e., the identification and definition of the services that make up a system, into something of an art. Adding insult to injury, mistaken decomposition decisions can result in what is termed a *distributed monolith* [43], a distributed system composed of tightly-coupled services that must be deployed together. A system in those conditions is prone to the drawbacks of both the monolithic

and microservice architectures.

There is also a significant operational complexity associated with MSA. Many moving parts must be managed in production. Successfully deploying microservices requires a high degree of automation. Typically, professionals will turn to technologies like Netflix Spinnaker, or Docker orchestration platforms like Docker Swarm or Kubernetes. These deployment options are, in themselves, complex pieces of software which require expertise to fully take advantage of.

Adopting the microservice architecture changes an organisation and its development process. Fundamentally, it transforms people's working environment. If ignored, people's emotions can complicate a transition to microservices [44]. Thus, teams must be prepared to manage the several stages of the refactoring process, from a worker's perspective. Developers must be sufficiently informed about the overarching procedure so they can understand (and hopefully support) the change.

Last but not least, developer tooling is still far from mature. Integrated Development Environments (IDEs) are designed for building monolithic applications and provide little or no support for developing distributed systems.

Each of the aforementioned challenges, and other minor issues, could hopefully be addressed by software- and systems-engineering researchers. To be sure, MSA is a popular research topic, with growing interest. Yet, many research papers and academic proposals have seen little impact on industry practice. One possible explanation for this is that academics have limited access to industry-scale microservice systems, complicating the scientific processes of replication and experimentation [31]. Further collaboration between industry and academia is evidently necessary.

2.1.2.1 Defining a microservice

Microservices are gaining tremendous traction in industry and a growing scientific interest in academia [23]. But what *is* a microservice? Unfortunately, there is no clear agreement on what makes a service a microservice. While the name itself seems to suggest that services should be as small as possible, teams tend to have wide-ranging interpretations of this tenet [31].

Nonetheless, they are all bound by the same set of principles. Microservices should be, according to Richardson [44]: highly maintainable and testable; loosely coupled; independently deployable; designed around business capabilities; and owned by small teams.

2.2 Software qualities

Business and design considerations determine qualities that must be abetted in a system's architecture. These qualities, the totality of which we can describe as *software quality*, are above that of functionality, i.e. above the system's capabilities and behaviour [6]. While there is a close relationship between functionality and quality, these often receive asymmetrical considerations in typical software development projects, spelling doom from the start [6]. Systems are frequently refactored not because they are functionally deficient but because they are difficult to maintain or scale or are too slow or have been compromised by hackers [6].

Given this, in software architecture, we express these qualities of software with quality attributes (QAs). Architecture is important because it allows a system to more clearly satisfy its QAs [43]. The architecture chosen, along with all the architectural decisions made in the design process, enables the complete enrichment of desired qualities [54].

Software architecture, in effect, is critical to the fruitful realisation of qualities in our systems, and these qualities can be evaluated at the architectural level. However, the definition of a given quality attribute is not operational [6]. To affirm that a system will be maintainable has no meaning. Maintainability, like other qualities, is only significant when considered with respect to a set of changes. Put simply, while architecture provides us the foundation for achieving quality, attention on the details and movements over time is a demand that cannot be cast aside.

2.3 Design patterns

The concept of design patterns has its origins in the city building language of Christopher Alexander's 1977 book "A Pattern Language" [1], wherein a network of solution blueprints is conceptualised. This framework was later adapted to the fields of software engineering and software architecture, among other domains, and enjoys a great deal of popularity within them. The most famous example of design patterns in software engineering are the object-oriented patterns defined in Gamma et al. [24], also known as the "Gang of Four".

A pattern is, hence, defined as a proven and established solution to a recurring design problem that is documented in a form that is agnostic of the technology and can be implemented in several different but not identical fashions [9, 14, 35]. Design patterns, hence, entail a set of reusable structures for similar problems found in different contexts and provide a joint vocabulary for the software design community [54].

While there is no standardised structure of patterns description, their documentation is typically presented systematically within a pattern language, i.e. an organised and coherent set of patterns [1]. Authors of these pattern languages will usually describe each patterns with, at least, the following (or semantically equivalent) properties:

- *Context*: Describes the issues present in the context of the system. These forces may be in direct conflict with one other, demanding an evaluation of the trade-offs depending on the goals.
- *Problem*: Describes the problem that the pattern seeks to solve, given the context.
- *Solution*: Describes the procedure required to address the problem.

We will make use of these properties to document particular design patterns in this document. The following serves as a taste, using the SERVICE REGISTRY pattern (adapted from Osses et al. [41]) as an example:

SERVICE REGISTRY

- *Context*: Many different business microservices comprise an application.
- *Problem*: How to decouple the physical address of a service instance from its clients so that the client code will not have to change if the address of the service changes?
- *Solution*: Map between a unique identifier and the current address of a service instance in order to decouple the physical address of a service from the identifier.

Architectural patterns reflect a notion akin to design patterns, but on a broader scope; they reflect an image of a system, i.e., a concept that solves and outlines essential elements of a software system's architecture. In this sense, we can conceive of a MONOLITH as an architectural pattern, or a MICROSERVICE pattern, *et hoc genus omne*.

A pattern language typically presents the relationship between patterns visually. These relationships reflect the concepts of:

- *Predecessor*: A pattern that motivates the need for this pattern.
- *Successor*: A pattern that solves an issue that was introduced by this pattern.
- *Alternative*: A pattern that provides an alternative solution to this pattern.

Understanding these relationships helps guide software architects towards better informed decisions, and prove invaluable for defining a set of qualities for an entire system. Figure 2.3 shows an example of a pattern language, from a high-level perspective – i.e., the domain-specific patterns are omitted.

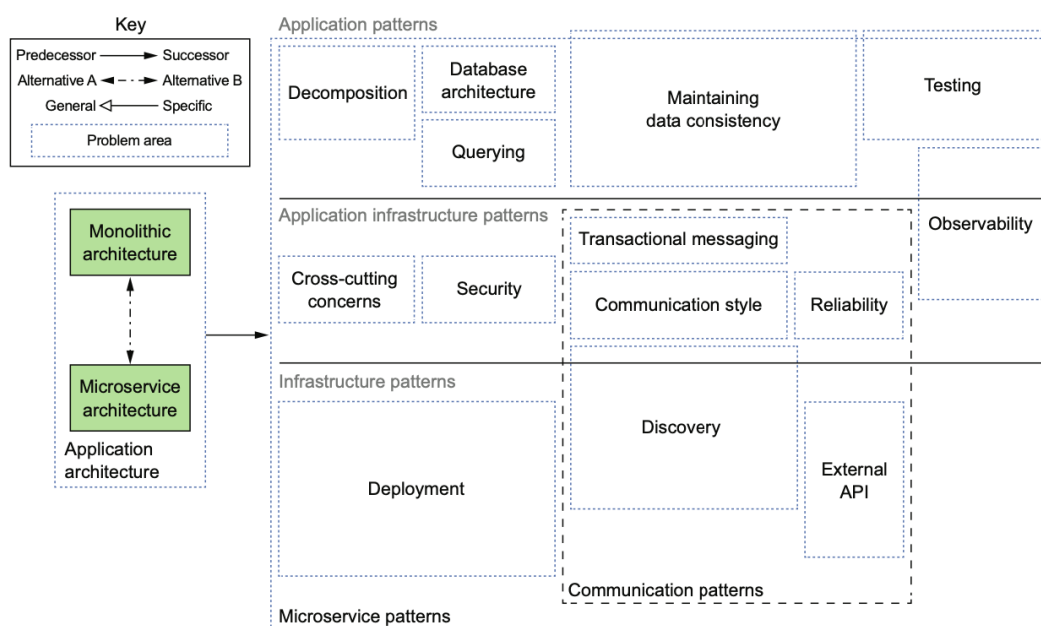


Figure 2.3: High-level view of a microservice pattern language, as defined and presented by Richardson [44].

Architectural patterns are sometimes used interchangeably with design patterns. In the interest of clarity, throughout this dissertation we will use the term "architectural pattern" as equivalent to "architectural style" (e.g. monolithic, microservices, etc.), and the term "pattern" to refer to design patterns of any sort.

2.4 Summary

With this chapter we have provided readers with a theoretical account of the constituting elements of our research. While we do not explore these topics in great depth, our overview allows the reader to appreciate the contours and the purposes of this work.

Our work revolves around the design of the microservices architecture, a recent development in the field of commercial distributed systems. We are particularly interested in exploring how to achieve stronger designs of these complex systems through the usage and application of design patterns. However, design patterns, like any other design tool, typically imply a conceptual re-thinking of the structure of our applications in order to draw the best usage out of them. That is, the usage of design patterns produces effects which reverberate through several domains, some of which are invariably negative. Put simply, design patterns come with a set of trade-offs which should be duly considered. We express these trade-offs through quality attributes. An analysis of these trade-offs gives both researchers and professionals an informed perspective which can be used to guide the decision making process.

Chapter 3

State of the Art

In this chapter we present a thorough overview of previously published work related to the goals of this dissertation. Section 3.1 describes the scientific literature on software qualities in a microservice architecture, e.g. how quality attributes are perceived, trade-offs of the architectural style in general, design patterns as tools for achieving better design, and other approaches. Section 3.2 describes the studies on conducting empirical investigations in software engineering of microservice systems. Hence, this chapter aims to reflect on the published studies about microservices and their dissimilarity with the present work, in order to help clearly locate and formulate the specific research problem to be addressed in this dissertation.

This search for literature was not systematic nor driven by any formal research questions. Our strategy was to look for articles most closely related to our primary objective – an empirical study on software architecture trade-off analysis – and, after exhausting those results, branch out into more general related works.

The studies mentioned in this chapter were found through various academic resource libraries, namely the Google Scholar engine, the IEEE Xplore Digital Library and Scopus. Several queries were used for discovery, but to constrain ourselves to studies on MSA, the following string was prepended to all queries:

(architect* OR design* OR system* OR structur*) AND (microservi* OR micro-servi* OR "micro servi"*)

Since we are chiefly concerned with patterns and quality attributes, we achieved a further narrowing of our search space with the addition of terms in a way illustrated by the following string:

("design pattern*" OR pattern*) OR (qualit* OR "software qualit*") OR (tradeoff* OR trade-off*)

We sort the following studies by year of publication primarily and by author secondarily, for each section.

3.1 Software quality in microservices

While there is a modest wealth of academic interest in identifying and defining patterns, there is relatively little work done to crystallise the knowledge about them and their impact on software quality [38]. This poses challenges for software architects deciding which patterns might best fit their needs. Since these decisions have long-lasting impacts on several aspects of a system [20], they also affect quality attributes such as testability, security, performance and reusability [50, 33]. Stated more clearly, each design decision generates trade-offs.

Hassan and Bahsoon [30] summarised views of microservices from informal (i.e. grey) literature, hoping to define a stronger fundamental context of MSA. Beginning with the premise that there is a *"lack of academic consensus regarding the definition and properties of the paradigm shift and corresponding design patterns for microservices"*, the authors' motivation was to aid in this understanding of the design problems in the microservitisation process. The authors reviewed the state of the art and formulate the design trade-offs of microservitisation, along with their proposed solution, which they define as a *"self-adaptive road-map"*. In essence, the authors identify two major trade-offs in the microservice refactoring process:

- The number of services vs. the size of each service. While the introduction of microservices may lead to better isolation of business functionalities, this comes with costs in inter-service network communications and object distribution complexity.
- The global vs. local satisfaction of non-functional requirements. The NFR satisfaction levels of individual microservices must be balanced with the satisfaction levels of the overall system.

The authors also attempt to draw a further distinction between SOA and MSA, affirming that microservices are *"autonomous fine-grained computational units"* and that this property enhances flexibility. The authors propose a step-by-step solution based on a MAPE-K loop aiming to improve system quality with knowledge accumulation over time.

Soldani et al. [48] conducted a grey literature review on *"pains and gains"* of microservices. An investigation of 51 industrial studies (e.g. whitepapers, videos, blog posts), published between 2014 and 2017, yielded many interesting observations on professional perception of microservices.

The authors class the pains and gains in terms of *stages*, which refer to common steps in the life cycles of software development – design, development and operation. We summarise their findings in the following list:

- In the *design stage*, professionals are (perhaps unexpectedly) chiefly concerned with the application's architecture. Within architectural concerns, the size (or complexity) of the application is called more often into question as a primary concern, followed by concerns related to the size of services and API versioning. Securing microservices, while not reported as often as a pain, typically relates to the challenges of access control and endpoint proliferation.

On the plus side, professionals report several concrete gains with MSA. The fact that microservices lend themselves to bounded contexts, cloud nativity, fault tolerance and flexibility was often praised in the surveyed data. Design patterns are also heavily taken into account as a boon for microservices, with DATABASE PER SERVICE (each service equipped its own database) most often reported, followed by API GATEWAY (public endpoint for clients of a microservices-based application).

- In the *development stage*, practitioners perceive storage and testing to be both significant concerns in microservices. Data consistency and distributed transactions are the most concerning pains when handling storage in MSA. Performance testing was overwhelmingly reported as a challenge, highlighting a need for solutions to the problem of measuring performance in applications based on possibly huge numbers of independently evolving partitions.

Despite this, the development of microservices is the most-reported source of gains in MSA. Developers frequently profess loose coupling and the freedom given to choose which technology to adopt when creating services as the most attractive gains. Furthermore, the architecture's natural inclination for supporting a development process that supports Continuous Integration/Continuous Delivery (CI/D) was considered an important gain.

- In the *operation stage*, concerns are more evenly distributed, suggesting a lack of tooling specifically oriented for microservices. Operational complexity are notoriously difficult to address in MSA management; a similar challenge is faced with regards to monitoring in MSA, with complexity reported as the largest obstacle, followed by logging.

Regardless, microservices seem to provide significant benefits in operations. Professionals primarily favour microservice independence, i.e. each service can be deployed or taken down independently, followed by containerisation. Consequently, scalability is the most often reported quality attribute, from a microservices management perspective.

The authors note that the understanding of microservices is already mature in industry, but not in academia. The importance of our work, which seeks to analyse current industry practice, is made evident.

Zdun et al. [60] propose a reusable architectural decision model, based on a UML2 meta-model, to help optimise quality aspects of service API design. By way of a qualitative study, the authors identified six architectural design decisions, with 40 decision options in total guided by 47 decision drivers.

Osses et al. [41] sought to explore and review design patterns and tactics in microservices in academic literature, with the motivation to described the design solutions proposed, as well as identifying the quality attributes associated with these self-same patterns and tactics. The authors synthesise 44 design patterns they found in the literature and performed a quality assessment to appraise the quality of each pattern in terms of completeness in documentation: the extent to which each pattern describes its applicability context, the problem it aims to solve, its proposed solution,

and whether or not each pattern clearly identifies quality attributes it is related to. From this assessment the authors find that 48% satisfy their quality criteria, i.e. almost half of the patterns have a complete description. Further, the authors propose a taxonomy of these patterns, classifying each into one area of interest. These areas are: *Front-End*, *Back-End*, *DevOps*, *IoT*, *Migration*, and *Orchestration*. Figure 3.1 shows the relationship between the identified design patterns and their classification. There is a predominance of design patterns concerned with migration, i.e. the decomposition of a system into smaller services.

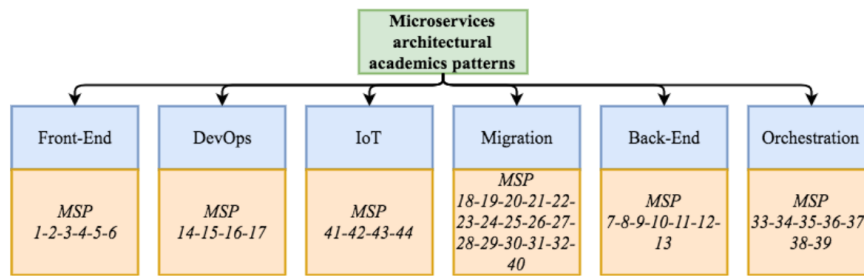


Figure 3.1: Proposed taxonomy of design patterns, as presented by Osses et al. [41]

Additionally, the authors did not find any sufficient evidence on architectural tactics for microservices directly, noting, however a presence of tactics from related disciplines like DevOps and IoT. Finally, the authors note that *scalability* and *performance* are the most frequently found QAs in microservice design patterns, followed by *flexibility*, *elasticity* and *testability*.

Valdivia et al. [53] observed that the mapping between quality attributes and patterns in microservices was not clear and sought to fill that gap with a systematic literature review. The authors presents a collection of patterns found and, for each pattern, the quality attribute(s) related to it i.e. the software qualities that are improved resulting from the adoption of the pattern. For example, the authors map the API GATEWAY pattern to reliability and security, the SIDECAR pattern to maintainability, etc.. The authors map this relationship by drawing strictly on the documentation available for each pattern but not all patterns have a documented association with a quality attribute. In these instances, the authors consulted an outside researcher with expertise in microservices to help confirm their assumptions. Additionally, the authors classify each pattern in terms of the areas in which most benefits will be seen. They use six classification groups: *Data persistence*, *Communication*, *Entry point*, *Distribution*, *Fault tolerance* and *Supplementals*. For example, the BULKHEAD pattern is mapped to *Fault tolerance*, the ANTI-CORRUPTION LAYER to *Distribution*, etc..

Regarding patterns in general, the authors note that applying a pattern or a collection of patterns has consequences on the structural software qualities, thereby demanding a thorough analysis of the trade-offs between patterns and quality attributes. Furthermore, the literature on microservice design patterns is still growing as more patterns are identified, so the list of patterns analysed is not final. Finally, the authors also identify metrics associated with the evaluation of quality

attributes in design patterns. These are: time, percentage of requests accepted, number of files modified, and energy consumption.

The authors follow up on this work in [54], attempting an improved synthesis of known design patterns in the academic and industrial literature and tying them together with QAs and associated metrics. By integrating grey literature into their work, the authors propose the first multivocal literature review about patterns in MSA. From this expanded perspective the authors identified 20 new design patterns, bringing the total to 54. Their results also show that the most frequently identified patterns are related to inter-service communication and coordination, e.g. ASYNCHRONOUS MESSAGE-PASSING, SERVICE REGISTRY, etc..

Rosa et al. [14] sought to develop a method for an architectural trade-off analysis based on patterns in microservices, with the goal of identifying the relationship between a set of twenty-four patterns and three structural attributes: the size of services, the degree of database sharing between services, and the coupling level between services. A five-step method for architectural analysis was formulated and applied by each author and conflict points were discussed in an effort to remove personal biases. Figure 3.2 shows a synthesis of the results obtained from a demonstration their method.

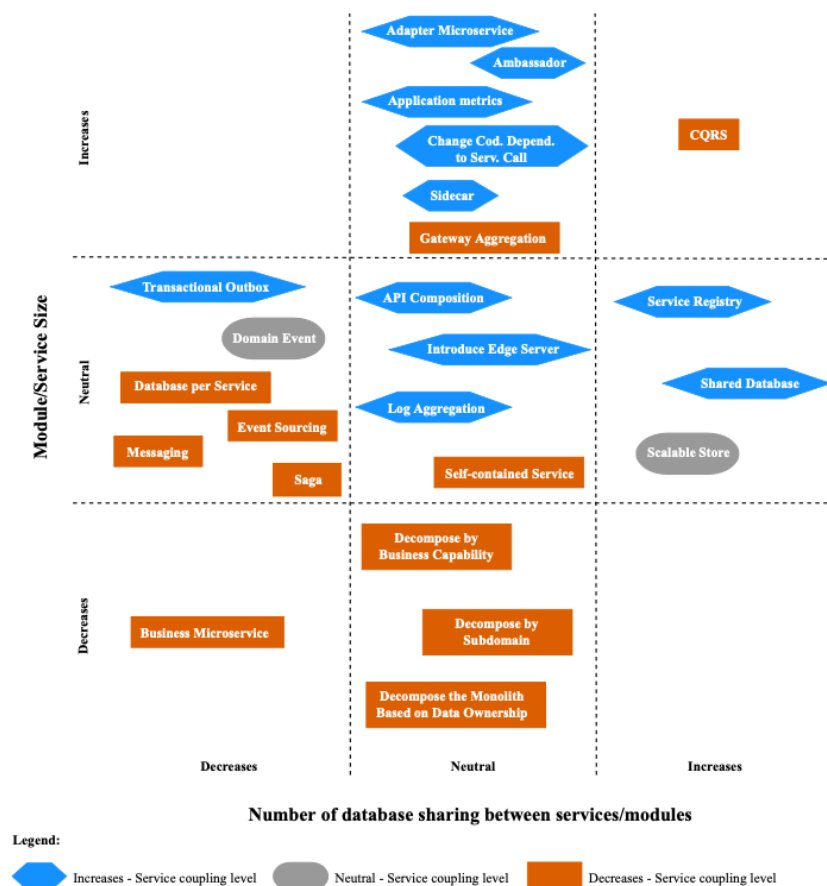


Figure 3.2: Pattern trade-off diagram, as presented by Rosa et al. [14]

Several takeaways emerge from their results, above all the level of influence exerted by each

pattern on the given structural attributes, enabling developers to choose an optimal set of patterns to apply in their projects.

