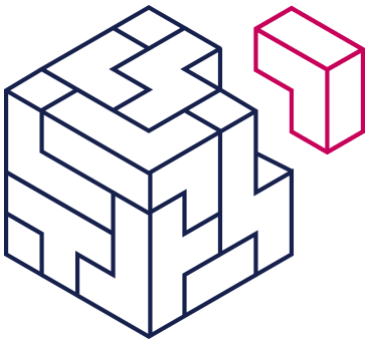
MASTER THESIS

DECISION-MAKING IN A MICROSERVICE ARCHITECTURE

Bjorn Goossens
EEMCS Faculty
Master Business Information Technology

Graduation Committee
Dr. Ir. M.J. van Sinderen
Dr. A.B.J.M. Wijnhoven

External Supervisor
H.W.K. Boenink



27/11/2019

UNIVERSITY OF TWENTE.

COLOFON

DATE
27/11/2019

VERSION
1.0

STATUS
Final

AUTHOR
Bjorn Goossens

E-MAIL
b.goossens@alumnus.utwente.nl

ADDRESS
Postbus 217
7500 AE Enschede

WEBSITE
www.utwente.nl

MANAGEMENT SUMMARY

In recent years, the subject of microservices has gained increasing popularity with software engineering practitioners and academics. The decomposition of system parts into separate services imposes new challenges, many of which related to how to manage the increased complexity in networking and communication between these services. In general, microservice architectures are considered to be confronted with a number of "nontrivial design challenges that are intrinsic to any distributed system" as well as others specific to microservices. Because of microservices' unique characteristics, new challenges arise on how to make microservices *communicate*, *integrate* and be *managed* effectively. A main question for organisations aiming to implement a microservice architecture is how to make well-supported software architecture design decisions regarding these categories of challenges. The aim of this work is to research and design a first step towards a decision-making framework backed by academic literature as well as insights from practice to help organisations better manage these challenges.

Through a structured literature review, an overview of microservice challenges found in academia was constructed. Challenges ranged from purely technical considerations to high-level management-related questions. These challenges were then mapped to the topics of communication, integration and management. The resulting challenges are used as a technical basis for the designed framework. A practical view on these challenges was then gathered through interviews with practitioners at Thales Naval to understand their view on microservice challenges. In general, they recognised the challenges found in literature, though for their organisation some challenges did receive more attention than they did in academia. The categories of challenges that were considered as hardest to manage were also identified, which showed to largely concur with the communication, integration and management challenge categories that were emphasised.

Through searching academic works and sources from practice, challenges were characterised, their dependencies shown, and possible decision alternatives and guidelines were documented. This overview of challenges and their dependencies is used in the framework design as reference for which challenges to consider, and in what order. A comparison of decision-making methodologies for selecting between solution alternatives to these challenges was made. From this, the most suitable methodology was chosen to serve as a theoretical foundation for establishing a decision-making process. Subsequently, the new decision-making framework was designed. It involves making decisions by choosing between solution alternatives and following guidelines where applicable to find the best way for addressing the identified microservice challenges. The Analytic Hierarchy Process was used to support decision-making by making pairwise comparisons; enabling more precise outcomes than competing approaches.

Two single-case mechanism experiments were done to validate whether the goals set for the design were accomplished. The outcome of the first case study suggested that the artifact was somewhat usable and useful, but decision-quality and perceived practicality were lacking. Additions to the artifact in the form of more guidance on what requirements to select and how to weigh them when comparing were made. The second case study showed minor improvements in usability and usefulness, but a significant improvement in decision-quality and perceived practicality. Though it cannot be said definitively that these increased ratings were due to the changes to the artifact, it is expected that a fair conclusion would be that on

average, the participants were neutral to somewhat confident about the quality of the decision outcomes and the framework's practicality. Qualitative observations show that the tooling used during the case studies had the biggest potential for improvement. Nevertheless, the framework, and in particular its use of pairwise comparisons and its contribution to evoking discussion, were viewed favourably.

Through this research, several contributions to academia and practice have been made:

- *Scientific:* A previously not found overview of microservice challenges has been constructed through literature research.
- *Scientific:* Knowledge on decision-making methodologies for software architecture design has been applied to a practical case in the field of microservices, providing insights in the behaviour of such methodologies in this context.
- *Practical:* Academic literature has been used to characterise microservice challenges in a clear and consistent way, providing insights in what challenges can be encountered when designing a microservice architecture, as well as possible decision alternatives and guidelines to consider.
- *Practical:* A previously non-existent decision-making framework to be used for managing microservice challenges in practice has been designed based on an academic foundation.

The findings are mainly limited by the fact that the case studies used for validation included few participants, and not all microservice challenges could be considered during these. Future research should focus on highlighting microservice challenges that were not included in the current design of the decision-making framework, comparing it to other methodologies and translating the decisions made to actual software architecture designs.

PREFACE

With the completion of this thesis, my time as a student at the University of Twente has come to an end. After completing the bachelor programme for Business & IT, I pursued the master programme of Business Information Technology with a specialisation in IT Management and Innovation. Throughout this time, I have always continued to be inspired by the intersection of the fields of business and IT. Bridging the gap between both has been a central theme in my studies and I truly believe that by combining those perspectives, real impact can be made in tackling challenges that businesses face today and tomorrow. In this thesis, I tried to do just that. I am happy with the results of this and inspired by how my findings have been received. Many people have contributed to completing this research, for which I would like to thank them:

First, I would like to thank my supervisors Marten van Sinderen and Fons Wijnhoven. Throughout this research, they have helped me to put my thoughts and ideas into perspective, find ways to approach my work and they have provided me with valuable feedback. Thank you for the interesting discussions we had, and the time you spent on supporting me during the writing of my thesis.

I would also like to thank the people at Thales for their interest in my research and for helping me find insights from practice through many discussions and coffee-machine talks. I am also inspired by their dedication to keep innovating and be at the forefront of using the newest available technologies in IT, despite the challenges that may come with using technologies that are not mainstream yet. In particular, I would like to thank my supervisor Willy Boenink for finding time to guide me during this research in his already full calendar, and his continued motivation to help me complete it. I would also like to specifically thank the Thales employees that participated in the interviews and case studies in this research. The insights from practice that arose from these, helped me in putting my theoretical findings in context.

Finally, I would like to thank my family and friends for supporting me during my studies. Your support has been invaluable throughout the past few years.

TABLE OF CONTENTS

ACRONYMS

AHP	Analytic Hierarchy Process
API	Application Programming Interface
CMS	Combat Management System
CNCF	Cloud Native Computing Foundation
CQRS	Command Query Responsibility Segregation
CR	Consistency Ratio
CS	Combat System
DDD	Domain-Driven Design
DDS	Data Distribution Service
DSM	Design Science Methodology
EQ	Exploratory Question
GDSS	Group Decision Support System
HMI	Human Machine Interface
HTTP	Hypertext Transfer Protocol
IDL	Interface Description Language
MADM	Multi-Attribute Decision Making
MS	Mission System
OS	Operating System
OSGI	Open Services Gateway Initiative
OSS	Open Source Software
OTS	Off-the-shelf
PaaS	Platform as a Service
QA	Quality Attribute
REST	Representational State Transfer
RPC	Remote Procedure Call
RPI	Remote Procedure Invocation
RQ	Research Question
SA	Software Architecture
SaaS	Software as a Service
SOA	Service-Oriented Architecture
TAM	Technology Acceptance Model
UA	User Adaptation

1 INTRODUCTION

In recent years, the subject of microservices has gained increasing popularity with software engineering practitioners and academics. A microservice can on a high level be defined as a "cohesive, independent process interacting via messages" [1]. A characteristic practice in microservices is to decompose a system into small services that are built around business capabilities and communicate through a standardised interface or API [2]. This decomposition of system parts into separate services imposes new challenges, many of which related to how to manage the increased complexity in networking and communication between these services. In a microservice architecture, information that needs to be passed between services is sent over the network connecting these services, rather than accessed in the shared memory of a single application. In general, microservice architectures are considered to be confronted with a number of "nontrivial design challenges that are intrinsic to any distributed system" [3] as well as others specific to microservices. The implications of using this architectural style should be carefully considered when developing a solution.

Because of microservices' unique characteristics, new challenges arise on how to make microservices communicate, integrate and be managed effectively. Challenges in this sense refer to difficulties encountered when developing microservices that need to be overcome for organisations to be able to realise their possible benefits. A main question for organisations aiming to implement a microservice architecture is how to make well-supported software architecture design decisions regarding these categories of challenges. In academia, a straight-forward answer to this seems to be lacking. Even though more and more works on microservices are being published and substantial academic knowledge already exists with regards to decision-making in software architecture design, none of these works seem to combine the two fields. That is; there is currently no common, comprehensive decision-making framework to assist software engineering practitioners seeking to overcome the *communication*, *integration* and *management* challenges of microservices. The aim of present work is to research and design a first step towards such a framework backed by academic literature as well as insights from practice. For any such framework to be of any use in practice, organisations should be willing to adopt it. In its application it should be usable, require limited effort, and preferably offer highly practical insights to organisations.

Part of the practical motivation to conduct this research originates from Thales Netherlands B.V. that mainly develops radar, communication and command & control systems for naval ships. Part of their product range is the TACTICOS Combat Management System (CMS) for combat operations and maritime security, for which a microservice architecture is being considered in its future development. The aim of including Thales in this research is to gain insights from practice to support the academic findings. This way, the relevant research can be compared to real-world scenarios to ultimately better align academic and practical views on the topics at hand.

1.1 Report Structure

This report is structured as follows:

- Chapter 2 describes the research design used in this report, and the design goals as well as the research questions that will be addressed.

- Chapter 3 discusses the problem investigation phase of the design, including motivations for conducting this research, stakeholders and their goals.
- Chapter 4 gives a general overview of the concept of microservices, as well as the challenges that arise in the development of systems involving microservices through a structured literature review and interviews with practitioners.
- Chapter 5 goes into more detail about the challenges relevant to this research, along with identifying possible solution alternatives and guidelines to address these.
- Chapter 6 details the start of the treatment design step in this research, including an overview of decision-making, and selecting a base methodology and requirements for the decision-making framework's design.
- Chapter 7 discusses the design of the decision-making framework in detail.
- Chapter 8 concerns the validation of the designed framework through two case studies carried out in practice.
- Chapter 9 discusses the implications of this research, evaluates their validity and reliability and limitations.
- Chapter 10 concludes this research.

2 RESEARCH DESIGN

Since the main goal of this research is to design and verify a first step towards a methodology to overcome challenges related to microservices, the research resides largely in the field of design science. To structure the design process, the Design Science Methodology (DSM) developed by Wieringa [4] will be used. Wieringa describes a design project in terms of designing an artifact that contributes to stakeholder goals. Such an artifact always interacts with a problem context to produce effects. The so-called engineering cycle as described in the DSM is used to structure the design process. A schematic overview of it is shown in Figure 1.

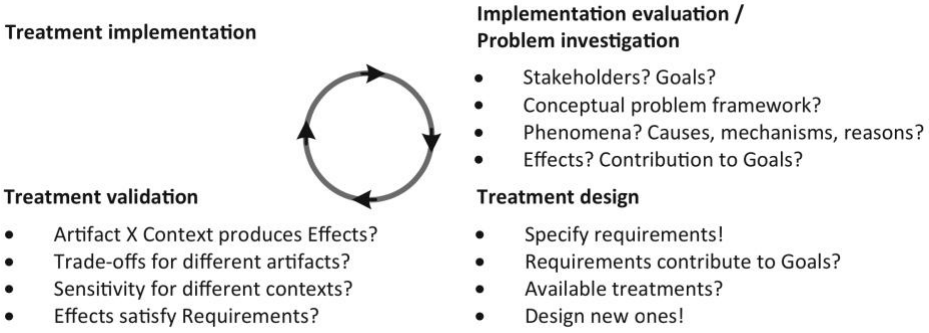


Figure 1 - DSM Engineering Cycle - Adopted from [4]

In this figure, question marks indicate knowledge questions and exclamation marks design problems. As can be seen, in the problem investigation step of the cycle, knowledge questions are used to clarify the problem which serves as input for the treatment design. These take the form of Research Questions (RQs) In the treatment design phase, the actual artifact is designed, of which the effects are then analysed in the treatment validation phase.

2.1 Research Goals and Questions

Wieringa identifies several different goals that a design science research project can have. The overall research goal can be seen as an artifact design goal. Underlying this can be several knowledge goals to describe so-called phenomena and to explain them [4]. A template for formulating design problems is also proposed, which is shown in Table 1. The different parts of this template can be filled in to determine the design goals for this project.

Table 1 – DSM template for design problems, adopted from [4]

Improve <a problem context> by <(re)designing an artefact> that satisfies <some requirements> in order to <help stakeholders achieve some goals>

First is the artefact; i.e. what will be designed. This is the prospective decision-making framework. This artefact interacts with a context, being the design of a microservice software architecture. The interaction between the artefact and the context is captured by the requirement for confidence, effort and practicality. This interaction is useful to fulfil the goals of allowing software architects to better manage the design challenges related to communication between, integration and management of microservices. Putting this all together in the template above, the following design problem results:

Improve *the design of a microservice architecture*

by *designing a decision-making framework*

that gives *confidence in its results,* and satisfies *effort and practicality requirements*

so that *decision-makers can better manage the design challenges related to communication between, integration and management of microservices*

This design problem will be central in this project. To be able to formulate this as a RQ, it is proposed to rephrase this problem as a technical research problem [4]. This can be done as follows:

> *How to design a decision-making framework that gives confidence in its results, and satisfies effort and practicality requirements so that decision-makers can better manage the design challenges related to communication between, integration and management of microservices in the design of a microservice software architecture?*

Underlying this design problem are several open descriptive knowledge questions. The RQs for this project are as follows:

RQ-1 What common design challenges related to communication between, integration and management of microservices can be found in academic literature?

RQ-2 What common design challenges related to communication between, integration and management of microservices can be found in practice?

RQ-3 What are the dependencies between the identified challenges and what possible alternatives and guidelines are available as solutions?

RQ-4 What decision-making methodology for selecting between design alternatives can serve as conceptual foundation for the framework to be designed?

RQ-5 How can the designed framework's fitness for purpose best be validated?

To illustrate the steps taken in this research to answer these RQs, a research model as described by [5] is shown in Figure 2. This model shows which parts of this report address what RQ, as well as how these steps relate to the stages in the aforementioned DSM that is used to structure the design process.

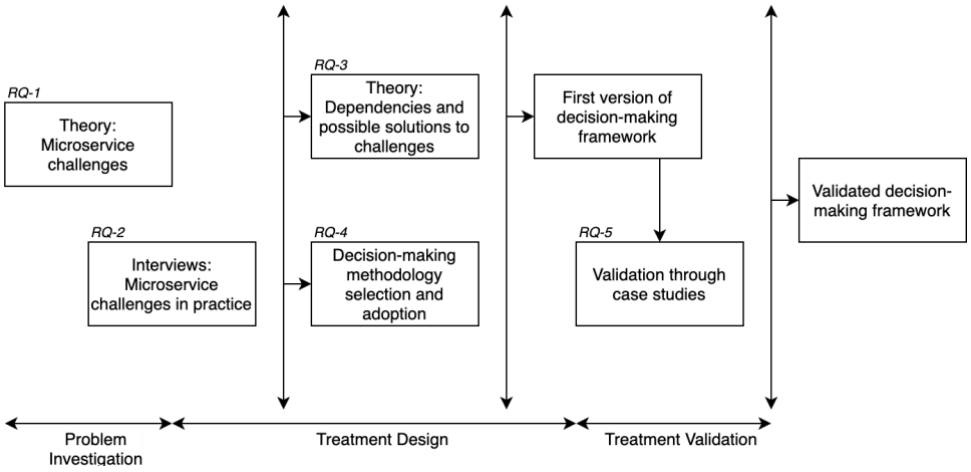


Figure 2 - Research Model as described by [5]

3 PROBLEM INVESTIGATION

To further understand the problem at hand, in this section the stakeholders, goals and context of the problem are analysed. This serves as the first step in the DSM – the problem investigation. This way, the involved stakeholders, their goals, the underlying problem and the motivation behind addressing it can be better understood.

3.1 Motivation and Scope

As said, the motivation to conduct this research originates from both an academic ambition to further investigate and aid in mitigating challenges when designing a microservice architecture, as well as a practical need to improve this process. It is essential to consider what distinctive types of challenges most set apart and complicate the design of a microservice architecture from other architectures to understand why this need for improvement exists.

Part of the practical motivation to conduct this research originates from Thales Netherlands B.V. – part of the worldwide Thales Group; one of the largest defence contractors worldwide [6]. Thales Netherlands B.V. – from here on referred to as Thales – mainly develops radar, communication and command & control systems for naval ships in this field. Part of their product range is the TACTICOS Combat Management System (CMS) for combat operations and maritime security, which is developed by the Thales Naval department. A microservice architecture is being considered in the future development of TACTICOS and is now the focus of Thales Naval for future development of the combat system.

From a practical point of view, Thales Naval asked the following question in the exploratory phase leading up to this research:

"How to do API Management in a Microservice Architecture within the Naval Domain?"

This question can be broken down into three parts. In discussion with infrastructure architect Mr. H.W.K. Boenink regarding this case, several parts were explained in more detail.

"*How to do API Management…*" refers to the expected outcome of the research assignment; a design, guideline or other artefact that describes in what way to manage APIs. An unequivocal definition that can be directly applied to this case seems to be lacking in academic literature. However, its characteristics have been described in technical writings from practice. One organisation describes API management as *"the process of publishing, documenting and overseeing APIs in a secure, scalable environment"* [7]. A report sponsored by Microsoft also mentions several characteristics that API management entails such as API definition, lifecycle management, decoupling APIs from service implementations, facilitating developer use of APIs, securing access and providing analytics and metrics of API use [8]. From this it can be seen that not only the technical workings of connecting and integrating services is considered, but also the use of managerial tools to facilitate the use of APIs. Nevertheless, in practical sources API management is mostly described in the context of organisations providing functionalities to third parties through *web* APIs. Hence, an unequivocal definition that can be directly applied to this case seems to be lacking. In this case, the focus lies on enabling API management functionalities mainly within the organisation itself. Many challenges are similar, though more focussed on the use of APIs within a system or microservice architecture rather than exposing functionalities to the outside world. Therefore, in this setting API management is considered to concern the *approach of managing the design challenges related to*

communication between, integration and management of microservices. Examples of this are how to manage interface design and extensibility, service discovery and communication mechanisms. It is still unclear though which specific challenges are the main focus for Thales Naval in this research.

"…in a Microservice Architecture…" identifies part of the context described in the previous paragraph. The choice has been made by Thales Naval to focus on a microservice architecture for future system development. Even though other architectural styles might also to an extent be suited for this system, microservices are seen as the most promising means to implement a SOA. As stated before, microservice architectures are considered to be confronted with a number of "nontrivial design challenges that are intrinsic to any distributed system" [3] as well as others specific to microservices. The implications of using this architectural style should be carefully considered when developing a solution.

"…within the Naval Domain?" also describes context to be considered in the research assignment. This identifies the setting in which the system is used. The fact that the system is used in the Naval domain has consequences for the requirements of the design. For example, security and latency are possible concerns that are more prevalent in this domain because of its nature. Furthermore, the Combat System can be considered a real-time system [9], because of the inclusion of sensors like radars. This has implications for the required system performance and responsiveness.

Considering these case specifics, it can be seen that some of the challenges that Thales Naval faces might be generalisable, whereas others are unique to their specific domain. Solutions to these challenges might thus also be in the form of generally applicable practices as well as less widely used ones.

The view from literature seems to largely support the notion that most difficulty lies in the communication between, integration and management of microservices. In a seminal work by Dragoni et al. [1], the authors discuss the past, present and future of microservices in the software engineering field. When discussing the impact of current microservice development practices, they touch upon implications for availability, reliability, maintainability, performance, security and testability that arise from microservices' characteristics. Examples of such implications are:

> *"Even if a single service is not available to satisfy a request, the whole system may be compromised and experience direct consequences."*
>
> *"Spawning an increasing number of services will make the system fault-prone on the integration level."*
>
> *"Particular attention should be paid to the reliability of message passing mechanisms between services and to the reliability of the services themselves."*
>
> *"The greatest threat to microservices reliability lies in the domain of integration."*
>
> *"Prominent factor that negatively impacts performance in the microservices architecture is communication over a network."*
>
> *"In order to achieve higher reliability, one must find a way to manage the complexities of a large system"*

Examples of microservice concerns described in [1]

Even though these are just a few of the concerns that are described by the authors, they do support the notion that microservice-specific challenges often touch upon aforementioned communication, integration and management categories. These categories will be the main

focus for the decision-making framework to be designed. The exact challenges that fit in these categories are still unknown though and will be further investigated through a structured literature research and interviews with practitioners.

3.2 Stakeholders

As a means of understanding the problem better, interviews with involved stakeholders will be conducted. For this it is important to first generate an overview of the foreseen stakeholders in this project. In a paper by Alexander [29], several possible stakeholders with their different roles are identified. These roles will be used to identify the stakeholders involved in this project. They also document the use of the so-called Onion Model to represent these stakeholders in different layers. An overview of the possible stakeholders identified based on the initial case description and context is shown in Figure 3.

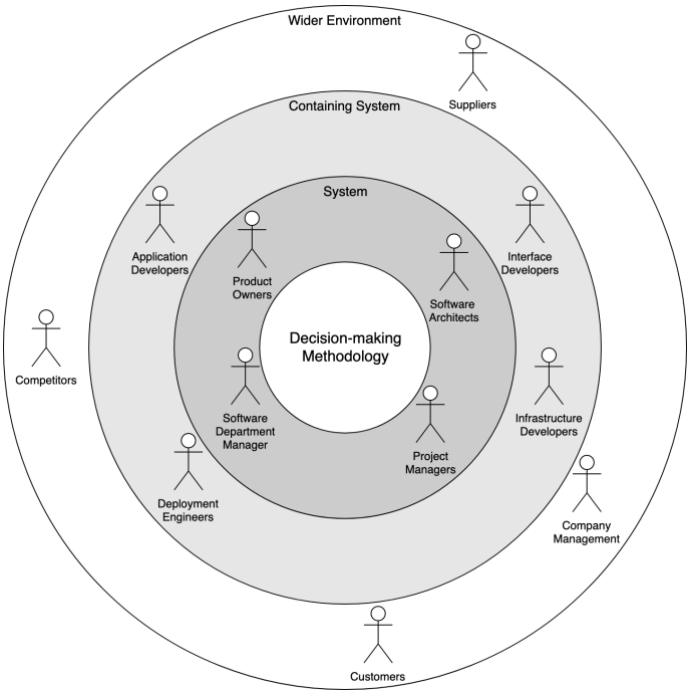


Figure 3 - Stakeholder Overview

The different layers or circles in this figure represent different stakeholder contexts. The inner circle is the artifact under development; the decision-making framework to be designed. The *system* circle contains stakeholders that directly interact with this artifact. The circle after that is the *containing system*, consisting of stakeholders that influence or are influenced by but do not directly interact with the artifact. Finally, there is the *wider environment* circle, to contain stakeholders situated in the environment that the artifact is part of.

The first and foremost stakeholders that will interact with the artifact are software architects. They act as *normal operators* with respect to the artifact and interact with the system on a day-to-day basis. The majority of the decision-making framework's usage will consist of them using it to make substantiated decisions on the topics of communication, integration and management of microservices. The main challenge for them is that there are many ways of implementing a microservice architecture and many tools and software solutions available to help solve this. There is no common or reference architecture that immediately suits their needs, and so they need guidance in deciding which combination of tools to use. They might also have conflicting objectives. E.g. one software architect might prefer a certain solution for communicating between services because it is easy to implement, whilst another architect

could find this solution unsuitable since it makes communication with interfacing systems more difficult.

These stakeholders may, however, not be the only decision-maker in this process; since stakeholders such as product owners, project managers, and a software department manager might also have a say in this process. Such software architecture decisions involve "involve several stakeholders with different knowledge, views, and responsibilities for the system" [10]. There may even be a wide range of stakeholders that influence what decisions are made in this respect. However, they are not always directly present in the decision-making process. Software architects try to capture and understand their needs and translate this to software architecture design decisions. Product owners can also act on behalf of stakeholders to represent their views. Another part of these stakeholders' involvement is when looking back to past architectural decisions and the rationale behind them. The steps taken in the framework can be analysed to retrieve this rationale behind past decisions. Together, these stakeholders in the system layer can be referred to as *decision-makers*. From them, *software architects* are often the most involved stakeholders when making decisions with implications on software architecture. They will therefore be regarded as the primary stakeholders.

The next layer – the containing system – is where the stakeholders that deal with the choices made using the decision-making framework by the immediate stakeholders reside. Their goal is to transform these architectural decisions into a working system. The stakeholders shown in Figure 3 are the most notable ones. Their role in this overview is that of *functional beneficiary* as they benefit from the output of the system. When more effective architectural decisions are made, they get better input for their work.

In the wider environment are stakeholders that do not interact with the artifact directly but are influenced by it. For instance, an organisation's management will benefit from a more effective design of a software architecture by being able to offer better products to their customers, more quickly delivering these or having a more future-proof product to sell. They act as sponsor and financial beneficiary as the changes to the architecture will possibly put the organisation at a competitive advantage. These advantages also influence suppliers, that for instance might need to integrate with a system produced by the organisation. Customers will on their turn receive a better solution, and as such act as political beneficiary. This is in contrast to an organisation's competitors, that are a negative stakeholder in this context.

Knowing which stakeholders are involved with this case helps to understand what topics affect whom and how their different views on the same topic might differ. This supports formulation of the research goals for this project.

3.3 Decision-Making Framework Goals

A sound decision-making framework should support decision-makers' work in the best way possible. A software architect's work plays a vital role in software development, as they are the main stakeholders in this case. Even though not all design decisions are by definition purely software architecture decisions, these do make up a large part of them. It is therefore useful to consider how software architecture decisions are made. Some insight in this is given by Falessi et al. [10]:

> *"Architectural decisions are crucial to the success of a software-intensive project. […] Therefore, software architects need a reliable and rigorous process for selecting architectural alternatives and ensuring that the decisions made mitigate risks and maximize profit."*

The authors explain several main influencing factors on software architecture design that are addressed by decision-making methodologies – or as they call it; techniques. They should deal with multiple stakeholders, competing and conflicting objectives, uncertainty both in the descriptions of requirements and in their associated solutions, and interdependencies between decisions [10].

By following a predefined process or structure to generate architectural decisions, one can have more confidence in the choices made during software architecture design. This is shown by the need for a *reliable* process. This also means that the process should be repeatable with similar results. The fact that a process should be rigorous shows that it should be comprehensive; all important aspects should be considered. The authors' view of a good decision-making technique is one that "guides the user toward better, perhaps optimal, alternatives, and, at the same time, is easy to use" [10]. There are a few parts about this quote that have implications for the decision-making framework to be designed. First, a main goal is to select between *alternatives*. This means that the alternatives to be considered should be discovered beforehand. Second, it cannot always be guaranteed that an *optimal* alternative will be found. 'Optimal' is defined by certain quality attributes and desired properties per alternative. Finally, there is a possible trade-off with *ease-of-use* in selecting an alternative. A technique that is hard to use, will likely not be utilised by software architects in practice.

In discussions with several Thales Naval employees working in the software architecture design field about the challenges that they face and goals during decision-making, many of the aforementioned characteristics could be recognised. A main desire that was indicated for such a framework was that it should be *practical*; focused at the actual software architecture design work rather than only concerning theory and ideas. In line with this were comments that the process should be understandable; software architects should be able to easily understand how a certain decision came to be. The output should be of direct value to software architects. In line with this, almost all employees saw a *limited effort* to use a framework as vital for its success. Furthermore, the used methodology should give *confidence* in its results. As one employee noted; that is not to say that the outcome should always be the perfect combination of alternatives but give alternatives as input to confidently make decisions on architecturally significant aspects. Many practitioners noted that it is to be expected that the process of synthesizing a software architecture is iterative, as decisions made in a later stage can change their view on those made earlier. More notably, most respondents felt that the main goal of any framework to help in decision-making should not be to impose too strict of a process or way of working, but rather guide the discussion on the challenges that are encountered when designing a microservice architecture. Having a predefined set of steps, clues and checks to use during this design process is seen as possibly being of great help.

Nonetheless, not all challenges are purely of a software architectural nature, and some also do not have clearly distinguishable alternatives to choose between. Therefore, some challenges may be better addressed by defining guidelines or finding industry practices. An example of this is deciding on *service granularity*; i.e. the 'size' and scope of a microservice. This challenge will be discussed further later on. This challenge is not solved by for example defining how many lines of code or function points a microservice should be. There are, however, guidelines and techniques to decide on this. A goal of the framework is to also provide guidance in solving these challenges; where no clear-cut decision can be made, but solution guidelines or common practices are available.

Also note that in order for the prospective decision-making framework to be useful, decision-makers need not necessarily have already encountered a problem during the design of a microservice architecture before applying it. The goal is to help practitioners from the

beginning of the design of a microservice architecture by showing the challenges they may encounter and hopefully address these before they become a problem in the first place.

All this gives rise to high-level goals of the decision-making framework to be designed. Whereas the DSM [4] does not yet prescribe to define requirements in this problem investigation stage, these goals can already be seen as goal-level requirements in terms of the goal-design scale as described by Lauesen [11]. They are shown in Table 2. These goal-level requirements are each assigned a number G(n) for future reference.

Table 2 - Decision-Making Framework Goals

	Goal-level requirements
G1	The framework shall improve decision-makers' work in managing design challenges related to communication between, integration and management of microservices.
G2	The framework shall give confidence in its outcomes.
G3	The framework shall require limited effort in its use.
G4	The framework and its outcomes shall be practical.

These goals help in filling out the DSM template by showing which aspects are to be addressed by the designed framework. Of these goals, **G1** is the main goal that should be contributed to by the future artifact designed in present research.

3.4 Problem Overview

To understand how all the facets of this assignment interact, a problem overview as described in [12] was created. This is shown in Figure 4. The aim of constructing a problem overview like this is to show the core problems that needs solving. The image illustrates how two sides of the preceding research come together to form the single question of how to design a microservice software architecture. This question arises from a *lack of knowledge* on how to solve the challenges related to communication, integration and management.

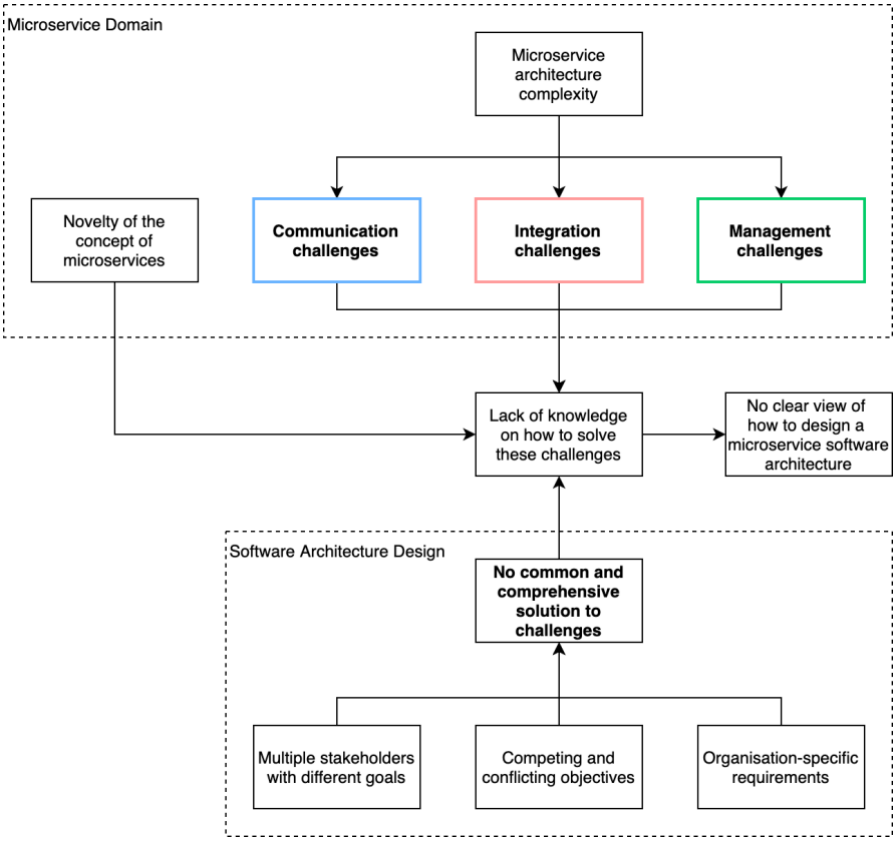


Figure 4 - Problem Overview

For a start, there are the microservice challenge categories. These arise from the inherent complexity of a microservice architecture as described. A further complication is the novelty of the concept of microservices, which results in little academic literature being available on the topic. Because of this novelty, there is also no de-facto way of dealing with such challenges when constructing a microservice architecture. This contributes to the stated lack of knowledge.

The intricacies of software architecture design also contribute to the existence of this problem. Software architects have to deal with multiple goals of different stakeholders that quite often are not aligned. Conflicting objectives and interdependencies between choices further complicate their work. Finally, there are often also organisation-specific requirements that need to be taken into account. These challenges contribute to the fact that there is no common and comprehensive solution to overcome the challenges when designing a microservice software architecture.

Heerkens [12] describes how the core problems can be found by tracing problems in this overview back to their roots and selecting problems that can be changed and are expected to have a sizeable impact on those further down the chain. When tracing back the core problems related to microservices, it seems clear that the inherent complexity to a microservice architecture cannot directly be solved. However, an attempt can be made to investigate the challenges on communication, integration and management to improve the knowledge on how to solve them. Therefore, these are seen as the three first core problems and are shown in bold text in the overview. As for the novelty of microservices; this will change in the future as more research is done but cannot be directly influenced by this research. The software architecture design specific requirements can also not easily be mitigated since these will expectedly always exist. Nevertheless, it can be studied how the complex system

requirements influence the design of a microservice architecture, and in what way they interact with the challenges in microservices. Therefore, the fact that there is no de-facto solution to solve architecture design challenges this chosen as the fourth core problem. This is also shown in bold text in Figure 4.

4 MICROSERVICES

In this section, the characteristics and challenges of microservices as described in academia are discussed. The goal of this is to understand what defining characteristics microservices have, and which challenges come with this. The overview of challenges from literature and practice will serve as the solution to RQ-1 and RQ-2.

4.1 General Overview

Microservices are a relatively new concept in the software engineering field and even more so in the academic research field on this subject. In 2014, Fowler and Lewis formalised microservices as an architectural term by describing common characteristics and industry practices [2]. These characteristics gave a first notion of what was considered a microservice at the time. In one of the first books published on the design of microservices, Newman [13] described microservices as “small autonomous services that work together”. A few years later, Dragoni et al. [1] wrote an article about the past, present and future of microservices in which they proposed to define a microservice as “a cohesive, independent process interacting via messages”. They describe a microservice architecture relatively straight forward as “a distributed application in which all its modules are microservices” [1].

In the early days of software engineering, many software systems used to be built as a monolithic system whose modules cannot be executed independently [1]. Over time, several approaches emerged to decompose systems into smaller parts, such as object-oriented programming or component-based software engineering. A Service-Oriented Architecture builds upon these concepts by using making application components provide services to and consume services of other components over a network interface. Although this statement is debated, microservices can be regarded as a particular implementation approach to SOA. In 2016, Zimmermann published a comprehensive analysis comparing the characteristics ascribed to microservices with SOA principles and patterns [3]. Through a viewpoint-based analysis they support the position that “microservices are not entirely new, but qualify as “SOA [implementation] done right”. More precisely, microservices comprise an organic implementation approach to SOA (just like Scrum is one, but not the only way to practice agile development).” The emphasis on ‘SOA done right’ is central to this, as they further explain how “microservices implementations have the potential to overcome the deficiencies of earlier approaches to SOA realizations by employing modern software engineering paradigms and Web technologies” [3].

A further indication that the microservices architectural style is currently undergoing a reality check comes from the Gartner Hype Cycle industry reports for the application architecture domain [14]. In these reports, Gartner uses their so-called Hype Cycles to help organisations and practitioners “get educated about the promise of an emerging technology within the context of their industry”. Their reasoning is that innovations tend to follow a predictable path of expectations over time in which their expectations are first inflated, they then move through a *trough of disillusionment*, after which they move onto mainstream adoption. In the report of 2015, microservices first appeared and were immediately considered to be at the *peak of inflated expectations*. This is not to say that they came out of thin air, but probably likely reflects the fact that only in 2014 the term was formalised to refer to this concept. Microservices stayed at this peak in the years 2016 and 2017. However, in 2018 they are first seen as on their way of “sliding into the through [of disillusionment]” [15]–[18]. A graphic representation of the

Gartner Hype Cycle including indications of where microservices were in 2015 through 2018 is shown in Figure 5.

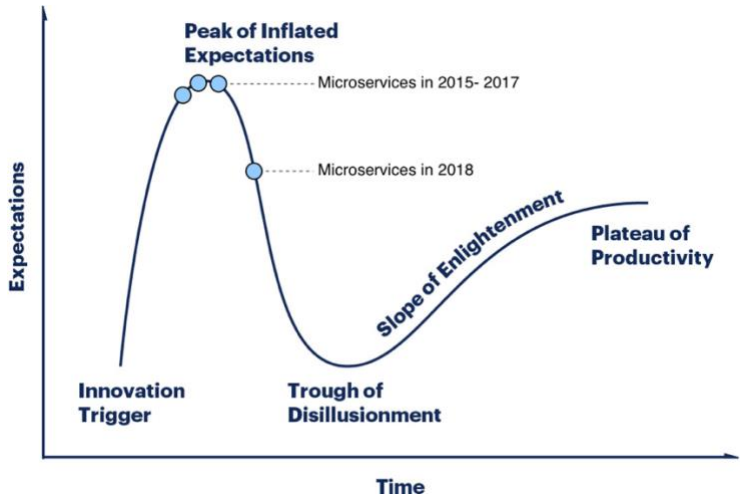


Figure 5 - Microservices on the Gartner Hype Cycle - Base image adopted from [14].

All this can be seen as supportive of the notion that microservices are a new and seemingly somewhat hyped approach to implementing SOA-like applications and architectures while using modern technologies. This view is also taken in this research. Using this viewpoint allows one to see microservices in perspective, and possibly use academic literature related to SOA as a secondary source of knowledge to gain a better understanding of their traits.

4.1.1 Characteristics

The different terms used in the aforementioned definitions imply certain characteristics that a microservice will typically possess. One of the most fundamental characteristics of microservices is that of componentisation via services. Even though a monolithic application can use some form of componentisation, “all too often these arbitrary in-process boundaries [i.e. boundaries between components] break down” [13]. According to Fowler and Lewis, “over time it's often hard to keep a good modular structure, making it harder to keep changes that ought to only affect one module within that module” [2]. Because of this, there are limits to the flexibility with which an application can be built or changed. By using services as a means of componentisation, these difficulties can be reduced.

Microservices are meant to be a separate entity. This enables microservices to be deployed in isolation and independent of other services, for example in containers or another type of platform as a service (PaaS). Newman states that “services need to be able to change independently of each other and be deployed by themselves without requiring consumers to change” [13]. This refers to ‘*autonomous’* in the definition by Newman [13] and ‘*independent’* in that of Dragoni et al. [1]. The design of APIs by which services communicate is an important factor here. The aim is to achieve a high degree of decoupling [2]. Newman again refers to the question of whether changes in a service can be made and deployed without changing other services [13] to show that decoupling is imperative to get microservices right. Often, communication between services is done by means of RESTful API requests. This is reflected in the definition by Dragoni et al. [1] in the part ‘*via messages’*.

What makes microservices ‘micro’ is the scope that a single service encompasses. Microservices are relatively small compared to a typical service found in a SOA. The focus is on having each microservice provide a single business capability and be as cohesive as possible [1], [2]. This reflects the characteristics ‘*small’* and ‘*cohesive’* in the definitions by

Newman and Dragoni et al., respectively [1], [13]. To further explain the term *'cohesive'*, Dragoni et al. state that this indicates that "a service implements only functionalities strongly related to the concern that it is meant to model" [1].

Furthermore, services can also be composed of other, often smaller, services. For example, a service providing customer data could in turn request a user's profile image and other details from two different underlying services. Another example is that of displaying a product page in a web shop. The service serving the page might invoke separate services to provide product images, prices, specifications, reviews and so forth. The part '*that work together*' from Newman's definition [13] reflects this. This also touches upon the characteristic of decentralised data management that microservices often incorporate. Whereas a monolith would often use a single database for data storage and retrieval, microservices regularly make each service manage its own database [2]. This way, multiple applications that might require customer data can request this data from the same service. Also, since data is requested through a service's interface, the consumer of this data is not concerned with the underlying database system. These concepts are neatly depicted in Figure 6 by Richardson [19], a microservice practitioner and advocate.

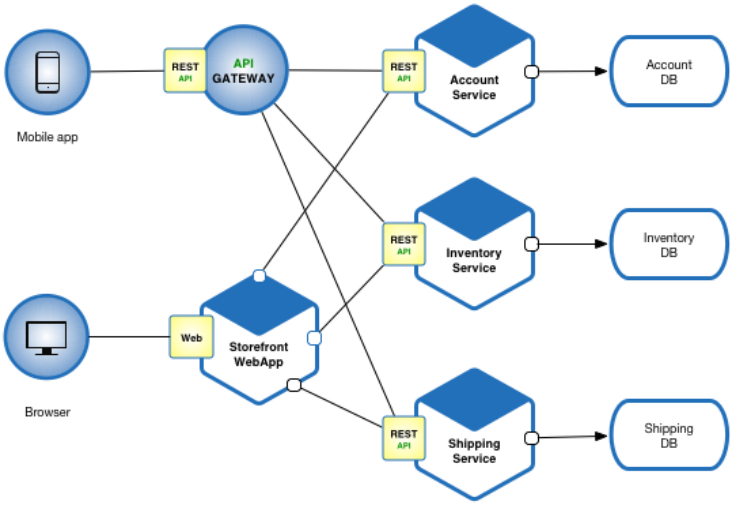


Figure 6 - Example microservice e-commerce application. Adopted from [19]

4.1.2 Motivations

For the same reason of a system being divided into microservices and data being requested through a service's interface, a consumer is often also unconcerned about the inner workings of a service. Or as described before; interfaces should be decoupled from the programming logic. Therefore, developers have a choice of programming language, styles and standards to build a service as fits best. Services that could have specific advantages by using a certain programming language are free to use this even if none of the other services use this language as long as they provide a valid response to a request. Fowler and Lewis do indicate though, that "just because you *can* do something [i.e. use various programming approaches], doesn't mean you *should"* [2]. Sharing useful tools to battle similar problems between services can be of great value, but the door is left open to choose another approach.

This independence of microservices also helps resilience of a system. Whereas an application error could make an entire monolithic system crash, if a microservice fails this should not directly impact other services, enabled by the aim for de coupling. Nevertheless, the unavailability of a single service should be properly handled to prevent cascading failures [13].

Microservices can furthermore be scaled independently. This means that when one service requires more resources than another, these can be allocated to those specific instances. This elasticity is especially advantageous when using cloud platforms such as Amazon Web Services[1] in which pricing is done based on dynamic resource usage.

4.1.3 Deployment

Microservices enable new ways of application deployment. In this scenario, a single service can be upgraded without directly affecting other services. This is a vast change from a typical deployment of a monolithic application in which "a one-line change […] requires the whole application to be deployed in order to release the change" [13]. Because of this, changes are often grouped for a planned release once in a while. Microservices enable continuous delivery and continuous integration [1], which in practice means that changes can be released more frequently.

A common practice in deploying microservices is to use containerisation solutions like Docker [20] as a deployment model, since they naturally lend themselves to this [1]. A service runs in a container that isolates it from the underlying operating system and hardware. This use of containers enables the aforementioned benefits of deploying and scaling parts of a system. Furthermore, by using containers, separation of different services is realised. For example, it tackles the problem of 'dependency hell' [20] that can arise when libraries or frameworks that services are built on top of are shared or missing. Finally, containers are portable and can be moved from one system to another. In the example of Docker, a container will run on any system that supports the Docker platform.

Moreover, microservices enable more flexible ways of deploying new functionality to a production environment. Therefore, downtime and service interruptions can be minimised or in some scenarios almost eliminated. One example is part of Netflix's delivery pipeline as described on their tech blog by Ben Schmaus [21]. After testing of a service when changes have been committed, they first deploy it as a so-called canary. In a canary release, new code (the canary) is run on a small subset of production infrastructure and is then compared to the old – baseline – code [21]. When the canary is considered to work as expected, it is then further deployed on the production infrastructure. Because microservices are independent entities, this can be a gradual process. Containers with the new code can be instantiated, after which a load balancer can seamlessly direct more and more traffic to this new version of a service. This way, downtime is mitigated.

4.1.4 Considerations

As is to be expected, microservices are no universal solution – or so-called 'silver bullet' – for all types of applications. Depending on the type of system that is being developed, choosing for a microservice architecture might introduce a substantial amount of complexity. This is because a microservice architecture comes with all the associated complexities of a distributed system [3], [13]. Therefore, in some cases – especially in less-complex systems – it might make more sense to build a monolithic system as less productivity is lost to the so-called 'microservices-premium' [22]. Furthermore, any organisation planning to run a microservices system successfully, should have the knowledge, tools and culture to support it. Prerequisites for this have also been described by Fowler and Lewis in [22].

[1] https://aws.amazon.com/

Fowler also goes into more detail about the possible trade-offs in developing microservices [23]. Complexity due to distribution is also seen as a main concern. Fowler states that “as soon as you play the distribution card, you incur a whole host of complexities”. Challenges of performance, asynchrony and reliability can come into play. Consistency issues are seen as a further hindrance, as is operational management of the typically many microservices.

Many of microservices’ advantages stated are also not straight-forward to realise. For example, the flexible ways of deployment that microservices allow must still be designed, implemented and managed properly by organisations. As Newman puts it: “if you’re coming from a monolithic system point of view, you’ll have to get much better at handling deployment, testing, and monitoring to unlock the benefits [of microservices]” [13].

This all demonstrates the decision to newly develop or embark on a transition towards a microservice architecture should not be taken lightly. There might be reasons to just stick with a monolith and in other cases the organisation requires just as much redesigning as their system’s architecture.

4.2 Challenges in Literature

As stated before, deciding to use microservices in a system’s architecture comes with certain challenges that are not straight-forward to solve. To identify what challenges exist, a systematic literature review was conducted. The aim is to answer RQ-1; *What common design challenges related to communication between, integration and management of microservices can be found in academic literature?* The literature review aims at giving insight in the academic part of this question.

At first, the search for challenges will not yet be limited to the challenge categories in RQ-1. This means that all the challenges found in the searched literature are considered and determining which of those are in and out of scope is done afterwards. This is done to not possibly miss any challenges of interest by overly restricting the search criteria. Since this report is aimed at formulating a research proposal any proposed solutions to the challenges found will not yet be considered in-depth. The focus is first on understanding the academic context of what challenges are commonly encountered when implementing microservices, to get a broad view of the problem. This provides a background to appropriately position the final research project.

4.2.1 Literature Review Process

The challenges will be identified using a systematic literature review. The PRISMA guidelines [24] are used to structure this process. Scopus[2] will be used as primary search engine for searching the top 25 IS journals described in [25]. Two of these top journals are not indexed by Scopus, namely Journal of MIS and Communications of the AIS. These will be searched independently.

The query used to search publications are as follows:

TITLE-ABS-KEY(microservice OR “micro service”) AND PUBYEAR > 2013

The TITLE-ABS-KEY field code indicates that the title, abstract and keywords of a publication should be searched. The key terms *microservice* and *micro service* are sufficient to select all publications because Scopus automatically ignores punctuation and includes plurals and

[2] https://scopus.com

spelling variants. Similar approaches were taken when searching the two journals not covered by Scopus. The results were limited to publications from 2014 onward, as the term ‘microservices’ is only consistently used since then [26]. This is reflected in the PUBYEAR constraint. Online sources, books, theses, talks and presentations were excluded for the literature review in order to keep a consistent and comparable view of microservices as a research area.

To first assess the usefulness of the found publications, their titles, abstracts and keywords were read. Based on the contents of these, the decision whether or not to consider a publication was made based on the selection criteria outlined in Table 3.

Table 3 - Selection Criteria

	Criteria
Inclusion	• Abstract or keywords include key terms. • Studies indicating a contribution towards discussing challenges in communication between and integration of microservices. • Studies focussing on specific challenges in microservices. • Studies describing challenges in distributed systems and how they relate to microservices.
Exclusion	• Studies using key terms but not referring to the microservices architectural style described in section 4.1 • Studies only in the form of abstracts, workshops or presentations. • Studies that do not have microservices as their primary research topic or analysis. • Mere mapping studies. • Studies primarily describing a solution design without explicitly discussing challenges in microservices. • Studies not written in English.

After this first pass, 43 studies of potential interest were selected. Then a backward search was done to identify studies that were referenced in those already found. This resulted in an additional 8 studies to be considered. For these 50 studies, the full text was screened to assess whether their contents matched the expectations after the first selection. Ten studies were removed from the selection for several reasons. Some turned out to not make an effort to discuss challenges in microservices. Others did not support the claims that were made about such challenges. A few studies from lower-ranked journals were of questionable quality and also lacked in evidence supporting claims. This resulted in a selection of 40 studies. An overview of the selection process is shown in Figure 7 and the full list of selected publications is available in Appendix A.

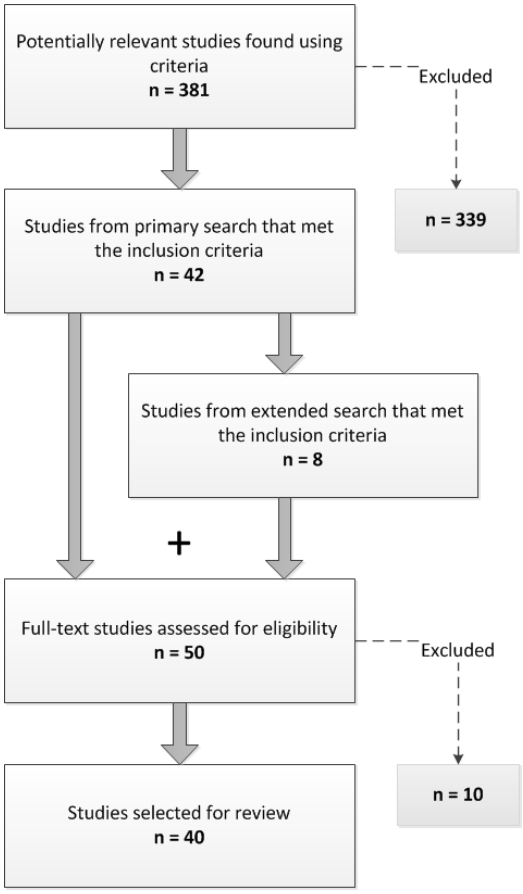


Figure 7 - Study selection process

Next, all publications were screened and parts discussing challenges in microservices were identified. These parts were then thoroughly read to find which challenges were discussed. This was done by assigning *keywords* to them and creating a list of these. For example, discussions on 'service discoverability' and 'how to discover services' were given the keyword *Service Discovery*. This was first done to find all keywords that appeared in the selected publications. After this, a second pass of reading was done to identify any challenges that might have been missed now that the list of keywords was complete. These keywords were then classified into eight categories to indicate the high-level topics that they mainly concern. An overview of this classification along with the amount of times a keyword occurred in the searched literature is shown in Table 4. Whereas some challenges might fall into multiple categories, the choice was made to assign them a main category for clarity. Furthermore, the 'performance' category could logically be seen as a subset of 'quality'. However, because many studies specifically discuss microservices' performance, this category was added to emphasize the prevalence of these challenges in literature. Appendix B contains Figure 44 visualising the occurrence of these different keywords and Figure 45 showing the prevalence of the different categories based the occurrences of their corresponding challenge keywords. When referring to one of the studies, a number sign following the corresponding entry in Table 4 are used in brackets. E.g. [#1].

Table 4 - Challenge Categories and Keywords with Occurrence

Service Management	#	**Quality**	#	**Performance**	#	**Organisation**	#
Service granularity	13	Fault-tolerance	7	Network performance overhead	6	Organisational structure and culture	5
Service logging and monitoring	11	Security	4	Memory consumption	3	Team composition	5
Service management	5	Reliability	3	Quality of Service	3	Legal responsibility	1
Service evolution	4	Maintainability	2	Performance isolation and characterisation	2		
Service boundaries	3	Availability	1				
Communication	#	**Service Development**	#	**Deployment**	#	**Development Process**	#
Communication mechanisms	11	Service orchestration	12	Automated deployment mechanisms	8	Monolith migration	11
Service discovery	9	Service composition	11	Deployment methods	7	Integration testing	5
Interface design	6	Data distribution	8	Service scheduling	7	Development process	3
Service interconnection	5	Number of services	3	Resource management	4	Resilience testing	3
Networking complexity	3	Service specification	2	Multi-cloud deployment	1	Development setup	1
Service contracts	2	Statefulness	2				
		Multi-tenancy	1				
		Service choreography	1				
		Vendor lock-in	1				

4.2.2 Results and Discussion

While Table 4 gives an overview of the challenges that were mentioned in the selected studies and which of these are most prominent in each category, this is just quantitative data. The identified keywords concern challenges with many different subtleties to them. In this section, the challenges in each category are discussed with regard to their meanings, viewpoints from the searched literature, as well as their possible implications on the development of microservices.

Service Management

A central concern in any microservice architecture is *Service Management*. In general, this category contains challenges concerning high-level reasoning about services. The most abstract keyword found in this category is identically named *service management*. This basically entails the supporting activities to guide the design, development and delivery of services. This is especially important because a microservice architecture inherently consists of many individual services. The studies that identified this as a challenge, mentioned several activities related to it such as the creation of an “enterprise wide service repository of all the microservices specifications” [#39]. Service management is also seen in the sense of lifecycle management [#8, #39], which ties in with *service evolution*. As indicated in [#29], “Determining compatibility and consistency between microservice versions is a continuous challenge for developers”. When developing microservices, this should be done in a way that allows for future changes to these services. Put differently, “a system should stay maintainable while constantly evolving and adding new features” [#7]. Challenges in operational management of

services are mainly expressed as *service logging and monitoring* concerns. The complex interactions between many microservices can be hard to comprehend, therefore making it harder to trace errors and understand the overall system health. Collecting logging and monitoring data from individual services is relatively straight-forward, but aggregating this data to create meaningful, actionable insight is far from it.

A step above all of this are the challenges concerning *service granularity* and *service boundaries*. As stated before, microservices are meant to be developed around business capabilities [6]. However, translating this guideline to practice often proves to be difficult. Granularity of services refers to "the trade-off between size and number of microservices" [#5], while s*ervice boundaries* concerns the decision of what functions are in and out of scope for a certain service. Questions of how to divide functions between services, how many functions a service should offer and whether to subdivide certain functions into more, even smaller functions are seen as hard to answer in practice by organisations. This is reflected by service granularity being the most mentioned challenge in this overall literature review.

Quality

As with any software system, there are several *Quality* requirements that must be satisfied before it can be used in production. Systems consisting of microservices do, however, impose new challenges to managing these requirements. *Fault-tolerance* is the most common challenge discussed in the searched literature. A fault-tolerant system is one that keeps functioning even in the presence of failing parts such as unresponsive services. As mentioned in [#7], "even if a single service is not available to satisfy a request, the whole system may be compromised and experience direct consequences". It is also stated that "spawning an increasing number of services will make the system fault-prone on the integration level". Similar concerns were raised in other studies discussing this challenge. Also, because of the generally complex interactions between different microservices, there is a possibility of cascading failures. In this case, if a single service fails it can trigger the failure of others. Failure of components should be foreseen and even embraced during the design and development process of a microservice system, so that the focus shifts from *how to prevent failures* to *how to best deal with failures* when they will unavoidably occur. *Reliability* and *availability* are both related to *fault-tolerance*. An exemplary definition of these terms and explanation of their differences is given in [#8]: "Reliability is not to be confused with availability which has a slightly different meaning, indeed an application may be able to answer requests (it is available), but its responses are not what should be expected for such requests according to its specification (it is unreliable)". They also state that "an application cannot be reliable if it is unavailable, which means that reliability combines both constraints and is thus a stronger requirement on a service" [#8]. The challenge of maintaining a highly reliable and available system is complicated by the fact that microservices are often deployed in a cloud environment, which is naturally unreliable [#8].

Security is another concern that should be considered very closely. The complexity of a microservice system "can result in security vulnerabilities affecting one or more vectors in the architecture" [#9]. Because the amount of services and connections between them, the amount of attack vectors in the system greatly increases. Part of the security concerns is how to deal with trust. As also stated by [#9], in a typical microservice architecture "there's a real risk that attacks can be easily propagated due to the microservices' dependencies and (blind) trust of peer components". In other words, if one service would be compromised but all other services trust it nevertheless, the entire system is vulnerable. As mentioned in [#7], "Considering a microservice trustworthy represents an extremely strong assumption in the "connectivity era", where microservices can interact with each other in a heterogeneous and

open way". They proceed to discuss an example of the Netflix infrastructure being compromised because of a single vulnerability in a subdomain.

A final challenge is that of *maintainability*. Because of the aim for loose coupling between microservices, a microservice architecture should inherently be more maintainable than a 'traditional' monolith. However, it is still completely possible to write microservices that despite this characteristic are hard to maintain by for instance writing obscure and counterintuitive code [#7]. The lesson here is that even though a microservice architecture enables the development of more maintainable services, this trait does not come 'for free'. In line with the recently popular DevOps movement [27], a useful tool to manage maintainability is the use of the "you build it, you run it" principle [28].

Performance

There are several challenges related to system *Performance* in a microservice architecture that have been identified in literature. *Network performance overhead* is the most prominent one. Microservices mostly talk to one another over a network interface. The overhead comes from the fact that because a microservice architecture consists of many small services, all calls to different parts of an application – other services – happen over the network. Previously these calls would be in-memory calls to other application components of the same application. Because accessing a server's memory is significantly faster and happens with less latency than having to communicate over the network, this overhead can cause performance issues. A real-world example of this is given in [#36], in which the authors deploy the same application as both a monolithic and microservice architecture. They reported having observed "a significant overhead due to the microservice architecture". In [#19] the link is made between service size and network communication overhead. They state that "communication between multiple microservices can introduce performance issues if the services are too fine-grained". The logic is simple, when a service is decomposed into more smaller services, more communication between parts of this aggregate service happens over the network. *Memory consumption* is also mentioned in [#36] as being higher in the microservice setting compared to a monolithic system. Part of the reason for this is the need for each service to have its own memory address space [#24]. The overall *quality of service (QoS)* is also seen as being harder to manage when for instance using the common HTTP protocol for communication. Many of the protocols often used for communications between microservices lack QoS guarantees with regards to transaction management, atomicity, reliable delivery of messages, once-only delivery and broadcast event-driven actions" [#39]. A final performance concern identified in literature is that of *performance isolation and characterisation.* This challenge is raised because of the de facto way that microservices are deployed; using containers. Compared to using virtual machines to deploy services, containers give less performance overhead. However, because multiple containers can run on a single host and thus share resources, performance isolation is weaker. That is, when different types of microservices are run on the same physical machine, they might differ in their performance needs. In [#10] the example is given of one microservice having dominating storage requirements, whereas another service might be more computation or communication intensive. They state that "balancing resource consumption and performance is critical in deciding where to deploy microservices".