Li et al. [34] sought a comprehensive understanding of quality attributes in the context of MSA through a systematic literature review. In the study's opening remarks, the authors state that "*in contrast to the rapid and widespread adoption of and migration to MSA in the software industry, the shortage in the comprehensive understanding on the QAs of MSA turns out to be significant*".

Based on the data from the 72 primary studies that were selected, the authors begin by identifying six quality attributes that reflect most concerns in the current literature. They are: *scalability*, *performance*, *availability*, *monitorability*, *security*, and *testability*. The authors ground the usually hazy and inconsistent understandings of these attributes in the coherent and more rigid definitions provided by commonly accepted standards like ISO/IEC 25010. Apart from these attributes the author found two other attributes being frequently mentioned in the literature, though these did not pass their initial research checklist designed to evaluate QAs as essential to microservices. They are *maintainability* and *interoperability*.

As an additional contribution, the authors conceptualise each QA in the context of MSA and synthesise 19 known tactics for addressing them. Figure 3.3 shows an illustration of these findings.

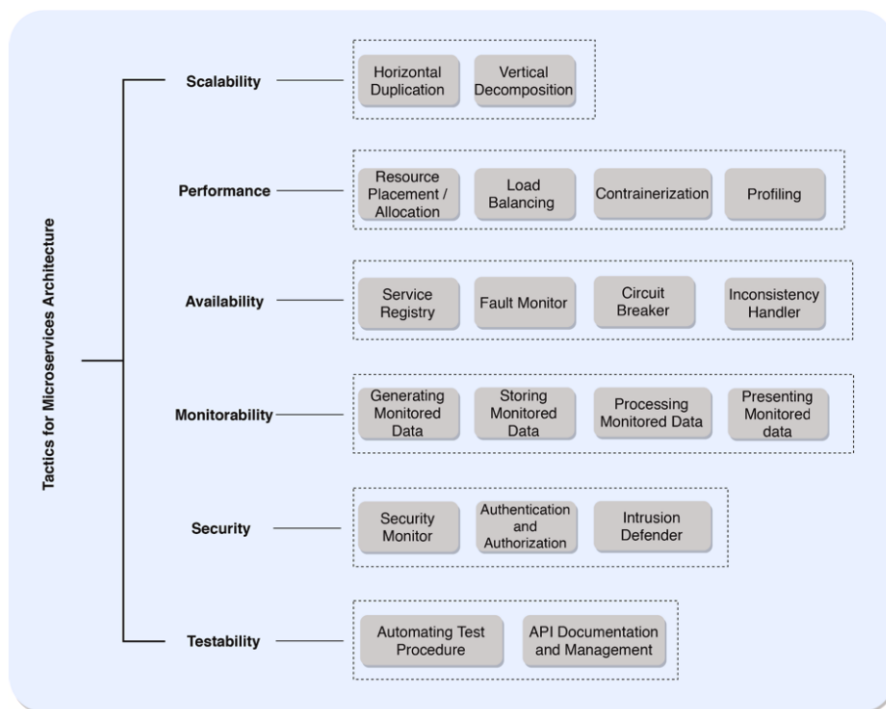


Figure 3.3: Known tactics of addressing essential microservice quality attributes, as presented by Li et al. [34]

The authors recommend a serious consideration of quality attribute trade-offs for practitioners looking to migrate to MSA. Additionally, they affirm a need for further empirical research to study how to address particular QAs like *maintainability* in MSA.

3.1.1 Discussion

This review presented a review of the published literature on software qualities within microservices. Outside of microservices, Wedyan and Abufakher [58] noted, in a recent work, that the research on impacts of design patterns on software quality, in all sorts of software architectures, has led to results which are contradictory and difficult to compare. The authors set out to illuminate the confounding factors, practices or programming constructs that affect QAs with the usage of design patterns, as well as which QAs are evaluated, how they are evaluated and with which metrics, through a systematic review of the literature.

- Providing well-documented examples of design patterns significantly reduces the cost of maintenance, in terms of time and errors, compared to bare bones examples.
- The size of a software module, e.g. a class in object-oriented programming, has a clear and inverse effect on several QAs, as large and complex modules become harder to maintain and test, thus becoming error-prone.
- Design patterns that cut across multiple design concerns without proper a structure model generate tangled code, increasing coupling and, consequently, decreasing modularity, negatively affecting software quality.

These remarks may explain why our search for studies specifically related to architectural consequences of design patterns on software quality within microservices yielded only a handful of articles. Even the works that aimed to map relationships between design patterns and quality attributes suffer from no empirical validation of their results, implying a need for direct contact with industrial reality.

Researchers show, instead, much more interest in making sense of concerns related to engineering microservices from different angles – exploring strategies for deploying services as containers, inter-service communication models, testing tools, etc.. For these areas, authors developed methods and decision models to take the best advantage possible of MSA.

3.2 Empirical research in microservices

In the systematic mapping study of the published literature on the microservice architecture by Di Francesco et al. [18], a classification framework for research studies is produced which gives us a clearer picture of the different sorts of research being conducted in the context of software architectures based on microservices. For our purposes, we highlight a key finding from the results of their study, which states that *evaluation research* is "very rarely performed", having found only one such study in a sample of 119 studies. The authors describe this type of research as industry- and practitioners-oriented studies – e.g. industrial case studies, controlled experiments, surveys, etc. – and assert that a shortage of these types of studies hampers the technology transfer of research results in industry.

In order to address the controversial evidence surrounding the general lack of empirical studies in microservice systems, we will present, in the following subsections arranged by method, an overview of the evaluation research conducted to alleviate this gap.

3.2.1 Case studies

A case study is conducted to investigate a phenomenon within a specific time interval, and involves a thorough collection of data on a single project during a sustained period of time [59]. It is an observational method, that is, performed by observation of an ongoing process. Case studies are useful tools for comparing two or more methods, typically to evaluate the effects of a change compared to some baseline. Case studies are advantageous in that they are easier to plan and execute but their results can be difficult to generalise and interpret [59].

The earliest empirical contribution to the microservices literature is the work by Villamizar et al. [56]. The authors present a case study that aimed to evaluate the contrast between deploying "in the cloud" an enterprise-grade application with a monolithic architecture versus with a microservice architecture. More specifically, the authors sought to identify areas in both the development and operations processes that are affected when the microservices architectural style is used. The results confirmed much of the preconceived benefits and drawbacks of microservices – perhaps unsurprisingly, since this architecture originated in industry before being an object of academic interest: authors noted an increase in agility, reduction in deployment costs, and granular scalability. They also mentioned that, compared to a monolithic deployment, microservices required a more rigorous and careful coordination and orchestration related to service versioning. The authors note that the adoption of microservices requires a new culture of development that should be complemented by company guidelines, i.e.: to maximise the usage of this new paradigm, developers should be well-informed, lest they fall into anti-patterns.

Balalaie et al. [4] empirically evaluate their own proposed catalogue of migration patterns by analysing three industry case studies. The authors observed a distinctive recurrence of the proposed patterns which reflect the basic tenets of microservices in practice, suggesting that their academic contribution can play a key role in supporting enterprise-grade migration efforts.

Haselböck et al. [28] performed a design space analysis to propose design spaces and decision models for MSA, and consequently applied their task proposals in an industrial context with a case study. The authors devise a several decision models for each of the design spaces they identify. Figure 3.4 shows their proposed microservice API management process and the sequence of design spaces. The authors found success with their case study, noting its worth in an industrial context, useful for exploring, structuring and capturing design spaces as well as improving the understanding of decisions made.

Cojocaru et al. [12] evaluated their own proposal through the use an industrial case study. Their goal was to provide a minimum recommended set of quality attributes applicable to MSA, based on two meta-criteria: meaningfulness in the migration process and feasibility of implementation. From an exhaustive list of quality attributes, the authors selected a minimum set of quality

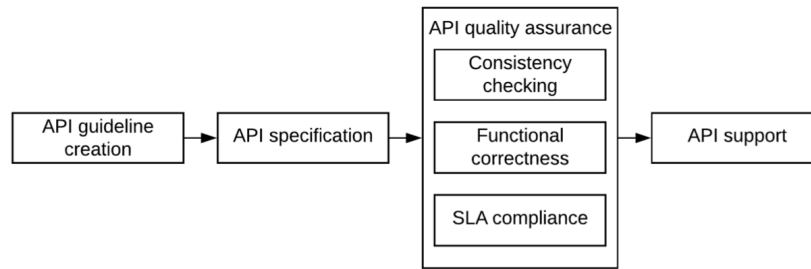


Figure 3.4: Microservice API management design spaces and process, as presented by Haselböck et al. [28]

attributes, and interviewed an industry expert for opinions on the proposal. The proposed set of quality attributes is:

- **Granularity:** also known as size, the granularity of a microservice is important since the architectural style places emphasis on services being small (hence the prefix *micro*). However, since there is not a generally-accepted definition for the target size of a service, this attribute may be controversial. Regardless, the size of a service relates to its deployment costs and also its quality assurance costs. Service granularity is important in industry because it relates to the size of the development team working on the service. Size can be determined through static analysis, either by measuring lines of code (LOC) or the number of exposed interfaces.
- **Cohesion:** the degree to which a service's features are oriented towards one functionality. A service with low cohesion usually indicates the need for further decomposition into smaller and more cohesive services that fulfil the "single responsibility" principle. An oft-quoted tenet of microservices, this attribute is, however, difficult to measure automatically. Assessing cohesion, in industry, usually relies on the architect's subjective opinion.
- **Coupling:** the degree of dependencies and connections between microservices. Along with cohesion, it is a defining feature of microservices, since services should be loosely coupled. Coupling can be measured by identifying network invocations and dependencies, e.g. using Wirehark or other tools.
- **Scalability:** the ability of a microservice to behave as designed independent of the change in amount of resources used without incurring performance penalties. Can be increased horizontally, i.e., by replicating the microservice) or vertically, i.e. by allocating more computing resources to a microservice. Scalability can be determined through dynamic analysis, by analysing the distribution of synchronous requests provided by the exposed interfaces, i.e. a low distribution would indicate good scalability.

- **Response time:** the expected delay between the instant when request is issued to a service and the instant when the response is delivered. Response time can be determined through dynamic analysis.
- **Security:** the degree to which a microservice is vulnerable to malicious disruptions. While its importance cannot be overstated in large-scale applications, security in microservices is rarely a topic of research.
- **Resiliency:** the ability of a service to handle failures, usually by gracefully recovering to a working state. Services should be resilient so as to avoid compromising key business functionalities that might be made impossible if one subsystem fails (e.g. failure of a service required for cross-service database transactions). Enterprises value resiliency because they reduce these risks of disruption. Resiliency is usually measured, in industry, by way of software tools such as Netflix Chaos Monkey.
- **Execution cost:** the monetary costs of the resources required for running a microservice. With obvious importance for industry practitioners, this attribute is usually determined by estimation at design time, based on the chosen deployment platforms and their respective costs.
- **Reusability:** the degree to which services in the system may fulfil multiple business needs. Reusable services may be difficult to implement but the resulting reusability may end up saving development costs, thus being an important attribute for enterprises.

The results of the interview show a predominant match in assessments, thereby helping confirm the strength of the research. The authors also enquired the specialist regarding the motivation for migrating from monolith to microservices, how such a process is carried out, and which type of migration process to choose (rebuilding vis-à-vis refactoring).

Matias et al. [37] conducted a case study to evaluate an original systematic approach for determining microservice boundaries using both static and dynamic software analysis, a popular topic of academic interest in MSA. The authors presented two suggestions of a system decomposition – one obtained through their original prototype and another tool that uses only static analysis – for a large web application to three software developers who work on it. Participants were then presented with a questionnaire, indicating their assessment of the feasibility and benefits of the proposed approach. The results regarded the authors' systematic approach as positive, strengthening the value of the work done into service decomposition, and helping assess the direction that future research should explore.

de Toledo et al. [16] looked specifically toward architectural debt issues in MSA, i.e. sub-optimal architectural solutions which bring with them increasing costs in the long term – examples include: applying design patterns in the wrong contexts; the presence of cyclic dependencies; betraying a decided architecture in favour of 'quick fixes', etc.. To identify the most common and critical architectural debt issues in MSA, the authors conducted a multiple-case study ($n = 7$) with

large international companies, aided by 25 interviews ($n = 22$) with employees in different roles. From this study, the authors formulate a catalogue of 12 architectural technical debts, some of which further developed into sub-types. Additionally, for most of these, solutions that interviewees deployed to minimise technical debts are identified. The state of maturity of MSA – rather, lack of maturity – is a particular point of interest, as this means different practitioners have wide-ranging interpretations of MSA, generating different opinions on solving problems. The problem is further compounded by the lack of standards, unlike SOA which has the WS-* family of standards to guide developers. For our purposes, this is an indication that design patterns should be mined from the solutions reported and disseminated to help professionals avoid pitfalls.

3.2.2 Controlled experiments

In controlled experiments, researchers have greater control over the study and how participants carry out their assigned tasks. Unlike case studies, where researchers are observers in the process, they dictate the structure of the study in this form of research. A big advantage of experiments is that they can be planned and designed to ensure high validity, but this is usually contrasted by the limitations in scope needed to carry out the study [59]. For instance, a complete web development project could be seen as a case study but a typical controlled experiment will not include all of the activities of such a project [59]. The purpose of these experiments is to identify and manipulate key factors and control all other variables at fixed levels. A statistical analysis of the effects of this manipulation is later performed to draw significant conclusions.

Bogner et al. [9] sought to empirically study the impact of four service-based patterns on the evolvability¹ of a system from the viewpoint of inexperienced practitioners, by conducting a controlled experiment with Bachelor students ($n = 69$). Two groups worked on functionally equivalent service-based systems, and were presented with tasks to add new features. One group worked on a version designed with four patterns – PROCESS ABSTRACTION, SERVICE FAÇADE, DECOMPOSED CAPABILITY, and EVENT MESSAGING – believed to help increase evolvability; the other group worked on a version built in an "ordinary" fashion, i.e., without the use of the aforementioned patterns. Their results suggest that the patterns adopted had a positive effect on the service coupling level but a negative effect on the size and granularity of each service. On the whole, there was "no clear evidence" for decisively concluding that the chosen patterns were considerably helpful for inexperienced developers. They remark that such a positive effect might only be seen with more experienced developers.

3.2.3 Surveys

Surveys are investigations performed in retrospect, e.g., when a method or tool has been in use for an extended period of time. Surveys take a sample representative of the population to be studied, and the results are analysed to derive descriptive and explanatory conclusions [59]. By taking a

¹Evolvability, also known as modifiability, is defined as "the degree of effectiveness and efficiency with which a software system can be modified to adapt or extend it" [46].

sample that is representative from the population to be studied, the results from the survey are generalised to the population from which the sample was taken. The two most common forms of surveys are questionnaires and interviews, wherein both qualitative and quantitative data can be gathered [59].

Haselböck et al. [29] investigated the use of decision models in MSA through the use of the technical action research method. The authors performed this study ($n = 2$) with companies working with MSA, and the objects of inquiry were use cases and stakeholders of decision models in microservices, alongside required elements for decision models and presentation of microservices. The authors went through four cycles, participating in design workshops each company conducted regularly as part of their Scrum cycles. The participants used the researchers' proposed decision models to discuss design concerns in their own professional contexts, while the researchers initially acted as observers and then interviewed participants to obtain feedback. As results of their work, the authors developed a set of design areas in MSA. Figure 3.5 shows a proposed taxonomy of these design concerns.

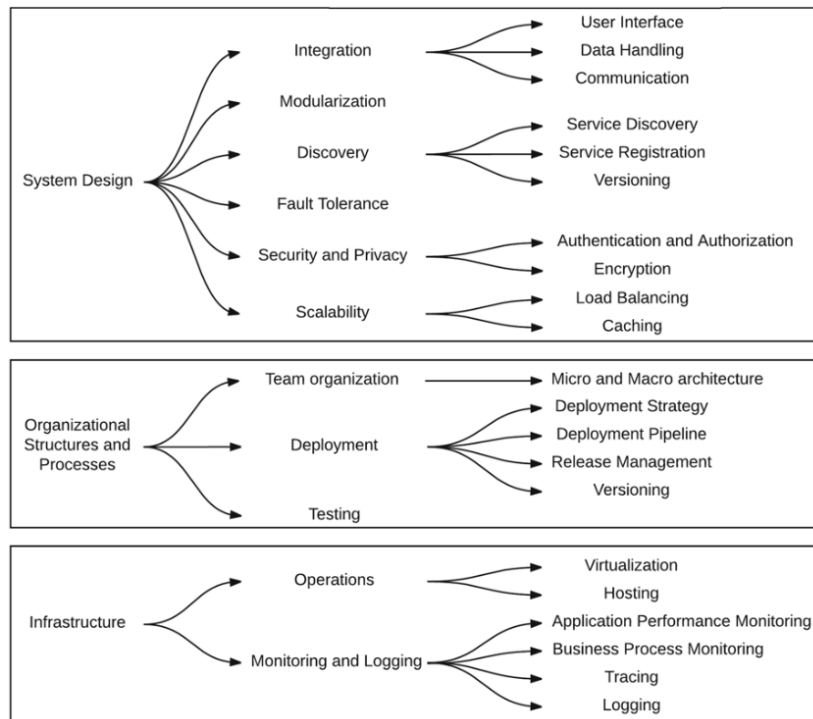


Figure 3.5: Areas of microservice design, as presented by Haselböck et al. [29]

Additionally, the primary stakeholders for decision models in microservices are *Software architects, Developers, Application engineers, Operations engineers, Quality assurers, and Managers*. By studying how participants applied the proposed decision model, the authors also identified the main use cases of such a tool. These are: *Design Space Exploration; Decision Guidance and Documentation; Design Review; Requirements Elicitation; and Evaluation of Technology Options*.

Di Francesco et al. [23] designed and conducted a survey ($n=21$) with practitioners involved in the monolith-to-microservices refactoring process. This quantitative study's main contribution is the industry insight on the increasingly common migration process. Participants were presented with a questionnaire composed of 35 questions, 31 of them closed-ended mandatory questions, the rest optional, with room for broader answers. This study provides us with a great window into designing this type of research. In an initial step, the authors performed 5 long-form exploratory interviews to become familiar with the industrial perception of the topic and thus guide the content of the questionnaire. The results shed light on how professionals approach refactoring their legacy systems to a microservice architecture. Participants overwhelmingly report a phased and iterative migration process, i.e. by slowly transforming the system into a suite of services according to new or prioritised features. When asked about the activities performed when designing the new architecture, a variety of approaches emerge – domain decomposition, service identification, application of DDD principles chief among them. This suggests that professionals go to great lengths to understand and define the bounded contexts of their systems.

Ghofrani and Lübke [25] conducted an online questionnaire survey ($n = 25$) with participants that indicated involvement in projects with a MSA, a majority of which were practitioners. The primary goal of the survey was to collect challenges found by professionals in microservices systems. The survey results show interesting points: security is an important concern for an overwhelming number of participants, with many emphasising a demand for better security-oriented third-party tools and libraries in microservices. Participants noted a serious lack of notations, tools, methods and frameworks to help guide the architecture process, further evidencing a demand for research in this direction.

Haselböck et al. [27] conducted interviews with industry experts ($n = 10$) to determine the importance of different areas of microservice design. Figure 3.6 shows the importance rating for each design area identified.

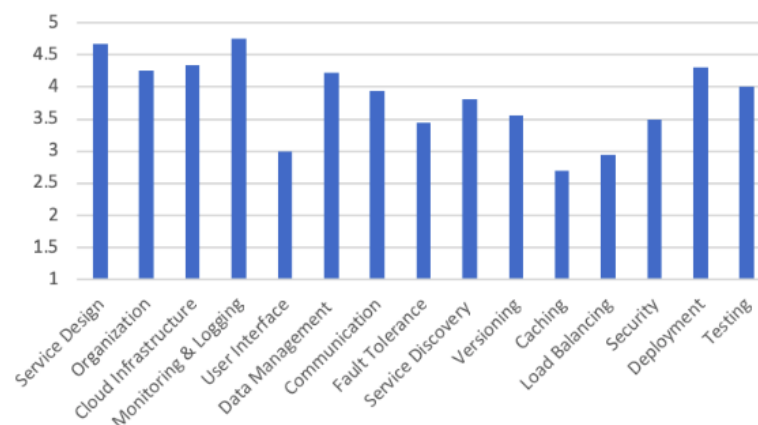


Figure 3.6: Importance of identified design areas in microservices, as presented by Haselböck et al. [27]

Monitoring and Logging receive the most importance, suggesting that practitioners see real

challenges in observing a complex distributed system based on microservices. Other areas trail close by in importance, but we can clearly see that *User Interface*, *Caching* and *Load Balancing* are not topics of particular concern for designing microservices.

Taibi and Lenarduzzi [51] collected reports of bad practices in MSA by interviewing ($n = 72$) developers experienced with microservice systems, and classified their answers into 11 'smells' considered harmful by practitioners. In addition, the authors give us insight into which of these bad practices have the greatest impact on software maintainability and how professionals are overcoming them.

Viggiato et al. [55] noted a lack of empirical evidence about the use of MSA and common practices found in industry and sought to remedy this with a questionnaire survey ($n = 122$). The authors started with a mapping study to gain a deeper understanding of the emerging field, namely its advantages and challenges. From this preliminary work, the authors extracted recurring topics and classified them as pros and cons. They proceeded to design a survey aimed at verifying the characterisation of MSA that they previously created. A 14-question, 3-section questionnaire was sent to developers active in the Stack Overflow community. Figure 3.7 shows the perception the survey respondents have of advantages and challenges in MSA. By and large, the authors note a healthy match between the literature and actual practice. Some dissonance was found, though, namely regarding the concern professionals have with *expensive remote calls* – while it was one of the challenges identified, most participants give it little or no importance.

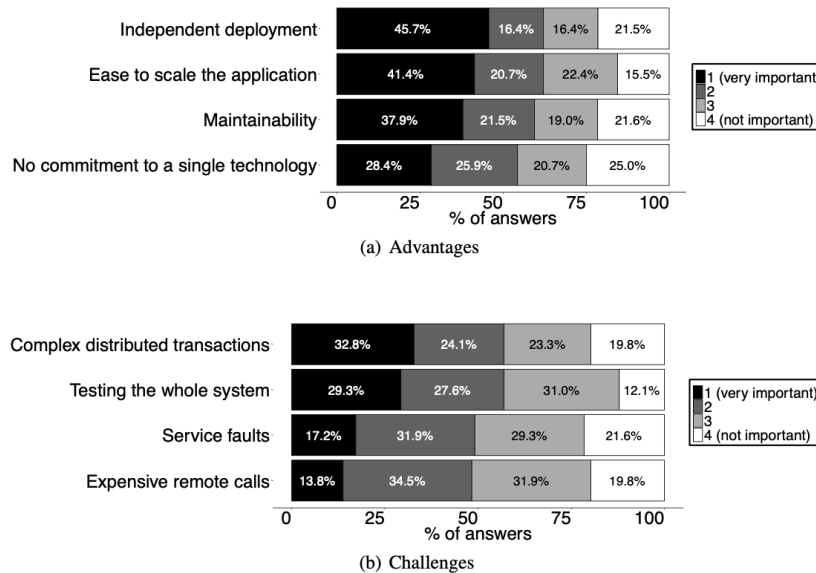


Figure 3.7: Microservice advantages and challenges felt by professionals, as presented by Viggiato et al. [55]

Bogner et al. [8] sought to paint a clear picture of the usage of microservices in industry. The authors present a qualitative study: in-depth interviews ($n = 17$) with software professionals from 10 different companies, hoping to find seasoned perspectives on applied service technologies, adherence to the microservice architecture and its influence on software quality. With regards to the

latter, participants generally agreed that maintainability was most positively impacted. Portability, reliability and compatibility were other quality attributes that saw a positive influence. The remaining quality attributes (*performance efficiency, usability, security, functional stability*) were generally not considered impacted in microservices implementations. However, security was the most controversial quality attribute, with 4 participants declaring it negatively impacted: the participants' reasoning generally revolved around the increased attack surface in this type of distributed architecture. The authors also state that the majority of the empirical research on microservices is quantitative and survey-based, noting that additional industry-focused research could be gained by qualitative and interview-based methods.

Carvalho et al. [11] sought to understand the criteria actually adopted in industry to support the extraction of microservices by performing an exploratory survey ($n = 15$). Their results suggest practitioners use around four criteria for assessing their decision, simultaneously, differing from academic solutions which posit only one or two criteria (typically coupling and cohesion). One important observation from the authors is "*Mistaken decisions on microservice extractions are mentioned to be often related to the lack of synthesised information about the relevant criteria and their trade-offs*", stressing the importance of closer contact between academia and industry.

Knoche and Hasselbring [32] conducted a survey among software development professionals ($n = 71$) in Germany, in order to gain insight into the reasons why companies might consider the adoption of MSA. The researchers attended industry meetings and conferences and forwarded their questionnaire to professionals, to be answered in paper or web form. The questionnaire sought to discover (a) primary drivers for companies to adopt microservices; (b) primary barriers preventing companies from adopting microservices; (c) modernisation goals for existing applications; and (d) how professionals rate the impact of MSA on runtime performance and data consistency. The authors found that the primary drivers for the adoption of MSA were scalability, maintainability and time to market; the primary barriers, however, were developer inexperience and resistance to the paradigm shift by operations staff. Respondents affirmed that modernisation of legacy applications into microservice-based systems were a primary goal of adoption. Finally, most respondents considered the loss of familiar ACID transactions a serious limitation in regards to ensuring data consistency.

Zhang et al. [61] performed interviews ($n = 13$) with different companies to determine what they term a gap "*between the ideal characteristics to be expected for MSA and the reality from practical experiences*". Additionally, they queried professionals on the benefits and expenses of actually implementing a microservice architecture. Based on eight of the nine characteristics provided by Fowler and James [22] – *Componentisation via Services, Organised around Business Capabilities, Smart endpoints and dumb pipes, Decentralised Governance, Decentralised Data Management, Infrastructure Automation, Design for failure, Evolutionary Design* – the authors asked participants about each and how they perceive them.

Sousa et al. [49] wanted to determine how a language of twelve cloud development patterns are regarded and adopted by practitioners. The authors investigated the extent to which system particularities influenced the adoption of the given patterns, and how likely they were to adopt others

after adopting one. An online questionnaire ($n = 102$) targeting cloud computing professionals was conducted. The authors were interested not just in which of the shown patterns professionals actually use, but if the rate of adoption varies depending on a project's operational complexity, active monthly users and the company's size.

Ayas et al. [2] investigated 16 cases of microservitisation by conducting a grounded theory interview study ($n = 19$) with participants that had recently been through a migration to MSA. The authors sought to get a feel for the decision-making process of organisations during migrations to microservices, as well as which solutions are typically chosen when faced with the many decisions implied in the transition to MSA. The authors found a multitude of decisions required in the migration process which they distinguish into two broad types: *procedural decision-points* – points during the migration involving how to continue the process – and *outcome decision-points* – points during the migration involving a team-wide agreement on specific architectures or technologies as outcomes of the process. From their findings, the authors chart a sequence of major decisions that should be made in order to ensure a smooth migration to microservices, and to determine beforehand whether or not this architectural style is a fit for a given context.

3.2.4 Discussion

This review presented several empirical approaches related to the microservices architecture, differing both in goals and methodology. Table 3.1 summarises these results. The rows are roughly arranged by research method.

The *Focus* row of the table indicates the primary subject of inquiry for each study. This classification was derived directly from the explicit wording in the studies or was inferred by us. In addition, the *Impact of Patterns* row refers to works wherein the impact of design patterns was evaluated; the *Software Quality* row refers to studies that concern themselves with how software quality in microservices is seen in industry, and *Migration* refers to studies that aimed at understanding practitioners' insight into the microservitisation process, from the point of view of expected benefits, challenges found and opinions on current tools.

Researchers are mostly interested in evaluating the way software quality is perceived in MSA compared to monolithic systems [30, 31]; this is typically measured through the perception professionals have of the quality attributes most affected by microservices [7].

Many empirical studies are related to the migration process. This reflects a high level of interest in microservitisation compared to other facets of the architectural style [48]. Decomposing a monolith into several services is one of the first major challenges to be addressed by teams looking to switch architectures, so understanding how professionals cross this hurdle is important. The data suggests that an incremental transition from monolith to microservices is the industry standard, due to the high costs required for a build-from-scratch endeavour, in large scale production systems. For this reason, the *Strangler* pattern, which outlines an incremental transition process, is the most cited approach.

Table 3.1: Overview of related empirical studies in microservices.

Approach	Research strategy			Focus		
	Case Study	Controlled Experiment	Survey	Impact of Patterns	Software Quality	Migration
Villamizar et al. [56]	✓				✓	
Balalaie et al. [4]	✓					✓
Haselböck et al. [28]	✓				✓	
Cojocaru et al. [12]	✓		✓		✓	✓
Matias et al. [37]	✓		✓			✓
de Toledo et al. [16]	✓		✓	✓	✓	
Bogner et al. [9]		✓		✓		
Haselböck et al. [29]			✓	✓		
Di Francesco et al. [23]			✓			✓
Ghofrani & Lübke [25]			✓		✓	✓
Haselböck et al. [27]			✓		✓	
Taibi and Lenarduzzi [51]			✓		✓	
Viggiato et al. [55]			✓		✓	
Bogner et al. [7]			✓	✓	✓	✓
Carvalho et al. [11]			✓			✓
Knoche and Hasselbring [32]			✓		✓	
Zhang et al. [61]			✓		✓	
Sousa et al. [49]			✓	✓		
Ayas et al. [2]			✓			✓

Only one controlled experiment studying microservices was conducted, again likely due to the difficulties researchers have emulating a microservices-based system that can be a good candidate for meaningful empirical insight.

There are still comparatively few empirical studies dealing directly with design patterns in microservices, with very little knowledge on how industry professionals perceive the application of a given pattern on the quality attributes of a system. The work by Bogner et al. [9] was one step in this direction, but their controlled experiment regarded only 4 patterns, leaving many unaddressed.

In a field still in its infancy, it is important to have direct contact with professionals more involved with the matter at hand. While surveys via questionnaires are a helpful (and cost-effective) method, long-form and exploratory interviews with experts can provide insights that could otherwise go unmentioned.

3.3 Summary

We have presented a thorough exploration of the research literature related to two broader topics, software qualities in microservices and empirical studies in microservices, with some overlap between the two.

On the former front, many steps have been taken to analyse the architectural properties of microservices, with a strong emphasis on microservitisation, a key process which still holds many challenges for researchers. Many studies concern themselves with defining the boundaries of MSA in terms of taxonomies of concerns, e.g., design, implementation, deployment, etc. With regards to patterns, researchers have extracted and identified many though they rarely attempt evaluations of the consequences these patterns have on quality attributes.

On the latter front, researchers are mostly concerned with gathering professional insights on the several facets of MSA, typically in terms of pains and gains. Additionally, many authors are studying how the migration process is performed in industry, identifying lessons learned and common pitfalls. The impact of patterns is a rare topic of interest for empirical studies. Surveys dominate the approach researchers take, followed by case studies. On the other hand, we identify only one controlled experiment in our review. To the best of our knowledge, there is no empirical study which focuses on design patterns and the way their impacts, both positive and negative, are perceived in industry. As a result, we chose this avenue for this dissertation.

Chapter 4

Problem Statement

In this chapter we will formally describe our research problem. We argue that there is a lack of scientific literature describing clear relationships between microservice design patterns and the software qualities that they may support or hinder as a result of their application in a software system; in one word, their trade-offs. In Section 4.1 we introduce our thesis statement. In Section 4.2 we present our research questions. Finally, in Section 4.3 we describe the research strategy that will direct us towards answers to our questions, and proceed to define the constraints of our research along the terms we deem feasible.

4.1 Thesis statement

The details available on the architectural trade-offs of microservice design patterns are often a result of personal experience, intuition and observation, biased to specific contexts and thus lacking in rigorous empirical validity. With few exceptions, as we have seen in the previous chapter, most design patterns in MSA are not validated against industry practice. Software designers and architects are left to depend on their prior experience and intuition to make crucial, long-lasting, structural decisions which shape the behaviour of the software systems as well as development practices and rationales. While there is no optimal way to design software systems, software architects are more likely to construct systems that better suit their requirements if provided with proven and precise resources.

We seek a deeper 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.

4.2 Research questions

Given our statement, we present the following three research questions (RQ):

- RQ1.** *"What is the rationale for the adoption of patterns in microservices systems?"* — Consider: 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 quality attributes influenced as a result of applying microservice patterns?"* — Consider: what are the trade-offs perceived by professionals? That is, what are the advantages and disadvantages of current microservice patterns?
- RQ3.** *"How are quality attributes measured in microservices?"* — Consider: Do practitioners evaluate their non-functional requirements? If so, how? With which tactics and resources? If not, why?

4.3 Research strategy and scope

We believe that face-to-face interviews are reliable for collecting unstructured data, as they can help establish a more effective communication channel with participants and make it easier to interpret the answers to open-ended questions [51]. Our strategy is, therefore, a qualitative interview study.

A complete and thorough trade-off analysis for all known microservice design patterns would not only be an endeavour more ambitious than the time constraints imposed by this dissertation would allow but, in the face of the current dynamic landscape of microservices research, would quickly become incomplete given the frequent addition of new design patterns to the literature. Similarly, a typical list of system quality attributes is quite large. Suffice it to say that, to carry out a feasible study, we must be more specific about the software qualities that are particularly interesting in the context of MSA. A clearer definition of the breadth of our research is required. This section aims to do exactly that.

The following sections detail our rationale for the selection of the design patterns and quality attributes that we use for analysis in our study, respectively, in Sections 4.3.1 and 4.3.2.

4.3.1 Design patterns

Several pattern languages specific to the microservices architecture have been published in the last few years. These collections of patterns are sourced from individual authors with industry expertise, academics identifying common practices seen in formal and informal literature, or even from researchers proposing their design patterns without specific mentions of industry adoption. The following are some of the published microservice-oriented pattern languages we found:

- *Implementation Patterns for Microservices Architectures*, by Brown and Woolf [10];

- *Microservices Migrations Patterns*, by Balalaie et al. [4];
- *Architectural Patterns for Microservices*, by Taibi et al. [52];
- *A pattern language for scalable microservices-based systems*, by Márquez et al. [36];
- *Microservice Architecture - A Pattern Language for Microservices*, by Richardson [44];
- *Engineering Software for the Cloud: A Pattern Language*, by Sousa [15].

While we initially considered the aforementioned work by Richardson as the source of patterns to use in our study due to its apparent popularity, we were not pleased with the way the book presents – or rather, doesn’t present – its patterns. In it, the patterns are broached as needed following an example the author develops throughout the book, in an effort to illustrate the usage of his design patterns in more concrete circumstances. As a result, the book lacks a glossary of patterns or an otherwise easy way to reference and digest them, with some patterns mentioned in very short off-hand remarks. The author also presents his pattern language in a website¹ dedicated to the book, but it is similarly plagued by problems: while the patterns are presented systematically following a template common to all on the website, we quickly found that many of the included patterns have little to no documentation. As a result, we completely discarded Richardson’s work from consideration.

Since our focus for this study is on industry practice, we thought it wise to also consider the grey literature as a source for other candidate pattern languages to consider. We eventually selected the patterns from the Azure Architecture Centre (AAC), a repository of software architecture documentation maintained by Microsoft, as it seems the more robust and complete documentation. The AAC has an extensive language of patterns termed the *Cloud Design Patterns*², divided into several categories: *Design and Implementation*, *Data Management*, *Messaging*, *Reliability*, *Security*, *Performance Efficiency* and *Operational Excellence*, with most patterns belonging to more than one category.

We selected the 14 patterns found in the *Design and implementation*³ category. The AAC has a related article titled "*Design patterns for microservices*"⁴ with 9 patterns listed in total, 8 of which found in the aforementioned pattern set, indicating that the authors see these as fundamental design patterns for the microservice architecture.

This documentation is available in a public GitHub repository⁵. Since it is an actively maintained repository, we analysed the documentation available as of commit with hash `eff408a`.

This section presents a summary of each of the patterns that will be analysed in this study. Each of the following subsections briefly describes an understanding of the patterns as presented in the AAC, chiefly the applicability context, the problem of and the solution reflected by each pattern. Additional notes about particular patterns are also produced when deemed necessary.

¹<https://microservices.io/>

²<https://docs.microsoft.com/en-us/azure/architecture/patterns/index-patterns>

³<https://docs.microsoft.com/en-us/azure/architecture/patterns/category/design-implementation>

⁴<https://docs.microsoft.com/en-us/azure/architecture/microservices/design/patterns>

⁵<https://github.com/MicrosoftDocs/architecture-center>

STRANGLER FIG

- *Context*: As systems age, the development tools, hosting technology, and even system architectures they were built on can become increasingly obsolete. As new features and functionality are added, the complexity of these applications can increase dramatically, making them harder to maintain or add new features to.
- *Problem*: How to replace a legacy system's architecture?
- *Solution*: Incrementally replace specific pieces of functionality with new applications and services. Create a façade that intercepts requests going to the backend legacy system. The façade routes these requests either to the legacy application or the new services. Existing features can be migrated to the new system gradually, and consumers can continue using the same interface, unaware that any migration has taken place.
- *Notes*:
 - * More commonly known as just STRANGLER, this pattern uses a *façade* abstraction to illustrate its purpose, inspired by the FAÇADE pattern found in the work by the Gang of Four [24].
 - * This pattern is also found in the *Operational Excellence* patterns of the AAC.

ANTI-CORRUPTION LAYER

- *Context*: Most applications rely on other systems for some data or functionality. For example, when a legacy application is migrated to a modern system, it may still need existing legacy resources. New features must be able to call the legacy system. This is especially true of gradual migrations, where different features of a larger application are moved to a modern system over time.
- *Problem*: How to maintain access between new and legacy systems without forcing the new system to adhere to the legacy system's APIs or other semantics?
- *Solution*: Implement a façade or adapter layer between different subsystems that don't share the same semantics. This layer translates requests that one subsystem makes to the other subsystem. Use this pattern to ensure that an application's design is not limited by dependencies on outside subsystems.
- *Notes*:
 - * The documentation explicitly traces the origins of this pattern to the work by Evans [21], which summarises its version of the pattern as follows:

Create an isolating layer to provide clients with functionality in terms of their own domain model. The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally, the layer translates in both directions as necessary between the two models.

However, the AAC version of the pattern sees the Anti-Corruption Layer as a physical, independent service, to be deployed and maintained alongside all other owned processes, a detail not broached by Evans.

- * This pattern is also found in the *Operational Excellence* patterns of the AAC.

SIDECAR

- *Context*: Applications and services often require related functionality, such as monitoring, logging, configuration, and networking services.
- *Problem*: How to offload common peripheral and shared processing tasks such as logging and configuration?
- *Solution*: Deploy components of an application into a separate process or container to provide isolation and encapsulation, known as a sidecar service. By co-locating a cohesive set of tasks with the primary application, inside the same process or container, we can provide a homogeneous interface for platform services across languages.
- *Notes*:
 - * This pattern is also found in the *Operational Excellence* patterns of the AAC.

AMBASSADOR

- *Context*: Resilient cloud-based applications require features such as circuit breaking, routing, metering and monitoring, and the ability to make network-related configuration updates. It may be difficult or impossible to update legacy applications or existing code libraries to add these features, because the code is no longer maintained or can't be easily modified by the development team.
- *Problem*: How to offload common client connectivity tasks such as monitoring, routing, security, and resiliency patterns?
- *Solution*: Put client frameworks and libraries into an external process that acts as a proxy between your application and external services. Deploy the proxy on the same host environment as your application to allow control over routing, resiliency, security features, and to avoid any host-related access restrictions.
- *Notes*:
 - * This pattern can be thought of an extension of the SIDECAR pattern if the ambassador service is deployed as a sidecar to accompany the life cycle of a consuming application or service.
 - * This pattern is also found in the *Operational Excellence* patterns of the AAC.