Organisation

It is always important to consider the context in which a microservice architecture is developed. This is where the *Organisation* category comes in. Apart from the technical challenges related to microservices, there are organisational aspects that have to be managed to either support the development of microservice architectures in the first place, and to deal with the

requirements that operating and maintaining such an architecture involves. For a start, having an *organisational structure and culture* that is in line with the work practices commonly attributed to developing microservices is seen as vital to successfully do so. In [#19] a quote from Newman [13] is used to illustrate this: “In order to develop good application, the organization must align their structure with the structure of the application architecture”. They also mention Conway’s law which states that “the organisation which designs the system will produce a system which structure is a copy of the organisations structure” [29]. This is in line with [#27], in which a practitioner claims that “domains evolve or are given by the organizational structure, which eventually matches the system structure”. A common way of creating an organisational environment suitable for microservice development is the adoption of DevOps practices. Part of the discussed structural alignment is the *team composition* of development teams. It is stated that cross-functional teams are vital for the development of a microservice architecture to be successful. Previously an organisation could have teams each specialised in a certain area such as database, UI or backend. However, given that microservices are built around business functions, it makes sense to build teams to focus on certain domains such as finance or accounting that consist of developers with all the required specialisations to build and also operate the services.

A further organisational challenge identified by the searched literature is that of *legal responsibility*. In [#9] this challenge is described in the light of security concerns regarding microservices. They state that because there are more attack vectors when operating a microservice architecture and because these are commonly deployed in a cloud environment – often at a third-party supplier – it is unclear who is responsible for a cybersecurity incident under the current legal framework. Even though this is a real concern, this is the only study eliciting this concern, and it is questionable to what extent it influences the actual development of microservices.

Communication

Because of their nature, *Communication* between services is a fundamental part in microservice architectures, or any distributed system for that matter. Microservice architectures generally consist of many services, all connected over the network. A first and rather general challenge that was mentioned in literature is the *network complexity* that this brings. The most important challenge related to communication is the choice of *communication mechanisms.* The question of how to conceptually deal with system-wide communication on a high-level is one that transcends and precedes any technology choices. The decisions made with regards to this should be well-considered as they affect the entire system. At this level, decisions such as the choice for synchronous versus asynchronous communication or request-response versus publish-subscribe patterns are made. In an aim to obtain a high degree of decoupling of microservices, asynchronous communication is mostly seen as preferable. However, as stated in [#12], “interestingly, microservices are most suitable for asynchronous communication, bringing performance, decoupling and fault-tolerance, but the paradigm shift implied has not been overtaken yet in practice”. Another implication of microservices to the choice of a communication paradigm is that one “should also take in to consideration the possibility that a service might not respond” [#19]. The authors in [#24] discuss choices of inter-service communication patterns that are common in microservice architectures. The three that they mention are direct calls to services, using a gateway or using a message or service bus, each with their own advantages and disadvantages.

The technical patterns for enabling this communication between services are also broadly discussed. A main concern in this is *service discovery*; the question of how certain services know about the existence of other services and how to connect to these. As described in many

of the studies such as [#22, #28, #31], service discovery mechanisms are an essential part of microservice architectures and often prove to be the greatest challenge to successfully implement such an architecture. This involves implementing a means of discovering which application instances – such as machines or containers – offer what services and keeping a service registry of this. When an application or service then wants to connect to another service, the available instances along with their addresses and exposed interfaces can then be read from this registry. The function of deciding which instance to connect to can then be made by the service itself or be taken care of by a load balancer. Besides such *service interconnection* challenges, another challenge is how to achieve the integration of services without being restricted by to some specific technology [#19] such as a certain programming language. However, as also discussed in [#19], other aspects such as *interface design* are also part of enabling this integration. Challenges in interface design are for example extensibility, backwards compatibility and abstraction of implementation details. *Service contracts* are sometimes seen as a means to aid this by defining standard communication patterns that services should adhere to. The overall goal in all these challenges is to ensure that changes to one service do not cause system failure and require as little effort as possible for adapting other services to for instance use new functionalities.

Service Development

Further challenges are seen when making decisions regarding *Service Development*. When services are being developed to be microservices, certain challenges become prominent. Within this category, *service orchestration* is the most frequently mentioned challenge in the searched literature. Orchestration involves the arrangement, coordination and management of interactions between services to allow for execution of higher-level functions. This is a critical function in operating a microservice architecture. Orchestration is done in a central way, requiring a conductor that "that will send requests to other services and oversee the process by receiving responses" [#7]. The contrary of service orchestration is *service choreography*, which imposes a decentralised structure to this and "uses events and publish/subscribe mechanisms in order to establish collaboration" [#7]. From literature it seems that orchestration is the dominant approach of the two.

A next concern – just as in almost all distributed systems – is *data distribution* in this new setting. Often a sharding pattern is used to implement this [#9]. Data consistency and distributed transactions are not always straight-forward to manage and these challenges are amplified by the *number of services* that microservice architectures generally consist of. The number of services also "poses challenges to evolving microservice-based applications" [#29]. Because system functionality now often requires multiple services to work together to complete a task, *service composition* is also identified as a challenge. It is for instance proposed that a microservices composition framework could be developed, which would "facilitate knowledge reuse and make it simpler for application engineers to interact with a complex computing platform" [#10]. The question is how services integrate functionally and fulfil complex tasks together and how to achieve the integration of services without being restricted by to some specific technology such as a certain programming language [#19]. In line with this is the question of how to describe services and their functionalities; i.e. how to do *service specification*. This should be done in a standardised way to also facilitate composition. As stated in [#10], "the topology specification and composition need to cover the whole life cycle […] of each microservice as well as the application as a whole". Further development concerns for microservices identified in literature are *statefulness* and *multi-tenancy*. Ideal microservices are stateless [#11], thus enabling them to be started scaled and stopped on the fly. Often though, especially when migrating from a monolithic system, some

services need to retain their state. To not interfere with the flexibility of stateless services, one could decompose an application in stateful and stateless services [#35]. Multi-tenancy is only mentioned in one publication, but an interesting challenge, nevertheless. It is described as “a system’s ability to fulfil the requirements of multiple groups of service consumers, organisations, and even competitors in an industry” [#11]. In other words, it is a challenge to serve multiple (types of) users with the same system instance. Finally, there is the risk of possible *vendor lock-in* when deploying microservices on a commercial third-party platform or using other proprietary technologies. An example of this could be having an application talk to a PaaS vendor’s proprietary API directly to request resources. This would inhibit the deployment of this same application on the platform of another vendor without changing any code. Another example is using a certain vendor’s software to facilitate service orchestration, which could lead to a dependency on this technology.

Deployment

After considering the development of services, their *Deployment* becomes the next challenging topic. A first challenge is the *deployment method* used to operationalise services. Because microservices may each be built differently and even use a different language, the chosen deployment method should provide support for these multiple types of services. While using virtual machines to deploy a microservice architecture is possible, using container-based solutions like Docker [20] is considerably more wide-spread. Microservices are also commonly deployed on third-party cloud platforms. The choice of deployment method is not to be made lightly, as “a poor deployment choice can increase cost, and hurt performance, scalability, and fault tolerance” [#29]. Furthermore, *automated deployment mechanisms* are ideally in place to deploy services. Because of the many services that a microservice architecture can consist of, managing deployments manually can require a lot of effort and be error-prone. Examples of such deployment automation are tools that enable continuous delivery and continuous integration practices. Ideally, also rolling upgrades should be supported [#8]. All of these requirements are important to fully utilise the benefits of a microservice architecture and to “save time and gain agility” [#37].

Resource management and service scheduling are somewhat related. Resource management is needed to allocate appropriate resources to a microservice application. Because of the complexity of interactions between services in such a system, this is no straight-forward task. Allocating resources should be done dynamically to ensure system performance under varying load but also reduce operational costs by downscaling when possible. Scheduling deals with the question of where to deploy what microservices and in what configuration. Here, there are also many dependencies to keep in mind, such as the technology heterogeneity, required functionalities such as a load balancer and control and data flow dependencies [#10]. Because of this complexity, “the mapping of microservices to datacentres demands selecting bespoke configurations from an abundance of possibilities, which is impossible to resolve manually” [#10]. These challenges become even more prominent when considering a *multi-cloud deployment* setup, in which a microservice architecture might be distributed over multiple data centres, possibly in different regions. While a multi-cloud or multi-region setup might be beneficial to for instance serve geographically distributed users, any communication that needs to happen between these regions “runs the risk of increasing the application execution time” [#15].

Development Process

Apart from the challenges related to the development and deployment of microservices, the supporting *Development Process* should provide the right support to achieve successful

implementation. A main concern seen in literature is how to deal with *monolith migration*. Quite often, organisations aspiring to develop a microservice architecture are redesigning an existing, monolithic system. An important question here is how to decompose this existing system into microservices. Frequently there is no clear-cut way of dividing application parts and multiple theoretical approaches are mentioned in the searched literature to tackle this challenge. This is for example expressed in [#23], in which the authors state that "identifying components of monolithic applications that can be turned into cohesive, standalone services is a tedious manual effort that encompasses the analysis of many dimensions of software architecture views". Furthermore, a transition away from a monolith and towards a microservice architecture is not merely technical. As also stated in the *Organisation* category, an organisational structure and culture should be in place to support this new paradigm of developing applications. This is especially important with regards to the *development process* used to implement such systems. As mentioned previously, working with DevOps practices is seen as a means of supporting this transformation and fully embracing the requirements to the development process that a microservice architecture demands.

There are also technical implications for the development process. In one study the authors mention the difficulty of deploying a microservice architecture in a *development setup*. "Although the application code is now in isolated services, developers must also deploy the dependent services to run the isolated services on their machine" [#2]. Other challenges relate to the testing of microservices. As stated before, microservice architectures can be fault-prone at the integration level. This is why *integration testing* now becomes one of the most vital parts in an application's testing process besides for instance unit testing. However, as stated in [#18], "unit and integration tests are insufficient to catch […] bugs". They propose to perform *resilience testing* – "testing the application's ability to recover from failures commonly encountered in the cloud". This way, one can better test the behaviour of a microservice application to be certain that it behaves as required.

4.2.3 Scope and Conclusions

For the literature research, the aim was to first get a general overview of all types of challenges that could be found and to help answer RQ-1. For this, the appropriate challenges that relate to *communication* between*, integration* and *management* of microservice*s* need to be selected. Some challenge categories correspond directly with these subjects, some partially and others are considered out of scope.

For a start, the Service Management category can be fully considered within the management scope of RQ-1. As for communication, all challenges in the identically named challenge category can be also considered here. The one exception here is service integration. Integration is a part of RQ-1 that was not identified as a separate challenge category during the literature review, mainly because challenges related to this topic often fit in different, larger categories. It is possible to select the right challenges for this, though. These are service composition, specification, the integration and resilience testing of microservices and the aforementioned service integration. Testing is considered to be in the scope of integration because – as mentioned before – microservices tend to be error-prone on the integration level. A mapping of the challenges per category of RQ-1 is shown in Table 5. In this, blue, red and green identify the Communication, Integration and Management categories, respectively. Table 6 gives a single overview of these selected challenges.

Table 5 - Challenge mapping per category

Service Management	#	Quality	#	Performance	#	Organisation	#
[M] Service granularity	13	Fault-tolerance	7	Network performance overhead	6	Organisational structure and culture	5
[M] Service logging and monitoring	11	Security	4	Memory consumption	3	Team composition	5
[M] Service management	5	Reliability	3	Quality of Service	3	Legal responsibility	1
[M] Service evolution	4	Maintainability	2	Performance isolation and characterisation	2		
[M] Service boundaries	3	Availability	1				
Communication	#	**Service Development**	#	**Deployment**	#	**Development Process**	#
[C] Communication mechanisms	**11**	Service orchestration	12	Automated deployment mechanisms	8	Monolith migration	11
[C] Service discovery	**9**	**[I] Service composition**	**11**	Deployment methods	7	**[I] Integration testing**	**5**
[C] Interface design	**6**	Data distribution	8	Service scheduling	7	Development process	3
[C] Service interconnection	**5**	Number of services	3	Resource management	4	**[I] Resilience testing**	**3**
[C] Networking complexity	**3**	**[I] Service specification**	**2**	Multi-cloud deployment	1	Development setup	1
[C] Service contracts	**2**	Statefulness	2				
		Multi-tenancy	1				
		Service choreography	1				
		Vendor lock-in	1				

Table 6 - Challenges per topic of RQ-1

[C] Communication	#	[I] Integration	#	[M] Management	#
Communication mechanisms	11	Service composition	11	Service granularity	13
Service discovery	9	Integration testing	5	Service logging and monitoring	11
Interface design	6	Resilience testing	3	Service management	5
Service interconnection	5	Service specification	2	Service evolution	4
Networking complexity	3			Service boundaries	3
Service contracts	2				

Part of RQ-1 is the search for common challenges. When evaluating the number of times each of the challenges in Table 6 is mentioned in the searched literature, one could think that a challenge that is only mentioned twice is uncommon. However, given that the topic of microservices has only recently attracted significant interest in the academic community, many challenges have a rather low occurrence. In Figure 8 a histogram is shown of how many of the challenges occur a certain number of times which demonstrates this. In fact, almost half of the identified challenges are mentioned three times or less. Therefore, no challenges will be disregarded based on their occurrence since this might cause important challenges that have just not gotten enough attention in academic literature to be overlooked.

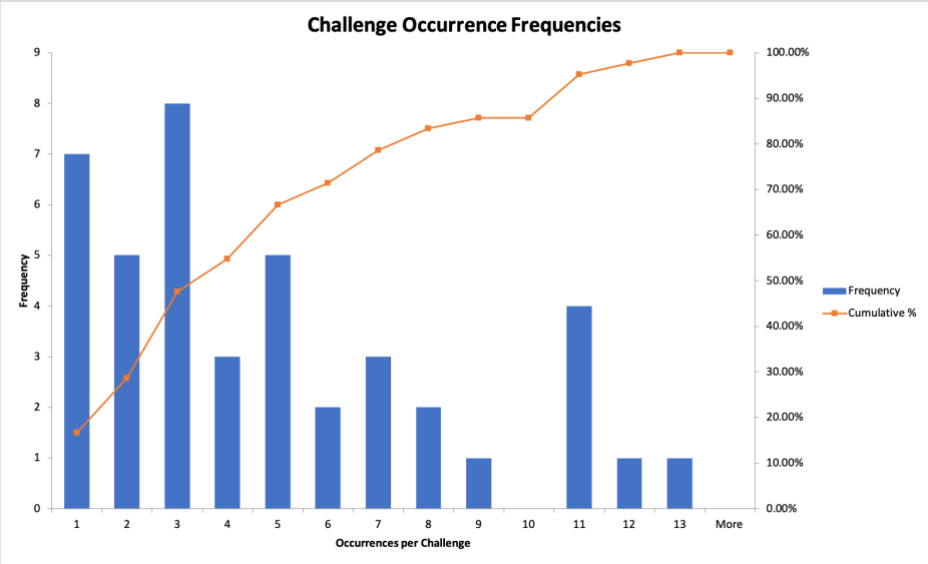


Figure 8 - Frequencies of Occurrences per Challenge

The challenges that are not directly linked to one of the aforementioned topics, are not necessarily of no importance to the project. Challenges related to deployment, security, quality, performance and monolith migration should be considered as context to those that are focussed on. For example, performance considerations could influence the choice of communication mechanism, but are not the primary focus in this research. Therefore, these challenges will not be explored in-depth as they are out of the primary scope.

Altogether, the challenges in Table 6 provide the answer to RQ-1: *What common design challenges related to communication between, integration and management of microservices can be found in academic literature?*

4.3 Challenges in Practice

To gain an appreciation of how decision-makers in practice view the challenges related to communication between, integration and management of microservices, interviews were conducted with Thales employees. To better interpret their views and opinions, first an overview of the organisational context is given. This demonstrates the current challenges that the organisation faces. This should help in answering RQ-2.

4.3.1 Context

In this section, first the systems landscape is examined to find out what complicating factors might be in place that complicate implementing a microservice architecture. After this, the motivations that drive Thales Naval to pursue this goal are discussed, followed by a preliminary stakeholder analysis to guide the exploration process. As discussed in these sections, there are many factors that make this project unique.

As stated before, the goal of Thales Naval is to transform the TACTICOS CMS to a microservices architecture. However, this system is no standalone entity as it has to interface with many connected systems that can be both physical and software systems. Together this forms a so-called Combat System of a naval vessel. This is contained in the Mission System (MS) which also includes other systems like communication and navigation. Added to this are the integration services that facilitate the interfaces with subsystems to end up with a complete Naval Mission Solution. The goal to move to a more flexible architecture also exists for the MS. However, for the scope of this assignment, only the CS is to be considered as this is also

the scope of development for Thales Naval. A high-level overview of the components of the entire system and how they relate to each other adopted from [30] is shown in Figure 9.

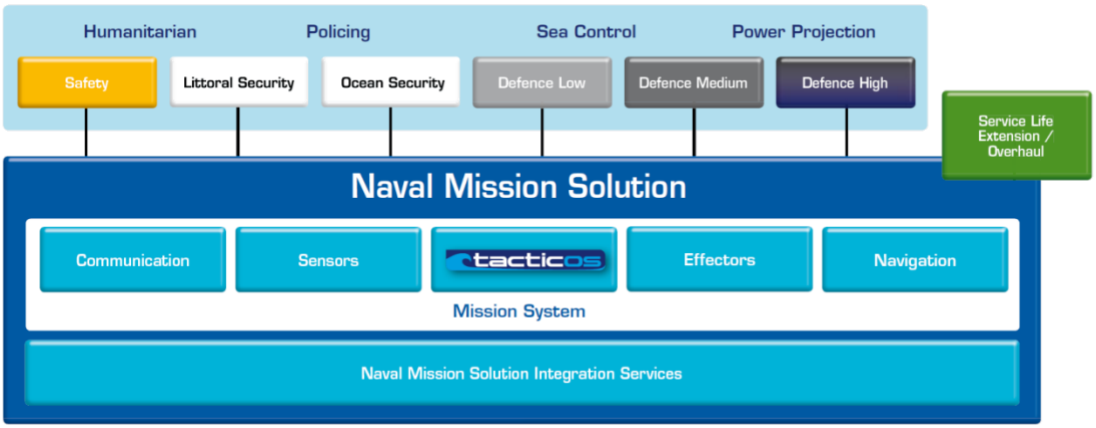


Figure 9 – Thales Naval Systems Overview – Adopted from [30]

The TACTICOS CMS is the central command and decision-making element of the CS aboard a naval vessel. It supports the sensor management, picture compilation, situation assessment, action support and weapon control functions of this system [9]. The current TACTICOS architecture has been in service since 1993 [31] and is used on board 160 ships that are operated by 20 navies [32]. The different applications in this system are distributed over several operator consoles – also called Human Computer Interfaces (HMIs).

A central part in the current architecture is the use of Data Distribution Service (DDS) middleware for Real-Time Systems standard, partially developed by Thales Group and approved by the Object Management Group [33]. Thales Naval uses Vortex OpenSplice [34] as implementation of this DDS standard that is commercially available from ADLINK Technology IST. The DDS is a data-centric publish and subscribe style of communications standard that "enables scalable, real-time, reliable, high performance and interoperable data exchanges between publishers and subscribers" [35]. This DDS is designed to be data-centric; rather than focussing on delivering messages regardless of the payload, in this system there is a data model in place. "The middleware understands the context of the data and ensures that all interested subscribers have a consistent view of the data" [35]. The widespread use of this DDS throughout TACTICOS has considerable implications on its architecture. For example, it dictates a publish/subscribe structure for communication between components. Also, all components need to implement the communication with DDS endpoints. Figure 10 illustrates how publishers of and subscribers to data communicate through a so-called global data space that allows the aforementioned consistent view of data.

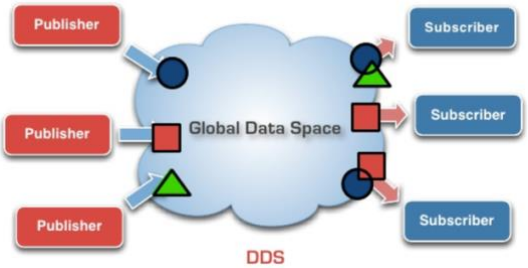


Figure 10 - DDS Structure – adopted from [36].

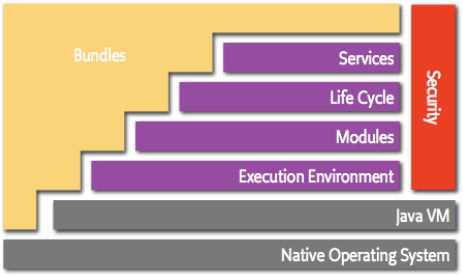


Figure 11 - OSGI Architecture - adopted from [37].

Many parts of the current implementation of TACTICOS are already built in a modular fashion. A main part of this is the implementation of the Open Services Gateway Initiative (OSGI)

specification that defines a dynamic component system for Java [37]. Modularisation is achieved by packaging application components as bundles. Underlying service, life cycle and module layers allow for dynamic connection between bundles, life cycle management and encapsulation of bundles. These layers are shown in Figure 11. As can be seen, the OSGI implementation runs on top of Java Virtual Machines. The OpenSplice DDS is used to facilitate communication between these instances. The fact that Thales Naval is already building modular software using the OSGI specification should help in a future transition towards microservices.

As shown in Figure 9, the CMS has to interface with so-called sensors and effectors. These are the physical devices used aboard a naval ship. Sensors are devices that monitor the environment, effector can interact with the environment. Examples of this are radar and sonar (sensors) as well as guns and missiles (effectors). Thales Naval manufactures some of these devices and often they are used together with the TACTICOS CMS. However, these sensors and effectors can also be sold separately to a navy that might use another CMS. Devices connected to TACTICOS might also be from other manufacturers. Therefore, the specification, design and management of interfaces in both the devices as well as TACTICOS is essential. This part of the system's communication structure is one of the most critical elements. It is also indicated that the question that Thales Naval asks largely originated from this work area.

Connecting to devices is often done by adapting any proprietary interfaces to a standard format. When devices from external suppliers need to be adapted towards TACTICOS, their interfaces are adapted to a generic format understood by the CMS first. The internally standardised format that Thales Naval uses in the devices they manufacture is called the Open Sensor Interface (OSI). It consists of different components to specify functionalities. Different components together make a composition of functionalities that altogether facilitate the functionality of a device. For example, certain functionalities between radar and sonar devices may be similar. The components for these base functionalities can then be shared and together with device-specific components form a composition. External customers often require an adaptation from this standard interface to their dedicated interface, controlled by themselves.

Thus far, Thales Naval has started several initiatives to investigate how a microservices architecture can be implemented for TACTICOS and the combat system in a broader sense. Apart from small-scale internal trials, there have also been larger, inter-organisational projects. The most prominent example of this is the INAETICS architecture project [38] with the goal to "define and demonstrate a dynamic service oriented reference architecture" that fits the requirements that real-time systems like TACTICOS face. This project is a collaboration between – among others – Thales Netherlands, OpenSplice and the University of Twente. This architecture's basic goals are to be technology agnostic, open and in the case of Thales Naval supports migration from the system currently in place. It is built upon principles of software modularity, a dynamic component architecture, dynamic application assembly and deployment, technology abstraction, service-based interaction and risk-adaptive security. Please note that these principles could be assigned to the basic principles of a microservice architecture as well. Even though in the INAETICS architecture microservices are not primarily considered, they are most certainly not ruled out.

Another current initiative is the GAUDI project with the goal to integrate the several self-contained systems such as the Combat, Platform and Bridge Management Systems and make all information aboard a naval vessel more easily accessible [39]. The aim is to also have a shared hardware platform and shared software services. This is seen as a way to increase

scalability of the systems and reduce life cycle costs. In this project, a SOA as well as microservices are considered as implementation approaches.

4.3.2 Interview Process

Seven employees of Thales Naval were interviewed in order to answer RQ-2. Respondents with different functions and hierarchical positions were chosen to get a situational overview that was as broad as possible. An overview of the different – anonymised – respondents is shown in Table 7.

Table 7 - Interview respondents

#	Function	Main Focus Areas
1	Chief Architect	CMS, microservices, interfaces
2	Chief Architect	Sensors, interfaces
3	Infrastructure Architect	Infrastructure, microservices, interfaces
4	Infrastructure Architect	Infrastructure, microservices, interfaces
5	Cloud Architect	Cloud Infrastructure, APIs
6	Software Architect	Service-orientation, interfaces
7	Software Architect	Sensors, service-orientation, interfaces

The interviews were conducted in a semi-structured qualitative way. Respondents were all asked several predefined questions and depending on their answers, follow-up questions were asked. The overall goal of the interviews was to get an in-depth insight in the challenges related to microservices that are perceived at Thales Naval. For this reason, the analysis of these interviews will also be done in a qualitative way. Just as in the literature review, the scope of the interviews was not limited to the challenge categories found when answering RQ-1 on beforehand so that no challenges would be missed. In line with the recommendations in [40], the interviews were not recorded as to not influence the respondents' answers and give them the feeling that they could talk freely.

The interview structure was as follows:

- First this study and its goal were introduced, as well as what the respondent's contribution to the research would be.
- Then a few basic questions were asked to the respondent such as their background, function and what they do in their daily work. This is to understand the context from which they view the discussed subjects.

  Questions:

  > *"What is your function within the organisation?"*
  >
  > *"What other functions have you had before this, and how did these influence your current work?"*

- After this, a baseline was set on what was meant with microservices. The respondent was asked if they were familiar with the subject and if so what their understanding of it was. If this differed significantly from what is understood by microservices in this study, or if they were not familiar with the subject, microservices' main characteristics were introduced to them. This way, all interviews started with a similar understanding of the subject for all respondents.

Questions:

> *"To what extent are you familiar with the term microservices?"*
>
> *"How would you best describe the subject of microservices?"*
>
> *"Could you name any defining characteristics of microservices?"*
>
> *"To what extent have you encountered microservices in your daily work?"*

- First the respondent's views on the proposed transition toward a microservice architecture were discussed. This could both include motivations for this shift as well as possible discouraging factors. Obviously, when a respondent would mention hindrances for moving to microservices these could already be seen as challenges.

  Questions:

  > *"Are you aware of a possible future transition of TACTICOS towards a microservice architecture?"*
  >
  > *"What is your view on this transition?"*
  >
  > *"What limitations do you see in the current system that might lead to this transition?"*
  >
  > *"What further motivations for moving to a microservice architecture would you consider?"*

- If the conversation on challenges had not yet started during the discussion following the previous question, the respondent was explicitly asked for any challenges that they foresaw or possibly have already encountered when considering a transition to a microservice architecture. The respondents were first free to speak their mind on the different types of challenges that they envisioned. This was done to not limit the discussed challenges to those already identified through literature research.

  Question:

  > *"What possible challenges do you foresee when transitioning TACTICOS towards a microservice architecture?"*

- After this, any categories of challenges that were not yet considered were discussed with the respondent. The categories from the literature review were used for this, along with the individual challenges in these categories. This way, it was possible to get the respondent's view on any challenges that they had not yet considered by themselves.

  Questions:

  > *"Would you consider challenges related to <category> as difficult in this transition?"*

- For the challenges identified in these discussions, the respondent was also asked why they felt like it was a challenge in the first place, as well as in what way they thought this would impact the implementation of a microservice architecture.

  Questions:

  > *"Why do you think <challenge> is a challenge in the first place?"*
  >
  > *"In what ways do you think <challenge> might affect the transition of TACTICOS towards a microservice architecture?"*

- During the interview, notes about the occurrence and difficulty of discussed challenges and categories were made. Before finishing the interview, the respondent was asked to verify whether these represented their views correctly.
- Finally, the respondent was of course thanked for their participation.

The challenges will be discussed using the categories identified in the literature review. Due to the qualitative nature of the interviews, each challenge category will be discussed qualitatively rather than counting how many respondents recognised each single challenge. For each category it is also indicated whether it was seen as challenging to identify the ones that have the highest priority for Thales Naval. This can then be used to answer RQ-2: *What common design challenges related to communication between, integration and management of microservices can be found in practice?*

4.3.3 Interview Results and Discussion

In general, all respondents were familiar with the concept of microservices. There was a small difference though in their nature of understanding; those with a more managerial function would describe microservices on a more abstract level and by their conceptual traits, whereas others would describe them in a more technical way. The concept of microservices as described in this study was discussed with all respondents, and they all agreed that this fit with their understanding. Some also indicated to have already had hands-on experience with implementing microservices through for instance pilot projects. All of them were also aware that moving towards a microservice architecture was considered as a future objective for TACTICOS. The respondents' opinions on this differed somewhat, with some considering it as a logical goal and others indicating that there were still many challenges ahead. This was where the discussion on challenges naturally started with those respondents. The limitation in the current system that was described most was inflexibility in development and configuration. Increased opportunity for modularisation and designing generic modules, as well as isolation through containerisation were seen as important motivations for the evolution of TACTICOS.

To follow the same order of challenge categories as in the literature review, *Service Management* is first discussed. All respondents indicated that a certain degree of management would be necessary to successfully operationalise and maintain a microservice architecture. Especially centralised specification and reasoning about service evolution are seen as challenges. One respondent mentioned service evolution early on in the interview as vital due to their experience with a pilot system using microservices. Other respondents indicated the importance and difficulty of logging and monitoring. One interesting notion with this challenge was that, while respondents understood the choice of including it in the service management category, many felt that a separation between the actual tools to implement logging and monitoring, and the managerial challenge of system observability should be kept separate. Observability in this case refers to the extent in which a system's state can be inferred and analysed from the logging and monitoring output. Furthermore, it was indicated that current systems already use some degree of componentisation. Nevertheless, service granularity was still seen as a possible challenge because of the generally narrow scope of a single microservice. Overall this category was considered to be important and challenging and therefore a priority.

As for *Quality* aspects of a microservice architecture, security is seen as the most important challenge to solve. This is mainly due to the nature of the software systems built by Thales Naval. Certain challenges such as how to deal with security zones and managing encryption certificates are not straight-forward to figure out in this new architecture. Maintainability is also

seen as difficult to get right, but the respondents' overall opinion is that this can be partially managed by managerial measures. The other challenges in the quality category are also seen as important, but not too challenging to solve. Rather, the general opinion is that when microservices' traits are embraced, they should be in a good position to ensure fault-tolerance, reliability and availability. Because of this, the priority of this category is neutral.

Respondents had mixed opinions on the challenges related to *Performance*. On the one hand, several respondents indicated to foresee or have already encountered noticeable network performance overhead. This challenge was seen as most prominent and impactful. However, while the importance of the several performance-related challenges was recognised, respondents also indicated that these would probably not cause many difficulties. If for instance any excessive network performance overhead would be encountered in the transition towards microservices, this could be fixed by improving the networking infrastructure to handle more load. The cost of fixing such performance challenges was not seen as impeding this transition. Therefore, managing performance-related concerns is not seen as a main priority.