COMPUTE RESOURCE CONSOLIDATION

- *Context*: A cloud application often implements a variety of operations. In some solutions it makes sense to follow the design principle of separation of concerns initially,

and divide these operations into separate computational units that are hosted and deployed individually. However, although this strategy can help simplify the logical design of the solution, deploying a large number of computational units as part of the same application can increase run-time hosting costs and make management of the system more complex.

- *Problem:* How to increase compute resource utilisation and reduce the costs and management overhead associated with performing compute processing in cloud-hosted applications?
- *Solution:* Consolidate multiple tasks or operations into a single computational unit. Tasks can be grouped according to criteria based on the features provided by the environment and the costs associated with these features. A common approach is to look for tasks that have a similar profile concerning their scalability, lifetime, and processing requirements. Grouping these together allows them to scale as a unit.

COMMAND QUERY RESPONSIBILITY SEGREGATION

- *Context:* Traditionally, the same data model is used to query and update a database. This is simple and works well for basic CRUD operations. In more complex applications, however, this approach can become unwieldy. Read and write workloads are often asymmetrical, with very different performance and scale requirements.
- *Problem:* How to optimise read and write workloads in a busy database?
- *Solution:* Separate reads and writes into different models, using commands to update data, and queries to read data. Commands should be task-based, rather than data-oriented. ("Book hotel room", not "Set `ReservationStatus` to `Reserved`"). Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously. Queries should never modify the database. A query should return a data transfer object that does not encapsulate any domain knowledge.
- *Notes:*
 - * This pattern is often abbreviated as CQRS.
 - * Though it is not a strict requirement, it is also possible to have separate databases: a database only for reading, and a database only for writing.
 - * This pattern is also found in the *Data Management* and *Performance Efficiency* patterns of the AAC.

EXTERNAL CONFIGURATION STORE

- *Context:* The majority of application run-time environments include configuration information held in files deployed with the application. In some cases, it's possible to edit these files to change the application behaviour after it's been deployed. However, changes to the configuration require the application be redeployed, often resulting in unacceptable downtime and other administrative overhead.

- *Problem*: How to simplify managing configuration data across multiple running instances of services?
- *Solution*: Store the configuration information in external storage, and provide an interface that can be used to quickly and efficiently read and update configuration settings. The type of external store depends on the hosting and run-time environment of the application. In a cloud-hosted scenario it's typically a cloud-based storage service, but could be a hosted database or other system.
- *Notes*:
 - * This pattern is also found in the *Operational Excellence* patterns of the AAC.

GATEWAY ROUTING

- *Context*: When a client needs to consume multiple services, setting up a separate endpoint for each service and having the client manage each endpoint can be challenging. For example, an e-commerce application might provide services such as search, reviews, cart, checkout, and order history. Each service has a different API that the client must interact with, and the client must know about each endpoint in order to connect to the services. If an API changes, the client must be updated as well. If you refactor a service into two or more separate services, the code must change in both the service and the client.
- *Problem*: How to supply a single client interface to access all current services?
- *Solution*: Place a gateway in front of a set of applications, services, or deployments, serving as the single entry point for all clients.
- *Notes*:
 - * This pattern is more commonly known as just API GATEWAY, though the AAC limits this pattern to just the functionality of request routing. Added capabilities of a gateway are encompassed in the three patterns that follow.
 - * This pattern is also found in the *Operational Excellence* patterns of the AAC.

GATEWAY AGGREGATION

- *Context*: To perform a single task, a client may have to make multiple calls to various backend services. An application that relies on many services to perform a task must expend resources on each request. When any new feature or service is added to the application, additional requests are needed, further increasing resource requirements and network calls.
- *Problem*: How to reduce the number of calls between the client and the server composed of smaller services?
- *Solution*: Use a gateway to aggregate multiple individual requests into a single request. This pattern is useful when a client must make multiple calls to different backend systems to perform an operation.

- *Notes:*

- * This pattern is also found in the *Operational Excellence* patterns of the AAC.

GATEWAY OFFLOADING

- *Context:* Some features are commonly used across multiple services, and these features require configuration, management, and maintenance. A shared or specialized service that is distributed with every application deployment increases the administrative overhead and increases the likelihood of deployment error. Any updates to a shared feature must be deployed across all services that share that feature.
- *Problem:* How to consolidate cross-cutting concerns shared by several backend services?
- *Solution:* Offload features into a gateway, such as certificate management, authentication, SSL termination, monitoring, protocol translation, or throttling, etc..
- *Notes:*
 - * This pattern is also found in the *Operational Excellence* patterns of the AAC.

BACKENDS FOR FRONTENDS

- *Context:* An application may initially be targeted at a desktop web UI. Typically, a backend service is developed in parallel that provides the features needed for that UI. As the application's user base grows, a mobile application is developed that must interact with the same backend. The backend service becomes a general-purpose backend, serving the requirements of both the desktop and mobile interfaces.
- *Problem:* How to optimise our back-end for clients with different requirements and concerns?
- *Solution:* Create one backend per user interface. Fine-tune the behaviour and performance of each backend to best match the needs of the frontend environment, without worrying about affecting other frontend experiences.
- *Notes:*
 - * This pattern is often abbreviated as BFF.
 - * This pattern is also found in the *Operational Excellence* patterns of the AAC.

LEADER ELECTION

- *Context:* A typical cloud application has many tasks acting in a coordinated manner. These tasks could all be instances running the same code and requiring access to the same resources, or they might be working together in parallel to perform the individual parts of a complex calculation.
- *Problem:* How to coordinate the actions performed by a collection of instances in a distributed application?

- *Solution*: Electing one instance as the leader that assumes responsibility for managing the others. The system must provide a robust mechanism for selecting the leader. This method has to cope with events such as network outages or process failures. In many solutions, the subordinate task instances monitor the leader through some type of heartbeat method, or by polling. If the designated leader terminates unexpectedly, or a network failure makes the leader unavailable to the subordinate task instances, it's necessary for them to elect a new leader.
- *Notes*:
 - * This pattern is commonly found in distributed computing, and has seen wide use for decades.
 - * This pattern is also found in the *Reliability* patterns of the AAC.

PIPES AND FILTERS

- *Context*: An application is required to perform a variety of tasks of varying complexity on the information that it processes. However, the processing tasks performed by each module, or the deployment requirements for each task, could change as business requirements are updated. Some tasks might be compute intensive and could benefit from running on powerful hardware, while others might not require such expensive resources. Also, additional processing might be required in the future, or the order in which the tasks performed by the processing could change.
- *Problem*: How to decompose a task that performs complex processing into a series of separate elements that can be reused?
- *Solution*: Break down the processing required for each stream into a set of separate components (or filters), each performing a single task. By standardising the format of the data that each component receives and sends, these filters can be combined together into a pipeline.
- *Notes*:
 - * This pattern is also found in the *Messaging* patterns of the AAC.

STATIC CONTENT HOSTING

- *Context*: Web applications typically include some elements of static content.
- *Problem*: How to serve requests for static content without consuming processing cycles that could be put to better use?
- *Solution*: Deploy static content to a cloud-based storage service that can deliver them directly to the client.
- *Notes*:
 - * In effect, this pattern reflects the usage of a Content Delivery Network, or CDN.
 - * This pattern is also found in the *Data Management* and *Performance Efficiency* patterns of the AAC.

4.3.2 Quality attributes

From our literature review in the previous chapter, we can make an informed selection of valuable quality attributes for industry experts and thus narrow down which ones to use in our analysis.

We initially strongly favoured the work by Cojocaru et al [12] regarding what they term attributes assessing the quality of microservices, as this seemed in line with our research goals. We considered their proposed minimum set of quality attributes for our analysis – the set of the following attributes: *granularity*, *cohesion*, *coupling*, *scalability*, *response time*, *security*, *health management*, *execution cost*, and *reusability*. After some deliberation, however, we concluded a mismatch with our goals, primarily on the basis that these attributes are located on different planes of abstraction – for example, cohesion and coupling are domain concerns, whereas security is more commonly an architectural concern. Given that the authors sought a ‘per service’ analysis of highly relevant attributes (as opposed to a ‘per system’, i.e., global architecture analysis), this makes sense. However, since our research is chiefly concerned with architectural trade-offs, encompassing relationships between services and their operation, we felt this was not the best way forward.

We then turned to the work by Li et al. [34], which dealt with questions of assessing quality attributes in microservices matching our desired level of architectural perspective. The authors present a set of six quality attributes and reinforce their definition rigorously by resorting to the conceptions recommended by the ISO. We were pleased with this work and decided it was a worthwhile foundation to draw inspiration from regarding a practical set of QAs that can be analysed within the limits of this dissertation. For our part, we extend the set to include a seventh element, *maintainability*.

The list that follows contains the set of quality attributes of our study, a brief exploration of each, and a description of their relevance to MSA.

Scalability. A measure of a system’s ability to handle a varying amount of requests [6]. MSA is generally adopted to maximise aspects of scalability, given its proximity to cloud-native technologies and its inherent inclination towards simplifying 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 [6]. As a distributed architecture, the latency in network requests involved in inter-service communication is an obvious source of performance hindrance. Battle-tested 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 [6]. As such, we can think of *availability* as interchangeable a commonly referred QA, *reliability* [34]. While we can minimise dependence between service with good design, rarely is a service truly self-sufficient or not depended on by others. Downtime in one faulty service can quite easily cascade into severe

limitations, bleeding into otherwise well-behaved services. Several tactics 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 [6]. Any distributed system, conceived as a microservice system or otherwise, exhibits a high level of dynamic structure and behaviour, representing a added challenge for monitoring. In MSA, monitoring should encompass data such as infrastructure information (VM or container, etc.) timestamps, identifiers of service instance, requested methods, etc. [34]. 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 unauthorised access while enabling access to authorised users and systems [6]. The distribution of application logic into different processes results in complex network interaction models between services [34]. 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 [6]. Due to the complex relationship between services, *testability* is important to pay attention to so as to detect and minimise impacts on other qualities like *performance*, *security* or *availability* [34].

Maintainability. A measure of a system’s ability to support the development staff in applying functional changes to the system [6]. 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. In essence, *maintainability* concerns the cost of change. Poor design can easily generate a plethora of undesirable circumstances which can wildly inflate the cost of change. The presence of coupling between different service implies scenarios where, for instance, making changes to one service cannot be realised without orchestrating similar changes in its ‘spouses’, a sign of poor maintainability.

4.4 Summary

This chapter defines the questions and constraints that will drive our research. As we have seen in the previous chapter, there is still little work done on the topic of surveying professionals about design patterns, particularly within MSA.

With this work, we hope to validate a pattern language by direct confrontation with industry practice. Solidifying the grounds of a maturing software architecture helps software engineers and architecture in general have a firmer and safer understanding of the tools they use day to day, and also any potential alternatives they might be interested. By exploring professional perceptions of trade-offs inherent to design patterns in MSA, we hope to help pave one step forward in this direction.

Chapter 5

Empirical Study

We conducted a survey among experienced developers, collecting their rationales on design pattern usage, their perceptions of architectural trade-offs in patterns, and how they go about measuring software qualities, all within the context of microservice systems.

We collected information in interviews with a semi-structured approach, taking cues from grounded theory for iterating over and refining our approach as well as for data analysis, using constant comparison when coding to generate subcategories and properties. However, we classify our study as a semi-structured interview study and not a grounded theory study, given that we did not make use of many of its formally recommended techniques such as *memoing* or sorting.

To restate the contours of our study, we seek an understanding of the architectural trade-offs of the patterns listed in Section 4.3.1, sourced from the Azure Architecture Centre, AAC for short. We express these trade-offs through the quality attributes they affect, either positively or negatively. For our purposes, the QAs we are interested in are listed in Section 4.3.2.

In this chapter we describe the work involved in conducting this study, from planning to execution to analysis to interpretation. In Section 5.1 we describe the interview study’s design in detail. In Section 5.2 we describe the actual experience of carrying out the study, thoroughly characterising the study participants, the process of the interviews themselves, and how the study was refined over its course. Finally, in Section 5.3 we describe our process for data analysis and extraction and then present our results.

5.1 Design

In this section we describe the process of conceiving our interview study. We detail all the procedures involved in preparing us for the realisation of this study and producing the research instruments for later application. We support our design decisions by reviewing literature related to interview studies.

5.1.1 Preliminary analysis

As a preliminary exercise, we applied the 5-step method for architectural trade-off analysis based on patterns by Rosa et al [14]. We read through the documentation of each pattern in the AAC, identifying quality attributes either by explicit mention in the text or by mapping properties of the system that are said to be affected by the system to a corresponding QA, and subsequently assigning it a positive (▲) or negative (▼) value. For example, take the following passage from the GATEWAY OFFLOADING documentation:

This pattern can simplify application development by moving shared service functionality, such as the use of SSL certificates, from other parts of the application into the gateway.

We map this sentence to the *maintainability* QA, with a positive value. In some cases, however, this is not so straightforward. Take the following passage from the GATEWAY ROUTING documentation:

The gateway service may introduce a single point of failure. Ensure it is properly designed to meet your availability requirements.

We map this sentence to the *availability* QA but we feel that assigning it either a positive or negative value is too simplistic a judgement. In such cases, we opt for a mixed (●) value, to indicate that this requires particular attention to the context at hand and more careful consideration. Table 5.1 contains the results of our trade-off analysis.

The purpose of this exercise was twofold. Firstly, we developed a thorough understanding of each pattern and a first impression of their architectural trade-offs. Secondly, and more importantly, we obtained a baseline trade-off to compare against our interviews. That is, with these findings we could expect our participants to highlight some of these QAs in response to our questions and, importantly, we could immediately bring to the conversation QAs that they may not have mentioned, for whatever reason, but which we knew to be pertinent. For example, if we had identified a positive impact on *availability* for a design pattern we're asking about at a given moment and the participant has not mentioned *availability* directly or indirectly in their answer, we ask a question along the lines of "*do you think this pattern had an impact on availability?*"

The accuracy of our value judgements is up for debate, although we take it to be a good assessment. In any case, we are more interested in extracting the quality attributes that are mentioned or alluded to in the source text rather than whether or not we can determine a positive or negative impact.

5.1.2 Population sampling

Regarding sampling, we searched for academic literature regarding an optimal number of interviews to include in order to obtain a sample size representative of a wider population and thus statistically significant. Both Baker & Edwards [3] and Guest & Johnson [26] sought to find

Table 5.1: Preliminary design pattern trade-off analysis.

Pattern	Scalability	Performance	Availability	Maintainability	Monitorability	Security	Testability
STRANGLER FIG	▲			▲			
ANTI-CORRUPTION LAYER	▲						
SIDECAR		▼		▼	▲		
AMBASSADOR			▲		▲	▲	
BACKENDS FOR FRONTENDS		▲		▲			
COMPUTE RESOURCE CONSOLIDATION		▲	▼	▼		▼	▼
COMMAND QUERY RESPONSIBILITY SEGREGATION	▲	▲		▼		▲	
EXTERNAL CONFIGURATION STORE				▲			
GATEWAY ROUTING		●	●	▲			
GATEWAY AGGREGATION	▲	▲					
GATEWAY OFFLOADING	▲		▼	▲	▲	▲	
LEADER ELECTION	▼	●	▼				
PIPES AND FILTERS	▲	▲		▼			
STATIC CONTENT HOSTING	▲	▲		▼			

answers to this question and found that, while there is no clear guideline for determining a non-probabilistic sample size, it typically relies on the concept of *saturation*, i.e., a point at which no new information or themes are observed in the data. Their results show that twelve interviews is an adequate threshold for saturation. However, this must be balanced with other constraints, chiefly the time we have available to perform the study as well as the number of study candidates we can find.

In any case, the target population for this survey must conform to the following acceptance criteria: (a) industrial practitioners (not students) who (b) have worked and/or are currently working on microservice systems. Since design patterns are more so the speciality of software architects, our study participants should preferably be familiar with as much of the system architecture as possible.

5.1.2.1 Web-based sign-up form

We designed a web-based sign-up form for interested participant to fill out. The questionnaire is composed of 6 closed-ended mandatory questions. This form can be found in Appendix A.1. The form follows a simple structure:

- An introductory statement to ease readers into the purpose of the study, as well as the planned methodology – an interview with an estimated duration.
- A confidentiality statement to make clear to interested participants that any sensitive data regarding the specifics of company products will not be published.

- A text field for the participant's name.
- A text field for the participant's e-mail address.
- A check box for the participant's company role, featuring the options 'Developer', 'Architect', 'Operations'; 'QA', 'CTO', and an 'Other' option for respondents to specify different answers.
- A check box for the participant's years of experience with professional software development, in the ranges: less than 5 years; 5 to 9 years; 10 to 14 years; 15 to 19 years; 20 years or more.
- A check box for the participant's years of experience with microservice systems, in the ranges: less than 2 years; 2 to 4 years; 5 years or more.
- A check box for the largest number of active monthly users regularly served by the microservice systems the participant had previously worked with, in the ranges: less than 100; between 100 and 1K; between 1K and 10K; between 10K and 100K, between 100K and 1M; more than 1M.

The form serves as a preliminary step for candidate evaluation, i.e. to determine the suitability of each study candidate, as well as a way to get their preferred contact method so we can schedule an interview in the future. We took great care to concentrate a minimal number of meaningful inputs in this form so as to not drive away potential participants who might otherwise feel overwhelmed with a litany of questions.

5.1.3 Interview structure

After preparatory remarks – greetings, personal presentation, study presentation, obtaining authorisation for recording, etc. – all interviews were planned to follow a three-act structure described as follows:

Act 1. Personal experience with microservices — We ask the participant to elaborate freely on their professional experience with microservice. We ask the participant to consider details like the number of services that were in development and operation in these projects, the number of LOC that composed the system, the inter-service mechanisms used, and the deployment strategies used.

The goal of this act is to get a better sense of the microservice systems the participant has worked with, so as to help contextualise and interpret their answers in the rest of the interview.

Act 2. Design pattern trade-offs — We present the participant with the design patterns we selected in Section 4.3.1, using a helper document, found in Appendix A.3. For clarity, the slides are ordered in the same sequence as in Section 4.3.1, with only one caveat: we

changed the title of the STRANGLER FIG to just STRANGLER, as this is its most common designation. The figures included in each slide to help illustrate the patterns 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 localised level (e.g. per service) but on a generalised level.

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

Act 3. Microservice quality metrics — We present the participant with the QAs we selected in Section 4.3.2, using the same helper document from the previous act, specifically the last slide in the document. 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 act is to determine the extent to which software qualities are perceived in industry, specifically the metrics used to evaluate them. While we don't necessarily connect this act to the previous one, intuition suggests that companies that put greater care into measuring their non-functional requirements will have more solid impressions of the trade-offs of design patterns.

To help us keep the interview on track, we produced an interview guide. This document can be found in Appendix A.2. This document is meant for the interviewer only, containing an array of questions that we expected to ask in the interviews, divided by each act as outlined in the aforementioned structure. After wrapping up each interview, we also ask participants to give us their feedback on the interview experience itself.

One of the constraints we were most concerned with when designing the interview was the duration. Given that we hope to draw as many participants as possible, we do not want to drive potential candidates away with an overly long estimate. Since we have defined a set of 14 patterns, a conversation of one hour strictly regarding patterns – i.e., if the interview was confined to Act 2 – would, in the best case scenario, allow for roughly four and a half minutes for each pattern. However, because some patterns are closely related in scope, e.g. patterns related to API gateways or migrations or service deployment, we expected 60 minutes to be a sensible period of time for gathering worthwhile findings. All things considered, we estimated the duration of the interviews to be around 75 minutes.

From the beginning we approached this design with a very flexible mindset. Inspired partly by grounded theory, we expected that possible changes in the direction and structure of the interviews would reveal themselves throughout the data collection process. As such, with the goals of this

research fixed in place, we foresaw and expected to make iterations over the study’s design during execution.

5.1.4 Quality assurance

To ensure quality and completeness of our work, we used the free online Empirical Standards Checklist tool¹, developed by the ACM Special Interest Group on Software Engineering, which provides a set of criteria that empirical studies should abide by for maximising scientific rigour, directed for the field of software engineering. For qualitative surveys, the tool defines three categories: essential, desirable and extraordinary. We reproduce the two former checklists in Tables 5.2 and 5.3, respectively, along with our assessment of this study’s fulfilment of each item. We omit the latter because our study does not conform to any extraordinary criteria.

5.2 Execution

All interviews were spoken in Portuguese and conducted online via Google Meet using a Google Workspace edition that enables us to record meeting video, provided by the University of Porto. While we would have preferred face-to-face interviews, we were met with this limitation due to the ongoing COVID-19 pandemic.

A general overview of the procedure of our study is illustrated in Figure 5.1. We present the three major steps linearly and chronologically, but it is important to note that, as is mentioned later in Section 5.3, the data analysis process was performed in lockstep with data collection. As soon as one interview was finished and transcribed, we glanced at the new data found and contrasted with what we had up until that point in order to discover possible new avenues for orienting our research and better refinements to our study design.

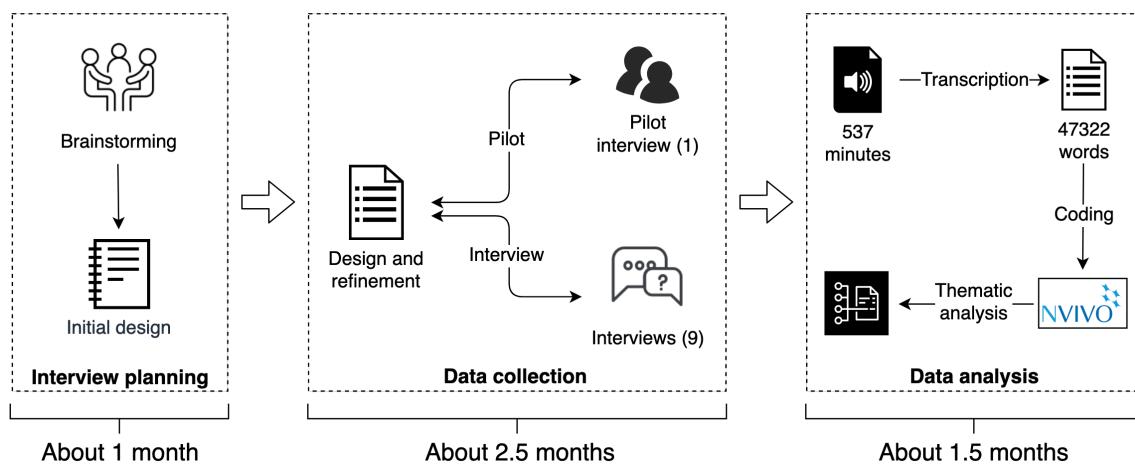


Figure 5.1: Interview study procedure.

¹<https://acmsigsoft.github.io/EmpiricalStandards/tools/>

Table 5.2: Quality assurance checklist: essential items.

Description	Achieved	Note
Clearly answers the research question(s)	✓	Section 6.2
Presents clear chain of evidence from interviewee quotations to findings	✓	Chapter 5
Discusses implications of the results	✓	Section 5.3
Supports main claims or conclusions with explicit evidence (data/observations) or arguments	✓	Section 5.3
Results directly address research questions	✓	Section 5.3
Presents results	✓	Section 5.3
Describes in detail how the data were analysed	✓	Section 5.3
Describes interviewees (e.g. demographics, work roles)	✓	Section 5.2.2
Explains how interviewees were selected	✓	Section 5.2.2
Balances the study's anticipated benefits with its potential risks or harms, minimising risk or harm wherever possible	✓	Section 5.3.5
Contributes in some way to the collective body of knowledge	✓	Chapter 5
Discusses the study's limitations and threats to validity	✓	Section 5.3.5
Describes in detail what, where, when and how data were collected	✓	Section 5.2
Defines jargon, acronyms and key concepts	✓	Chapter 2
States a purpose, problem, objective, or research explains why the problem, objective, or research question is important	✓	Chapter 4
Provides evidence of saturation; explains how saturation was achieved	✓	Section 5.3.5
Methodology is appropriate (not necessarily optimal) for stated purpose or questions	✓	Section 4.3
Language is not misleading; any grammatical problems do not substantially hinder understanding	✓	Throughout
Visualisations/graphs are not misleading	✓	Throughout
Discusses and validates assumptions of any statistical tests used	✗	Not applicable

Table 5.3: Quality assurance checklist: desirable items.

Description	Achieved	Note
Provides an auto-reflection or assessment of the authors' own work	✓	Section 6
Concise, precise, well organised and easy-to-read presentation	✓	Throughout
Evaluates an <i>a priori</i> theory using deductive coding with an <i>a priori</i> coding scheme based on the prior theory	✓	Chapter 5
Uses direct quotations extensively to support key points	✓	Section 5.3
Visualisations advance the paper's arguments or contribution	✓	Section 5.3
Reasonable attempts to investigate or mitigate limitations	✓	Section 5.3.5
Provides plausibly useful interpretations or recommendations for practice, education or research	✓	Chapter 6
Clarifies the roles and responsibilities of the researchers	✓	Chapter 5
Compares results with prior theory or related research	✓	Section 5.3
Findings plausibly transferable to different contexts	✓	Section 5.3.5
Reflects on how researchers' biases may have affected their analysis	✓	Section 5.3.5
Clearly describes relationship between contribution(s) and related work	✓	Chapter 6
Summarises and synthesises a reasonable selection of related work	✓	Chapter 3
Discusses study's realism, assumptions and sensitivity of the results to its realism/assumptions	✓	Section 5.3.5
Openly shares data and materials to the extent possible within practical and ethical limits	✓	Appendix A
Provides supplemental materials including interview guide(s), coding schemes, coding examples, decision rules, or extended chain-of-evidence table(s)	✓	Appendix A
Appropriate statistical power or saturation	✓	Section 5.3.5
States epistemological stance	✗	
Uses multiple raters for any subjective judgements	✗	
Includes highly diverse participants	✗	Section 5.3.5
Validates results using member checking, dialogical interviewing, feedback from non-participant practitioners or re-search audits of coding by advisers or other researchers	✗	

This section describes the process of data collection for our research which, fundamentally, was achieved through the interviews previously designed.

5.2.1 Pilot interview

Given the design of the interview and the concerns described in Section 5.1.3, we opted for a pilot interview. The intent behind this interview was not for data collection as such but as an aid to the design of our later research. It served as an initial attempt to test the suitability of our design, both in terms of relevance of questions, but chiefly to get a feel for the duration of the interview.

Thus, we conducted a pilot interview with a fellow Masters student involved in research regarding design patterns in MSA, which had a duration of 43 minutes. This is considerably less than the estimate we mentioned previously (75 minutes), but a few notes should be considered. Firstly, the participant could not provide us with an account of their prior professional experience with microservice development (corresponding to Act 1 of our interview), because he had had none to speak of. Secondly, and similarly, the participant had not had a professional experience from which to rely on to respond to the questions regarding metrics of software qualities (corresponding to Act 3), since he was an academic and not a practitioner. Nonetheless, the section of the interview dedicated to architectural trade-offs of patterns (corresponding to Act 2) was eventful and a worthwhile emulation of an interview with a professional.

In effect, the pilot interview was a serviceable small-scale implementation of our research plan. We concluded that 75 minutes still served as a good estimate for an upper-bound duration of our interviews, and the confidence in the design of our study was strengthened.

5.2.2 Interviews

Due to the oft-mentioned challenges of recruiting a representative sample of study participants in software engineering, we made use of purposive sampling [5] and our personal network, i.e., we reached out to colleagues or associates in the industry who either served as participants or forwarded us to potential participants. We also attempted a heterogeneity sampling, by contacting candidates from a breadth of different industry domains and performing different roles.

We reached out to 16 contacts, each working in a different company, through several means (telephone, email, direct messaging) and got back 11 informal responses claiming an interest in participating in our project, 3 of which forwarded us to a different person in the company. The remaining 5 contacts did not respond.

Two of the ten people who had informally told us they were interested in participating dropped out of the study, i.e. they did not submit a response to the sign-up form described in 5.1.2.1, even after being reminded several times. The remaining 8 participants proceeded with the study and hence compose our body of interviewees – our sample.

All things considered, we have interviewed, over a period of two months, 9 professionals, all of which male, from 9 different companies; 8 Portuguese companies and 1 French company with a development office in Portugal. An overview of the participants is found in Table 5.4. We

omit the names to preserve interviewee privacy and protect potential commercial interests of our participants' employers. Instead we replace the names with an identifier i_n , where n is a unique number from 1 to 9. Worth noting that n is not an indicator of chronology; i_3 was not necessarily interviewed after i_2 , etc..

The interviewees work in widely different industries, with only 2 industry domains in common among all, two participants work in the Artificial Intelligence industry and two others in the Finance industry. All participants have at least 5 years of experience with software development. A third of our participants (3/9) reported 5 or more years of experience with microservices, the remaining two thirds (6/9) have between 2 to 4 years of experience with microservices. Over half of our participants (5/9) are in roles highly familiar with the system architecture: 2/8 participants are CTOs in their respective companies, another participant is an Architect, another is a Head of Engineering, and another is a Scrum Master. The remaining participants (3/9) are in more localised roles: one is a Developer, another is in Operations, and another is in QA.

Table 5.4: Participants of interview study. Experience is reported in years, values in brackets are years of experience with microservices.

Interviewee	Role	Experience	Industry
i_1	Architect	10 to 14 (5 or more)	Artificial Intelligence
i_2	CTO	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 computing
i_6	CTO	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

Prior to recording each interview we obtained affirmative consent from the interviewee. The transcription of all interviews was done manually, using the free *oTranscribe* tool² to aid in the process, as it provides intuitive keyboard shortcuts for quickly navigating the audio and inserting timestamps.

An overview of the interviews in terms of recording duration and words transcribed is found in Table 5.5. The interviews ranged from 42 minutes to 79 minutes. A total of 47322 words were transcribed from 537 minutes and 40 seconds of recorded video. The duration is presented with respect to the length of the video recording; in reality, a few minutes of greetings and introduction transpired before starting the recordings. Additionally, the count of words transcribed is not exact for all interviews. In the early interviews we painstakingly transcribed every intelligible detail, including chit-chat and interjections which were ultimately irrelevant. From our third interview onward, we omitted those inconsequential remarks and, in their place, inserted a bracketed ellipses, that is, a literal [. . .].

²<https://otranscribe.com/>

Table 5.5: Statistics on interviews transcribed.

Interviewee	Words	Duration
i_1	6169	56 minutes, 12 seconds
i_2	5911	79 minutes, 19 seconds
i_3	3530	42 minutes, 42 seconds
i_4	4889	65 minutes, 39 seconds
i_5	5939	61 minutes, 32 seconds
i_6	5763	62 minutes, 33 seconds
i_7	6070	60 minutes, 53 seconds
i_8	4932	58 minutes, 46 seconds
i_9	4118	50 minutes, 20 seconds

5.2.3 Iterations

Throughout the interviewing process several minor and major changes were made after our retrospectives, wherein we considered the data we had obtained at that point, trends we were noticing in the participants' responses, possible gaps in the data we were looking for but were finding trouble obtaining, and feedback given directly by interviewees. The list that follows showcases, in chronological order, the most substantial alterations made to the interview structure:

5th interview. Up until this point, the interview helper document featured 15 slides, 14 for the design patterns and 1 with our set of QAs. We had noticed that participants were not expressing trade-offs in our desired terms. Instead of proclaiming impacts on quality attributes explicitly, participants instead elaborated on their answers in terms of gains and pains on development and organisational levels. From their remarks we followed up with questions hoping to draw out pertinent quality attributes, using phrasing like "*so you think this impacted mostly your system maintainability?*", but this was frequent and awkward, so we looked to solve it.

Our solution was to show, at the start of Act 2, the slide with our QAs to each participant before showing the patterns. The intent was to help define the vocabulary that we use in software engineering to refer to trade-offs, i.e. in the terms of software quality attributes. In the interviews that followed, however, we made it clear that we did not want participants to feel pressured to give their answers in these terms, instead noting it to be something to keep in the back of their head.

8th interview. Up until this point, when presenting the design patterns in Act 2 we followed up with a question like "*have you ever used this pattern?*", usually obtaining a simple affirmative or negative answer which determined how to proceed. Often, however, participants would volunteer information like "*I've never heard of this pattern before*" or "*I have but I didn't know it had a name*", leading us to believe it important to determine not only whether interviewees had applied (or not) the patterns, but also if they had known (or not) the patterns.

Our solution was to ask "are you familiar with this term?" after presenting each pattern, i.e. we prepended our existing pattern-related questions with this new question. Unfortunately, this change came late into the study, seeing only effect in the final interview..

5.3 Findings

The data analysis in this study was performed with a combination of manual and computer-assisted techniques: we used the NVivo qualitative data analysis software solution for coding the large amount of text data we obtained but we manually identified themes and other findings, since we found NVivo's automatic theme identification feature quite lacklustre, likely due to inadequate support for the Portuguese language.

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. By envisioning a set of expected topics that we had foreseen a priori from our research questions, we developed the top-level codes *Preliminary*, *Design Patterns*, *Measuring QAs*, and *Miscellaneous* and a code for every pattern and QA. These form the structural areas of our analysis. Through later axial coding, a higher level of coding [17], 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 re-conceptualised 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 visible in Figure 5.2.

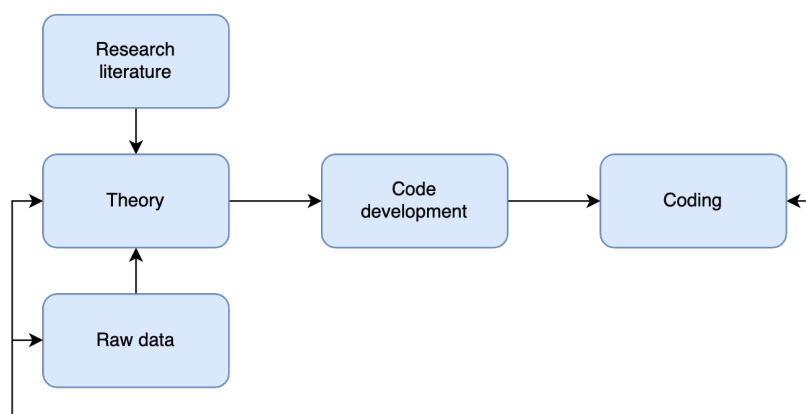


Figure 5.2: Coding process.

The following sections show our results. We map the first three sections to our research questions, in Sections 5.3.1, 5.3.2 and 5.3.3, respectively, wherein we thoroughly explore and discuss

the implications and limitations of our findings. In Section 5.3.4, we present miscellaneous comments derived from interviewees which we found to be of interest, though not necessarily related to the goals of this work. Finally, in Section 5.3.5 we discuss the threats to validity of our results.

We present our findings mostly in table form. These tables contain a row "#" which reflects the number of unique respondents that mentioned a particular theme. Additionally, to reinforce our interpretations of the findings, we present quotations of our participants when relevant, faithfully translated to English.

5.3.1 Rationale for pattern adoption

Table 5.6 presents an overview of the adoption of design patterns reported by our participants. Additionally, Figures 5.3 and 5.4 show the count of patterns adopted by each participant and the number of participants that adopted each pattern, respectively.

It is worth noting that the information on prior awareness of the patterns (expressed most obviously through the \bullet^\times or \circ^\times values) is not necessarily complete: as stated in Section 5.2.3, we did not explicitly ask interviewees if they were familiar with the names given to the patterns until the final interview (i_9). As such, we were only able to assign those values in instances when participants directly mentioned knowing or not knowing about this or that pattern.

Table 5.6: Reported patterns adopted by participants. \bullet is used when the pattern was said to have been applied by the participant; \circ is used when the pattern was said to not have been applied by the participant; \bullet^\times is used when the pattern was said to have been applied but the participant was not familiar with the given pattern's name; \circ^\times when the pattern was said to have neither applied nor even known to the participant and \odot is used when 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.

Pattern	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9
STRANGLER	\bullet	\bullet	\bullet	\bullet	\bullet	\bullet	\bullet^\times	\bullet^\times	\bullet^\times
ANTI-CORRUPTION LAYER	\circ	\bullet	\circ^\times	\bullet	\circ	\circ^\times	\circ	\bullet^\times	\bullet
SIDECAR	\bullet	\bullet	\circ	\bullet	\bullet	\bullet	\circ	\circ	\circ
AMBASSADOR	\bullet	\bullet	\odot	\circ	\bullet	\circ	\circ	\odot	\circ
COMPUTE RESOURCE CONSOLIDATION	\bullet^\times	\bullet^\times	\circ	\circ	\bullet^\times	\bullet	\circ^\times	\circ	\bullet^\times
COMMAND QUERY RESPONSIBILITY SEGREGATION	\bullet	\bullet	\circ	\circ	\odot	\bullet	\bullet^\times	\bullet^\times	\bullet
EXTERNAL CONFIGURATION STORE	\bullet	\bullet	\bullet	\circ	\bullet	\bullet	\bullet^\times	\bullet	\bullet
GATEWAY ROUTING	\bullet	\bullet	\bullet	\bullet	\bullet	\bullet	\bullet	\bullet	\bullet
GATEWAY AGGREGATION	\bullet	\bullet	\circ	\circ	\bullet	\circ	\circ	\circ	\circ
GATEWAY OFFLOADING	\bullet	\bullet	\bullet	\circ	\bullet	\bullet	\circ	\bullet	\circ
BACKENDS FOR FRONTENDS	\bullet	\bullet	\circ	\circ	\bullet	\bullet	\circ	\circ	\bullet
LEADER ELECTION	\circ	\bullet	\circ	\circ	\bullet	\circ	\bullet	\circ^\times	\circ
PIPES AND FILTERS	\bullet	\bullet	\bullet	\bullet^\times	\bullet	\bullet	\bullet	\circ	\bullet
STATIC CONTENT HOSTING	\bullet	\bullet	\bullet	\circ	\bullet	\bullet	\bullet	\bullet	\bullet

The average number of patterns adopted from the set is 9. Out of our 9 participants, only 1 (i_2) reported having had experience with all 14 patterns. Additionally, only 2 patterns were applied

Figure 5.3: Number of patterns adopted by each participant.