Next there are the *Organisational* challenges. When discussing this topic, one respondent immediately mentioned the importance of ensuring that people have the right mindset when embarking on this transition to microservices. The challenges of organisational structure and culture as well as team composition were seen as important and somewhat challenging. Respondents acknowledge that in their case the right structure and culture should at least be in place in their department, and that building the right teams is key in enabling the microservice transition. Thales Naval already implements agile practices in their development process. Therefore, they are already off to a good start in this regard and this category is not seen as too challenging. They could also see ways in which the legal responsibility challenge would be applicable, but this challenge was not discussed in-depth because of its mainly legal nature.

The *Communication* category was seen as both important and challenging by the respondents and is therefore a main priority. Respondents often indicated that the choices made on for example the communication mechanisms topic have a large impact on the overall system structure and behaviour. Also, when considering the transition from Thales Naval's current systems, it was indicated that the current widespread implementation of the OpenSplice DDS might complicate the move to microservices. This is because many decisions about communication mechanisms, interface design and interconnection of system components are currently constrained by this underlying middleware. Furthermore, interface design is seen as a challenge, especially considering that some services should provide multiple different interfaces and the system should support future changes well. Thus, an adaptable and extensible interface is required. Networking complexity was specifically mentioned as a challenge by one respondent, after their experience in a pilot project concerning microservices.

Participants indicated that the concerns related to *Service Development* were also challenging to manage and that this category was of high priority. One respondent said that in a way currently there is a set of fixed patterns, assumptions and choices - a kind of 'recipe' of doing things in system development. This will obviously become obsolete when developing microservices. There are now also more challenges related to distribution of data and the flow of data through the system will change drastically. Furthermore, it is indicated that even though applications are already developed to be modular, individual modules are generally still relatively 'large'. Statefulness is also considered hard to minimise. In contrast, service specification is not seen as particularly challenging by the respondents because of experience with this from previous projects. Service orchestration was furthermore seen as vital to get

right, but through previous projects there was already a lot of knowledge about this challenge. Interestingly, even though the challenge of vendor lock-in was only sparsely mentioned in literature, many of the interview respondents thought that this challenge was of high importance. The concern was not as much about being dependent on a single vendor, but rather a single technology. Because Thales Naval's systems generally are expected to have a long lifespan, it is seen as important to use proven technologies that are expected to be supported well in the future either by their vendor or developer community.

Deployment was seen as important to get right, but not highly challenging. Thales Naval has already been working on multiple pilot projects that use a distributed, containerised service architecture. From this there is already a lot of knowledge on challenges such as automated deployment mechanisms and deployment methods. A recurring requirement to the chosen solutions is their 'maturity'; i.e. how stable and production-ready technologies are. One respondent described another pilot project in which service scheduling and resource management aspects were also investigated. These examples demonstrate that Thales Naval already has certain experience with the challenges in this category.

Challenges related to the *Development Process* were also considered of high importance but not all too difficult. Just as in literature, monolith migration was by far considered the hardest challenge in this category. Because of for example the aforementioned existing technology choices and communication structures, it is not straight-forward to plan a migration process from the current systems to a microservice architecture. The development process itself is expected to be easily adaptable to support microservice development. Currently, Thales Naval is already following many agile practices in their process. This is not to say that no changes are needed, but these will probably be easy to implement. Several changes will have to be made to the process for testing though. These are seen as possibly difficult to solve by the participants.

4.3.4 Conclusions

As stated, some challenge categories were considered more important than others by the interview respondents. Due to the qualitative nature of the interviews, the results will also be discussed in a qualitative way in order to answer RQ-2.

A so-called PICK-chart is used to show the results graphically in Figure 8. This chart has its origins in Lean Six Sigma (LSS) literature, where it is used to prioritise solution ideas. As described in [41], these ideas or solutions are represented in four quadrants; *possible, implement, challenge* or *kill*. Their classification depends on the expected effort required to implement it and the expected payoff. It is important to note that items are merely classified in one of these quadrants and the axes do not represent a continuous scale. The foreseen use in LSS literature is to first implement those solutions that maximise payoff while requiring limited implementation effort. These would fall in the *implement* quadrant. The addition of 'Just do it' in this quadrant has become common in practice to show that these solutions can be accomplished immediately with little difficulty [42]. In the case of representing the interview outcomes, chis chart can be read as a classification of a category's importance (payoff) and difficulty (implementation effort) by participants.

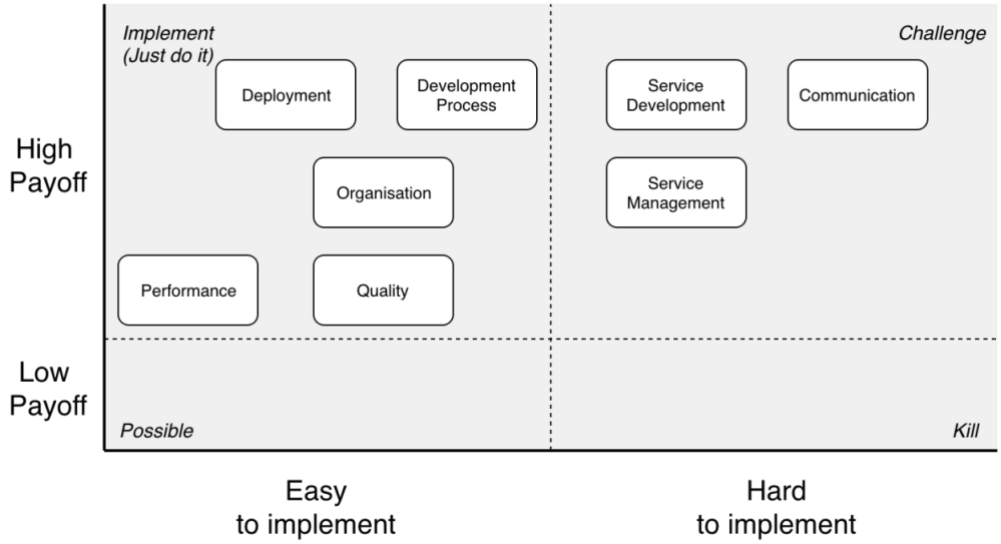


Figure 12 - PICK Chart of Challenges from Interviews

In general, it can be seen that all categories were seen as important to consider by the participants. They recognise that in order to successfully implement and transition to a microservice architecture, none of the categories should be discarded. There are, however, several categories that are seen as more challenging than others by the respondents. The most notable categories are *Service Management, Communication* and *Service Development*. These categories concern challenges and design decisions that are generally difficult to solve and have a notable impact on the overall system design. Therefore, they are seen as a first priority to figure out. That is not to say that the challenges in other categories are easy to resolve; they are just considered less of a priority than those in the aforementioned categories. In some categories there were also individual challenges that were seen as particularly challenging, apart from the category as a whole.

The view that arose from the interviews is that the success with which the challenges in the categories that have a high payoff but are hard to implement are addressed, can make or break the success of transitioning TACTICOS to a microservice architecture.

When comparing Thales Naval's view on challenges from the interviews with that from the literature review, several details stand out. A first interesting insight is that the aforementioned categories that were seen as challenging in the interviews, also accounted for a large part of the challenges found in literature. When looking at Figure 45 in Appendix B, it can be seen that these categories cover almost half of the occurrence of the found challenges. On the contrary, other prominent categories such as *Deployment* and *Development Process* were not seen as highly challenging by the respondents. This can probably be explained by current experience in these areas or the fact that Thales Naval has already taken steps to facilitate certain of these challenges. As stated before, for deployment there have already been some pilot projects and they are already implementing agile practices in their development process. Nonetheless, it should be considered that because these categories are well-represented in literature, those challenges might be harder to solve than is currently thought.

Some individual challenges also stood out during the interviews. While these were not always found to be present in much literature, they do illustrate what difficulties respondents foresee in the process of implementing a microservice architecture. These can largely be explained by the business context that Thales Naval operates in. For instance, security is naturally a prominent in naval systems engineering. As was also described, the long lifespan of the systems developed by Thales Naval cloud make for the need to avoid vendor lock-in more

than in other types of organisations. Furthermore, the suggestion of adding observability as an additional Service Management challenge seems to make sense, since this builds on top of output of monitoring and logging tools. In the description of the logging and monitoring challenge from literature, the focus on creating meaningful and actionable insight was already noted. By splitting up this challenge in a management-related one that can use guidelines for the observability aspect and choose between software alternatives to implement logging and monitoring, both challenges can likely be better addressed. The logging and monitoring challenge can then be placed in the integration category. A revised version of Table 6 is shown in Table 8 below. The occurrence for the challenge of observability is kept empty, since this was not directly counted during the literature review.

Through these interviews, a view from practice at Thales Naval on the challenges when implementing microservices could be compared to that from literature. This helps to better understand the practical context and nuances surrounding these challenges.

Table 8 - Final Challenges Overview

[C] Communication	#	**[I] Integration**	#	**[M] Management**	#
Communication mechanisms	11	Service logging and monitoring	11	Service granularity	13
Service discovery	9	Service composition	11	Observability	
Interface design	6	Integration testing	5	Service management	5
Service interconnection	5	Resilience testing	3	Service evolution	4
Networking complexity	3	Service specification	2	Service boundaries	3
Service contracts	2				

5 CHALLENGE DEPENDENCIES AND POSSIBLE SOLUTIONS

To enable the decision-making framework to be adapted and used to make decisions related to *communication between*, *integration* and *management* of microservices, the dependencies between challenges related to these categories must be established. Also, to help give practical insights about the alternatives and guidelines available for managing these challenges, possible solutions should be investigated. By giving examples of solution domains to these challenges, decision-makers have a starting point for solving them. To end up with possible decision alternatives and their characteristics, it is first important to gain an understanding of the conceptual aspects of each challenge and what high-level, architecturally-important decisions can be made regarding them. There might also be general guidelines that are important to consider regardless of how a challenge is dealt with. Furthermore, it is important to note the related concepts to these decisions and guidelines to facilitate the search for solution alternatives. Therefore, for each challenge a set of possible decision alternatives, guidelines and related keywords for further searching for solutions are established. If any dependencies on superior choices are found, these are also documented. Note that this overview is not exhaustive, as there are too many possible alternatives and guidelines available to document in a coherent way. It should rather serve as a starting point from which solution alternatives and guidelines can be searched and selected.

In the preceding literature research, many academic papers were searched for microservice challenges. The descriptions in these papers serve as a starting point for explaining each challenge in more detail and finding possible solutions to them. Often, descriptions and solutions are also described in practice, through sources such as technical blogs, large-scale software development organisations or collaborations between organisations through for instance the Cloud Native Computing Foundation[3] (CNCF). Even though these practical sources might not give information as objective as in academia, their input can enhance the challenge descriptions with practical insights. Together with academic literature, this should make for a well-balanced starting point for describing these challenges and their possible solutions. The CNCF claims to be vendor-neutral and is part of the non-profit Linux Foundation. It has also been referenced in microservice-related academic works such as [43], [44]. CNCF also maintains a so-called landscape [45] to show what software solutions can be used for different parts of a microservice architecture. This can serve as a viable source of solutions that are used for different challenges in practice.

The descriptions in this chapter form the answer to RQ-3: *What are the dependencies between the identified challenges and what possible alternatives and guidelines are available as solutions?* This can then be used as theoretical input for the artifact to be designed.

5.1 Management

Even though microservice challenges are often seen as rather technical, many deal with management-level decisions to be made in organisations. Several challenges for which this is true have been identified through the preceding literature research and are described in this section. A neat example of a management-related challenge is that of service evolution; i.e.

[3] https://www.cncf.io/

how to build microservices so that future changes can easily be made. Even though this challenge is partially of influence on the actual implementation of a microservice, the bigger question is what process is used to facilitate these changes and how to agree on the way of implementing new functionalities. Zimmermann puts it in these words: *"such architecture design issues transcend both style and technology debates"* [3]. Generally, there seem to be no clear-cut decisions to choose between for addressing these management challenges. Therefore, for the five challenges in this category guidelines are provided to help manage them. An overview of the management challenges is shown in Figure 13. The challenges are not connected by arrows indicating dependencies, since no clear dependencies between them could be found. This is most likely due to the absence of distinct decisions to be made for each challenge.

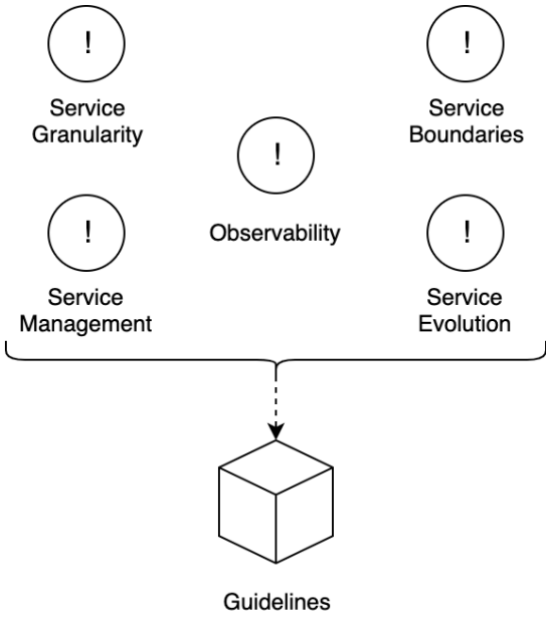


Figure 13 - Management-related Challenges

Service Granularity

The most mentioned challenge in the preceding literature research was that of service granularity; i.e. how 'big' individual services should be. Microservices are meant to be developed around business capabilities [6], though translating this guideline to practice often proves to be difficult. In a paper discussing this challenge, Hassan and Bahsoon [46] identify two main trade-offs that are made when determining a granularity level. The first is that of the size versus number of microservices in a system. On one hand, it could be said that more services should allow for more separation of business functionalities. On the other hand, this also implies that as services get more numerous, increased communication between individual services is required. The next trade-off identified by the authors is between satisfying local versus global non-functional requirements. The example given here is that of minimal performance requirements such as load time for a system as a whole, as well as for an interaction between two services. The overall system load time is dependent on the speed of underlying service communication, which in turn is influenced by the number and size of services. The authors take the view that "aggressive isolation of business functionalities is not necessarily ideal for all scenarios of the [system] environment" [46]. Dragoni et al. explain how service granularity is supposed to be preserved even in an evolving system. "Idiomatic use of the microservices architecture suggests that if a service is too large, it should be split into two

or more services, thus preserving granularity and maintaining focus on providing only a single business capability" [1].

What makes addressing this challenge so hard is that one cannot easily give a number of function points or lines of code that a service should be limited to for 'optimal granularity'. As Gouigoux and Tamzalit outline: "there is currently no commonly-accepted definition of the desired size of a microservice" [47]. This is also highly dependent on the context a system is developed in, an organisation's goals for the system, as well as development and operational trade-offs. Besides this is the cost factor of testing and operating a microservice architecture. Service granularity affects these costs as shown in Figure 14 adopted from [47].

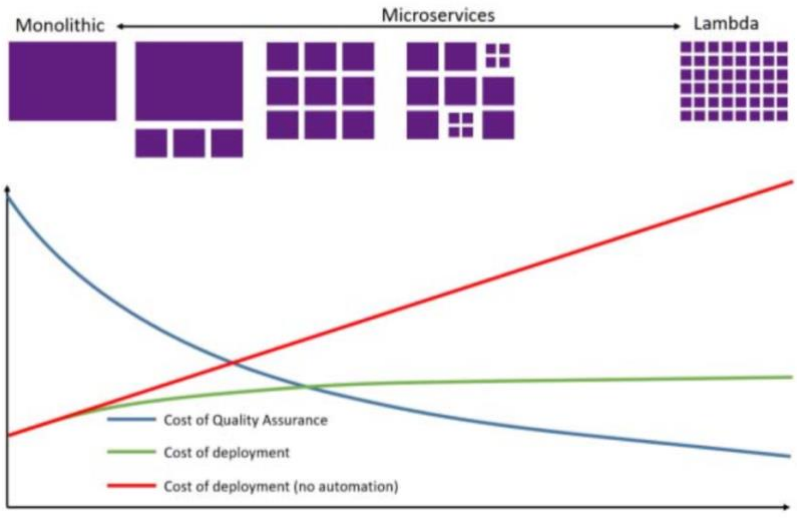


Figure 14 - Service Granularity versus Costs - adopted from [47]

On the top, there is a range of granularity levels for microservices. The extremes are shown as monolithic and lambda. The finest-grained example of Lambda comes from Amazon's Lambda[4] service in Amazon Web Services and describes so-called "nanoservices" which are functions executed in a serverless fashion in response to events. As a concept, these are out of scope for present research. The main point here is that they are an example of an extremely fine-grained architecture. The three lines in this graph each represent cost. The blue line refers to cost of quality assurance. The authors argue that as microservices become more fine-grained, the costs of testing and validating services decrease as this can now be done one-by-one. The red and green lines illustrate costs involved with deploying and operating microservice architectures with a certain granularity level. In the case of the red line, no deployment automation for automatically managing and assigning resources to services is in place, whereas with the green line there is. This graph first of all suggests that deployment automation definitely pays off when developing a microservice architecture and is required to keep deployment costs in check with increased granularity. Also, higher granularity only marginally increases deployment costs from a certain point.

[4] https://aws.amazon.com/lambda/

Guidelines:

- Consider the balance of the size versus the number of microservices
- Assess the implications on non-functional requirements and costs of different granularity levels
- Ensure deployment automation is in place to make more fine-grained microservices economically feasible

Keywords: *service granularity, service size, number of services*

Service Boundaries

While service granularity focusses on the size and number of services, service boundaries concerns the decision of what functions are in and out of scope for a certain service. Questions of how to divide functions between services, how many functions a service should offer and whether to subdivide certain functions into more, even smaller functions are seen as hard to answer in practice by organisations. When cutting up a system into microservices, basically service granularity deals with how many cuts to make, and service boundaries with where to make these cuts.

As previously described, microservices are often built around business capabilities. Richardson [48] discusses how to do this, stating that “an organization’s business capabilities are identified by analyzing the organization’s purpose, structure, and business processes”. Each business capability can have several sub-capabilities. These can then be mapped to services. This is not always one-to-one, as business context can influence when it is logical to develop separate services for several sub-capabilities or capturing these in one service if they are similar. Also, service granularity can come into play in the decision whether to split up a service. This mapping can also change over time as the system evolves and functionalities change.

Another approach to this that is well-known and widely used in the microservice field is that of Domain Driven Design (DDD), outlined by Evans [49]. DDD defines a separate domain model for each subdomain which are established similarly to the approach to decomposing around business capabilities. The scope of such a domain model is referred to a bounded context. According to Richardson [48], each bounded context can then be mapped to a service or set of services. In their view, DDD and the microservice architecture are in almost perfect alignment because of the easy mapping from bounded context to microservices and the similarity between autonomous teams that own services (proposed in microservice literature) and DDD prescribing that each domain model should have its own team.

Guidelines:

- Use a structured approach to define service boundaries
- Ensure that the approach used aligns with microservice objectives
- Consider using Domain-Driven Design, as it is widely known and used in the microservice field

Keywords: *service boundaries, business capabilities, domain-driven design*

Observability

The complex interactions between many microservices can be hard to comprehend, therefore making it harder to trace errors and understand the overall system health. Collecting logging and monitoring data from individual services is relatively straight-forward, but aggregating this

data to create meaningful, actionable insight is far from it. Observability refers to the extent in which a system's state can be inferred and analysed from the logging and monitoring output. The actual tools to support gathering information about a system's state and enabling such analyses are part of the integration category of challenges. On a management-level, it is important to determine the patterns used to enable observability in a microservice architecture.

The behaviour of microservice architectures is intrinsically hard to keep track of. In his book, Newman states: "We cannot rely on observing the behaviour of a single service instance or the status of a single machine to see if the system is functioning correctly. Instead, we need a joined-up view of what is happening" [13]. Different patterns to enable this are outlined by Richardson [48]. Commonly used first steps are to have services provide health check APIs, and aggregating system-wide logging output. Health check APIs are useful for keeping track of whether a single service is able to fulfil requests. The implementation of such health checks can be tailored to fit with a service's functionality. The tests that are done to assess a service's health can for instance differ between a service that accesses a database and an upstream service that aggregates data from multiple underlying APIs. In the event of a failure, these health check APIs could also provide more information about what is going wrong, which is useful for observability. Log aggregation is aimed at understanding the system-wide status. Logs of all services are aggregated in a centralised database and can then be searched and used for alerts about unexpected system behaviour.

More advanced patterns are also outlined by Richardson [48]. Distributed tracing is a pattern in which each external request is assigned a unique identifier and its flow through the system is recorded by a centralised server. Subsequently, the flow of this request can then be analysed and visualised through special tooling. This approach is popular in the microservice field, as it allows to get a better understanding of the dynamic behaviour of a microservice architecture. Other patterns for observability include reporting application-specific metrics to a centralised system and exception tracing. Audit logging is a final example discussed by Richardson and is aimed to help customer support, ensuring compliance and detecting suspicious behaviour by recording user actions in a database.

Industry reports also shed some light on realising observability in practice. Engineers Lew and Narayanan at Netflix share their lessons learned from building observability in practice in a tech blog article [50]. Several examples are relevant in this context. First, on the scale at which Netflix operates, aggregating all raw logging information was seen as not scalable enough. This resulted in a switch to analysing streaming logs, filtering and analysing these and only storing relevant information for longer. They also argue that distributed tracing should be augmented with additional information to truly be able to understand the behaviour of their microservices-based system. Furthermore, when alerting based on metrics, the engineers argue for focussing on anomaly detection and analysis of metrics correlation to focus on truly relevant system behaviour, rather than just alerting when a certain threshold for a metric is exceeded.

Guidelines:

- Determine which observability patters best fit with the system's characteristics
- Aim at understanding the system's behaviour to generate actionable insights
- Check whether the observability patterns used support the scaling requirements to a system

Keywords: *observability, service health, service health checks, log aggregation, distributed tracing, application metrics logging, exception tracing, audit logging*

Service Management

One of the most abstract challenges found in this category is service management. This entails the supporting activities to guide the design, development and delivery of microservices. Because a microservice architecture inherently consists of many individual services, their management becomes increasingly important. As Yu states: “Microservices though independent, do not exist in silos. […] They are part of the enterprise landscape and need to participate in the enterprise business process” [51]. In several studies discussing this challenge, various activities related to it were mentioned. One example is the creation of an organisation-wide service repository of all the microservices specifications. Additional information like status, ownership, can also be included. Such activities would improve organisation-wide understanding of what services exist and are used, and facilitate reuse of existing services where possible [51]. Furthermore, information on services’ lifecycle status can also be included in this repository. Also, best practices and organisation-specific knowledge on the development and deployment of services should be accessible to the development teams concerned. This way, the development of new functionalities can be accelerated. In these aspects, the requirements for management of microservices are somewhat similar to those of managing a SOA. Though, given that a microservice architecture typically consists of many services, these activities become all the more important. Also, service management related activities need not always be standalone initiatives. For example, organisations developing a microservice architecture can already be using a DevOps approach to software development. Zimmermann call this a form of *“lean but comprehensive system/service management”* [3]. In this case, activities related to service management are already being put into place.

Guidelines:

- Ensure an organisation-wide service repository is in place
- Develop best practices for developing and operating a microservice architecture

Keywords: *service repository, service management, system management, service lifecycle management*

Service Evolution

Changes to a system during its time in operation are almost inevitable. Industries change, as do business goals and requirements to a software system. The fact that microservices generally exhibit a high degree of decoupling [2] – i.e. changes in a service can be made and deployed without changing other services [13] – enables organisations to more easily change the functionality of individual services. However, microservice architectures are more dependent on the interactions between such individual services. As stated in [52] by Sampaio et al.: “Determining compatibility and consistency between microservice versions is a continuous challenge for developers”. Put differently, “a system should stay maintainable while constantly evolving and adding new features” [1]. Other services can – when developed well – be largely unaffected by changes in one. However, this means that providing new or deprecating old functionalities in which multiple services are involved requires coordination between the teams that developed them. This question is therefore not limited to merely technical solutions. It also concerns the process that is used to facilitate these changes and how to agree on the way of implementing new functionalities.

Several approaches exist to managing changes in software development. In the field of microservice design, much emphasis is already put on mechanisms to tolerate changes in services without directly influencing others. The goal is to anticipate that other services will

change or malfunction and figure out how to handle this. Generally, changes to services will translate in their APIs changing, as these are the primary means of communication with other services. Richardson [48] puts forward some strategies to manage evolving these APIs, such as Semantic Versioning and aiming to make changes as backwards-compatible as possible. Services should be built following the so-called Robustness principle, originally written by Postel in the specification for TCP: *"Be conservative in what you do, be liberal in what you accept from others"* [53]. In practice, this translates in 'older' clients ignoring any unknown extra information in an API call's response, as well as services offering default values when certain request attributes are missing. On the other hand, Fowler proposes that it is best to *"only use versioning as a last resort"* [2]. In their view, "versioning can significantly complicate understanding, testing, and troubleshooting a system". Nevertheless, they describe similar approaches to ensuring compatibility as Richardson, including following the Robustness principle. Thus, both authors share similar ideas about handling service evolution.

In the case of having to make major, breaking changes, offering different API versions or endpoints simultaneously is seen as a way to ensure that older clients keep functioning as expected [19]. Breaking changes cannot always be avoided at all costs. Subsequently, the way and timeline for deprecating the old major version of the API should be coordinated within the organisation. For example, it can be agreed upon that all existing services should implement the new API within a certain amount of time.

More detailed theoretical approaches are also available, such as the one by Sampaio et al. [52]. In their paper, the authors propose a service evolution model in which they "combine static and dynamic information to generate an accurate representation of the evolving microservice-based system". This should for example help practitioners manage service upgrades and architectural evolution. Other insights from practice are described by Bogner et al. [54] in a paper on the evolvability of microservices. Through interviews they found that generally "there were two different approaches for assuring evolvability: very decentralized with very autonomous teams […] vs. centralized governance for macroarchitecture, technologies, and assurance combined with a varying degree of team autonomy for microarchitecture […]" [54]. The former approach focusses on individual responsibility for assuring evolvability by for instance relying on the "you build it, you run it" principle [28]. The latter focusses more on guidelines, principles or standardisations that software engineers have to take into account.

Guidelines:

- Ensure service robustness to anticipate changes to services when developing a microservice architecture
- Coordinate the implementation and timeline of major, breaking changes
- Decide and communicate where the responsibility for assuring service evolvability lies
- Consider tools for gaining insight in service evolvability

Keywords: *service evolution, service evolvability, API evolution, robustness principle*

5.2 Integration

Challenges related to integration consider the not merely technical behaviour and design of a microservice architecture. For example, the service composition challenge is just as much a challenge related to the theoretically designing how services work together as it is a technical challenge of how to accomplish this. The challenges and dependencies and are shown in Figure 15.

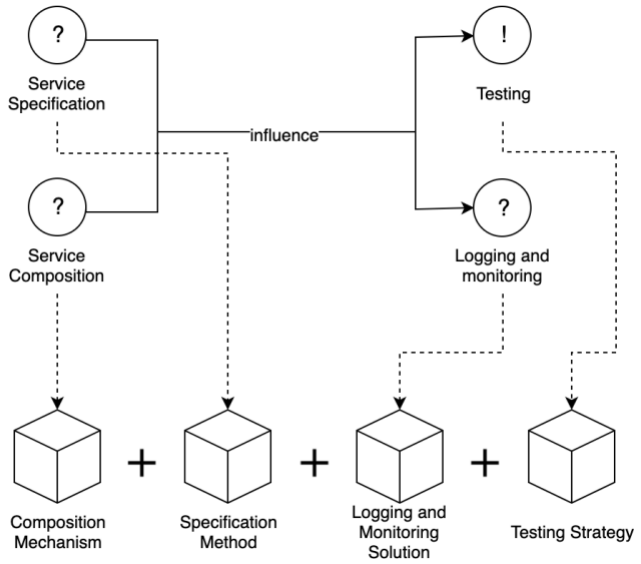


Figure 15 - Integration-related Challenges and their Dependencies

Service Composition

The challenge of service composition deals with the question of how services integrate functionally and fulfil complex tasks together. Because in a microservice architecture, system functionality often requires multiple services to work together to complete a task, service composition an often-mentioned challenge. One challenge is how to achieve the integration and composition of services without being restricted by to some specific technology such as a certain programming language [55]. It is for instance proposed that a microservices composition framework could be developed, which would “facilitate knowledge reuse and make it simpler for application engineers to interact with a complex computing platform” [56]. However, such a framework is currently non-existent.

A paper by Dustdar and Schreiner [57] goes into detail about the different strategies to service composition one can take. It is an often-cited work that explains and describes these strategies based on existing technologies. Dynamic composition is one such strategy. Static composition is also a thing, but given the aim for flexibility by using microservices, this is out of scope. Dynamic composition is based on the premise that services should automatically adapt to unpredictable changes. Services composition is realised in a way that is not predetermined, by for instance using frameworks to describe the interactions between services. Model-driven service composition builds on top of this and aims to define interactions through high-level languages such as UML or OCL. There are also approaches that aim to automate compositions by extending service descriptions or specifications. Context-based approaches also exist that for instance take into account information and requirements from a client to change the composition of services. An example of this is using a different composition of services to serve clients using one operating system versus another. The authors describe various frameworks used in practice that each use a unique approach to enabling service composition.

Besides the descriptions by Dustdar and Schreiner, a more recent work by Sheng et al. [58] goes into great detail about subsequent developments to service composition and describes standardisation efforts to service composition, research prototypes and platforms used in practice today. Given that this paper is more recent, it can serve as a good starting point for finding service composition alternatives. However, potentially not all commercially available solutions support microservices as well, as they can impose quite rigid constraints to a system's design, thus possibly limiting the benefits of using a microservice architecture.