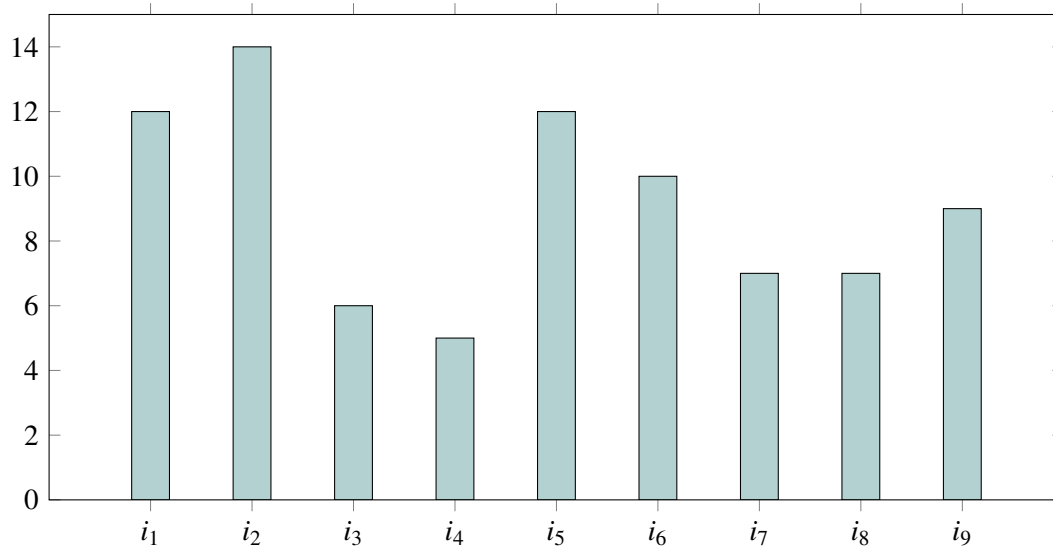
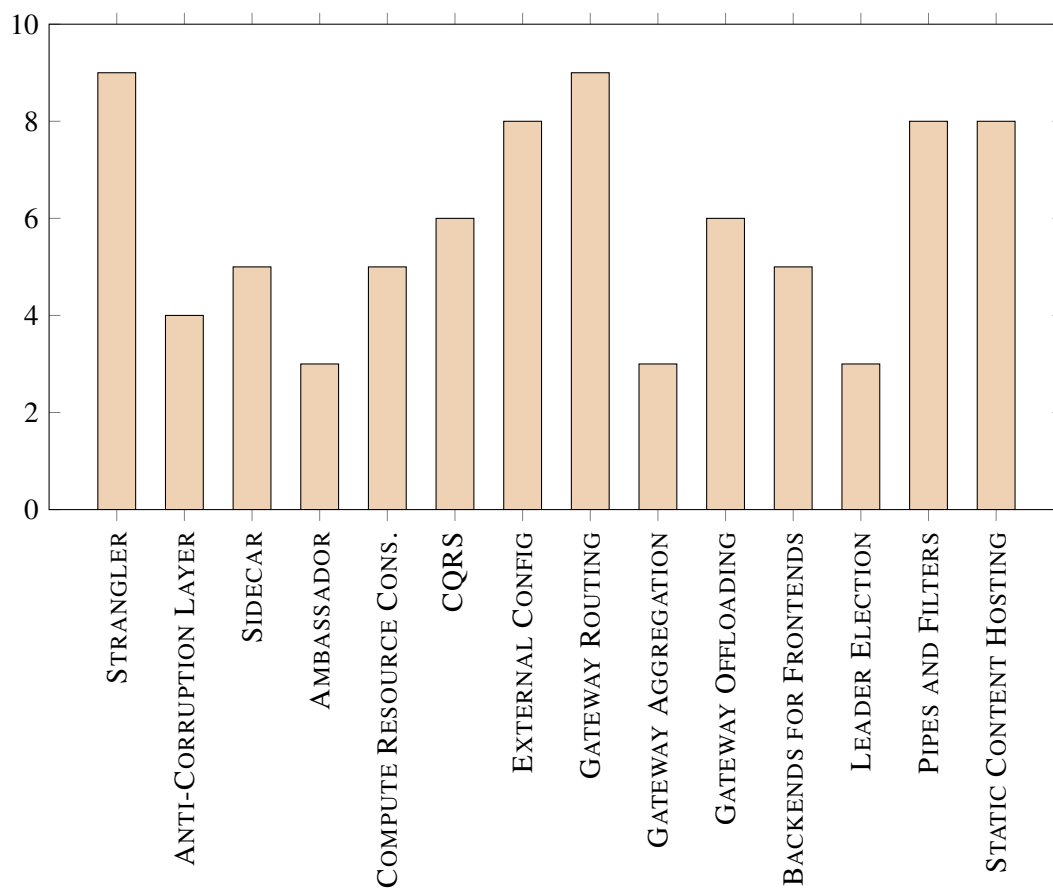


Figure 5.4: Number of participants that adopted each pattern. The names of some patterns have been abbreviated due to space limitations.



by all participants, STRANGLER and GATEWAY ROUTING, with the former receiving particular praise from most respondents.

If you're not starting from scratch, if you have a legacy service you can't just drop immediately, I would honestly say [the strangler approach] is the only way you can keep the legacy system running while being able to develop new features in a modern system. — i₁

Many participants recognised many of the patterns but were only familiar with them to the extent that they had used them as clients. The LEADER ELECTION pattern is probably the best to illustrate this point: all but one participants were aware that leader election is a strategy used in distributed systems and implemented ‘under the hood’ by database management systems and container orchestration mechanisms but only three reported having had actually implemented it in their products.

I know this pattern because, for example, Kafka uses this. I've also used Cassandra which uses this as well. I have made use of several technologies which implement this pattern but I've never had a need to use this. — i₉

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.

Typically, this is done at the system level. With Kubernetes we have mechanisms to address this need for management, with a gateway for instance. — i₃



In our environment we have a gateway which is the interface for when there are requests coming in, and it later deals with the application code. In a sense we end up having an ambassador in there. — i₈

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 were getting too many reads in our DB for large amounts of data and what we saw was that writes were getting blocked often, so we created read-only replicas. — i₅

We assign these instances a ● value. These are most likely reflections of 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 so as to not jeopardise or corrupt our findings in the next section.

5.3.2 Pattern trade-offs









In the following sections, we explore the findings related to the professional perception of the software qualities affected by each design pattern. We express the trade-offs of the patterns through the form of gains and pains, systematically, by way of table representation. We attach one of two symbols to each concern raised by respondents. We use the  symbol to denote remarks which are already identified in the AAC documentation, and the  symbol to denote novel remarks which are not present in the AAC.

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

5.3.2.1 STRANGLER

Applied by all 9 participants, Table 5.7 contains a summary of the trade-offs of this pattern as reported by participants.

Table 5.7: Reported gains and pains of the STRANGLER pattern.

Gains	#	Pains	#
 Scalability	5	 Costly to maintain	2
 Maintainability	4	 Planning transitions	2
 Consumer transparency	4	 Managing data consistency	1
		 Initial resistance by developers	1
		 Backwards compatibility	1

Over half of our participants mention **scalability** as a gain received from application of 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 facet that practitioners extolled.

An advantage that isn't shown in the diagram here is that we have consumers calling this façade, and normally they are barely affected by a well done migration. — *i*₂

If you have many external systems consuming this legacy API, or even if you don't know how many are calling it, this is clearly the way to go. — *i*₄

We use this for keeping our operation running without any harm to our clients, ensuring we can keep making this transition, updating and maintaining our systems without regression or data loss issues, things of that nature. — *i*₈

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.

You have to evaluate the effort you want to put in here: maybe you want to take the legacy system and move the user information out to a new one? Those are very critical decisions which can turn out well if they're well thought out, but it demands a very high level of attention to detail. — *i*₅

One participant reported difficulties **managing data consistency** between subsystems. Additionally, another participant reported that **initial resistance by the development team** was met when presenting the idea for this pattern.

Due to the Big Design Upfront paradigm, people will look at the monolith and want to remake it from scratch. And it's a challenge having to say "no, we'll just take this piece here and extract it". It's a communication issue. — *i*₆




Finally, another participant noted **backwards compatibility** as a particular challenge introduced by this pattern.

The difficulty here is backwards compatibility, by which I mean being able to implement new features while not losing any previous functionalities. — *i*₉

5.3.2.2 ANTI-CORRUPTION LAYER

Applied by 4 participants, Table 5.8 contains a summary of the trade-offs of this pattern reported by participants.

Table 5.8: Reported gains and pains of the ANTI-CORRUPTION LAYER pattern.

Gains	#	Pains	#
 Decoupling between systems	4	 Layer complexity	3
		 Performance	1

All four participants note the same advantage with this pattern: the **decoupling between systems**.

This allows you to develop your service without having external interference: changes from the outside don't force you to corrupt your service, so to speak. — *i*₉

On the other hand, the **complexity of the layer** was mentioned by three participants as a serious drawback to be considered.

In reality what happens is this layer in the middle is going to accumulate all the complexity that each system exposes. So, for me, the weight of this layer should not be too great: it becomes impossible to manage, we should contain its entropy and growth. — *i*₂




Additionally, a negative impact on **performance** was noted by one participant.

There's an added latency here. To deal with that we used some cache mechanisms or sometimes local replication. But yes, a performance requirement is something to consider. — i_2

5.3.2.3 SIDECAR

Applied by 5 participants, Table 5.9 contains a summary of the trade-offs of this pattern reported by participants.

Table 5.9: Reported gains and pains of the SIDECAR pattern.

Gains	#	Pains	#
 Monitorability	5	 Operational complexity	1
 Maintainability	4		
* Scalability	1		
* Security	1		
* Testability	1		

All participants who applied this pattern noted **monitorability** as the major gain of this pattern.

We use this for logging. We have a sidecar with Kibana and then one with Consul, for service discovery. — i_6

Maintainability also followed in the set of benefits of sidecars.

We had all these [cross-cutting concerns] already implemented before we started using Kubernetes and sidecars. But what ended up happening is we were able to remove many shared packages because now a service doesn't need to import all of those dependencies, the service simply assumes the infrastructure is going to give it these features. — i_1

One participant additionally suggested **scalability** and **security** as further gains from this pattern. One other participant mentioned **testability** as a facet enhanced by sidecars, with an interesting use case to illustrate.

In our test environments we do not want direct communication with our providers, we want mock servers. What's the simplest way to do that? We placed a sidecar container which functions as a mock server. It has a SOAP UI running, or something like that, and we can easily change the request URLs to hit the mock server instead of the providers. — i_5

Participants seem happy with this pattern, with only one pointing out **operational complexity** as a pain brought about by the adoption of this pattern.





There's an operational complexity here: debugging the sidecar is quite complicated because now we're in several containerised environments, all of which have dependencies at build time, deploy time and run time. And this coupling is tricky to manage. Not only the obvious coupling between the service and the sidecar but also all the other services the sidecar might need, such as a login service.

I believe that an organisation with the capacity to operate this should bet on this model because of the advantages it brings, otherwise it can never reach the scale and standardisation demanded by microservices. — *i₂*

5.3.2.4 AMBASSADOR

Applied by 3 participants, Table 5.10 contains a summary of the trade-offs of the AMBASSADOR pattern reported by participants.

Table 5.10: Reported gains and pains of the AMBASSADOR pattern.

Gains	#	Pains	#
 Availability	3	 Operational complexity	3
 Monitorability	2		
 Security	2		
* Maintainability	1		
* Scalability	1		

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.



There are things here which can increase coupling in the sense that, typically, when you add these libraries they start to have too much knowledge over the business and then you end up with all your services running with the same code, and that couples everything up. — *i₆*

5.3.2.5 COMPUTE RESOURCE CONSOLIDATION

Applied by 5 participants, Table 5.11 contains a summary of the trade-offs of this pattern reported by participants.

All five participants 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

Table 5.11: Reported gains and pains of the COMPUTE RESOURCE CONSOLIDATION pattern.

Gains	#	Pains	#
 Scalability	5	—	
 Cost savings	2		

strategies, calculating the best consolidation of instances automatically. One participant noted that the scalability enabled by this patterns adds greatly to the serverless functions they use, as the serverless paradigm encourages processing on-demand rather than always running and waiting for requests.

Having this with Kubernetes is easy, you have dozens of projects which give you this automatic scaling flexibility, of managing thousands or millions of functions in production. It's a pretty mature approach at this point. — i_1

With [a serverless] solution we're able to scale pretty much without any limits, in a way that is fast and automated. In the case of AWS, for example, it's super easy to get autoscaling of some services. — i_6

Additionally, with the conjunction of sophisticated techniques like autoscaling, **cost savings** were noted by two participants as a welcome addition.





This pattern is pretty much required if you have volatile resources. For example, during Black Friday we have to have 10 times our current infrastructure but we can't pay for this the whole year or the company will bleed money. So we choose this approach. — i_2

No participant reported pains with this pattern. This suggests that participants think of this pattern as a boon with minimal to no downsides.

5.3.2.6 COMMAND QUERY RESPONSIBILITY SEGREGATION

Applied by 6 participants, Table 5.12 contains a summary of the trade-offs of this pattern reported by participants.

Table 5.12: Reported gains and pains of the COMMAND QUERY RESPONSIBILITY SEGREGATION pattern.

Gains	#	Pains	#
 Scalability	3	 Complexity	5
 Performance	2	 Ensuring data consistency	2
* Availability	2		

Participants note gains in **scalability**, **performance** and **availability** as a result of this pattern.

The main advantage is you can scale reads and writes differently. You want reads to be as fast as possible with no downtime, whereas with writes it's not quite the same. [...] This let you see these as two different pieces, which you can define accordingly.

— *i*₁

For us this was very useful for scaling our product processing. [...] We have a wealth of processes which read data, perform some functions on them and re-inject data back in the system. So we needed to scale reads per consumer without impacting the processing of our backend.

— *i*₂

As for drawbacks, the **complexity** introduced with this pattern was mentioned by all but one participants.

You have to think about whether or not you want to take on the costs of this overhead. We have this implemented in a big part of our services but we still have some services which are purely CRUD and with those we opted against CQRS. We've noticed this makes sense when you want to see [reads and writes] as two separate pieces and when you have a complex business logic which is best when decoupled.





— *i*₁

Finally, **ensuring data consistency** was another problem felt by two participants.

5.3.2.7 EXTERNAL CONFIGURATION STORE

Applied by 8 participants, Table 5.13 contains a summary of the trade-offs of this pattern reported by participants.

Table 5.13: Reported gains and pains of the EXTERNAL CONFIGURATION STORE pattern.

Gains	#	Pains	#
 Decoupling	6	 Complexity	3
 Maintainability	6	 Single point of failure	2
* Security	2	* Introducing breaking changes	1
* Scalability	1		
* Availability	1		

Our results suggest a consensus among our participants: the **decoupling between development and operational concerns** and **maintainability** are the primary gains of this pattern.

This is more of an organisational decision, not technical. At the company we want teams to be as autonomous as possible, and as decoupled as possible.

— *i*₁

From the perspective of team management, you can have someone working on changing config properties and another person working on new features, at the same time.

— *i*₇

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.

There's a cost of communication here between who writes the code and who's looking at the configurations. However, there's nothing against it being the same person, which is most often the case. — *i*₃



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.

There's a danger here which is if you introduce a breaking change to a config. You may think you're making a positive change by updating one URL here but then you end up breaking other services over there. [...] You have to pay attention when you change things, it's not just your world, it's the world of other applications, too. — *i*₅

5.3.2.8 GATEWAY ROUTING

Applied by all 9 participants, Table 5.14 contains a summary of the trade-offs of this pattern reported by participants.

Table 5.14: Reported gains and pains of the GATEWAY ROUTING pattern.

Gains	#	Pains	#
 Decoupling	7	 Single point of failure	4





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.

This is good because you don't have to expose your infrastructure to the outside. You can bounce around requests between your own services without clients knowing anything, all they care about is the endpoint. — *i*₉

However, a gateway represents a **single point of failure**, a concern raised by four participants.

This can be a headache depending on the amount of traffic. If this is highly scalable, that's fine. If not, you have a potential breaking point here. — *i*₅

Table 5.15: Reported gains and pains of the GATEWAY AGGREGATION pattern.

Gains	#	Pains	#
 Performance	3	 Single point of failure	2
		 Scalability	2
		 Coupling	1

5.3.2.9 GATEWAY AGGREGATION

Applied by 3 participants, Table 5.15 contains a summary of the trade-offs of this pattern reported by participants.

The three participants who applied this pattern 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.

This is dangerous. Why? You have several services here. If the requests are sequential, the time to process this is the sum of the time to process all requests. If they are parallel, the time to process this is the time taken to process the slowest. And as soon as you have a service outage you're screwed. [...] This works but it has to be very well managed.

— *i*₅





This kind of solution is often necessary but it creates a complicated coupling here.

— *i*₆

5.3.2.10 GATEWAY OFFLOADING

Applied by 6 participants, Table 5.16 contains a summary of the trade-offs of this pattern reported by participants.

Table 5.16: Reported gains and pains of the GATEWAY OFFLOADING pattern.

Gains	#	Pains	#
 Security	5	 Defining shared functionality	1
 Maintainability	3		
 Monitorability	2		
* Performance	1		

Widely praised by all who adopted it, this pattern gives practitioners ways of tackling **security** foremost, followed by **maintainability** and **monitorability**.

This allows us for better data isolation to reduce the risk of attacks – we use whitelisting of outbound traffic, things like that. This is configured at the gateway level.

— i_8

For me this makes a lot of sense because it enables a separation of concerns. We can avoid cascading a requirement throughout the entire flow.

— i_2

I think the fundamental aspect here is monitoring. We have an entry point here and we can control who enters our home and we can track what they do.

— i_6

One participant noted **performance** as an extra gain, which logically followed from the security benefits gained.

The second advantage is in terms of performance, because you don't have to keep doing this encrypting and decrypting between services.

— i_3

As for drawbacks, only one participant mentioned **difficulties defining which cross-cutting concerns to share** in the gateway.




One risk could be wanting to use this for everything. The so-called 'specialised service functionality' should be well-defined at a company-wide architectural level: we have an external API manager doing this, we have an internal API manager doing routing, etc., and then we have service and application APIs which only expose business concerns.

— i_2

5.3.2.11 BACKENDS FOR FRONTENDS

Applied by 5 participants, Table 5.17 contains a summary of the trade-offs of this pattern reported by participants.

Table 5.17: Reported gains and pains of the BACKENDS FOR FRONTENDS pattern.

Gains	#	Pains	#
 Performance	5	 Complexity	3
 Maintainability	1		

All participants note **performance** as a QA positively affected by the application of this pattern, with one participant noting that **maintainability** was also improved.

I think an advantage is the 'fit to purpose' feature, it allows us to better respond to the needs of each endpoint or device which is sometimes necessary because of very specific constraints. If we try doing this all in one backend, we can't, it becomes too complex.

— i_2

I think what we get here the most is the flexibility of work between teams that this allows. Front-end teams can be more autonomous, they don't depend as much on the backend people, because now there's this component which abstracts away a lot of things for them. — *i*₇

However, three participants mention costs in the form of **complexity**.

There is an increase in complexity here that we have to manage. — *i*₆

5.3.2.12 LEADER ELECTION

Applied by 3 participants, Table 5.18 contains a summary of the trade-offs of the this pattern reported by participants.

Table 5.18: Reported gains and pains of the LEADER ELECTION pattern.

Gains	#	Pains	#
* Availability	3	* Managing leader resources	1

All three participants note **availability** as the major gain from this pattern.

The advantage here is availability. [...] It's an elegant solution because it's reactive and the code is same in all [nodes]. For me this is very useful, I can't even think of a downside. — *i*₂

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

As soon as you have a leader process, one of your servers is going to have a heavier strain on it, it's going to require more resources and this is something you need to think about when you're allocating resources to all of the processes in your cluster, because any of them could eventually become a leader. — *i*₅





5.3.2.13 PIPES AND FILTERS

Applied by 8 participants, Table 5.19 contains a summary of the trade-offs of the this pattern reported by participants.

Performance is cited by five 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.

This is basically the distribution of some complex business logic. We have all this separated, it gives us segregation of complexity – that is one of the fundamental aspects about this. — *i*₆

Table 5.19: Reported gains and pains of the PIPES AND FILTERS pattern.

Gains	#	Pains	#
 Performance	5	* Monitorability	2
* Maintainability	3	 Complexity	2
 Scalability	1		
 Reusability	1		

In terms of maintainability, teams have a narrower scope here, they end up focusing on more isolated features. — i₇

This simplifies the way you think about problems and how you process data. When you have data transformations you want to reuse, this approach helps a lot. So, here you get a performance boost because you're processing by chunks and you can reuse those transformations later as you perform the steps. — i₉

As for drawbacks, two participants reported **difficulties monitoring** their systems with this programming model.






It's very hard to track these requests, oftentimes. The client will send you an e-mail asking "is it done yet?" but that's the problem with asynchronous processing. [...] You can add logs here but you'll have to be tracking IDs and all that sort of stuff. — i₅

Finally, **complexity** was noted by two participants as a challenge.

5.3.2.14 STATIC CONTENT HOSTING

Applied by 8 participants, Table 5.20 contains a summary of the trade-offs of this pattern reported by participants.

Table 5.20: Reported gains and pains of the STATIC CONTENT HOSTING pattern.

Gains	#	Pains	#
 Performance	5	* Eventual consistency	1
 Availability	3	 Complexity	1
 Maintainability	1		
 Security	1		

Over half declared **performance** as a benefit gained from the application of this pattern.

We were initially hosting things on disk and with a CDN our performance increased a lot. We can have clients all over the world and provide them with static assets much more quickly. — i₇

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.

Files may not be available, may not have been replicated, there's a few problems here.

We've had some issues [with eventual consistency]. — *i*₂

5.3.2.15 Discussion

As was mentioned at the start of this section, our findings directly mix quality attributes 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 two reasons. Firstly, we wanted to present the results in as raw form as possible. Secondly, such a mapping would invariably produce unsatisfactory ambiguities which could jeopardise our (and the reader's) interpretation of the results.

All in all, we were able to gather a plethora of insights into each of these particular patterns, some of which differing quite significantly from their projected intention originally written 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 were, unfortunately, able to only go skin-deep due to a mix of time constraints, missed opportunities and participants not having had experience with some of the patterns, impairing our ability to extract more findings.

In any case, for all these patterns **we have identified 14 new gains and 6 new pains** which were not noted in the AAC documentation, out of a total of 35 gains and 25 pains reported by participants. This suggests that the available knowledge in the AAC is adequate but 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.