In practice, often rather 'lean' approaches to enabling service composition are used in microservice architectures. Richardson describes the API composition pattern [48], with three options of implementing it. In the first case, a client service aggregates requests to different services. So basically, a service makes multiple requests and aggregates the results itself. An alternative is to make these requests through an API gateway. This way, the service receives the same data, it just makes these requests through a single gateway. This is often used in exposing service functionalities to an external client [48]. A third option shown by Richardson is to implement a separate service that acts as composer. This service can query the necessary data and output this as a single response to other services. When such aggregated functionality is used multiple times in a system, it may make sense to implement service composition this way. However, Richardson also notes that these approaches come with certain drawbacks, such as a risk of increased overhead, less availability and transactional data consistency problems. An alternative to this is the Command Query Responsibility Segregation (CQRS) pattern, which is alike a Relational Database Management System (RDBMS). This pattern is built around "the notion that you can use a different model to update information than the model you use to read information" [59]. When queries get more complex, a service using the API composition pattern could be inefficient since possibly not all data can be retrieved in similar ways. This leads to receiving and joining large data sets and other concerns. By using CQRS, such issues can be resolved. However, it does make the architecture more complex, and replication lag for data can be in play.

Decision alternatives:

- API composition / CQRS pattern

Keywords: *API composition, CQRS, web service composition, service composition.*

Service Specification

In line with this is the question of how to describe services and their functionalities; i.e. how to do service specification. This should be done in a standardised way to also facilitate composition. As stated in [56], "the topology specification and composition need to cover the whole life cycle […] of each microservice as well as the application as a whole". Two main alternatives can be found for this; defining services by business capability or by sub-domain [48].

When using business capabilities as a driver, services are defined by the capabilities or activities a business does in order to generate value. Definition by subdomain largely revolves around using Domain-Driven Design (DDD) described by Evans [49]. In this case, building services is centered around the development of an object-oriented domain model.

In terms of ways to implement such specifications, the CNCF lists solutions to facilitate this. In their landscape [45], solutions related to *App Definition and Development* can be investigated for use in a microservice architecture.

Decision alternatives:

- Define by Business Capability / Define by Sub-Domain

Keywords: *Service specification, web service specification, domain-driven design.*

Service Logging and Monitoring

The complex interactions between many microservices can be hard to comprehend, therefore making it harder to trace errors and understand the overall system health. Collecting logging and monitoring data from individual services is relatively straight-forward, but aggregating this data to create meaningful, actionable insight is far from it. Possibly due to the fact that this is a prominent challenge in microservice architectures, many existing software solutions are available to support the logging and monitoring. Service Logging and Monitoring is mainly dependent on Service Composition because of its impact on the overall system architecture.

Tools specifically developed for microservice logging and monitoring may be required, since existing tools are often not built with the number of services and possible architectural complexity of a microservice architecture in mind. Given that microservices often assume that services can fail, makes the need for insight in a systems behaviour even more prominent since system's behaviour can be less predictable.

In the CNCF landscape's [45] *Observability and Analysis* category, more than 70 monitoring, logging and tracing solutions are available. Prominent examples that are often mentioned in practical sources are Prometheus or Grafana for system monitoring, Fluentd for logging and Jaeger for tracing to get information about service operations. Not coincidentally, these projects are ones that have been labelled as rather mature by the CNCF and are thus more and more often already seen in production use. Many tools each have their own advantages and disadvantages, and are often built with a specific purpose, type of system or use case in mind. Therefore, no high-level choices on this can directly be identified. Nevertheless, alternatives can be compared based on criteria set during decision-making to ensure that the right one is chosen. Also note that the CNCF is not the only source of possible alternatives, though it is a good place to start for the search of logging and monitoring solutions.

Decision alternatives:

- Choose between alternatives, likely from CNCF

Dependent on:

- Service Composition

Keywords: *Service monitoring, service logging, tracing.*

Testing

Microservice architectures can be fault-prone at the integration level [1]. This is why testing now becomes one of the most vital parts in an application's testing process besides for instance integration testing. However, in [60] it is proposed to perform resilience testing – "testing the application's ability to recover from failures commonly encountered in the cloud" since the authors argue that "unit and integration tests are insufficient to catch […] bugs".

In the literature review, two separate types of testing were identified; integration and resilience testing. However, since these challenges have no clear-cut alternatives, they are described by guidelines for testing. These guidelines do not differ too much between the two types of testing, it is just important to understand that they are different and may both be used. For this reason, they are discussed together and from here on out will be treated as a single challenge.

Based on the guidelines, a testing strategy can be developed. Testing is also mainly dependent on Service Composition for the same reason that Service Composition has a large impact on the overall system architecture.

A first and foremost guideline to be given here is to *automate testing as much as possible*. By doing this, testing can happen continuously and any changes being deployed can be tested quickly and be put into production. Richardson describes the need for automation very well: "Automated testing is the key foundation of rapid, safe delivery of software. What's more, because of its inherent complexity, to fully benefit from the microservice architecture you must automate your tests" [48].

As said before, *pay close attention to integration testing*, since in this regard a microservice architecture is most different from a SOA [1]. When looking at the so-called test pyramid – shown in Figure 16 – that Richardson [48] uses, integration test are a level above unit-tests. This focus on integration is not to say that unit-tests are not important, though. It is assumed that standard unit tests are in place. Testing a system's behaviour can be done through an end-to-end test, in which all services are launched and tested through their APIs, but this takes substantial effort. Integration tests are more lightweight, while still considering the interactions between separate services by focussing on specific interactions.

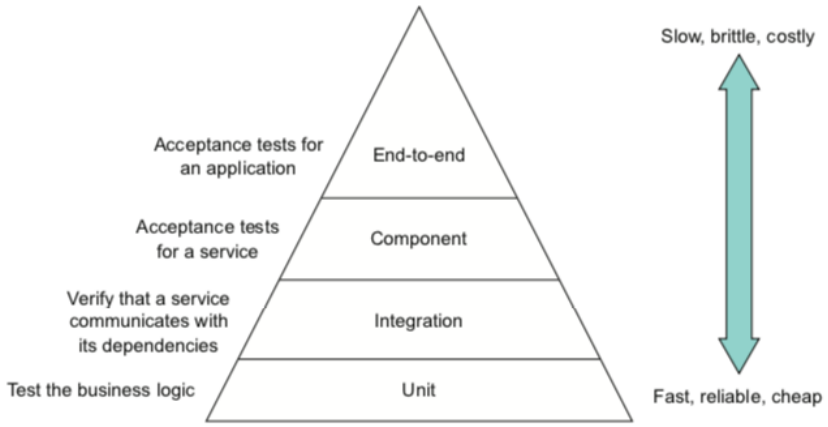


Figure 16 - Test Pyramid – adopted from [48]

Testing does not stop there, however. Heorhiadi et al. [60] argue that resilience testing should be in place for "testing the application's ability to recover from failures commonly encountered in the cloud". The authors propose a tool called Gremlin [5] to facilitate such testing. This tool mainly introduces failures in the network and tests if services handle this well. It can suddenly inject faults in application interactions to find out what this does to their behaviour. Similar to this is an approach – though a bit more risky – used in practice by Netflix called Chaos Monkey [61]. This tool randomly terminates services in production use. Netflix' motivation behind this is that "exposing engineers to failures more frequently incentivizes them to build resilient services". The thought is that when engineers know and anticipate that services can terminate at any point in time in production use, they will program services in a way that can handle this. The assumption that well-developed services will not crash does not hold up, especially in a microservice architecture. This approach can be generalised as a form of chaos engineering [62].

[5] https://www.gremlin.com/

This approach is also used in other organisations and could be of value when developing a microservice architecture. However, there may also be reasons for organisations to not want to introduce deliberate failures in their system. The guideline here is therefore to consider this approach, but not by all means.

Guidelines:

- Automate testing as much as possible
- Focus on thorough integration testing
- Use resilience testing to be more confident about how microservices handle failures
- Possibly consider chaos engineering type approaches

Dependent on:

- Service Composition

Keywords: *Integration testing, resilience testing, chaos engineering, automated testing*.

5.3 Communication

The communication-related challenges are those that touch upon the most technical parts of the microservice architecture. These choices do, however, have a large impact on how the eventual system behaves and performs. In Figure 17, the challenges in this category along with their dependencies and outputs are depicted. Note that the challenge of *networking complexity* is not included in this overview. When investigating this challenge further, it was concluded that it arises from several choices made in this category and cannot be solved by deciding between certain alternatives or following guidelines. The combination of choices made in the communication category determine the extent to which networking complexity is a challenge. It could therefore be used as a criterium in decision-making, to express how certain alternatives improve upon it or not. Furthermore, *service contracts* have been included in the interface design challenge because of their overlap and the similar considerations involved.

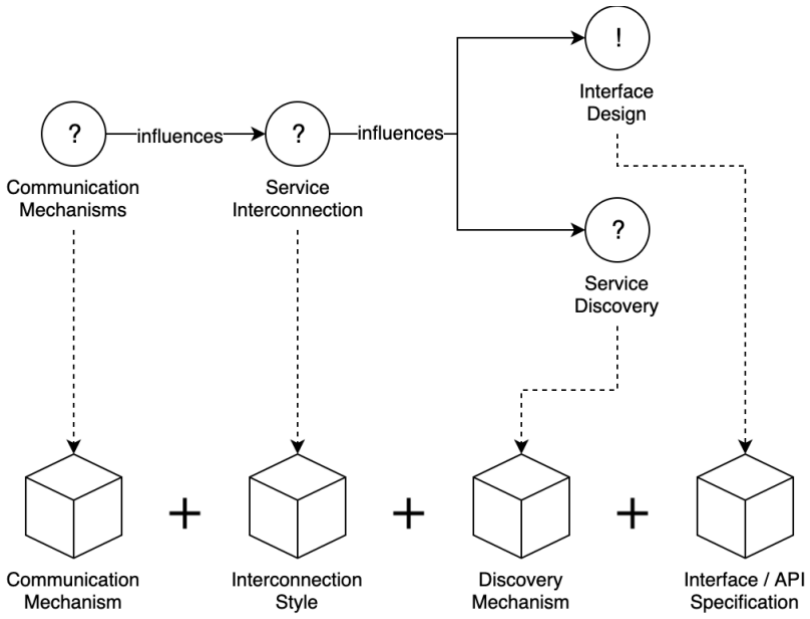


Figure 17 - Communication-related Challenges and their Dependencies

Communication Mechanisms

The question of how to conceptually deal with system-wide communication on a high-level is one that transcends and precedes any technology choices. The decisions made with regards to this should be well-considered as they affect the entire system. As Richardson [48] describes, “thinking first about the interaction style will help you focus on the requirements and avoid getting mired in the details of a particular IPC [Inter-process Communication] technology.” They categorise the possible interaction styles in two dimensions. Interactions can differ in multiplicity; i.e. be one-to-one or one-to-many. They can also be either synchronous or asynchronous. Richardson proceeds to give an overview of interaction styles that fit with certain characteristics as shown in Table 9 below.

Table 9 - Service Communication Mechanisms - Adopted from [48]

	One-to-one	One-to-many
Synchronous	Request/response	-
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/asynchronous responses

In one-to-one communication, exactly one service is invoked by a client request. When using one-to-many communication, obviously more than one service can be involved. The main difference between synchronous and asynchronous communication is that in the first case, the client invoking a service waits and possibly blocks while waiting for a response. Because of this, synchronous messaging can typically only occur in a one-to-one fashion, using the ubiquitous *request/response* interaction style.

When using asynchronous communication, the client does not block and the response might come at a later point in time, or not at all. The *asynchronous request/response* interaction style is basically a request/response mechanism in which the client does not block but does wait for a response for a certain amount of time. The o*ne-way notifications* interaction style on the other hand is self-explanatory; a client invokes a service but does not require or process a response.

A further well-known interaction style is that of *publish/subscribe*, in which clients can publish messages which are then communicated to any number of services subscribed to them. This approach is used widely in distributed systems. *Publish/asynchronous responses* is a variation to this, in which the client sending the message waits for some time to receive responses from subscribed services.

Richardson further argues that “synchronous communication with other services as part of request handling reduces application availability. As a result, you should design your services to use asynchronous messaging whenever possible” [48]. However, as stated in [63], “interestingly, microservices are most suitable for asynchronous communication, bringing performance, decoupling and fault-tolerance, but the paradigm shift implied has not been overtaken yet in practice”. Another implication of microservices to the choice of a communication paradigm is that one “should also take in to consideration the possibility that a service might not respond” [55].

Decision alternatives:

- One-to-one / one-to-many communication
- Synchronous / asynchronous communication

Keywords: *one-to-one, one-to-many, synchronous, asynchronous, request/response, asynchronous request/response, one-way notifications, publish/subscribe, publish/asynchronous response.*

Service Interconnection

The concern of service interconnection deals with the implementation connecting services to each other and exchanging data. The choices that can be made with regards to this depend on the choice of *Communication Mechanism*. Especially the choice of whether communication is done in a synchronous or asynchronous way is decisive. The authors in [64] discuss choices of inter-service communication patterns that are common in microservice architectures. The three that they mention are direct calls to services, using a gateway, or using a message or service bus, each with their own advantages and disadvantages. In his book on microservice patterns, Richardson [48] describes two main alternatives; synchronous 'Remote Procedure Invocation' (RPI) – conceptually analogous to RPC (invoking a function in another service) – and asynchronous messaging.

When using synchronous or request/response style communication, invoking a microservice is most commonly done over either *REST*, which uses HTTP, or a binary RPC protocol such as *gRPC*. REST stands for Representational State Transfer and is built around resources that typically are business objects such as orders or users. REST uses the default HTTP request methods such as GET, POST and DELETE to perform operations. This way of communication is popular in the microservice field because of its simplicity, familiarity, easy to test and directly supports the request/response style of communication. This last point is also one of its drawbacks, as this is in fact the only style of communication that it supports [48]. Furthermore, updating multiple entries at once is seen as difficult using REST. This limitation is addressed by opting for a protocol such as gRPC which addresses this, and also has a more extensive set of update operations as compared to the HTTP operations used by REST. Since gRPC is a binary protocol, it is able to provide more efficient communication between services. However, it does require adding code to services for consuming and serialising requests and responses. There are off-the-shelf solutions available for this serialisation, such as the popular Protocol Buffers solution developed by Google[6]. A further advantage of gRPC is that besides supporting request/response style communication, it can also be used in messaging-based communication [48]. Furthermore, to facilitate failures and provide resilience against unresponsive or unavailable services in this style of communication, a *gateway* can be put in place. By implementing functionalities like these, this logic does not need to be part of a service itself. However, this is yet another component in the architecture program and maintain.

In asynchronous communication, services no longer synchronously receive responses to their requests, but rather exchange messages. As such, a *messaging* pattern should be used to facilitate this exchange. Richardson [48] refers to work by Hohpe and Woolf [65] to help describe messaging using channels. "A sender (an application or service) writes a message to a channel, and a receiver (an application or service) reads messages from a channel" [48]. Such a message channel is an abstraction of the actual messaging infrastructure. Message channels can be either *point-to-point* for one-to-one interactions or of a *publish/subscribe*

[6] https://developers.google.com/protocol-buffers

nature for one-to-many communication. The underlying messaging infrastructure is often implemented in the form of a *message broker*. This takes some implementation effort of the development of the actual microservices, at the expense of introducing a possible single point of failure in the architecture. A *brokerless* approach can also be used, in which services exchange messages without the additional service broker component. Other advantages of brokerless messaging are less network traffic and no potential performance bottleneck that a message broker could introduce. However, this approach also comes with its own drawbacks such as the lack of message buffering capabilities that message brokers often implement and the need for services to be aware of each other's locations through service discovery (to be discussed further). Furthermore, in a brokerless setup the availability can be reduced since both the sending and receiving service must be available at the time of exchanging a message. Richardson concludes that "because of these limitations, most enterprise applications use a message broker-based architecture" [48].

As said, the style of service interconnection is dependent on the chosen communication mechanism. When using asynchronous communication, a REST approach cannot be used. The same goes for one-to-many interactions. Conversely, a messaging pattern can still support one-to-one styles of interaction and even mimic a request/response pattern to an extent in the case a client blocks while waiting for an asynchronous message as response to its request.

Decision alternatives:

- RPC / messaging
    - In case of RPC:
        - REST / gRPC mechanisms
        - Gateways
    - In case of messaging:
        - Point-to-point / publish/subscribe
        - Message-broker / brokerless

Dependent on:

- Communication Mechanisms

Keywords: *RPC, messaging, REST, gRPC, point-to-point, publish-subscribe*, *message-broker, brokerless.*

Service Discovery

A main concern in this is service discovery; the question of how certain services know about the existence of other services and how to connect to these. As described in many of the studies such as [66]–[68], service discovery mechanisms are an essential part of microservice architectures and often prove to be the greatest challenge to successfully implement such an architecture. This involves implementing a means of discovering which application instances – such as machines or containers – offer what services and keeping a service registry of this. When an application or service then wants to connect to another service, the available instances along with their addresses and exposed interfaces can then be read from this registry. The function of deciding which instance to connect to can then be made by the service itself or be taken care of by a load balancer.

The difference between service discovery approaches is in how a client finds out the location of a service and how it connects to it. There are two main means of accomplishing this; *client-side* or *server-side* discovery [63]. Client-side discovery works by having a client request the locations of services from a service registry. The client then selects an instance and directly connects to it. When using server-side discovery, the client makes a request to a service

through a server, which is commonly a gateway or load balancer. This then selects an instance and forwards the client's request.

The decision of whether service discovery is needed, depends on the type of service interconnection that is being used. For example, service discovery is common when using RPC-type interaction, but in a messaging-based setup with a message broker it is redundant since the message broker already facilitates this functionality. On the other hand, when using brokerless messaging, service discovery is needed again. Therefore, decisions on service discovery are not directly dependent on communication mechanisms, but rather service interconnection style.

Decision alternatives:

- Client-side / server-side discovery

Dependent on:

- Service Interconnection

Keywords: *service discovery, load balancer, gateway, client-side discovery, server-side discovery.*

Interface Design

The aspect of interface or API design is part of enabling service integration. Challenges in interface design are for example extensibility, backwards compatibility and abstraction of implementation details. Service contracts are sometimes seen as a means to aid this by defining standard communication patterns that services should adhere to. The overall goal in all these challenges is to ensure that changes to one service do not cause system failure and require as little effort as possible for adapting other services to for instance use new functionalities. In the end, the API specification, defines what a service does and in what way. "A well-designed interface exposes useful functionality while hiding the implementation" [48].

A so-called Interface Description Language (IDL) should be used to specify an API. Such a language describes an API in a language-independent way. The use of and IDL is both vital as there are typically many services involved and key in enabling organisations to make full use of the advantages of a microservice architecture. For REST, a common IDL is the Open API specification [69]. Other examples include – but are not limited to – the RESTful API Modelling Language (RAML) [70] and the Web Services Description Language (WSDL) [71]. A further advantage of specifying an API using a standardised IDL is that often there are tools available to help implementing it. For example, Swagger [72] provides solutions to help design and build APIs based on the OpenAPI standard. When using a gRPC approach, the specification for how to design an API is already set as the Protocol Buffers-based IDL [73] should be used.

When describing an asynchronous message-based API, the message channels, message types distributed over these channels and their formats should be specified. There is, however, no widely adopted standard for this and thus this is done rather informally [48]. Besides this, the available operations that clients can invoke should be described as well as the events that services can emit.

In general, literature seems to agree that whenever possible, a standardised IDL should be used to design and document APIs. Given that the options to do so are rather constrained by the choice of service interconnection style and the fact that the different available IDLs are quite similar apart from the actual representation of the API specification, there are no high-level choices to be made in this regard.

Guidelines:

- Use a standardised IDL whenever possible
- Consider using provided tooling for these IDLs

Dependent on:

- Service Interconnection

Keywords: *interface design, IDL, API design*

6 TREATMENT DESIGN

In the description of the Design Science Methodology, Wieringa [4] refers to the artifact to be designed as a *treatment*, which interacts with the problem context. In their words: “the design science researcher designs not just an artifact, but designs a desired interaction between the artifact and the problem context, designed to treat the problem” [4]. For this research, this respectively refers to the interaction between the decision-making framework and the design of a microservice architecture. This section discusses the first steps in the design of this interaction. A first step in this is the definition of requirements, showing their link to the stakeholder goals, and researching any available (partial) treatments to the problem.

6.1 Requirements

To explore the requirements that need to be fulfilled in order to fulfil the stakeholders’ goals, first the software architecture design process and its interaction with decision-making should be understood. As said, not all challenges are merely architecture-design problems, but many of them are. It is therefore useful to consider what a software architecture design process looks like in general, and how the practice of decision-making fits into this. The prospective decision-making framework will need to support this process, but not necessarily be limited by it.

6.1.1 Software Architecture Design and Decision-Making Process

In software engineering, software architectures are used to design and analyse software systems. This is done on a high level, so that one can reason about the structures of large and complex systems. It describes such structures by their elements, relations and the properties of these elements and relations [74]. As Kruchten et al. state, this is “the key to achieving intellectual control over a sophisticated system’s enormous complexity” [75] and facilitates both system design and maintenance [74]. When used at the start of a system’s design process, a software architecture “significantly constrains and facilitates the achievement of requirements and business goals” [10]. Therefore it can be used to validate whether a system being developed is adhering to the set objectives [76]. Falessi et al. [10] also rightly indicate that every software system has a software architecture; implicit or explicit.

The driving factors behind software architecture design were described by Kruchten [77] as *reuse, method* and *intuition*. They state that many software architects rely on their own experience and *intuition* when designing software architecture elements. Some follow a certain *method* – i.e. language or process model – to help designing a software architecture. *Reuse* is also used quite broadly; both in the sense of reusing parts of previous or similar systems, as well as organisation-wide experience and domain-specific knowledge. A driving force behind *reuse* is that it can help simplify the difficult task of architecting [77].

The possible designs of a software (sub)architecture are also referred to as *architectural alternatives* [10]. Often, there are multiple viable ways to design part of a software architecture. Such possible designs of software architectures are also influenced by the means by which they can be implemented and the required effort to do so. There might also be (commercial) off-the-shelf (OTS) solutions that can solve specific challenges in a (sub)architecture – possibly in conjunction with other solutions. ‘Commercial’ is optional here, since the nowadays there are many Open Source Software (OSS) solutions [78] to architectural challenges

available, sometimes for free. Such solutions could be used by multiple organisations facing similar challenges, and these same organisations can in their turn sometimes also contribute to the development of this software. Depending on the available solutions for architectural challenges and the requirements and constraints to a system, the decision to either develop part of a software system from the ground up or use an OTS solution can be made.

With usually many possible architectural alternatives for solving an architectural problem, the question becomes how to choose between them. This is where decision-making techniques become relevant to help software architects choose between architectural alternatives by describing a process and methodology to systematically compare options.

The basic process of deciding between architectural alternatives is relatively straight-forward. Falessi et al. [10] describe it on a high level using three main phases in an iterative process. A schematic representation of this is shown in Figure 18.

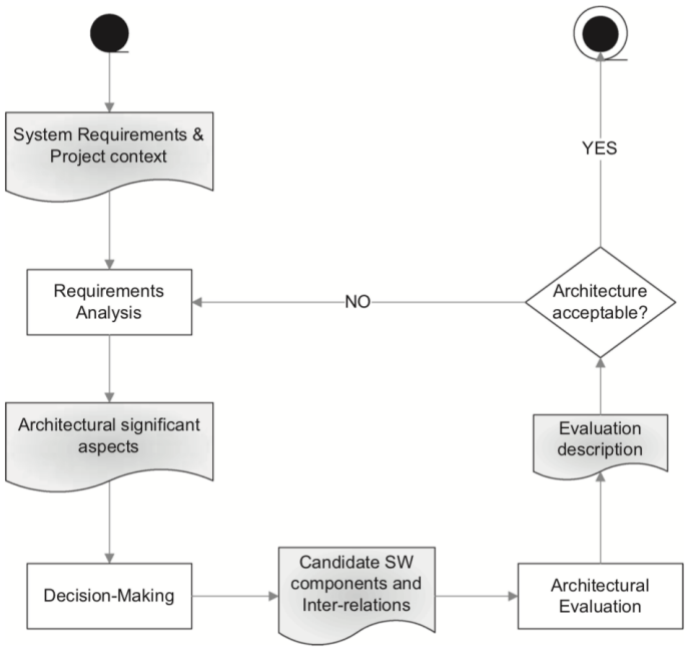


Figure 18 - Software Architecture Design Process - Adopted from [10].

First is the *requirements analysis* phase, in which a software architect aims to develop an understanding of the problem by “extracting the most critical needs from the big, ambiguous problem description”. These system requirements are influenced by the project context. Also, an organisation’s business strategy or goals may be of influence [79]. The output of this process should be requirements that are architecturally significant. Quality attributes are determined which together with business goals form the basis for architectural decisions. In the next phase of *decision-making*, the solutions that fulfil the established requirements are sought. The properties of different available options and their relationships are defined to allow comparing these options. This results in candidate solutions. Finally, the question of how well these alternative solutions solves the problem is answered in the *architectural evaluation* phase. The process can then be repeated if the end result is not an acceptable architecture.

While this process description given by Falessi et al. [10] is clear and highlights the main activities, it is not explained in too much detail. A work by Kontio [80], in which they describe the so-called OTSO process for reusable (OTS) component selection, gives more comprehensive though still high-level guidance on the decision-making process. For example,

they explicitly address the searching and selection process of possible architectural alternatives. An overview of the main phases they define is shown in Figure 19.

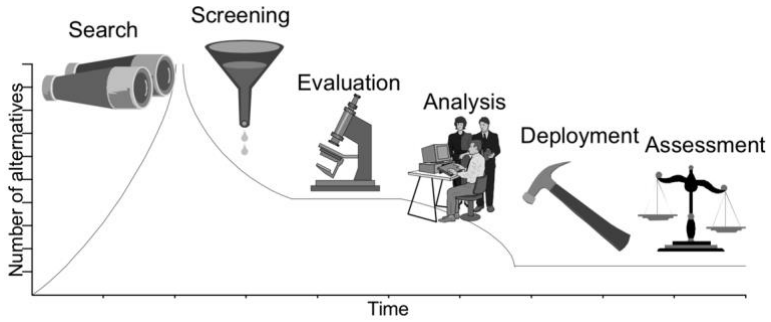


Figure 19 - Main Phases in Reusable Component Selection Process - adopted from [80].

In this graph-like overview, the vertical axis is used to show the number of alternatives being considered in a phase and the horizontal axis to show time progression throughout the selection process. During the *search* phase, there can be many alternatives that arise from literature, the internet, vendors, experts and many more sources. The goal here is to identify all possible candidates. These are then put through a *screening* process to determine which of the found alternatives are worth to consider in a more detailed evaluation. These are then – similar to what was described by Falessi et al. [10] – *evaluated* and *analysed* to select the most promising alternative. After this, Kontio also puts emphasis on *assessing* how successful the selected alternative or component was in solving the given problem after *deployment*. This might impact the potential further reuse of this same alternative. Besides this, the design process can be further improved given the experiences of the just completed selection process. As stated before in section 3.3, decision-making techniques used in this process should deal with multiple stakeholders, competing and conflicting objectives, uncertainty both in the descriptions of requirements and in their associated solutions, and interdependencies between decisions [10].

To show where these decision-making processes fit in the overall processes for software architecture design exist, a general model of software architecture design – shown in Figure 20 – developed by Hofmeister et al. [81] is used. This model is based on five approaches from industry, for which the commonalities were analysed.

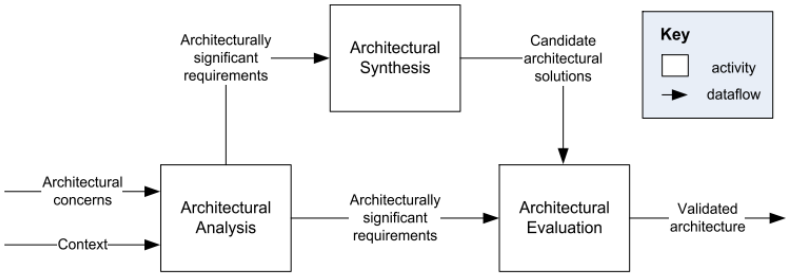


Figure 20 - Architectural Design Activities - adopted from [81]

The decision-making process used to decide between architectural alternatives largely lies in the *Architectural Synthesis* step in this model. Requirements are used as input to select between alternatives. However, the view shown before from work by Falessi et al. [10] shows that in general, most methodologies aimed at choosing between architectural alternatives take into account more than the mere decision-making step. Often, they also describe how to identify and formulate relevant requirements as well. As for analysis of the choices made; this is mostly done in the form of comparing the decisions made with the requirements and

constraints set at the start of the process. If these are fulfilled, the process is seen as complete. That still leaves out the actual implementation and operational evaluation to assess how well these decisions perform into practice. Furthermore, the *Architectural Analysis* step generally not given too much detail. How to arrive at the architecturally significant requirements used as input for the decision-making process is often not described in too much detail. Therefore, the main focus of these decision-making methodologies can be said to be on the *Architectural Synthesis* part of software architecture design.

6.1.2 Requirements Definition

These descriptions together with those given in the Problem Investigation stage can be translated into requirements for the decision-making framework to be designed. The goal is that in fulfilling these requirements, the desired interaction between the artifact and the problem context can be created. As was done before for the goal-level requirements, the requirements are subdivided according to the goal-design scale formalised by Lauesen [11] and are shown in Table 10. The initial goal-level requirements are also included in this table for completeness.

Table 10 - Decision-Making Framework Requirements

	Goal-level requirements
G1	The framework shall improve decision-makers' work in managing design challenges related to communication between, integration and management of microservices.
G2	The framework shall give confidence in its outcomes.
G3	The framework shall require limited effort in its use.
G4	The framework and its outcomes shall be practical.
	Domain-level requirements
D1	The framework shall support the selection of optimal architectural alternatives.
D2	The framework shall support the use of quality attributes for rating alternatives.
D3	The framework shall support input from multiple stakeholders.
D4	The framework shall deal with competing and conflicting objectives.
D5	The framework shall take into account uncertainty both in the descriptions of requirements and in their associated solutions.
D6	The framework shall deal with interdependencies between decisions.
D7	The framework shall support providing guidelines and industry practices when no clear decision can be made for managing a challenge.

6.2 Contribution to Goals

The DSM prescribes that the set requirements should be justified by providing a contribution argument in the following form [4]:

(Artifact Requirements) x (Context Assumptions) contribute to (Stakeholder Goal)

For the justification of requirements G2 through G4 in their contribution to G1, the following argument can be given:

- *If the designed decision-making framework gives confidence in its outcomes, requires limited effort in its use and is practical in use,*
- *and assuming that it is used in the design of a microservice architecture as described in this project,*
- *then the framework contributes to decision-makers' goal of better managing design challenges related to communication between, integration and management of microservices.*

As described in section 3.3, goals 2 through 4 are vital to fulfil in order to ensure that stakeholders will find the framework useful and be inclined to use it in practice. These requirements describe non-functional properties of the artifact. These should be made measurable when validating the artifact. The question is how to measure confidence, ease of use, usefulness and practicality? In other words, indicators for these properties should be defined. For measuring usefulness and ease of use, the questions defined by Davis [82] in their paper describing the Technology Acceptance Model (TAM) can be used. The question of confidence comes down to measuring decision quality. This can be done using the six elements of decision quality described by Spetzler et al. [83]. The authors argue that decision quality has six requirements, shown in Figure 21. These can be used to phrase questions to assess the decision quality indicator.

Figure 21 - "The Decision Quality Chain" by Spetzler et al. [83]

Practicality can be hard to measure, and mostly comes down to how practical stakeholders find the artifact and its outcomes to use. In interviews, this can for example be operationalised by requiring that interviewed stakeholders must not find any elements of the framework and its outcomes that inhibit their usefulness in practice. This then acts as an acceptance norm.

The domain-level requirements are of a functional nature and their fulfilment can therefore be tested by observing the interaction between an artifact and its context that contributes to a service to a stakeholder [4]. The fulfilment of D1 through D7 can be measured by observing this interaction. These requirements were set based on information from literature as well as insights from practice that indicate what functions the decision-making framework should support. The contribution argument that can be given here is as follows:

- *If the designed decision-making framework satisfies the domain-level requirements,*
- *and assuming that it is used in the design of a microservice architecture as described in this project,*
- *then the framework supports the required functions indicated by literature and practice, and thus contributes to decision-makers' goal of better managing design challenges related to communication between, integration and management of microservices.*

6.3 Available Treatments

The next step in the DSM is to search for any already available treatments to the design problem. A logical first step here is to explore what decision-making methodologies used in software architecture design are already available. As said before, a large part of managing microservice challenges involves choosing between software architecture alternatives but is not necessarily limited to it. Therefore, an existing decision-making methodology might serve as a conceptual foundation to build upon. This search should answer RQ-4: *What decision-making methodology for selecting between design alternatives can serve as conceptual foundation for the framework to be designed?*

6.3.1 Decision-Making

There are many different decision-making methodologies for selecting between architectural alternatives in existence, each with their own strengths and weaknesses. In the aforementioned paper by Falessi et al. [10], the authors provide a very thorough comparison to help software architects decide between these methodologies. They also elicit the conceptual elements and possible difficulties that come with each methodology in great detail.

Even though the authors compared 15 different decision-making methodologies, they found that all follow the basic process described previously and shown in Figure 18 and have a lot in common at the conceptual level, even though they might look quite different [81]. All these methodologies incorporate multiattribute decision-making methods to select between a finite number of alternatives.

Falessi et al. [10] go on to describe the various elements that the selected decision-making methodologies comprise and the mechanisms of implementing these elements they found in the methodologies. The first is how *quality attribute description* is done to facilitate a clear understanding of each attribute by all involved stakeholders. *Quality attributes' performance* can also be expressed in various ways to show which are most important to stakeholders. The *description of fulfilment* is used to communicate how well alternatives fulfil the previously described attributes. This fulfilment can come with an *uncertainty description* to quantify risks associated with a certain choice of alternative. Together, these elements are used to classify and compare the selected methodologies.

Table 11 - Elements and Mechanisms of Decision-Making Methodologies as described by [10]

Element	Mechanism	Example
Quality attribute description	Just a term	Performance
	Term and use case	Performance to perform <function>
	Term and measure	Performance in seconds
	Term, measure, use case	Performance to perform <function> in seconds
Quality attribute importance	No articulation	-
	Direct weight	Weight rating 5/10
	Elicited weight	Weight rating compared to other attributes
	Utility curve	Weight depending on fulfilment
Description of fulfilment	On/Off	Satisfies <attribute>
	Order	Medium/good/excellent fulfilment of <attribute>
	Direct ratio	Score 0.5 of fulfilment of <attribute>
	Elicited ratio	Fulfilment level compared to other alternatives
Uncertainty description	Not expressed	-
	Inferred from disagreement	Uncertainty if stakeholder 1 and 2 disagree on how well <alternative> satisfies <attribute>
	Related to quality attributes	Alternative certain to fulfil <attribute 1>, but uncertain to fulfil <attribute 2>
	Included as quality attribute	Alternative inherently has low uncertainty

The authors describe their view of a *good* decision-making methodology as "one that avoids the selection of a worse alternative and, at the same time, is easy to use" [10]. This is partially because all researched methodologies share the characteristics that they:

- involve several stakeholders,
- deal with competing and conflicting objectives,
- show a level of uncertainty in both the description of requirements as well as in their associated solutions,
- show that the decisions taken have strong interdependencies.

Because of these similarities, the authors compare the methodologies based on certain difficulties that might arise based on the mechanisms in which they implement the various elements shown in Table 11. Hence, the authors conclude that "there is no 'best' decision-making technique; however, some techniques are more susceptible to specific difficulties" [10]. They therefore argue that architects should compare decision-making methodologies based on the difficulties that they wish to avoid. The authors ranked the methodologies based on the susceptibility to each difficulty to facilitate this. They further state that some methodologies are merely more susceptible to certain difficulties, and that these can be overcome by adopting or tuning a methodology to reduce the impact of these difficulties.

6.3.2 Decision-Making Methodology Selection

Before relying on this comparison made by Falessi et al. [10] for selection of a decision-making methodology as conceptual basis, one should consider whether any new methodologies treating these decision-making problems have been published since the publication of their paper in 2011. When searching academic literature on this topic on Scopus[7] the search query used was as follows:

[7] https://scopus.com

TITLE-ABS-KEY(software architecture decision making) AND PUBYEAR > 2010 AND (LIMIT-TO (SUBJAREA,"COMP"))

As can be seen, the search was limited to the field of computer science. From these results, the title, abstracts and keywords of any works that had been cited more than 10 times were analysed to see if they were concerned with decision-making in software architecture design by selecting between architectural alternatives. No such methodologies were found. Therefore, the works discussed by Falessi et al. are considered as possible options to base the framework design of present work on.

Going with the authors' advice of selecting a decision-making methodology based on difficulties, the question first becomes which of these to avoid. In their paper, Falessi et al. [10] define 9 difficulties based on related literature to the susceptibility of researched decision-making methodologies' characteristics to difficulties.

As stated in section 6.1, the goal is to design a decision-making framework that is practical, requires limited effort and gives confidence in its results, while supporting a group decision process and dealing with competing objectives and interdependencies between decisions. Fortunately, the requirements of facilitating group decision making and dealing with competing objectives and interdependencies between decisions are often all addressed by the methodologies selected in the study by Falessi et al. [10]. This leaves the main characteristics of practicality, required effort and confidence as the primary factors for identifying candidate methodologies. These will be used to identify the difficulties that are most important to avoid.

From the difficulties identified by the authors, there are some that show a wide disparity in susceptibility of different methodologies to them. The first two are the either a *too complex* or *too simple description* of a solution [10]. The example given of a too complex situation are having to select between many alternatives using many quality attributes with many rating options. The opposite is the case for a too simple situation, few alternatives, quality attributes and rating options. Both extremes are detrimental to making the right decisions. Given the requirements, a logical decision is to aim at finding a technique that balances these difficulties. Intuitively, those methodologies that are least susceptible to describing solutions *too simple*, might be more prone to use overly-complex descriptions.

The next difficulty that distinguished the researched methodologies most is *coarse-grained indication* of stakeholders' satisfaction of alternatives or fulfilment levels. If this indication is too coarse, relevant details can be neglected [10]. The example given here is how a yes/no scale for indicating whether a solution's costs are acceptable can neglect the fact that, whilst both acceptable, some solutions can be considerably less expensive than others. This situation is undesirable from a *confidence* perspective, as likely not all relevant quality attributes of solutions are compared well enough. This again possibly results in sub-optimal decisions being made.

Furthermore, the methodologies show some variation in the susceptibility to the difficulty of having *insufficient time* to analyse a decision. As in many situations there are time constraints to finding a solution, a methodology should limit the time – or *effort* – required for analysis. This difficulty also has interdependencies with others, where a methodology requiring less time to carry out, can increase the risk of evoking other difficulties. For instance, the methodology described as least susceptible to having insufficient time available showed to be highly susceptible to utilizing a *too simple* description of solutions [10].

To compare the various methodologies, the ranking table as by Falessi et al. [10] was altered to highlight those that are least susceptible to difficulties overall, as well as to those discussed.

This was done by first calculating the methodologies' sum of rankings as indicated by SUM. Another attribute – SUM' – was assigned to show their rankings on the discussed challenges that most divide the methodologies; *too complex* or *too simple description*, *coarse-grained indication* and *insufficient time*. In both the SUM and SUM' ratings, a lower number means less overall susceptibility to the associated challenges. The methodologies were then sorted first by SUM, then by SUM' to end up with the ranking in Table 12. The variation between rankings is shown at the bottom of the table and the ranking scores are emphasised by showing methodologies with a worse ranking on a criterium in a darker colour. Also, even though the other identified difficulties showed less differences between methodologies' susceptibility to them, they are nevertheless important to consider. These will therefore be used in further comparing the selected methodologies in-depth.

Table 12 - Annotated Comparison of Methodologies – adopted from [10]

Methodology	Too complex description	Too simple description	Coarse-grain. ind.	Insufficient time	Coarse-grain. des. of needs	Linear and mon. satisfaction	Value perception	Underest. uncertainty	Perception of a term	SUM	SUM'
OTSO	7	2	1	2	1	1	1	1	1	17	12
Impact Estimation	4	3	2	2	1	2	1	1	2	18	11
Quality driven (Svahnberg et al. [2003	7	2	1	2	1	1	1	1	2	18	12
CBAM 2	7	1	1	4	1	1	1	1	1	18	13
Quality driven (Al-Naeem et al. [2005]	5	3	1	2	1	1	1	3	2	19	11
A framework for design tradeoffs	5	3	1	3	1	1	1	3	1	19	12
CBAM 1	6	2	2	2	1	2	1	1	2	19	12
BAREMO	4	3	2	2	2	1	2	2	2	20	11
ISO 9126	4	3	2	2	1	2	1	3	2	20	11
NeMo-CoSe	3	3	3	2	2	2	2	2	2	21	11
Quantitative methods	4	4	2	2	1	2	1	3	2	21	12
CEP	8	2	2	2	1	2	1	1	2	21	14
Softgoals	2	4	4	2	2	3	2	2	2	23	12
Dabous and Rabhi [2006]	1	5	5	1	3	3	3	1	2	24	12
PORE	1	6	5	2	3	3	3	1	1	25	14
Ranking Variation	8	6	5	4	3	3	3	3	2		

When weighing all rankings, the OTSO methodology by Kontio [80] shows the least overall susceptibility to any difficulties. However, this methodology does have a high susceptibility to the difficulty of a *too complex description* of a solution. The same is true for two of the three next best ranked ones; namely the methodologies by Svahnberg et al. [84] and Moore et al. [85]. As stated before, the aim is to balance the risks of having a *too-complex* and *too-simple description* of solutions. Furthermore, the methodology by Moore et al. shows to be highly susceptible to having *insufficient time* to carry out the analysis.

The Impact Estimation methodology by Gilb and Brodie [86] seems promising though, and so does the quality-driven method by Al-Naeem et al. [87]. These both have the lowest found score for SUM'. The methodology by Gilb and Brodie ranks better for avoiding a *too complex description,* whereas that by Al-Naeem et al. is less susceptible to *coarse-grained indication* of stakeholders' satisfaction. They further differ in that Al-Naeem et al.'s methodology introduces a risk of *underestimating uncertainty*, but that by Gilb and Brodie scores worse on susceptibility to *linear and monotonic satisfaction*. The uncertainty is important to keep in mind as this says something about the risks involved with a certain solution. The risk of linear and monotonic satisfaction can occur when, in order to cope with complexity, "the satisfaction of each quality attribute is modeled to grow linearly as its fulfillment grows" [88]. An example would be when a network solution that provides a data throughput rate that is significantly higher than needed, while scoring worse on other attributes such as latency. When satisfaction is modelled in a linear way as described, this solution could get a higher score because of it – unnecessarily – outperforming other solutions performance, therefore increasing the risk of

neglecting other solutions that provide lower but adequate data throughput rates at a lower latency.

To elicit where these differences in ranking come from, the so-called *'Impact Estimation'* by Gilb and Brodie [86] and *'ArchDesigner'* by Al-Naeem et al. [87] methodologies were studied in more detail. A first difference is the description quality attributes that are used; *ArchDesigner* uses just a term, whereas *Impact Estimation* adopts a term and a measure. According to the comparison by Falessi et al. [10], this decision influences the methodologies' rankings on the *insufficient time* and *perception of a term* difficulties. Both rank equal on the perception of terms, but the approach in *Impact Estimation* scores worse on insufficient time. Further differences arise in the mechanisms by which quality attributes' importance and the description of fulfilment are determined. *Impact Estimation* uses a direct weight and ratio approach to these, respectively. *ArchDesigner* elicits these elements by using pairwise comparisons following the Analytic Hierarchy Process (AHP) as described by Saaty [89], and as such its mechanisms are classified as an elicited weight and elicited ratio. Falessi et al. describe that AHP generally yields more precise results, albeit with an increase in required effort. Refer to Table 11 to for examples of these mechanisms. This difference in mechanisms influences many of the methodologies' rankings on difficulties described by Falessi et al. [10], with the elicited weight and ratio mechanisms generally being more susceptible to having *insufficient time* and having a *too complex description*, but being less susceptible to the *linear and monotonic satisfaction, coarse-grained indication* and *too simple description* difficulties.

The argument given by Saaty [89] against directly assigning scores using *absolute judgement* is subjective and can therefore be error-prone, stating it is "considered to be a questionable practice when objectivity is the norm". Using pairwise comparisons in the form of the AHP for decision making is seen as a means to aid these shortcomings. Instead of asking "how good is alternative A at fulfilling quality attribute X?", the question becomes "how much better or worse is alternative A at fulfilling quality attribute X compared to alternative B?". By comparing all possible combinations, an overall score can be assigned to each alternative for its fulfilment of quality attributes.

When choosing between the approaches in the two decision-making methodologies, the weights of conflicting requirements of *required effort* and *confidence* needs to be considered. I.e. is the extra confidence that using elicited weight and ratio mechanisms provide by implementing the AHP worth the required additional effort? Given that both *Impact Estimation* and *ArchDesigner* show a moderate ranking of being susceptible to having *insufficient time* in the comparison by Falessi et al. [10], the choice for the more systematic AHP approach that yields more precise results seems justifiable.

However, in the description of uncertainty to quantify risks the *ArchDesigner* methodology differs vastly from *Impact Estimation* in the fact that it incorporates no mechanism to express uncertainty, whereas *Impact Estimation* expresses it by relating uncertainty to each quality attribute. As can be seen in Table 12, this makes the methodology by Al-Naeem et al. highly susceptible to *underestimating uncertainty* about the risks involved with a certain solution. The approach by *Impact Estimation* is ranked as second-best with regards to this difficulty. Disregarding the chance of *underestimating uncertainty* would be risky and would harm the confidence in the methodologies' outcome. This point is also addressed by the authors when discussing the limitations of their methodology as they state that "Judgment consistency level was not measured before computing value scores" and "measuring the consistency level of stakeholders' judgements would help ensure the accuracy of the judgements" [87]. With this, they seem to propose a mechanism of determining uncertainty by inferring this from disagreement between stakeholders. The methodology described by Svahnberg et al. [84],

helps in clarifying how to manage this uncertainty. After all, Falessi et al. propose that a selected decision-making methodology can be adopted and tuned to reduce the risk of any identified difficulties [10]. On a high level, the approach by Svahnberg et al. follows a process similar to that by Al-Naeem et al; also incorporating the AHP as main supporting method of prioritisation and selecting architecture candidates. To elicit any uncertainty in these outcomes, they propose calculating the variance in scores given by different stakeholders to have a measure of disagreement. Because of the methodologies' similarities, this technique could also be applied to *ArchDesigner*, therefore reducing the risk of *underestimating uncertainty*.

A valid question then is: if the techniques by Al-Naeem et al. and Svahnberg et al. are largely similar, why not choose that by Svahnberg et al. in the first place? A first argument in this is that the approach taken by Al-Naeem et al. in creating *ArchDesigner* is intended to be more practical. This is motivated by the requirement of practicality for selecting the right decision-making methodology. The authors state that previous approaches – in which they also refer to the work of Svahnberg et al. – "evaluate and select among given coarse-grained SAs [software architectures] without giving guidance on how to arrive at these architectures" [87]. *ArchDesigner* is said to be aimed at evaluating and selecting candidates in a fine-grained fashion, which should help stakeholders at arriving at a suitable SA solution. The authors say that similar techniques often assume that a small set of architecture candidates have already been created without giving any guidance on how to arrive at these architectures. *ArchDesigner* is said to offer "guidance quite early during the architectural design process. This is achieved through the evaluation of various fine-grained design options, which together produce the resulting SA" [87]. Furthermore, the solution of using stakeholder disagreement to manage the difficulty of underestimating uncertainty when using this approach is straight-forward to adopt, whereas it is unclear to what extent the methodology by Svahnberg et al. can be altered to be more practical in nature.

Given the ease with which the risk of *underestimating uncertainty* can be reduced, the use of a more precise mechanism for determining *quality attribute importance* and *description of fulfilment* and a more practical attitude to decision-making by the authors, *ArchDesigner* by Al-Naeem et al. [87] is thus considered the best decision-making methodology to serve as conceptual basis for the design of the artifact, and is thus the answer to RQ-4. This methodology will be adapted to reduce the risk of encountering the difficulties it is most susceptible to. It will need to be further adapted and built upon to realise the requirements and goals set in this research, as the prospective artifact needs to address microservice architectures, and in particular the selected management, integration and communication categories of challenges.

6.4 Overview of ArchDesigner

Al-Naeem et al. [87] structure the ArchDesigner methodology in several distinct steps. These steps are shown in Figure 22 below. One can see how each design decision is treated separately in the first steps, but ultimately the process leads to an optimisation problem that attempts to find the best overall solution across multiple design decisions. The authors also describe how the overall solution can be subject to certain constraints such as cost or time.

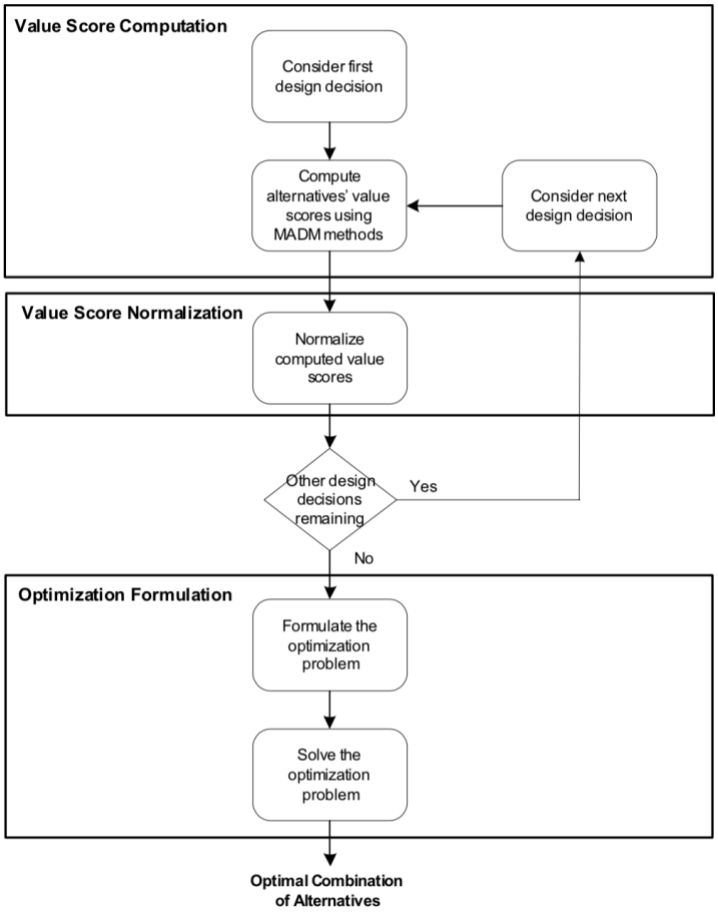


Figure 22 - ArchDesigner Process – Adopted from [87]

The ArchDesigner methodology essentially comes down to first identifying which requirements or Quality Attributes (QAs) should be considered and what alternatives are available, as well as their interdependencies. These QAs are then ranked to establish their comparative importance or weights. The identified alternatives are then compared based on how well they fulfil the selected QAs. After this, each alternative is assigned an overall score based on the fulfilment of the QAs combined with each QA's weight. Alternatives' scores are then normalised in order to express which decisions are more important than others. For example, an architectural choice that has great impact on other decisions might be given a relatively high weight in this step. The authors propose to ultimately formulate the scores obtained in the first steps towards an optimisation problem and subsequently try and solve this problem. This gives an indication of the optimal alternative for each decision made based on the scores assigned to them when applying the methodology. Constraints to the overall solution can also be expressed when solving this optimisation problem; for instance, that the overall should not exceed a certain cost.

Even though – according to the authors – others methods could be used, the ArchDesigner methodology relies heavily on the use of the AHP described by Saaty [89] as Multi-Attribute Decision Making (MADM) method. Central in AHP is the pairwise comparison between items to establish their relative significance. For defining the overall importance of QAs, each is compared to all others and assigned a number to show its relative importance. In practice one could ask:

- *When comparing QA 1 and QA 2, which one is more important, and to what extent?*

The extent to which one QA is more important than another is expressed on a 1-9 scale, where 1 means both are equally important, 3 means one is moderately more important, up to 9 which signifies one requirement as being extremely more important than another. The AHP then describes how to aggregate these pairwise scores to arrive at an overall ranking of requirements.

The same pairwise comparison is done when evaluating alternatives, only this time both sides are compared in relation to a QA to assess their fulfilment of this QA. For example:

- *When comparing Alternative A and Alternative B, which one fulfils QA 1 best, and to what extent?*

Each pair of alternatives is compared in relation to each significant QA. After also aggregating these rankings, this gives an overview of how well each alternative fulfils – or rather, is expected to fulfil – each QA. Combined with the established weights of QAs, the highest scoring alternatives can then be determined.

As said in section 6.3, ArchDesigner was susceptible to underestimating uncertainty according to the findings of Falessi et al. [10]. It was proposed that including a variance calculation as done by Svahnberg et al. [84] in scores given by different stakeholders to have a measure of disagreement. Even though this number on its own does not have direct implications for the choices made, it can be indicative of uncertainties in the rankings. As Svanhberg et al. state: “If there is high uncertainty, this may indicate that the architecture candidates and quality attributes are not so well understood, and that further investigations are necessary before the final architecture decision is taken.” [84] This advice can also be incorporated when applying ArchDesigner.

6.4.1 Applicability in Practice

Before embracing ArchDesigner as conceptual foundation to build a decision-making framework with, some expert interviews were conducted to get a first indication of its applicability in practice. Practitioners that would be prospective stakeholders of the framework at Thales Naval were asked to comment on the ArchDesigner methodology and how it could be used in practice. The overall process proposed by the authors was described to interview respondents at a high level – leaving out details like score calculation methods that stakeholders would not directly be concerned with in day-to-day use. Practitioners indicated particular interest in the pairwise comparisons between QAs and alternatives and thought that this might give them more consistency in their rankings. They recognised that this could reduce bias as compared to directly ranking options.

A concern that was often voiced was distrust in the way that ArchDesigner treats interdependencies between decisions as an optimisation problem that should be solved at once. Respondents feared that there might be many dependencies to take into account, and that solving the whole as one optimisation problem, even taking into account the normalisation step, would lead to encountering a ‘dependency hell’, and would decrease the

understandability and thus confidence in the process. They did however indicate that the main dependencies between decisions should be taken into account, but just were not sure whether creating one big optimisation problem including all of them was the best way to do this. Another concern was that in practice, people might not trust a single number or ranking in their decision to choose one alternative over the other, especially when two alternatives' scores are close to each other. This might partially be due to a distrust in the quality of the rankings given by stakeholders and possible inconsistencies between them and a perceived unclarity in how the scores are aggregated. However, with for example measures of variance and explaining how overall results are generated from individual comparisons, this could be treated. Nevertheless, some practitioners felt like they missed a kind of 'sanity check' at the end of the process. This initial feedback indicates some possible strong and weak points to the ArchDesigner methodology when used in practice and will be taken into account in the design of the artifact.

7 ARTIFACT DESIGN

Now that the problem investigation is complete, the requirements have been set and the available alternatives have been considered, the design of the artifact commences. In this section, the first artifact design is explained along with the integration of parts of ArchDesigner. Some artifact design decisions are motivated by academic literature, whereas others are mainly inspired by insights from practice. Together, this should allow for creating a framework that is well-grounded in theory and also easily applicable in practice.

7.1 Design Decisions

Design decision:
Explicitly show detailed overview of steps to be taken

With the ArchDesigner methodology [10] as a conceptual basis and the knowledge of microservice challenges from literature in practice, the design of the artifact to treat the problem of designing a microservice architecture starts. First, the high-level process was defined as shown in Figure 23. A more extensive model, also showing the challenges involved is shown in Figure 24. In this process, clues were taken from ArchDesigner, but made more explicit to show the different steps more clearly. It was also extended to specifically show discussion and evaluation steps – which will be discussed in more detail – to be taken, as well as an arrow looping back to the first step to indicate that the process is often iterative.

Figure 23 – High-Level Artifact Decision-Making Process

Design decision:
Focus on selecting impactful and distinguishing requirements

The first step is to capture requirements that are significant for the decisions to be made. As said before, these can be given at the start of the project, and also be influenced by a project context and the business strategy of an organisation. There can be many requirements, so it is vital to select those that are of most importance and have most implications on the system design when making decisions. The goal is to select those requirements and QAs that are *most impactful* and *most distinguish* possible alternatives from each other. As described by Lindblom [90] in their paper on the *Science of Muddling Through*, often many of the alternatives under consideration all fulfil a large part of the requirements. The alternatives then differ only on a marginal number of requirements. Therefore, it is important to focus on those requirements that best set the alternatives apart. That is not to say, though, that selecting between alternatives this way is promoting an informal evaluation process. In terms of different evaluation modes described by Mintzberg [91], the decision-making process should still mainly be done by analysis. In some cases, though, support for certain QAs can be hard to quantify. An example of this is maintainability. In this case some judgement by decision-makers can still be involved, making the process more bargaining-like. As Mintzberg also points out in their research, “judgement seems to be the favoured mode of selection, perhaps because it is the fastest, most convenient and least stressfull of the three [evaluation modes]” [91].

Whereas functional requirements can often either be fulfilled or not, QAs are generally expressed with a measure and possibly a norm. For QAs, it is again important to select those that mostly distinguish the different alternatives from each other and leave out those that are only important as constraint. For example, if latency would be a distinguishing factor between alternatives, then it should be considered. If, however, the only requirement to latency is that a system should respond within a set amount of time, this can be seen as a constraint and therefore it need not necessarily be included in the requirements by which to compare alternatives.

In the next step, the requirements are prioritised. This is done in a pairwise manner as described by ArchDesigner using the AHP. These pairwise comparisons ultimately result in a weight assigned to each requirement. Refer to [89] for a more detailed explanation of how such calculations are done. It is proposed to allow multiple decision-makers to first make their own comparisons, and then aggregate these to a consolidated list of requirements and weights. This is in an effort to reduce bias that could be introduced when communally deciding on relative weights in a group setting. In group settings, certain decision-makers could also be more forward about their views and aiming to convince the group of these. Another possible bias is that of *Groupthink* [92]; the phenomenon in which judgements are altered because of a desire for harmony and conformity within a group. One other possible source of bias is the phenomenon of *anchoring* – described by Kahneman et al. [93]. A description given by the authors is as follows: "[…] different starting points yield different estimates, which are biased toward the initial values. We call this phenomenon anchoring" [93]. In this case, it concerns the situation in which group members might subconsciously make their judgements depend on a weight suggested by one group member. E.g. when decision-makers are unsure about what weight to assign to a certain comparison and are still undecided, one decision-maker might say that they believe it should be a certain value. At this point, the other decision-makers start comparing their views with this set value – the *anchor* – and depend on it to compare their own weight ranking. This way, the value that a decision-maker assigns to a comparison may be different than the one they would have assigned without having first been presented with an anchor.

Design decision:
Allow for prioritisation input from multiple decision-makers

After establishing a prioritised list of requirements, the previously mentioned constraints to alternatives should be identified. These can be used to include or exclude candidate alternatives for comparison. In terms described by Kontio [80], these can be used in the screening stage of finding suitable alternatives. Constraints can be in the form of a QA as described before, but also other motivation that are not always captured in the requirements of a project. One example of this might be an organisation not selecting alternatives made by a certain supplier for competitive reasons. With these constraints in mind, the actual search for alternatives can start. Kontio describes several sources where alternatives can come from, such as in-house libraries, online sources, magazines and journals, vendor offerings, experience, colleagues, experts and consultants [80]. These alternatives are searched for each decision to be made.

Then comes the decision-making part. The challenges that need to be considered in the case of a microservice architecture arise from the literature research as well as interviews from practice as discussed in chapter 4. They are subdivided by their categories: management, integration and communication. In order to show the dependencies between decisions, further research was done to each challenge and how they relate to one another. This was discussed in chapter 5. The outcomes for each

Design decision:
Initially make decisions one-by-one

decision are the highest scoring solution alternatives, each serving as a building block for the microservice architecture to be designed. Because of the number of challenges that need to be addressed, and when considering the unfavourable feedback from practitioners on dealing with interdependencies by merely solving an optimisation problem, the choice was made to approach these decisions as a process in which decisions are initially treated one-by-one. When one decision influences another, that first decision is first made and then used as input for the next decision. This way, the results are consolidated before moving on to each next decision. This does however imply that now sub-optimisation is happening; having the optimal alternative for each decision does not guarantee the best *overall* solution. To deal with this, the next step is introduced.