5.3.3 Measuring and addressing quality attributes

Architectural trade-offs are expressed in the language of software quality, using quality attributes to refer to areas of the software engineering process that are affected with the application of a given tool or technique. Getting a concrete understanding of changes in software quality, though, should ideally go beyond a developer's subjective perception; it should be aided by real metrics one can point to and track over time.

A curious development followed from our interviews. When asked for metrics for each quality attribute, practitioners often jumped directly to techniques for addressing and improving those attributes, and we typically had to ‘pull back’ and ask them again for particular markers or indicators that can be used to measure their degree of perception of a given attribute. Because of this, we were able to collect additional information that we did not originally set out to find. Similar to the previous section, we present these results systematically: for each quality attribute we list our reported quality indicators for measuring and techniques for improving it. The following sections show our findings for each QA, respectively.

5.3.3.1 Scalability

Table 5.21 presents the reported quality indicators for measuring and techniques for addressing scalability reported by participants.

Table 5.21: Reported indicators for measuring and techniques for addressing scalability.

Indicator	#	Technique	#
Through performance	4	Autoscaling	4
Subjective interpretation	2	Load testing	3
No interest	1	External configs	2
		Dashboards	2
		Segregation of responsibility	1
		FinOps	1

When it comes to getting a feel for scalability, almost half of our participants note a close and **direct relationship between scalability and performance** (see Section 5.3.3.2), which makes sense if we understand a scalable system as one which shows increased performance in a manner proportional to the resources added.

It's always very strongly related to performance: as we improve performance we also become more scalable. — i_6

We end up mixing scale with performance in our metrics, and that's how we can infer one from the other. — i_7

Additionally, 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 understand the resistance

of their applications by evaluating the way they handle increases in requests. **Dashboards** were mentioned by two participants as tools that collate statistics and metrics about their applications which they turn to for guiding their business decisions – see Section 5.3.3.4. **External configurations** were also noted as a form of addressing scalability, as it allows teams to reconfigure application container properties or other run-time settings without any constraints.

We're concerned with removing factors that may limit our ability to scale: externalising our configs is one way we do that. — *i*₂

The **segregation of responsibilities**, encouraged naturally by microservices, was raised by one participant 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 for addressing scalability.

We have a FinOps dynamic: we can scale if the cost-benefit is worth it, because sometimes it's not necessary to scale as much. That aspect of scalability is important. — *i*₂

5.3.3.2 Performance

Table 5.22 presents the reported indicators for measuring and techniques for addressing performance reported by participants.

Table 5.22: Reported indicators for measuring and techniques for addressing performance.

Indicator	#	Technique	#
Infrastructure (CPU, RAM, etc.)	2	Dashboards	3
Server (requests, queries, etc.)	2	Performance testing	2
Business performance metrics	1		
No interest	1		

With performance, participants are aware of lower-level metrics, namely **hardware or infrastructure properties** (e.g., CPU and RAM usage, etc.) and **server properties** (e.g. number of active requests, time taken handling requests, database queries, etc.), for assessing performance levels.

During development we have automatic performance tests: you run a deploy and it executes the performance tests. — *i*₅

Outside of technical metrics, one participant referred **business performance metrics**, also known as key performance indicators, as an additional instrument for evaluating performance. One participant stated that his company has **no interest** in run-time performance, as it poses little cause for concern for their particular business.

Our bottlenecks typically reside in the persistence layers, in databases, in blob storage. Performance just isn't a big concern in our world. — i_1

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 5.3.3.4.

For scalability and performance we have dashboards with alerting. With Grafana and things of that nature. — i_2

Our measurements revolve around looking at Google Analytics and understanding our page load times and, at every release, tracking the change over time. — i_6

Dashboards effectively synthesise a range of indicators into one, sitting at a higher level of abstraction. Further, **performance testing** was noted by two participants as a strategy for measuring performance, integrated into continuous deployment pipelines.

5.3.3.3 Availability

Table 5.23 presents the reported indicators for measuring and techniques for addressing availability reported by participants.

Table 5.23: Reported indicators for measuring and techniques for addressing availability.

Indicator	#	Technique	#
Latency, traffic, errors, saturation	2	Health checks	3
SLAs, SLIs, SLOs	2	Service replication	2
No interest	1	Load balancer	1

For measuring availability, two participants noted run-time statistics like **latency, traffic, errors and saturation** as ways to evaluate availability.

We measure a lot of these availability metrics; response time, error rate, the kind of metrics in Google's SRE Golden Signals³. — i_2

Additionally, two participants mention tracking Service Level Agreements, Service Level Objectives and Service Level Indicators (**SLAs, SLOs and SLIs**, respectively) for judging availability.

We measure SLAs, SLIs, SLOs, etc. in all of our services in production so we can measure the availability of our system. — i_1

³https://sre.google/sre-book/monitoring-distributed-systems/#xref_monitoring_golden-signals

Everyone talks about the five nines or the four nines – that issue of service uptime, whether it's 99.999% or 99.99% – and SLOs; measuring availability an entire area of analysis which is super important for our businesses. — *i*₆

One participant reported **no interest** in availability due to not having concerns for critical or otherwise real-time processes in their business.

Our business isn't about the real-time, it's about the "when it has to work it has to work", that is, I need that report by today and that report has to be done and available to me. Those 99.99999 [five nines] aren't really important on our end. — *i*₃

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.

In microservices, since we have smaller processing units, we're able to increase redundancy by increasing the number of service replicas. — *i*₉

5.3.3.4 Monitorability

Table 5.24 presents the reported indicators for measuring and techniques for addressing monitorability reported by participants.

Table 5.24: Reported indicators for measuring and techniques for addressing monitorability.

Indicator	#	Technique	#
Subjective interpretation	1	Third-party monitoring	5
		First-party monitoring	3

This QA presented a challenge because, to the extent that monitorability is defined as the ability to support the operations staff in monitoring the system at runtime (see Section 3.1), measuring this ability is left unclear. 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 subjective account.

Monitorability isn't exactly an analysis, it's subjective. I can't think of anything that would tell us how to measure it. It's subjective but there is a big focus on it. Every company must have monitoring tools, logging tools, performance tools, alerting tools. But it's always a challenge. — *i*₆

As for addressing monitorability, most respondents cited using **third-party monitoring** for assisting their observing needs. Figure 5.5 presents the third-party tools that the participants reported using actively. As is made evident, no company is an island; some trends are visible.

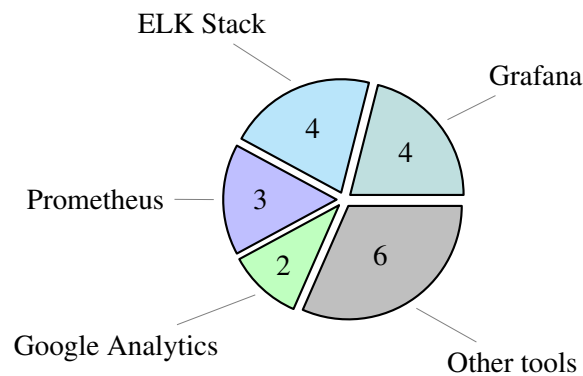


Figure 5.5: Reported third-party monitoring tools and the number of participants who use them.

Grafana and the ELK stack (simultaneous usage of Elasticsearch, Logstash and Kibana) received the most mentions, followed by Prometheus and Google Analytics.

Other third-party tools were mentioned by only one participant each. They are: 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.

We're currently developing our own monitoring platform because, from the experience I've had in the past and I've heard in other companies, no single monitoring tool will solve all of our problems so there is a tendency to have at least two and then this depends on the company's budget. What we did was focus on logging, metrics and health checks and then develop tools that focus on each of these concerns that can protect us from vendor lock-in.

— i_3

5.3.3.5 Security

Table 5.25 presents the reported indicators for measuring and techniques for addressing security reported by participants.

Table 5.25: Reported indicators for measuring and techniques for addressing security.

Indicator	#	Technique	#
Code security	1	Internal audits	5
		Penetration testing	3
		External audits	2
		Bug bounties	1
		Multi-tenancy	1
		SIEM	1

Security seemed a puzzling question for our participants, with only one pointing out any concrete metrics, in the form of **code security**.

We look at code vulnerability metrics, we do that through SonarQube. — i_2

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.

We have a security team that does that before things go into production. They check [man-in-the-middle] attacks, how we respond to [distributed denials of service], how we handle headers, they try to use weird and unexpected inputs, etc.. We have all that kind of stuff. — i_5

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 tactics 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.

5.3.3.6 Testability

Table 5.26 presents the quality indicators for measuring and techniques for addressing testability reported by participants.

Table 5.26: Reported indicators for measuring and techniques for addressing testability.

Indicator	#	Technique	#
Number of automated tests	3	Unit testing	8
Code coverage	3	Integration testing	4
		Performance testing	3
		Penetration testing	1
		Functional testing	2
		Smoke testing	1
		Non-regression testing	1
		QA Manifesto	1

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.

For addressing testability, participants turn, naturally, to testing. **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 also commented that he felt it was the most neglected QA out of the set.

For me this is a personal war but I think it is the least considered by software companies. — i_2

One participant mentioned their company having a **QA Manifesto**, an internal document containing their own definition of quality revolving around tests.

Internally we wrote a QA Manifesto which is basically about, what does quality mean to us? It means having 80% test code coverage, it means having integration testing, and so on. For now, we have a shared vision at least on the foundation which is 80% coverage of unit testing. That implies an extra awareness whenever we're writing code.

— i_3

5.3.3.7 Maintainability

Table 5.27 presents the quality indicators for measuring and techniques for addressing maintainability reported by participants.

Table 5.27: Reported indicators for measuring and techniques for addressing maintainability.

Indicator	#	Technique	#
Code coverage	2	Static code analysis	4
Time to fix bugs	1	Automated pipelines	3
Architectural debt	1	Code review	3
		Domain-driven design	2

With respect to metrics, participants are more divided. Similar to testability, **code coverage** was mentioned by two participants as an indicator of maintainability.

If it is not easy to test what you're working on – if the coverage is low – there is a strong chance that it is not maintainable or even that it is not respecting a segregation of responsibilities.

— i_4

Beyond that, one participant mentioned tracking, through Slack and Jira, bug reports and the **time taken to fix 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.

Something we have started doing recently is measuring the architectural debt of our application domains: what is the cost to repair, the risk to the business, the probability of it being necessary and the impact it may have?

— i_1

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 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.

We use domain-driven design in the first place, which is a great start [for maintainability]. Just from that we get this very helpful separation into modules. — *i*₅

5.3.3.8 Discussion

Given that the data extracted for these findings comes from the final act of the interviews, almost always nearing the end of our allotted time, in some instances even going over it, we are disappointed but not surprised by the short and superficial answers we obtained, which only scratch the surface of exploring industrial reality in observing software qualities in MSA.

With some participants, we were not able to obtain any details regarding either indicators or techniques for measuring or addressing, respectively, a quality attribute. 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 research questions and lack of depth. These hurdles give us less confidence in asserting saturation of our findings, though we do feel we have provided a substantial account of industrial perception of software quality attributes.

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.

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 quality attributes 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 inter-personal agreements on a semantic basis.

Given all these, I would say there are QAs here you can truly measure if you have observability in your system, but there are others you can't. Take maintainability — you can ask a developer and he'll tell you 'this is more maintainable', another will tell you 'this is less maintainable'. It's susceptible to opinion. — *i*₁

In all, these findings seem to support the arguments presented by Moses [39], particularly the author's note of scepticism 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 [34], we see several techniques practitioners resort to for addressing these QAs that weren't covered by their systematic literature review.

5.3.4 Other findings

In this section we present topics of interest that were raised by our participants of their own volition which are not related to our research goals but which we feel are valuable points to mention, as they represent industrial perspectives on microservices.

5.3.4.1 Event-driven architecture

Event-based processing, or an event-driven architecture, seems to be a rising tendency within microservices. Nearly all participant (8 out of 9) were familiar with Apache Kafka and RabbitMQ and had experience with at least one, suggesting these are flagship solutions for implementing inter-service communication based on events and message passing.

With messaging you get this huge decoupling between producer and consumer. All the producer service knows is that it is going to send a message and then it doesn't care which subsystems read it. If you want to add new services that listen to that event, that's fine, just another subscription. Nowadays our entire architecture rests on a very strong event-based foundation. — i₁

We have used RabbitMQ for messaging between services. One service would send another service a message which would, in turn, trigger processes in other services, through events. I'll give you a concrete example: marketing campaigns. We had a service that prepared the mailing lists and another service that prepared the e-mail templates. All of this was done through Rabbit[MQ], not REST. — i₆

5.3.4.2 Feedback

The following comments were conveyed at the end of each interview, where we asked interviewees their overall perception and value judgement of the study. We received no negative reviews of the interview process, with several positive comments regarding the relevance and interest of the topic of design patterns in software architecture. One participant stated an architectural conception of their product is something they had recently begun investing in terms of career plans in the company.

This is one of the things we have scheduled here for career plans development at the company, on the engineering side, this notion of design patterns, their names and applications. We end up using patterns sometimes that we didn't even knew had names. And I think it's useful to know these names because that allows us to have more mechanisms to grow and try new things. That is to say that I think it's a very interesting topic and the interview was useful for me, also – I've already written down some notes. — i₈

One participant suggested emphasising the organisational aspect of microservices. He considered it one of the essential features of the architectural style, as it enables further optimisation of the division of labour in large projects.

Choosing microservices is more a matter of scaling teams, you can partition the tasks and become more agile in the way you develop your software, rather than the particular technology you're using. I think it's an important point in the consideration of microservices.

— i₉

5.3.5 Threats to validity

The credibility of results obtained through empirical studies is always hampered by threats to validity [58]. We can think of validity through four different aspects [47]: construct validity; internal validity; external validity; and reliability.

Construct validity refers to the meaningfulness of measures, i.e. to what extent they measure what they claim to be measuring. The following threats were identified in this respect:

- An interview study is **susceptible to social desirability bias**, which refers to the tendency for participants to present themselves in the most flattering light [58]. To mitigate this threat, we made it clear to participants that no identifying data would be shared.

Internal validity refers to the cause-and-effect relationships identified and to what extent it rules out other possible interpretations for the findings. The following threats were identified in this respect:

- 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. To mitigate this threat, we informed ourselves on the quality attributes most important to practitioners from published literature.
- This study demanded extensive knowledge of all design patterns and quality attributes that were analysed as well as a proficiency for conveying this knowledge to participants in the event of miscommunication or confusion. While we sought a serious depth of knowledge of the theoretical elements involved in this study, all interviews were performed by the author, who **cannot claim 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. To mitigate this threat, we accompanied questions with a presentation that included brief descriptions of the patterns and example illustrations.

- In general, 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 quality attributes.

External validity refers to the degree to which the study produces results that can be generalised, i.e. applied to other contexts. The following threats were identified in this respect:

- We **cannot claim representativeness of our study demographics** for the software industry in general, as the population was sampled through our personal networks and using a voluntary, opt-in procedure. To mitigate this threat, we selected participants from diverse backgrounds so as to cover companies of different sizes and business domains.
- We **cannot claim saturation** of our findings on pattern trade-offs. From our study, we are confident asserting that a sample of seven proved to be a point of informational redundancy for design pattern trade-offs, that is to say we are confident affirming saturation of the trade-offs identified for patterns applied by 7 or more participants. Regarding pattern with less than 7 applicants, we expect that more themes would arise following further interviews.
- A voluntary survey design is inherently **susceptible to self-selection bias** [2]; uninterested respondents are unlikely to participate in this type of study.
- All interviewed participants are currently residing and working in Portugal. Due to this fact, it is possible that a regional technology culture might have rendered a **biased lens into industrial reality** of MSA.

Reliability refers to the extent to which replications of the study will achieve the same results if applied by a different researcher. The following threats were identified in this respect:

- The steps described in this study were **performed only by the author**. It is possible that changes on an attitudinal level might have drawn different answers and findings from our participants. We were, unfortunately, not able to mitigate this thread, e.g. through a second author repeating either the data collection process or the data analysis process.

5.4 Summary

We have described an interview study with nine professionals experienced with developing microservice systems. All participants reported at least a tacit understanding of design patterns though few showed a particular interest in the topic, suggesting that patterns do not yet make up part of the major lexicon of professional software development in MSA. With regards to quality attributes, practitioners seem less interested in including them in their day-to-day vocabulary, except more widely used QAs like scalability, performance and maintainability. Despite this, practitioners are eager to observe all facets of their products, with many showing initiatives to reach architectural excellence, perceived as a business advantage.

One major lesson learned is that we should have selected fewer patterns for analysis. Our roughly hour-long interviews were fruitful but this was, ultimately, very little time to explore, in any significant depth, any one particular pattern or quality attribute. In hindsight, we should have focused our pattern selection into particular domains of design, development or operation, instead of casting a wide net which proved difficult to go deep with. For instance, we suspect it might have been more interesting if we focused on design patterns that seek to address inter-service communication.

Chapter 6

Conclusions and Future Work

In this chapter we conclude this dissertation. In Section 6.1, we provide a summary of the entire project. In Section 6.2, we present the answers to the research questions which motivated this study. In Section 6.3, we describe potential avenues for future research that we have identified as a result of this work.

6.1 Overview

We began with the observation that the microservice 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 the 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 not been an object of rigorous empirical research.

A popular truism goes *"practice without theory is blind; theory without practice is empty"* – there is a symbiosis between theory and practice, one cannot flourish without the other. Research articles carry a literature review for this reason, i.e. to establish their theoretical topicality and validity. Design patterns, however, typically lack this highly desired empirical edifice with which to affirm their grounds as solid.

We researched the state of the art on the microservices architecture from the perspective of design strategies and their trade-offs, expressed mostly through impacts on quality attributes. We found, among other things, a noticeable lack of empirical studies performed within the context of applying design patterns in microservice systems. We set out to alleviate this gap by way of conducting an interview study.

The results and findings from this dissertation offer implications for both professionals and researchers. For professionals, this work provides a comprehensive understanding of QAs impacted by design patterns, which can serve as a useful guide for strategic engineering decision-making

on moving forward with adopting MSA. For researchers, we provide a review of the literature surrounding software quality, design patterns and empirical studies in microservices, as well as a thorough report of an interview study's design and execution, with findings that add to the development of the existing body of knowledge.

By providing an extensive account of our study we hope to have shone a light on the process of designing and conducting interview studies related to trade-offs in software engineering, both in the successes found but also the hurdles met along the way.

6.2 Research answers

Our research questions, initially posited in Section 4.2, can now be answered the following way:

RA1. *"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 professionals currently lack the language to explore solutions for hurdles seen in microservices which might lead to risky and untested paths being taken unnecessarily. With regards to experience with 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. See Section 5.3.1 for the rates of adoption of each pattern reported by our participant, as well as comments on their relevance and usage.

RA2. *"How are quality attributes 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, suggesting that the Azure Architectural Centre serves as a good resource that can be used for precisely studying how a pattern works in a wider, concrete context. New gains and pains were identified in some patterns as a result of this study, however, which should be properly reflected in the AAC. For example, GATEWAY AGGREGATION was viewed very poorly by all participants who applied it, suggesting 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. See Section 5.3.2 for the architectural trade-offs reported by participants.

RA3. *"How are quality attributes measured in microservices?"* — Practitioners show interest in tracking several key facets of software quality, though they do not directly turn to quality attributes to measure and express the changes to their systems. Instead, they use a wealth of tools and techniques which they perceive as improvements. Similarly, practitioners show little interest in discussing metrics, i.e. artefacts which can be directly measured, preferring

instead to think of quality indicators considered to be related to quality attributes. Thus, our results suggest QAs remain outside the vocabulary of the typical developer, with the exception of well-known terms like *scalability*, *performance* and *maintainability*. See Section 5.3.3 for the quality indicators and strategies reported by participants.

6.3 Future work

Despite having fulfilled our stated objective, this work ultimately represents just one more step in the quest for a full understanding of microservices in industry. We have picked up the brush and added our own contributions to the landscape. But the painting is far from over.

Future work can **apply the design of our study** using different inputs, i.e. a different set of design patterns and/or quality attributes, preferably smaller and more focused sets. We hold faith in the design of this study to produce valuable discoveries, and, as such, it forms a framework that can be used freely by researchers.

The construction of a **software prototype** which synthesises the findings of this study, perhaps in the form of a web application, is one of the possible next steps for increasing the reach of this research.