Design decision:
Discuss best decision alternatives to define the overall solution

In step 6, the decisions are discussed to see how they fit together. Practitioners indicated that they would likely not take the best alternative at face value based on merely a ranking. Therefore, it is proposed that the outcome of each decision to be made are one or more *best* alternatives instead of a single optimal one. In the discussion stage, these are then combined to an overall solution to the problem. Guidance in determining the interdependencies between decisions is given through their descriptions as was discussed in chapter 5. This way, decision-makers can combine alternatives in a way that makes most sense to them. Recall that during the problem investigation phase (chapter 3) of present research, practitioners indicated that the framework should in their view mainly *guide the discussion on the challenges that are encountered when designing a microservice architecture*. The goal is to achieve this by including this distinct discussion phase in the process. Together with considering one or more best alternatives instead of a single optimal one, the goal is to also increase practitioners' confidence in the framework by making it more transparent and not deciding on the overall solution merely by solving an optimisation problem.

Design decision:
Evaluate outcomes and iterate to improve outcomes and framework

Finally, there is the evaluation stage. In this stage, the outcome of the decision-making process is put into practice and the quality of the solution is assessed. The outcome of this evaluation can serve as new input for next iterations. This step is crucial in both improving future outcomes as well as the decision-making framework itself. First, the decision-making process is iterative. If during evaluation it turns out that the outcomes are not sufficient, the lessons learnt from one iteration can be used to improve the next one. For example, the selected requirements could be altered, their relative weights, the alternatives under consideration and so on. On the other hand, if it turns out that certain parts of the decision-making framework itself can be improved, this could also be considered.

Design decision:
Find solutions in guidelines and techniques when no clear decision between alternatives can be made

As stated before, not all challenges found in literature and practice translate to decisions with clear alternatives to choose between. One example used before is that of *service granularity* – how 'big' a single microservice should be. The solution to this challenge cannot just be expressed as a number of lines of code or function points that are optimal. There are, however, guidelines and techniques to decide on this. While such challenges have less to do with the main decision-making process put forward by this framework, the goal is to also provide guidance on managing these challenges. To accomplish this, first a distinction should be made between challenges that can be decided upon by selecting between alternatives, and those for which this is not the case. For each challenge that comes with guidelines instead of

alternatives, solution examples are given to help practitioners toward finding a way to manage these challenges as well. It can also be foreseen that the framework is used in cases where some decisions are already set. For example, if the communication mechanism used in a system is predetermined by for example external compatibility requirements, this challenge might not need further addressing and can be filled out with this choice. Another example is the case in which only changes to part of a system need to be made without completely overhauling its implementation of service interconnection.

To assess how reliable the decision outcomes are, the aforementioned advice by Svahnberg et al. [84] of assessing uncertainty can be incorporated. Saaty [89] describes the use of a Consistency Ratio (CR) for this to assess how well pairwise comparisons are in line with one another. If this ratio becomes too large – higher than 0.1 according to Saaty – then the comparisons can be inconsistent or contradicting. In this case, possibly the comparisons may need adjustment or need to be re-done as continuing with inconsistent rankings as input can make for unreliable decision outcomes.

7.2 Process Overview and Meta-Model

To make the theoretical description of the decision-making framework usable in practice, thought must be put into how to present and communicate it. This way, decision-makers can more easily get an overview of the different parts of the framework and the steps involved. A first step in this is to explain the different steps in the process, as previously depicted in Figure 23. Next, a process overview – shown in Figure 24 – was created to show how all parts in this process interact with one another.

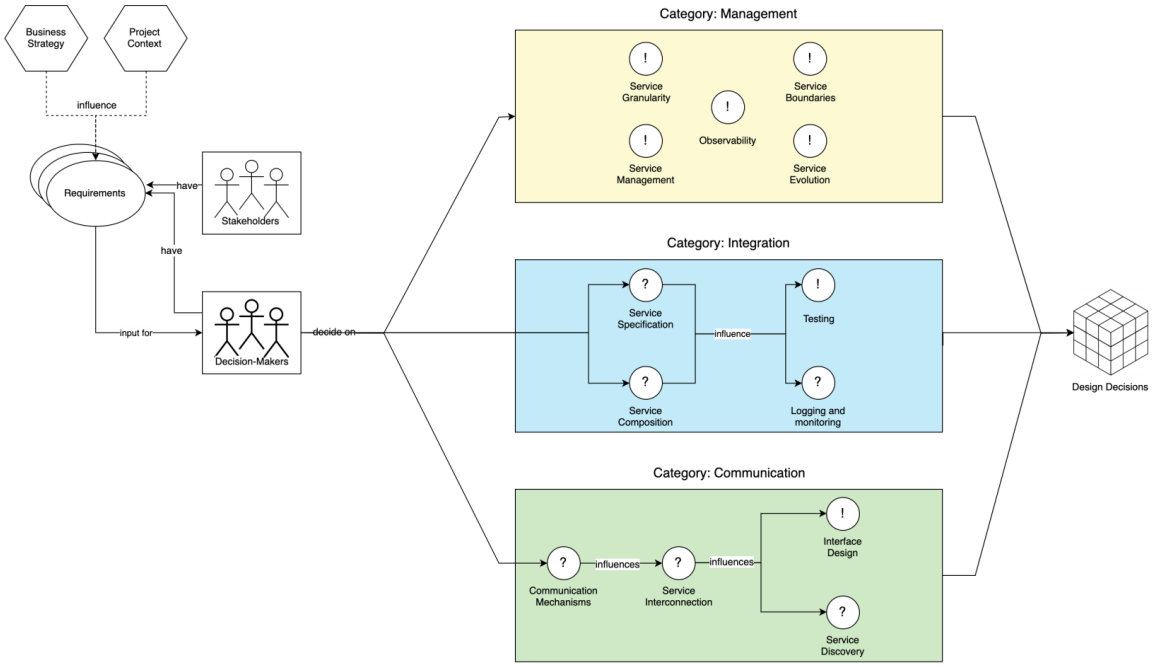


Figure 24 - Decision-Making Process Overview

As can be seen, central at the start of the process are the requirements to the solution that is worked towards using the decision-making framework. These can be defined by the decision-makers themselves, as well as other stakeholders that may not necessarily be directly involved with decision-making. When referring to the stakeholders as discussed in chapter 3, the ones in the *system* layer of the stakeholder overview are most likely to be directly involved as decision-maker. The requirements are also dependent on the project context; each system

has its own intricacies, and these are reflected in the requirements that decision-makers should take into account. A business' strategy that supersedes a single project, including business-specific requirements and goals can also be of influence [79].

Design decision: Focus on dependencies within category

Next are the challenges, each with their own category. These are the final challenges from Table 8 in chapter 4 These categories are depicted as three distinct groups of decisions to be made. Between the challenges in each category are arrows showing the dependencies between them. For example, in the communication category, the *communication mechanisms* challenge should be addressed before *service interconnection*, as the choice of communication mechanism – i.e. how to structure communication – restricts the alternatives that are available for service interconnection – i.e. how to implement this style communication. This is called an Alternative-Based Dependency in the work by Al-Naeem et al. [87]. In this case, the communication mechanism decision is superior to service interconnection. The authors propose that there can also be Context-Based Dependencies; in which an alternative's support for a certain quality attribute can change based on the alternative chosen in a superior decision. The challenges along with their interdependencies are discussed in chapter 5. There could also be interdependencies between challenges outside of a single category. In fact, the complete dependency graph could become fairly intricate fast with this number of challenges. This is one of the reasons the choice was made to not decide on all of the challenges in one go as a single optimisation problem. The aim is to address such category-transcending dependencies in the discussion stage within the framework. This way, the framework should be more practical and transparent to practitioners, expectantly increasing their confidence in it.

The challenges are not solved by choosing between alternatives but rather by incorporating guidelines or industry practices are also shown in this overview. The distinction between decisions and guidelines is made by showing them as a question or exclamation mark, respectively. The output of this process are decisions and guidelines that each act as a building block in the overall design for managing the design challenges related to the communication between, integration and management of microservices.

To further explain how all parts of the process interact and which actors are involved in what activities of the decision-making activities, a meta-model was created. This is shown in Figure 25.

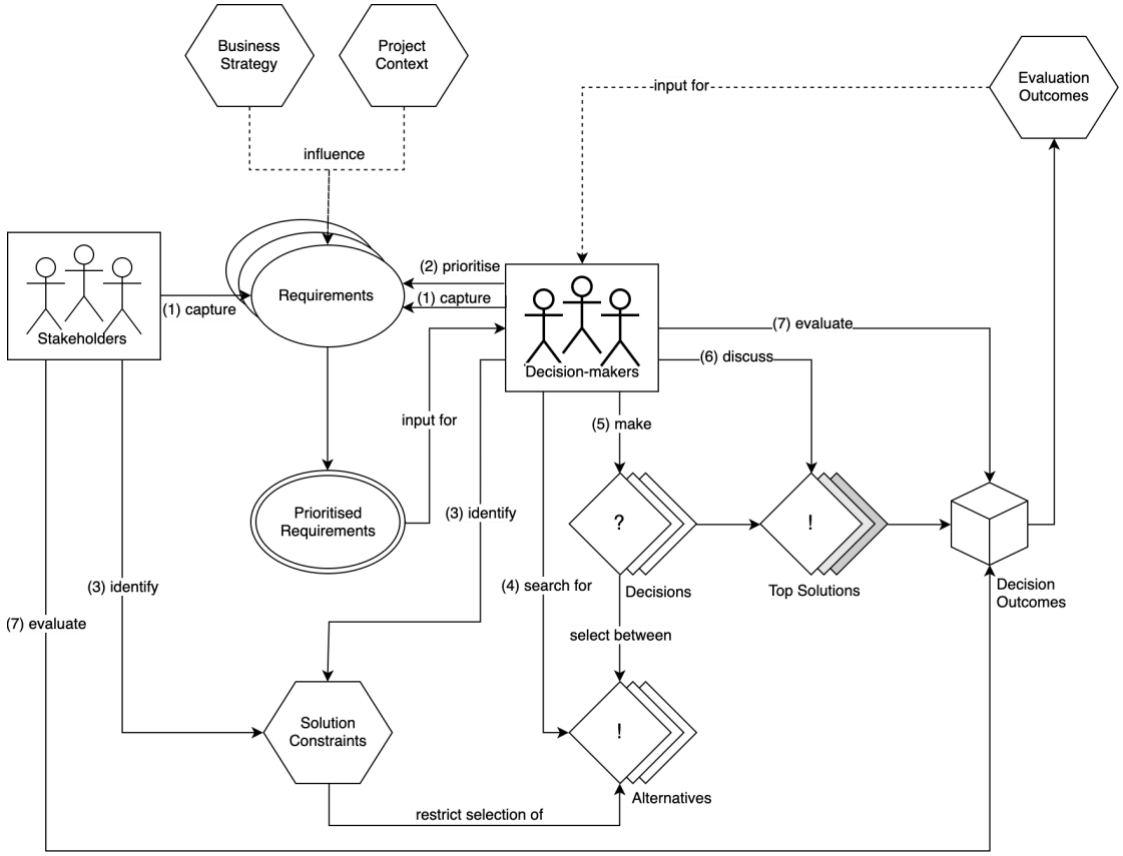


Figure 25 - Artifact Meta-Model

In this model, certain parts of the process overview can be recognised. The requirements are still central at the start of the process, along with the actors, and influencing factors. Also, this picture makes the separate steps as previously shown in Figure 23 explicit; showing how each step influences another along with the inputs and outputs of each step. This way, the interactions between all parts of the framework are made clear.

Several Thales Naval practitioners were asked for initial feedback on these descriptions and figures. They were presented with the figures in this chapter and an explanation was given on how it all fits together. Most were interested in the artifact and were interested in how it could help them in their day-to-day work. The overview of challenges also looked correct and complete to them. They did have a hard time imagining how it would work in practice. They indicated that this was in part due to them not having seen it be used in a real-life setting yet, but also because the theoretical description was not practical enough yet in their view. Practitioners indicated that a simplified explanation outlining the steps to be taken, why and in what way would probably be helpful in making the framework more usable in practice. Therefore, this was provided in presentation-form during the case studies, as well as a short cheat-sheet-style manual to explain the main ideas and goals of the artifact together with descriptions of the challenges and their possible solutions.

7.3 Fulfilment of Requirements

The goal-level requirements set in section 6.1 will be assessed through validation later in this research. However, the domain-level requirements can already be discussed based on the current artifact description. The domain-level requirements previously described are repeated in Table 13.

Table 13 - Domain-Level Requirements to the Artifact

	Domain-level requirements
D1	The framework shall support the selection of optimal architectural alternatives.
D2	The framework shall support the use of quality attributes for rating alternatives.
D3	The framework shall support input from multiple stakeholders.
D4	The framework shall deal with competing and conflicting objectives.
D5	The framework shall take into account uncertainty both in the descriptions of requirements and in their associated solutions.
D6	The framework shall deal with interdependencies between decisions.
D7	The framework shall support providing guidelines and industry practices when no clear decision can be made for managing a challenge.

D1 and D2 are addressed since they are intrinsic to the ArchDesigner [87] methodology, and this functionality is kept in the artifact design. As described by Falessi et al. [10], all decision-making methodologies considered in their comparisons support D3 through D6. Thus, this is also the case for ArchDesigner. The artifact design again includes these functionalities as well. The most notable addition that was not previously found in software architecture decision-making methodologies is that of D7; the support for providing guidelines. These have been described in section 5 and are incorporated into the challenge overview used in the artifact. Therefore, the artifact fulfils the seven domain-level requirements that were set. However, the goal-level requirements are harder to measure. Through thorough validation research, these will be addressed.

7.4 Usage Requirements

Besides the theoretical design of the artifact, it is important to determine when the framework can be used. By having a *population predicate* it can be assessed whether cases fall into the intended use or not. Making sure that the situation in which it is applied helps in enabling its full potential. A first aspect to this is the organisational culture that it is used in. Given that the decision-making framework is meant to be iterative, it is probably best used in organisations that apply an iterative or agile-like software development process. When using waterfall-like approaches, the goal is often to finish a certain design phase and not change its products in a later stage. This conflicts with one of the goals for which the artifact was developed; not always aiming for the perfect combination of alternatives, but helping organisations make decisions on architecturally significant aspects with confidence, then evaluate and iterate. Also, the business strategy and especially project context should be clear beforehand, since the artifact does not give guidance on these subjects. Because they can influence the requirements and their prioritisation quite substantially, it is vital that the goals for the system are clear. That is not to say that there can be a case in which such goals only become apparent when selecting requirements or even during the actual decision-making, though. However, when are clear from the start, better decisions can be made earlier; thus, reducing the time or number of iterations needed. Besides this, it is imperative that decision-making can be done by selecting between architectural alternatives and using quality attributes to express the extent to which these alternatives are suitable for their intended use.

It is proposed that the decision-making process be guided by a group facilitator. Their role is to guide the process itself; not actually be a leader in deciding the decision outcomes. It would

be unnecessary to always have all decision-makers know the particulars of the artifact. Only one decision-maker needs to be knowledgeable about how to execute the process and calculate the appropriate scores for the group to be able to use it. Some introduction to for example the details of pairwise comparisons can be given, but after that the group can focus on the actual decisions. The group facilitator can choose to also act as decision-maker besides only guiding the process. This choice largely depends on whether they are directly involved with the project – and thus are a direct stakeholder.

7.5 Tooling

Given the involvement of the AHP by Saaty [89] during the decision-making process, specific tooling is recommended – if not necessary – to calculate the weights, scores and priorities related all requirements and alternatives. Because AHP is widely known and used, some software made specifically for it is available. These are often in the form of a Group Decision Support System (GDSS). A GDSS "combines communication, computing, and decision support technologies to facilitate formulation and solution of unstructured problems by a group of people" [94]. Examples of such software that can be used to perform calculations following the AHP process are summarised in work by Pólkowski [95]. When deciding on what tooling to use, many of the tools in this paper were examined, along with those found when searching online. There are a few commercial software products available, of which some seem useful. There are also a few free software packages which offer basic functionality, but do not always excel in usability. Furthermore, as software vendors are increasingly offering their services online using a subscription-model; some that previously offered standalone software have now migrated to offering Software as a Service (SaaS). At their core, all software for use with the AHP support pairwise comparisons of requirements and alternatives, as well as prioritisation of both based on these scores. Many also offer support for involving multiple decision-makers. One notable example of a free online tool is AHP-OS[8], mainly developed and even academically documented very well by Goepel [96]. This software's user interface is fairly basic but clear to understand and use. It supports all necessary functionalities, along with options for sharing links for group members to fill out their pairwise comparisons for a given project set up by a group facilitator. Furthermore, it supports in-depth analysis of the results. Of the tools found during this research, AHP-OS seems like one of the best starting points for academics working with the AHP method.

So, if organisations that aim to utilise the decision-making framework want to adopt tooling to help with AHP-related calculations, plenty of options are available. However, not all tools are suitable for all organisations. For example, when discussing with Thales Naval practitioners about what software would best fit their needs, most pointed out that using SaaS solutions was unpreferable since often sensitive information needs to be processed during decision-making. This restricts the available alternatives. For the scope of the case studies to be conducted, it was also not feasible to purchase a software product already. Some software vendors offered trial versions of their software, but these were severely limited in their functionality by for example imposing a limit of the number of alternatives or requirements that could be considered. This would be detrimental to the case studies since the tooling used should not be limiting the participants in working with the artifact. An effort was made to use free software packages such as Super Decisions[9], but while its support for the required

[8] http://bpmsg.com/ahp/
[9] https://www.superdecisions.com/

functionalities was there, the user-interface was too complicated, to an extent it would be unlikely that it would ever be used in practice.

To be able to conduct case studies for validation of the artifact, tooling was needed that is free or could be trialled for free, would not impose limitations that would inhibit the proper execution of the decision-making process, and could be used offline. This resulted in two types of tooling being chosen to be used when conducting the case studies. The first one is a Microsoft Excel template developed Goepel [97] – the author and developer of the aforementioned online AHP-OS system. This template can be used to specify requirements or alternatives, have multiple decision-makers input pairwise comparisons, calculate their individual rankings and also consolidating these into a unified ranking. The template also shows useful statistical data such as a consistency ratio (CR) to assess the amount of contradictions and inconsistencies between pairwise comparisons. The advantage of using such an excel template would be that it would probably be easy to learn for practitioners, since generally many are already familiar with how to use Microsoft Excel. A further benefit is that the template shows results immediately, thus helping decision-makers understand how their inputs change the outcomes. A large downside, however, is that this template only supports these calculations for one set of comparisons at a time. For example, it can be used to calculate priorities of requirements, but the outcome of this cannot directly be used as weighing factors when comparing decision alternatives. Nevertheless, it is seen as a useful tool to experiment with pairwise comparison by multiple decision-makers and subsequently calculating consolidated priorities or weights for requirements or alternatives.

Regardless of the potential that such an Excel template can have, it was still seen as too limiting to only use this during the case studies. An abundance of manual work would be involved to calculate intermediate results between steps or decisions. An alternative that was more technical to use, but with a far better feature set was found in the form of an open-source R package called AHP [98], which is developed by Christoph Glur and available through GitHub. R is a programming language for statistical computing. Packages or libraries such as AHP can be developed to provide specific functionalities and calculations that can easily be used by the end-user. In the case of the AHP package, many helpful features are added, including a GUI to display a systematic overview of how requirements and decisions relate to each other, and to show the numerical outputs including extensive statistical data to easily show how the scores for alternatives are constructed. A very helpful addition to this is that alongside with the package, a file format was created to systematically express decision-making problem. This AHP file format [99] is then used as input for the calculations to be done. An example given by the author of this file format is as follows:

```
Version
Alternatives
  alternative 1
    property 1 (optional)
    property 2 (o)
    ...
  alternative 2
    property 1 (o)
    property 2 (o)
  ...
Goal
  decision-makers (o)
  preferences
    decision maker 1 (o)
      scoreFunction or
      score or
      pairwiseFunction or
      pairwise or
      priority
    decision maker 2
      ...
  children
    criteria 1
      preferences
      children
        sub-criteria 1.1
        sub-criteria 1.2
          children: *alternatives
      ...
    criteria 2
      ...
```

Several entries in this example have been highlighted to show the main parts to defining an AHP problem using this file format. At the *Alternatives* section, alternatives for a given decision can be specified. Optionally, each alternative can be assigned certain properties that can later be used in optional calculation functions for determining their score on a certain requirement. Individual decision-makers can also be specified using this format, including the option of giving certain decision-makers a higher weight in the overall score calculation. Under *Preferences*, the requirements – called criteria here – and their pairwise comparison can be

expressed per decision-maker. Other functions such as direct weighting of requirements are also supported, but are out of scope for the sake of this research. Under *Children*, for each criterium the scores per alternative can be expressed using pairwise comparisons per decision-maker. Again, other modes of calculation are also supported. For example, when measuring latency, decision-makers could compare alternatives in a pairwise fashion as described before, but using this package, a score can also be calculated based on latency defined as a number as alternative property. E.g. a latency between 0 and 10ms gets a score of 9, 10-20ms gets assigned 7 etc. Such functions are not the main focus of present research, but possibly a helpful addition in practice, nonetheless. Still, this example is quite abstract. During the execution of the case studies, real-life examples will be given to make it easier to understand how this AHP file format can be used in practice.

The fact that the AHP package needs to be used in R and comes with its own file format that needs to be used makes this solution harder to learn by practitioners aiming to use it for supporting the decision-making process in the future. However, for the sake of this research and carrying out case studies to assess the artifact's validity, the AHP package is deemed more useful because of its functional superiority over the aforementioned Excel templates. Furthermore, while defining a decision-making problem in the AHP file format can be time-consuming, the fact that after completion the prioritisation of requirements and scoring of alternatives are integrated makes for the expectation that ultimately it should be the easier to use solution in practice. Thus, using the AHP R package is selected as most viable option to implement tooling during the case studies.

8 VALIDATION

With the first artifact design complete, its fitness for purpose can be assessed in order to answer RQ-5: *"How can the designed framework's fitness for purpose best be validated?"*. To accomplish this, two case studies will be conducted with Thales Naval for validation. This chapter describes the set-up and scope of each case study, observations, their outcomes and changes made to the artifact design based on these.

8.1 Validation Methodology

As Wieringa states, "To validate a treatment is to justify that it would contribute to stakeholder goals when implemented in the problem context" [4]. For this, a model of the artefact interacts with a model of the intended context that both resemble their real-world counterparts. This is illustrated in Figure 26. The interaction between the model of the artifact and the model of the context is then observed to build a theory about how this interaction would work in real-world scenarios. This theory can then be used to make generalisations and predictions about the observed phenomena.

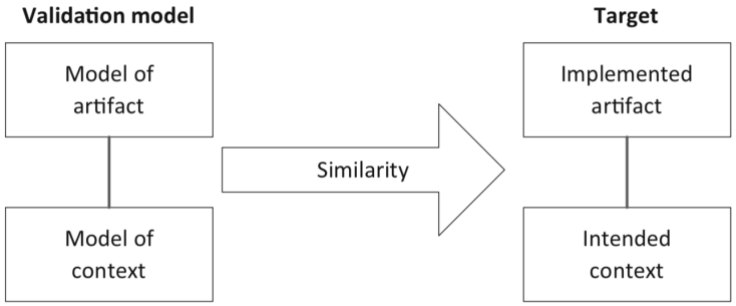


Figure 26 - Relation between Validation Model and Target - Adopted from [4]

The validation will be done through two case studies in the form of *single-case mechanism* experiments. For the case studies at Thales Naval, several software architects were asked about what current projects they and their team were currently working on that involved decision-making in microservice architectures, and which of those they would see fit to serve as model case for this research. The current description of the artefact acts as model of the future implemented artifact and is applied to two cases that resemble practical scenarios that will be used as models of the intended context. The guidance given in chapter 18 in Wieringa's book [4] on how to structure such experiments will be used to design the case studies. Those parts that both case studies share are discussed together, and case-study specific aspects – mainly the model of the context – will be explained in their respective sections.

The *knowledge goal* in these case studies is validation of the designed artifact or treatment. The *higher-level goal* of the treatment is to improve decision-makers' work in managing design challenges related to communication between, integration and management of microservices. The *knowledge context* consists of the problem overview, background literature and treatment design discussed in present research.

The *conceptual framework* of this validation research consists of the problem description, theoretical background and artifact design as well as the measures for operationalising indicators described in this research. The *generalisation* is that the steps in the decision-

making framework contribute towards decision-makers' goal of better managing microservice design challenges. The *generalization* predicts that this will happen when the decision-making framework is used. *Knowledge questions* that need to be answered in these case studies are about whether the designed artifact satisfies the goal-level requirements described in section 6.1.2. Besides this, there is the question of whether decisions on the identified microservice challenges can actually be made using this decision-making framework; i.e. the functional correctness needs to be verified. The designed framework's usefulness, ease of use and the quality of the decision outcome are the main factors in contributing to the stakeholders' goal of better managing design challenges related to communication between, integration and management of microservices. These can be expressed as follows:

- Can the artifact be used to support decision-making in a microservice architecture?
- To what extent do decision-makers find the artifact *useful*?
- To what extent do decision-makers find the artifact *usable*?
- How confident are decision-makers about the artifact's *decision outcomes*?
- To what extent do decision-makers find the artifact and its outcomes *usable in practice*?

The *intended population* is the set of microservice architecture design projects that:

- Are executed in an organisation with an iterative or agile-like software development process;
- Have a clear business strategy and project goals set at the start of the project;
- Employ a group facilitator to lead the decision-making process.

The *object of study* is the first design of the artifact and will be applied in two different but fairly similar case studies, the specific of which will be described in their respective sections. These cases do not support directly gathering data for mathematical analysis. Questionnaires to be filled out by participants can be used to get an overview of the outcomes of using the artifact. Furthermore, qualitative data such as the decisions itself, as well as remarks by the participants during the process can be used to describe and possibly explain some of the outcomes. The model will abstract from the real world in a number of ways. First, the number of participants, requirements, decisions and solutions to be considered will be limited because of time constraints. Ideally, one would ask an entire team to join and make decisions for an entire microservice architecture in as much time as needed. However, such a commitment of organisation resources cannot easily be expected for this experiment. Furthermore, the case studies will be conducted at Thales Naval. Even though the organisation is part of the intended population, it is not necessarily the de-facto model for the population. Nevertheless, it is expected that many of the observations done in the Thales Naval context can be generalised.

Both case studies reside in a context that resembles a real-world scenario. The decisions to be made are similar to those that decision-makers face more often, but this time the designed decision-making framework will be used to structure this process. The process will be guided by the researcher, who will act as facilitator. An introduction to the decision-making framework will be given in a presentation. The actual decision-making is supported with the tooling described in section 7.5. Little control was exerted over the treatment; the role of the researcher, materials and tooling will be to educate participants about the decision-making framework and show its intended use, but not restrict them or intervene in their actual use during decision-making. This should best support analogic inference to the real world.

The case studies will start with an introduction by the researcher, explaining the decision-making framework and its intended use as well as the concept of pairwise comparisons through a presentation. The goal of the artifact to mainly be of a supportive role in the participants' day-to-day work will also be highlighted. This is done to clarify that it would not tell them how to do their job but aims to make it easier. The case to be studied will be explained to ensure that all participants would be on the same page about the system to be discussed. A check will also be done to assess whether all participants are familiar with microservices. Finally, the participants are informed that because of the limited time available for these case studies, the number of requirements, decisions and alternatives to be considered would be limited. Nevertheless, they will be encouraged to approach the process in a way that they also would in the real world, to make it as true-to-life as possible.

In both case studies, data on usefulness, ease of use and decision-quality will be gathered by letting participants fill out questionnaires. The questions defined by Davis [82] in their paper describing the Technology Acceptance Model (TAM) can be used for measuring the first two variables. The question of confidence comes down to measuring decision quality. No detailed information about previously made decisions in microservice architectures and their performance is available. It is therefore hard to objectively judge decision quality based on the performance of decision outcomes in practice. As an alternative, this can be done using the six elements of decision quality described by Spetzler [83]. These questions have been augmented with three additional ones, directly asking participants about how confident they feel about decisions, the practicality of the decisions and about the contribution of the artifact to making these decisions. The question about the perceived practicality, together with asking about the framework's technical correctness are used to assess practicality. An overview of the questions used for these measurements can be found in Appendix D. From this data, descriptive statistics can be derived to gain insights in the participants' views on the decision-making framework. The data from the questionnaires will be mainly used to support *descriptive inference*. Furthermore, the events, remarks and outcomes of the decision-making process can be used to further analyse the model artifact's effects in the model context. An attempt will then be made to generalise from this.

8.2 Case Study 1

The first case study involves the design of a control system that on one side communicates with one or more client systems, and on the other side instructs external input and output devices and gathers data from these devices. As per request by Thales Naval, the precise description and application of this system will not be discussed beyond those aspects relevant to this case study. This is in no way limiting though, as many of the requirements and challenges found in this system could also occur in systems with similar functions in other types of businesses. Refer to Figure 27 for a schematic overview of the system that is central in this case study.