An empirical analysis of design patterns in microservices is still a matter which requires further work. More sophisticated research methods can be conducted in this domain, such as a **fully grounded theory-based study** or **controlled experiments**. Alternatively, aiming for a broader sample with a **web-based questionnaire** geared towards aiding the goals of this work is another possible strategy for further validation of our results. In follow-up studies, it would be valuable to characterise, in greater detail, the roles of participants: how they apply MSA in their contexts, e.g., how the organisation is structured- – what roles exist, how is the division in teams made, etc.. This can help us know more about specific configurations and spot relationships between potentially different contexts and the patterns that get applied in them.

In sum, industrial interest in event-driven microservices should be met with academic interest. As this architectural style matures, researchers should keep pace with the processes and strategies being adopted by practitioners in order to extract and identify new design patterns which, in turn, should form the objects of further study.

Appendix A

Interview resources

This appendix presents the resources produced and described in Chapter 5. In Section [A.1](#) we present the form sent to study candidates, described in more detail in Section [5.1.2.1](#). In Section [A.2](#) we present the interview guide, the document used to assist in our interview questions, described in more detail in Section [5.1.3](#). In Section [A.3](#) we present the interview helper, the document used during Acts 2 and 3 of our interviews, that was shown directly to participants.

A.1 Sign-up form

This form was produced with Google Forms. It was sent to people we had contacted and shown an informal interest in our research.

Empirical Study on Microservices Design

Thank you for agreeing to participate in our study about microservice architectures. You will be contributing directly to new understandings on how teams perceive and make architectural decisions in service-based systems, and we will be happy to share the conclusions of the study with you, first-hand.

The study is composed of several parts. Participating in this part of the study doesn't commit you to the following ones. In this first part, we would like to understand how professionals perceive the consequences (good and bad) of using certain architectural solutions. We want to find which solutions have worked best for you, and why. For this, we would like to interview you in a remote call, which should not take more than 1h30m.

To make sure that you qualify to participate in the study, please fill the short form below, and we will get in touch again to schedule the interview.

If you have any questions, feel free to get in touch with us:

- Filipe Correia <filipe.correia@fe.up.pt>
- Guilherme Vale <up201709049@edu.fe.up.pt>

*Required

Confidentiality Statement

The interviews will be recorded and transcribed. We can ensure that all the collected data will be used only to support the scientific research in question, and can ensure confidentiality regarding the companies' products, architectures, and processes, should they surface during the interview. In all works published by the researchers, the names of the interviewees and organization will be anonymized. Only the researchers in this study will have access to the raw data.

By agreeing to participate, the participant allows researchers to use the anonymous information as described below. The participant understands that:

- The participant can contact the researchers with any questions about the study;
- All information is confidential and that any objects of study will be anonymized if included in publications in journals, conferences, or blog posts;
- The participant can contact the researchers with any questions about the study;
- The researchers will maintain the data collected in perpetuity and may use it for future academic work.

1. Name *

2. Email address *

3. Role in company *

Mark only one oval.

- ☐ Developer
- ☐ Architect
- ☐ Operations
- ☐ QA
- ☐ CTO
- ☐ Other: _____

4. How many years of experience do you have with professional software development? *

Mark only one oval.

- ☐ Less than 5 years
- ☐ 5 to 9 years
- ☐ 10 to 14 years
- ☐ 15 to 19 years
- ☐ 20 years or more

5. How many years of experience do you have with developing microservices-based systems? *

Mark only one oval.

- ☐ Less than 2 years
- ☐ 2 to 4 years
- ☐ 5 years or more

6. Considering the microservices-based systems you've worked on, what was the largest number of active monthly users regularly served by the system? *

(please provide your closest estimate if you don't know the exact number)

Mark only one oval.

- ☐ Less than 100
- ☐ Between 100 and 1K
- ☐ Between 1K and 10K
- ☐ Between 10K and 100K
- ☐ Between 100K and 1M
- ☐ More than 1M

This content is neither created nor endorsed by Google.

Google Forms

A.2 Interview guide

This simple text document was produced with Google Docs. The guide does not discount the possibility of follow-up questions that may arise ad hoc.

Interview Guide

Act 1 - Personal experience with microservices

Tell me a bit about the microservices systems you have worked on or are working on, currently.

- How mature were these systems?
- Do you have an idea of the number of services that were in development and operation?
- How many lines of code do you figure made up these systems?
- What inter-service communication mechanisms have you used?

Act 2 - Design pattern trade-offs

I'm going to show you a presentation and for each slide I will briefly explain it and ask you a few questions about your experience with it.

1. Strangler
 - a. Are you familiar with the name of this pattern?
 - b. Have you used this pattern?
 - c. (if yes) Why did you opt for this pattern?
 - d. (if yes) What do you think are its pros and cons?
 - e. (optional follow-up questions)
 - f. (ask these questions for every following pattern)
2. Anti-Corruption Layer
3. Sidecar
4. Ambassador
5. Compute Resource Consolidation
6. Command Query Responsibility Segregation
7. External Configuration Store
8. Gateway Routing
9. Gateway Aggregation
10. Gateway Offloading
11. Backends for Frontends
12. Leader Election
13. Pipes and Filters
14. Static Content Hosting

Act 3 - Microservice quality metrics

Are you concerned with the following quality attributes, in your company?

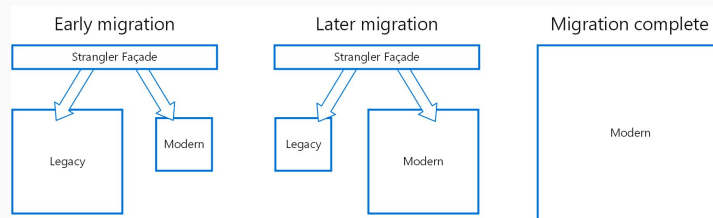
1. Scalability
 - a. (if yes) How do you measure scalability? How often?
 - b. (if not) Why not?
 - c. (ask these questions for every following attributes)
2. Performance
3. Availability
4. Monitorability
5. Security
6. Testability
7. Maintainability

A.3 Interview helper

Each of the first 14 slides corresponds to one design pattern. The 15th slide contains our set of seven quality attributes.

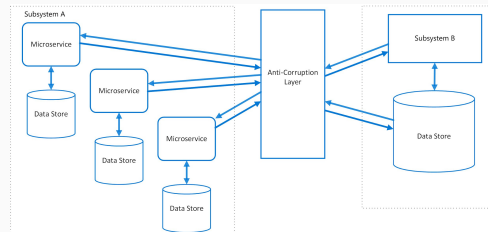
Strangler

Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.



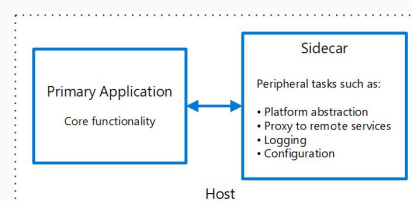
Anti-Corruption Layer

Implement a façade or adapter layer between different subsystems that don't share the same semantics. This layer translates requests that one subsystem makes to the other subsystem.



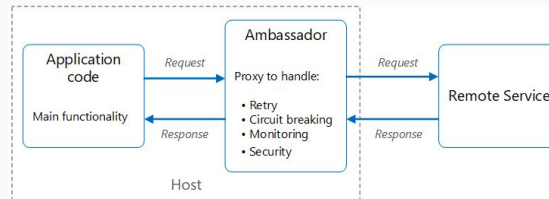
Sidecar

Deploy components of an application into a separate process or container to provide isolation and encapsulation.



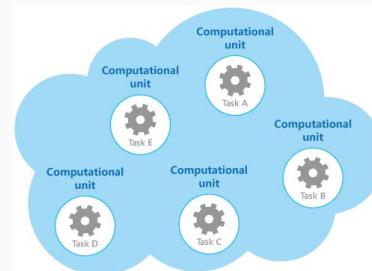
Ambassador

Create helper services that send network requests on behalf of a consumer service or application. An ambassador service can be thought of as an out-of-process proxy that is co-located with the client.



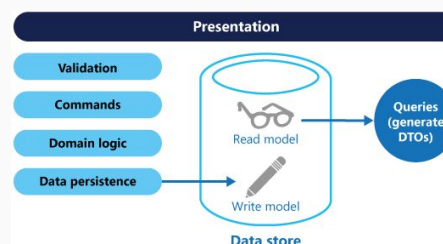
Compute Resource Consolidation

Consolidate multiple tasks or operations into a single computational unit.



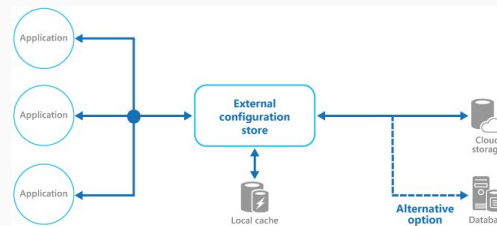
Command Query Responsibility Segregation (CQRS)

Segregate operations that read data from operations that write data by using separate interfaces.



External Configuration Store

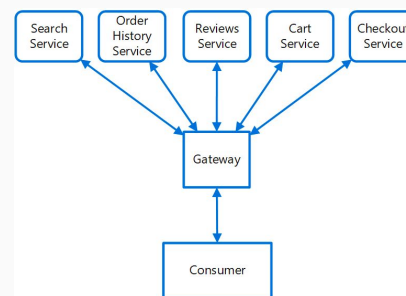
Move configuration information out of the application deployment package to a centralized location.



Gateway Routing

Route requests to multiple services using a single endpoint.

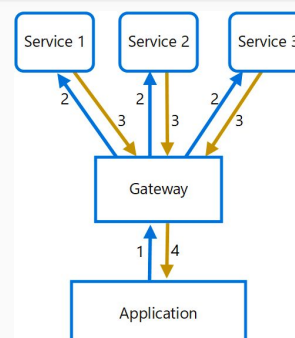
aka API Gateway



Gateway Aggregation

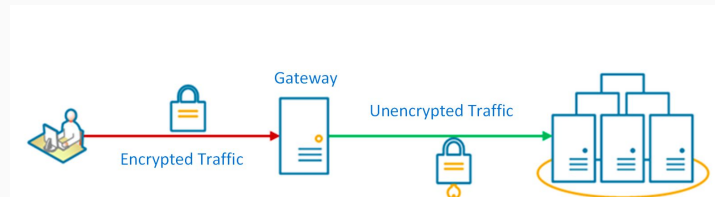
Use a gateway to aggregate multiple individual requests into a single request.

aka API Composition



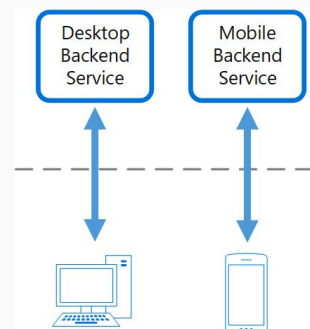
Gateway Offloading

Offload shared or specialized service functionality to a gateway proxy.



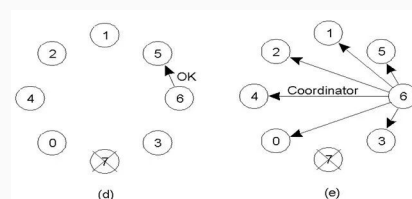
Backends for Frontends

Create separate backend services to be consumed by specific frontend applications or interfaces.



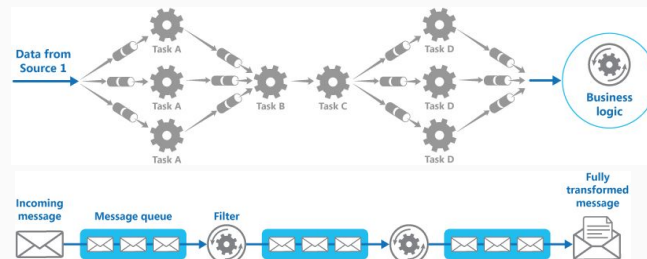
Leader Election

Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.



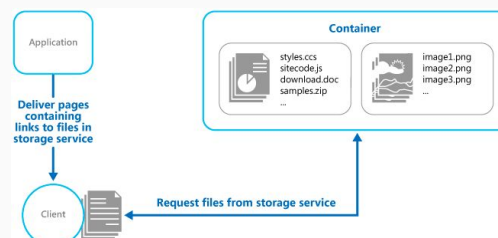
Pipes and Filters

Decompose a task that processes data streams into a sequence of processing steps using a common interface.



Static Content Hosting

Deploy static content to a cloud-based storage service that can deliver them directly to the client.



Scalability

Performance

Maintainability

Availability

Monitorability

Security

Testability

References

- [1] Christopher Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.
- [2] Hamdy Michael Ayas, Philipp Leitner, and Regina Hebig. Facing the giant: a grounded theory study of decision-making in microservices migrations. *arXiv preprint arXiv:2104.00390*, 2021.
- [3] Sarah Elsie Baker and Rosalind Edwards. *How many qualitative interviews is enough*. NCRM, University of Southampton, March 2012.
- [4] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A Tamburri, and Theo Lynn. Microservices migration patterns. *Software: Practice and Experience*, 48(11):2019–2042, 2018.
- [5] Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *CoRR*, abs/2002.07764, 2020.
- [6] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.
- [7] J. Bogner, J. Fritzsche, 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)*, pages 546–556, September 2019. Issn: 2576-3148.
- [8] J. Bogner, J. Fritzsche, S. Wagner, and A. Zimmermann. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 187–195, March 2019.
- [9] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. On the impact of service-oriented patterns on software evolvability: a controlled experiment and metric-based analysis. *PeerJ Computer Science*, 5:e213, August 2019. Publisher: PeerJ Inc.
- [10] Kyle Brown and Bobby Woolf. Implementation patterns for microservices architectures. In *Proceedings of the 23rd Conference on Pattern Languages of Programs, PLoP '16*, pages 1–35, Usa, October 2016. The Hillside Group.
- [11] L. Carvalho, A. Garcia, W. K. G. Assunção, R. de Mello, and M. Julia de 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 (SERIP)*, pages 22–29, May 2019. Issn: 2575-4793.

- [12] M. Cojocaru, A. Uta, and A. Oprescu. Attributes Assessing the Quality of Microservices Automatically Decomposed from Monolithic Applications. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 84–93, June 2019. Issn: 2379-5352.
- [13] S. Dasanayake, J. Markkula, S. Aaramaa, and M. Oivo. Software Architecture Decision-Making Practices and Challenges: An Industrial Case Study. In *2015 24th Australasian Software Engineering Conference*, pages 88–97, September 2015. Issn: 1530-0803.
- [14] Thatiane de Oliveira Rosa, João Francisco Lino Daniel, Eduardo Martins Guerra, and Alfredo 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*, pages 1–8, 2020.
- [15] Tiago Boldt Pereira de Sousa. *Engineering Software for the Cloud: A Pattern Language*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2020.
- [16] Saulo S de Toledo, Antonio Martini, and Dag IK Sjøberg. Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. *Journal of Systems and Software*, 177:110968, 2021.
- [17] Jessica T DeCuir-Gunby, Patricia L Marshall, and Allison W McCulloch. Developing and using a codebook for the analysis of interview data: An example from a professional development research project. *Field methods*, 23(2):136–155, 2011.
- [18] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97, April 2019.
- [19] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, Today, and Tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer International Publishing, Cham, 2017.
- [20] Peter Eeles. Characteristics of a software architect. *The Rational Edge, IBM Resource*, 2006.
- [21] Eric J. Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. Google-Books-ID: xColAAPGubgC.
- [22] Martin Fowler and James Lewis. Microservices - a definition of this new architectural term, 2014.
- [23] P. Di Francesco, P. Lago, and I. Malavolta. Migrating towards microservice architectures: An industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, page 29–2909, April 2018.
- [24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.
- [25] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture: A survey on the state of the practice. In *Zeus*, pages 1–8, 2018.
- [26] Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough?: An experiment with data saturation and variability. *Field Methods*, 18(1):59–82, February 2006.

- [27] S. Haselböck, R. Weinreich, and G. Buchgeher. An Expert Interview Study on Areas of Microservice Design. In *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 137–144, November 2018. ISSN: 2163-2871.
- [28] S. Haselböck, R. Weinreich, G. Buchgeher, and T. Kriechbaum. Microservice design space analysis and decision documentation: A case study on api management. In *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, page 1–8, November 2018.
- [29] Stefan Haselböck, Rainer Weinreich, and Georg Buchgeher. Decision Models for Microservices: Design Areas, Stakeholders, Use Cases, and Requirements. In Antónia Lopes and Rogério de Lemos, editors, *Software Architecture*, Lecture Notes in Computer Science, pages 155–170, Cham, 2017. Springer International Publishing.
- [30] S. Hassan and R. Bahsoon. Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 813–818, June 2016.
- [31] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.
- [32] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption - a survey among professionals in germany. *Enterprise Modelling and Information Systems Architectures (EMISAJ) - International Journal of Conceptual Modeling*, 14(11):1–35, January 2019.
- [33] Philippe Kruchten. What do software architects really do? *Journal of Systems and Software*, 81(12):2413–2416, 2008.
- [34] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology*, 131:106449, March 2021.
- [35] Gastón Márquez and Hernán Astudillo. Actual use of architectural patterns in microservices-based open source projects. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 31–40. Ieee, 2018.
- [36] Gastón Márquez, Mónica M Villegas, and Hernán Astudillo. A pattern language for scalable microservices-based systems. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–7, 2018.
- [37] Tiago Matias, Filipe F. Correia, Jonas Fritzsche, Justus Bogner, Hugo S. Ferreira, and André Restivo. Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis. In Anton Jansen, Ivano Malavolta, Henry Muccini, Ipek Ozkaya, and Olaf Zimmermann, editors, *Software Architecture*, Lecture Notes in Computer Science, pages 315–332, Cham, 2020. Springer International Publishing.
- [38] Mawal Mohammed and Mahmoud Elish. A Comparative Literature Survey of Design Patterns Impact on Software Quality. In *2013 international conference on information science and applications (ICISA)*, pages 1–7, June 2013.

- [39] John Moses. Should we try to measure software quality attributes directly? *Software Quality Journal*, 17(2):203–213, 2009.
- [40] Sam Newman. *Building microservices: designing fine-grained systems*. O’Reilly Media, Beijing Sebastopol, CA, first edition edition, 2015. OCLC: ocn881657228.
- [41] Felipe Osses, Gastón Márquez, and Hernán 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*, February 2019.
- [42] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. In *Closer (1)*, pages 137–146, 2016.
- [43] Mark Richards. *Software architecture patterns: understanding common architecture patterns and when to use them*. O’Reilly, Febrero, 2015. Oclc: 1047880356.
- [44] Chris Richardson. *Microservices patterns: with examples in Java*. Manning Publications, Shelter Island, New York, 2019. OCLC: on1002834182.
- [45] Richard Rodger. *The tao of microservices*. Simon and Schuster, 2017.
- [46] David Rowe, John Leaney, and David Lowe. Defining systems evolvability-a taxonomy of change. *Change*, 94:541–545, 1994.
- [47] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.
- [48] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, 146:215–232, December 2018.
- [49] 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*, page 1–1, 2021.
- [50] D. Spinellis. The changing role of the software architect. *IEEE Software*, 33(6):4–6, 2016.
- [51] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE software*, 35(3):56–62, 2018.
- [52] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural Patterns for Microservices: A Systematic Mapping Study. In *CLOSER*, March 2018.
- [53] 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)*, pages 181–190, October 2019.
- [54] J. A. Valdivia, A. Lora-González, X. Limón, K. Cortes-Verdin, and J.O. Ocharán-Hernández. Patterns Related to Microservice Architecture: a Multivocal Literature Review. *Programming and Computer Software*, 46(8):594–608, December 2020.
- [55] Markos Vigiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. Microservices in practice: A survey study. *arXiv:1808.04836 [cs]*, August 2018. arXiv: 1808.04836.

- [56] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. Ieee, 2015.
- [57] Hulya Vural, Murat Koyuncu, and Sinem Guney. A systematic literature review on microservices. In *International Conference on Computational Science and Its Applications*, pages 203–217. Springer, 2017.
- [58] Fadi Wedyan and Somia Abufakher. Impact of design patterns on software quality: a systematic literature review. *IET Software*, 14(1):1–17, 2020.
- [59] Claes Wohlin, Martin Höst, and Kennet Henningsson. *Empirical Research Methods in Web and Software Engineering*, page 409–430. Springer-Verlag, 2006.
- [60] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice apis. In *International Conference on Service-Oriented Computing*, pages 73–89. Springer, 2018.
- [61] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang. Microservice architecture in reality: An industrial inquiry. In *2019 IEEE International Conference on Software Architecture (ICSA)*, page 51–60, March 2019.