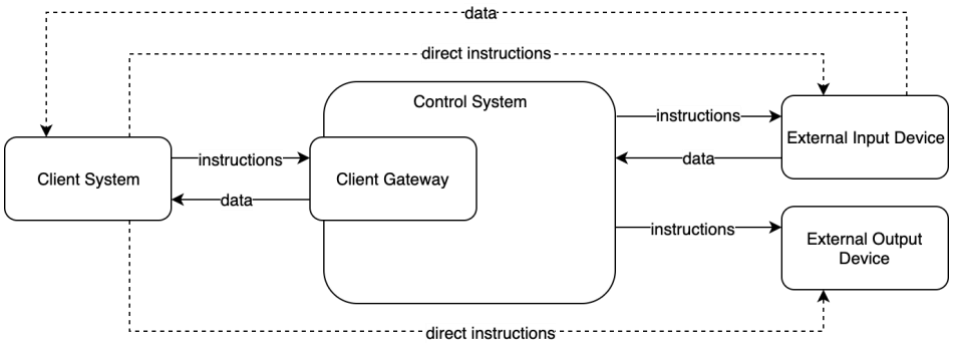


Figure 27 - Schematic Overview of Case Study 1 System

As can be seen, communication between the control system and client systems happens through a gateway. The gateway provides a standard interface that clients can integrate with, and this way abstracts away the external devices' specific interfaces. However, not all communication from this client system to the external devices need necessarily run through this gateway; as for some use cases direct communication may be necessary or preferred. For example, client systems might still use legacy code that cannot communicate with the client gateway yet, or some device-specific functionality unsupported by the client gateway can be needed by the client system in certain cases. The data sent between the different parts in this system can consists of both request/response-style communication – e.g. requesting an action to be performed by an external device – and notification-style communication – e.g. an external device announcing a change in its state. This control system will be newly built, using a microservice architecture.

Since time would not allow the design of the entire control system to be defined with help of the decision-making framework, the decision was made to focus on the part that connects to the external input and output devices. Services in the control system would together provide functionalities accessed through the gateway and would need to connect with possibly proprietary interfaces of these external devices. Beforehand, it was also expected that there would also not be enough time to consider all the decisions captured in the artifact. Practitioners indicated that most of the challenges they expected to face were of technical, communication-related nature. Therefore, the challenges in the communication category were selected to be focussed on in this case study. These are shown in Figure 28 below. The team consisted of six participants, from varying age, experience and function. Four participants were involved with software architecture design in their day-to-day work, and the two others were lead software engineers. Of them, four participants – three software architects and one lead software engineer – were able to attend the case study. Having more participants would have been preferable, but this was not possible at the time due to planning constraints. The total time available for this case study was roughly two hours.

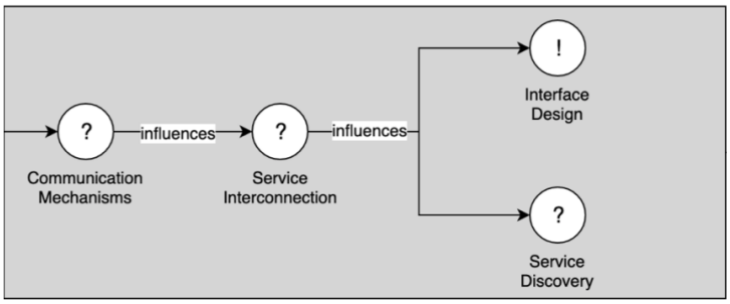


Figure 28 - Challenges in the Communication Category

8.2.1 Observations

First, a discussion was held about what requirements to involve. The participants were instructed to try and select those that they would consider the best for differentiating between alternatives for the subsequent ranking of alternatives. This resulted in seven main requirements that would be considered. Consequently, these requirements were ranked by the participants using pairwise comparisons as described. This resulted in the prioritisation of criteria shown in Figure 29. Besides the four quality attributes of reliability, availability, performance and security, there are three functional requirements present. The *Device_Access* criterium captures the need for support of direct connections between a client system and an external device. The *Subsystem_Var* and *Feature_Var* criteria are specified to show that alternatives should support variability or changes in both the kinds of external devices (subsystems) that are connected, as well as the features from these that are supported by the control system. It can be seen that the first two functional requirements make for a large portion of the overall weight of criteria. Please also note that a small weight for an individual criterium does not necessarily imply that it is of low overall importance to the project. This ranking merely shows the relative importance of the criteria.

One constraint to the overall solution was that its performance needed to meet a certain level. At the time of the case study, the participants could not immediately assign a measure and norm to this, but they indicated that in the real-world this could well be done. Performance is also present as criterium, since better performance is seen as always preferred. Therefore, it can be used to compare alternatives, with the constraint that it should at least be within a certain threshold.

Criteria	Weight
Criteria	100.0%
Device_Access	32.5%
Subsystem_Var	30.9%
Reliability	11.2%
Availability	10.6%
Performance	8.7%
Security	4.0%
Feature_Var	2.1%

Figure 29 – First Prioritisation of Criteria

Criteria	Weight
Criteria	100.0%
Reliability	36.0%
Availability	36.0%
Performance	19.7%
Security	8.4%

Figure 30 - Adjusted Criteria

The first decision of *Communication Mechanisms* was then considered. Within the scope of the case study, the search for alternatives involved practitioners' experience or previously used approaches. Some examples were also suggested by the researcher, but the ultimate selection was up to the participants. For communication mechanisms, two alternatives were deemed as possible candidates; a publish/subscribe (PubSub) or publish/asynchronous (PubAsync) response mechanism. These two alternatives were then ranked using pairwise comparisons for each criterium, resulting in the ranking shown in Figure 31.

	Weight	PubAsync	PubSub
Choose Between Alternatives	100.0%	57.7%	42.3%
Device_Access	32.5%	24.4%	8.1%
Subsystem_Var	30.9%	15.5%	15.5%
Reliability	11.2%	8.4%	2.8%
Availability	10.6%	5.3%	5.3%
Performance	8.7%	1.4%	7.2%
Security	4.0%	2.0%	2.0%
Feature_Var	2.1%	0.7%	1.4%

Figure 31 - First Ranking of Communication Mechanism Alternatives

When this outcome was shown to the participants, they indicated that they largely felt that the outcome fitted with their beliefs. However, upon seeing the differences in weight that now exist because of the large influence of the top two functional requirements, the participants felt that these were too influential. When for instance looking at the support for *Device_Access* between *PubAsync* and *PubSub*, the difference felt superficial to participants. They indicated that there could indeed be differences in the support for this requirement between the two alternatives, but that in practice they expected that they could make both alternatives support this function. They felt the same for the other *Subsystem_Var* and *Feature_Var* functional requirements; as long as these features were supported, the relative differences between alternatives with regards to these were not as important for the comparisons as the quality criteria. Because of this, the choice was made to omit the functional requirements from the comparisons and keep the requirements for support of these functions as constraints. The ranking of criteria and alternatives for communication mechanisms could then be recalculated. The results of this are shown in Figure 30 and Figure 32. The participants indicated that the new comparison between alternatives was better suited with their beliefs and gave them more insight. As can be seen, the publish/asynchronous response mechanism has the highest score in this case, though not by much. The practitioners were confident though that they should choose *PubAsync* for this decision and continue.

	Weight	PubAsync	PubSub
Choose Between Alternatives	100.0%	52.4%	47.6%
Reliability	36.0%	27.0%	9.0%
Availability	36.0%	18.0%	18.0%
Performance	19.7%	3.3%	16.4%
Security	8.4%	4.2%	4.2%

Figure 32 - Adjusted Ranking of Communication Mechanism Alternatives

The next decision to be made would normally be *Service Interconnection*. However, at this point in the case study, there was not much time left since the discussion on and adjustment of the requirements took quite some time. Participants indicated that they were highly interested in looking at the *Service Discovery* decision, as they were encountering it in their daily work at the time. The decision was therefore made to move on and focus on that decision. Two feasible alternatives as way of implementing service discovery were selected; either client-side or server-side discovery. The choice between these was not constrained by the

choice of communication mechanism. The same pairwise comparisons with regards to criteria were then made, resulting in the ranking shown in Figure 33.

	Weight	Server-Side	Client-Side
Choose Between Alternatives	100.0%	78.9%	21.1%
Reliability	36.0%	30.0%	6.0%
Availability	36.0%	30.0%	6.0%
Performance	19.7%	14.8%	4.9%
Security	8.4%	4.2%	4.2%

Figure 33 - Ranking of Service Discovery Alternatives

In this case, a clear preference for server-side discovery is observed. Participants felt that this outcome fitted well with their views and scores given during comparisons.

Another step in the communication category is to consider interface design; which is shown as a guideline in the decision-making overview. The main guideline here is to use a standardised IDL whenever possible. Participants were asked whether it would make sense to include this in the overall decision-making process and whether the guideline is usable. They indicated that both were indeed the case.

The next step would then be to discuss the decision made. Putting the separate chosen alternatives together to form a coherent solution requires some discussion and a reality-check. With the limited decisions made in this case study, this discussion by the participants was quite short. They felt that the best scoring alternatives were a good fit for the system and were confident that they could also find a solution to the service interconnection challenge that fit with the others. The final step of evaluating the decision outcomes by putting the solutions into practice was obviously out of scope for this case study, but participants indicated that they felt this was a logical and necessary step in the process. Especially since they use an agile software development method, this step felt natural.

8.2.2 Results

After the execution of the case study, the participants were first asked to comment on how they felt about the use of the decision-making framework for making these decisions. The first and foremost remark made by most participants is that they liked the use of the pairwise comparisons by the AHP method a lot and were pleasantly surprised by it. They felt that by comparing requirements and alternatives in pairs, the discussion about *why* one is better than another is stimulated as opposed to having to rank all options by direct scoring. The fact that each participant could input their own comparisons and eventually consolidate all into a single overview was also seen as helpful. Participants recognised the possible sources of bias that could be present when this would be done in a plenary fashion. One software architect indicated that this could also be very useful as justification for the choices made for a certain system, as these are now made explicit and after analysis give good insights. Participants were also asked about whether they thought the overview and dependencies between challenges in the artifact were technically correct. They indicated that they could not find any apparent shortcomings in both.

Points of improvement that were mentioned at this point were mainly about tooling; this was seen as still too ad-hoc and of a quite technical nature. Participants indicated that it might help to make the entire process a bit more ‘hands-on’ by for example creating a planning-poker-

like approach to the pairwise comparisons or using post-its on a whiteboard to show pairwise rankings. Besides this, participants indicated that they would have liked a more practical description of what decisions to make and how to make them. Having a short and to the point 'how-to' guide would in their opinion help to make it easier to use for practitioners.

The most notable observation during this case study was the fact that the requirements selection and prioritisation needed to be changed during decision-making. This is contrary to the intention of prioritising requirements and only changing them in subsequent iterations of decision-making. When participants were asked about why they thought the requirement selection was off in the beginning, they indicated that they probably needed more information about how to select the most architecturally significant requirements. They also noted that after they became more familiar with the decision-making framework, they felt that they could better foresee how the requirements would be used later in the process. Thus, better explanation beforehand and practice with the framework could also aid in improving requirement selection and prioritisation. Furthermore, participants indicated that they had some trouble in deciding what weight to give in comparisons. The 1-9 scale used here was not completely clear; especially the question of when to give a score of 9. More guidance on this would expectedly also help them make better judgements according to the participants.

In short, observations from practice can be summed up as follows:

- Allowing each participant to give their own rankings and then consolidating was seen as helpful to reduce biases present in collective decision-making;
- The outcome of the decision-making process can possibly be used to document and justify the decisions made;
- No shortcomings could be found in the overview of and dependencies between challenges;
- The requirements selection process in this case study was suboptimal, resulting in having to change the considered requirements during its execution;
- The tooling used during the case study was seen as too complicated and could be more 'hands-on';
- More guidance may be needed to explain the selection and ranking of requirements and alternatives;
- The use of AHP for comparisons was received well and thought to be useful for evoking discussions.

All four participants filled out the survey to assess usefulness, ease of use and decision quality. The scores for each question in Appendix D were recorded and then processed to show overviews of the scores for each indicator. No data was missing, and no outliers were removed. All answers were coded on a scale of -3 through 3. Corresponding answers are shown in Appendix D. The average score per question is shown by a bar, and the overall average for the indicator is shown on top of the diagram. It can be seen that the participants ranked the usefulness and usability of the artifact somewhat favourably. The decision quality indicator received a low score, though, meaning that participants on average neither agree nor disagree on the questions regarding this. It must be noted though that the scale used for measuring these questions was different from that for usefulness and usability, so these scores cannot directly be compared. These two were measured by asking about the likeliness that certain statements would apply to the respondent, whereas decision quality was measured by asking about the extent to which respondents agree with statements. The average score that participants gave about perceived practicality of the outcomes was -0,25,

though no technical shortcomings inhibiting practicality of the framework were found during the case study.

When asked about why participants gave the decision-quality and practicality related indicators low scores, the consensus was that they were hesitant to trust their judgements because during the process the requirements and their prioritisation had changed. Participants expected that when they were more comfortable with the use of the framework, and had more guidance in requirements selection and prioritisation, they expect to be more convinced that the choices made are the right ones.

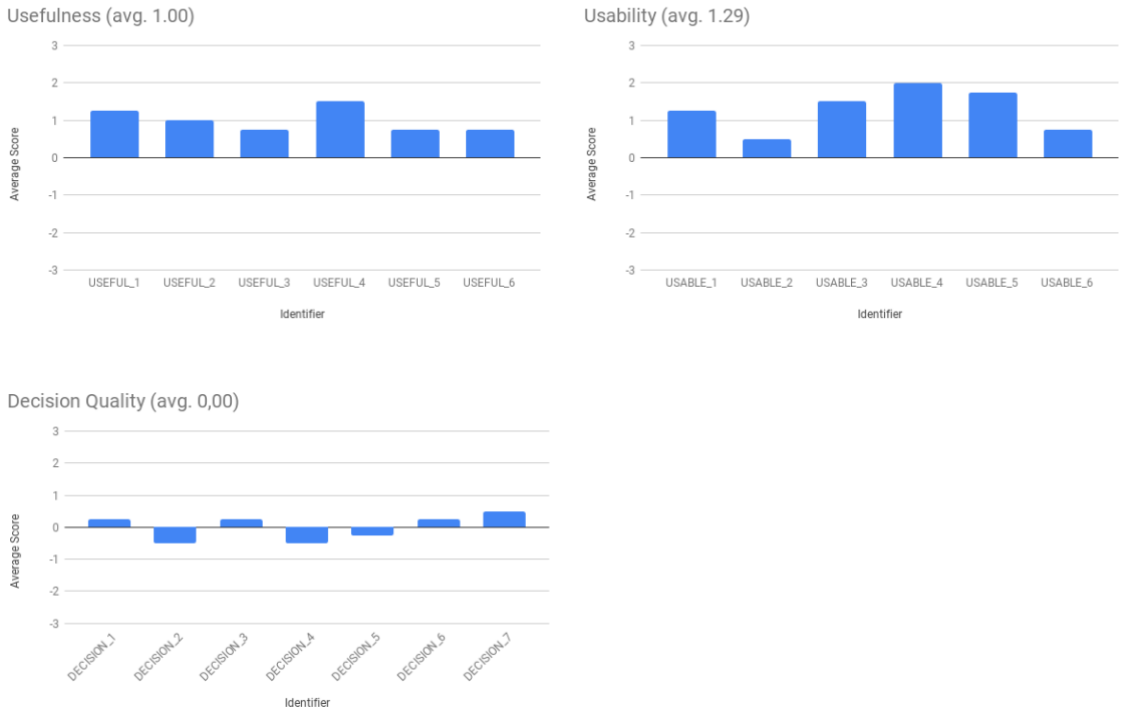


Figure 34 - Survey Results Case Study 1

8.3 Changes to the Artifact Design

Based on the observations and feedback in the first case study, the choice was made to improve on the artifact design by including more guidance on how to select the right requirements to include during decision-making. The scale of weights to be assigned to comparisons is also explained in more detail.

Without additional guidance, it may still be hard to determine which requirements to take into account. Chen et al. [100] aim to explain what characterises architecturally significant requirements through a framework developed in an empirical study with practitioners. In Figure 35, a graphical depiction of the framework in question is shown. As can be seen, the authors define an architecturally significant requirement as one that has “a measurable impact on the software system’s architecture” [100]. Significance, in their view, is measured by “high cost of change” – either monetary or not.

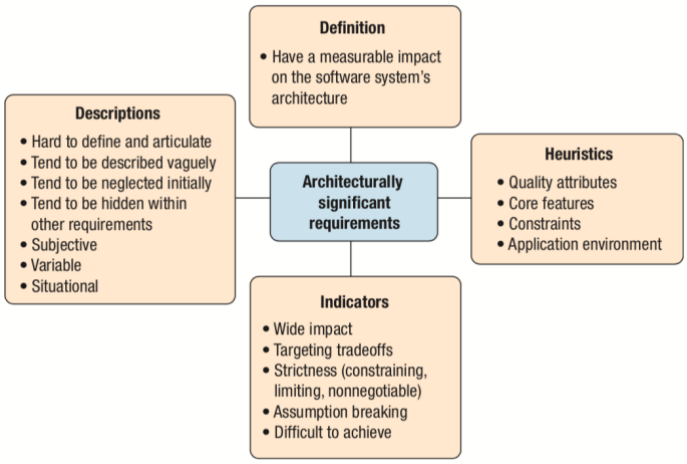


Figure 35 – "A framework of the characteristics of architecturally significant requirements" - described in [100]

Adaptability	Extensibility
Availability	Modularity
Configurability	Portability
Flexibility	Reusability
Interoperability	Testability
Performance	Auditability
Reliability	Maintainability
Responsiveness	Manageability
Recoverability	Sustainability
Scalability	Supportability
Stability	Usability
Security	

Figure 36 - Software Systems' Quality Attributes found in [100]

Design decision:
Explain what makes an impactful requirement

The authors go on to explain the different characteristics that can make a requirement likely to be architecturally significant; through their descriptions, indicators and heuristics [100]. The items shown in the framework can help recognise architecturally significant requirements. For example: if a certain requirement is described vaguely, has a wide impact on a system's design and touches upon a system's core features, there is a high likelihood that this requirement is architecturally significant. Even though not all decisions that the artifact concerns are of a purely architectural nature, the insights from this work can help in identifying those requirements that are most impactful. Therefore, the framework by Chen et al. [100] is used to help explain to decision-makers which requirements to focus on. In their paper, the authors also give concrete examples of quality attributes that practitioners mentioned in their study. These could be used as examples to possibly be used in the decision-making framework. Other sources of possible quality attributes also exist, such as the FURPS+ classification system by Robert Grady at Hewlett-Packard [101] or the ISO/IEC 25010:2011 standard [102].

Design decision:
Provide guidance on how to assign weights in comparisons

To give more guidance on the weights to be assigned to pairwise comparisons of requirements and alternatives, the scale described by Saaty [89] – shown in Figure 37 by Al-Naeem et al. – is included explicitly so practitioners can get a better feeling for when to choose what weight. As for the question of what comparison warrants the extreme weight of 9, Saaty describes the justification for this weight as follows: "The evidence favouring one activity [in this case; alternative or requirement] over another is of the highest possible order of affirmation" [89]. Saaty also suggest that the higher the weight assigned to a comparison; the more evidence should be in place to support this judgement. When describing weights 3 and 5, Saaty talks in terms of *experience and judgement*, whereas with 7 the explanation includes *demonstrated dominance in practice* and as stated for 9 the *highest order of affirmation*.

If A is … as (than) B	Quantitative Weight
equally important	1
moderately more important	3
strongly more important	5
very strongly more important	7
extremely more important	9

Figure 37 - AHP Weighting Scale - Adopted from [87]

With these clarifications in place, the expectation is that the requirements selection and prioritisation steps at the start of the decision-making framework give better results on the first execution. This should help practitioners in making logical choices here and thus better trust the decisions made accordingly.

8.4 Case Study 2

In the second case study, the same general set up as case study 1 was used. The first meaningful difference was the system to be studied and make decisions about. This time, a system is investigated that facilitates connecting several applications to a data store through services that provide specialised functionality. Just as in the first case study, the precise description and application of this system will not be discussed beyond those aspects relevant to this case study.

The services that implement this functionality are being designed as microservices. In the schematic overview shown in Figure 38, the data access service is used to interact with the data store, and to abstract away the direct control of this store. Other microservices connect to this and provide several types of information and functionality for use in different kinds of applications. One part of this is the *system configuration service* – also called *SysConf* by the participants. Applications or upstream services can request information about the system that they are running on such as system state and user info (request/response). Another part to this interaction is that the system configuration service can send notifications to these applications when something in the system configuration changes and they are subscribed to be notified of it.

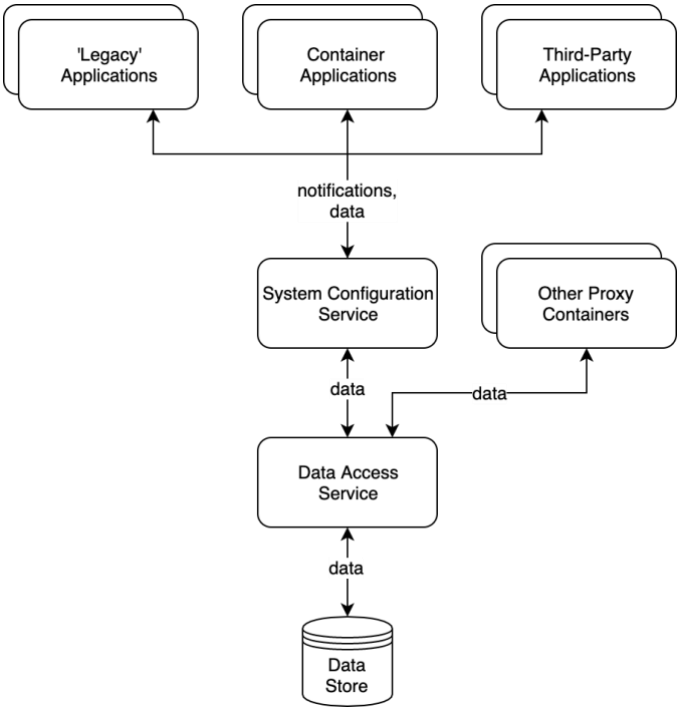


Figure 38 - Schematic Overview of Case Study 2 System

Several aspects to this case study are similar to the first one. Again, the time available for the case study was limited. Therefore, the decision was made to focus on the part where the system configuration connects to applications. Also, practitioners again indicated a strong preference to first consider the technical, communication-related challenges. This category (previously shown in Figure 28) was therefore selected to be focussed on in this case study also. This time, again four participants were available. More would have been better, but this was wat time and planning constraints allowed for. This case study was also conducted in two hours.

8.4.1 Observations

The second case study started off with capturing requirements and selecting those that are expected to be most impactful when making decisions. The requirements were as follows:

- Support for enabling High Availability
- Support for Third Party Access to the service
- Support for Polyglot programming
- Support for Technology Agnostic development
- Support for evolution → in this case regarded as interchangeable with being technology agnostic
- Support for accountability on user → for now seen as constraint; probably able to make this work with many if not all options
- Support for health checks → required in all cases, so seen as constraint

The first four were used as input for this case study, since others could be regarded as constraints that were expected to not contribute much towards distinguishing decision alternatives. Polyglot programming and Technology Agnostic development seem to be alike but are intended to specifically refer to the programming language, and the technology stack

used, respectively. The support for being able to easily evolve the service in the future was for this case study regarded as interchangeable with being technology agnostic. The required support for accountability on a user was a constraint that arose from the project and business context. As indicated, participants were convinced that this could be implemented in most – if not all – cases and therefore chose to not involve this requirement for comparisons. Furthermore, the required support for health checks of the service was non-negotiable and thus also seen as constraint. This could be taken into account when searching for alternatives to consider for each decision. The first four requirements were then prioritised by making pairwise comparisons. The outcomes of this are shown in Figure 39.

	Weight
Criteria	100.0%
HighAvailable	42.8%
Polyglot	34.3%
TechnAgnos	15.9%
ThirdParty	6.9%

Figure 39 - Prioritisation of Criteria

The first decision in this was that of *communication mechanism*s. This deals with the choice between synchronous and asynchronous communication, as well as one-to-one and one-to-many styles. When discussing this decision for the SysConf interface, it became clear that for some functionalities a synchronous approach would fit best, whereas for others asynchronous communication was more natural. For example; to request certain data on demand, a request/response pattern would suffice, but to notify services of a change in system state when for instance a user logs in with another role, a publish/subscribe mechanism would be better suited. This resulted in a discussion about how to combine the two functionalities in one service. At that point, it became clear that there was no consensus yet as to whether this should be handled by a single service, or multiple. To not overcomplicate the case study within the limited time, the choice was made to focus on the use case of notifying services of a change in system state. In line with this, for communication mechanism publish/subscribe was selected by the participants as it was seen as the only viable alternative in this case and thus no comparisons were necessary. One nice insight from this was that this distinction was big enough to warrant the use of two (or more) interfaces. Thinking of microservices, these could also be implemented using two separate services, each providing their own different interface.

The next step was to look at how to implement this service interconnection. Several alternatives were considered here. ActiveMQ and ZeroMQ are well-known implementations of messaging, with a broker-based and brokerless approach respectively. A native DDS solution and the java-based HazelCast were also included as alternatives. An overview of the considered requirements and decision alternatives as output by the AHP R library is shown in Figure 40. The alternatives were then ranked using pairwise comparisons. It turned out that in this comparison, the TechnAgnos requirement did not meaningfully divide the alternatives. In line with how such a situation is handled in ArchDesigner [87], the comparisons regarding this requirement were all assigned the weight 1 – with the assumption that they were equally strong in supporting this requirement. Furthermore, the Polyglot requirement was seen as an all-or-nothing requirement; it is supported or not. This resulted in ranking of alternatives using the extreme ends of the scale; when one supported Polyglot and another did not, the highest

rating was assigned and so forth. In other requirements, more nuance could be shown in the comparisons. The outcomes were as shown in Figure 41.

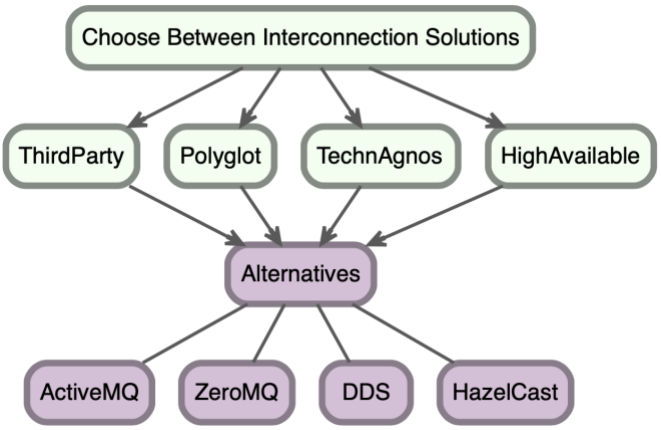


Figure 40 - Decision Overview for Service Interconnection

	Weight	ZeroMQ	DDS	HazelCast	ActiveMQ
Choose Between Alternatives	100.0%	59.7%	20.4%	12.3%	7.6%
HighAvailable	42.8%	27.2%	8.5%	5.5%	1.6%
Polyglot	34.3%	24.4%	7.3%	1.3%	1.3%
TechnAgnos	15.9%	4.0%	4.0%	4.0%	4.0%
ThirdParty	6.9%	4.0%	0.7%	1.5%	0.7%

Figure 41 - Ranking of Service Interconnection Alternatives

As can be seen, ZeroMQ is highly preferred over other solutions based on the comparisons with regards to the four set requirements. A large part in this decision was the fact that ZeroMQ uses a brokerless model, which was seen as positively influencing the high availability requirement. Since this requirement has much weight in the overall decision, this is of great influence.

After this decision, the time was almost up. This meant that there was no more time to choose between alternatives for service discovery. In order to verify that providing guidelines was also seen as helpful by participants, the interface design aspect was briefly discussed. The same guideline of using an IDL whenever possible was presented, and participants indicated that they would find such guidelines helpful, as long as they were to-the-point; i.e. gave advice that could quickly be used in practice.

The next step would then be to discuss the decision made. Putting the separate chosen alternatives together to form a coherent solution requires some discussion and a reality-check. With the limited decisions made in this case study, this discussion by the participants was quite short. They felt that the best scoring alternatives were a good fit for the system and were confident that they could also find a solution to the service interconnection challenge that fit with the others. The final step of evaluating the decision outcomes by putting the solutions into practice was obviously out of scope for this case study, but participants indicated that they felt this was a logical and necessary step in the process. Especially since they use an agile software development method, this step felt natural.

In the discussion step, the question is how to combine the alternatives in a way that makes sense for the entire system. This discussion was brief, since not too many decisions were made. The two decisions that were made on the use case of the SysConf container notifying

services when the system state is changed were clear and usable in practice: utilise a publish/subscribe mechanism and implement this using ZeroMQ. Practitioners indicated that it would be likely that this combination would also be used in practice later on. When asked whether feedback of a future implementation could be used in the evaluation stage, participants noted that this could indeed be the case, but that they might need guidance in how to use this feedback in a next iteration of the decision-making framework. For example, if a certain alternative is found to eventually be of insufficient quality, it could of course be removed from the comparisons for a particular decision. However, the underlying reasons behind this could be numerous. It might be that the requirements and their prioritisation have been off, or the scoring of alternatives in different comparisons were based on inaccurate estimates for how well alternatives supported criteria. These would therefore probably also have to be revisited to ensure that future decisions are based on reliable inputs.

8.4.2 Results

After the execution of this second case study, the participants were again asked to comment on how they felt about the use of the decision-making framework. The general consensus was that they felt that it could be a supportive tool to support decision-making. Participants indicated that they liked the use of pairwise comparisons, as they felt this would help them make more informed decisions. Participants again indicated that they could not find any apparent shortcomings in the overview and dependencies between challenges in the artifact. Another insight was that it would be nice to document the rationale between certain comparisons. E.g. why is option A better than B with regard to a certain requirement? By documenting this during decision-making, a large part of the design-rationale can be established according to the participants.

Further insights came from the discussion of what requirements were more important than others. Often, the practitioners questioned what requirements were more important on the short or long term. I.e. when looking at the short term, some requirements might be more important whereas this importance might change when considering long-term development and use of the system. It also turned out that during the decision-making process, sometimes the choice and ranking of requirements was questioned. This was not necessarily due to a distrust in the way these rankings were calculated, but rather the participants asking themselves whether they gave the right weights when comparing criteria. Some indicated that maybe after having some practice with how it all works, in a next iteration the requirements to include and their ranking may be changed by decision-makers. Furthermore, the consensus was that probably more than one interface would be needed to support all the SysConf functionalities. It is to be decided whether these are split up into multiple services. This discussion was not foreseen and could have maybe been avoided by first considering the *service granularity* challenge from the Management category in the overview of challenges.

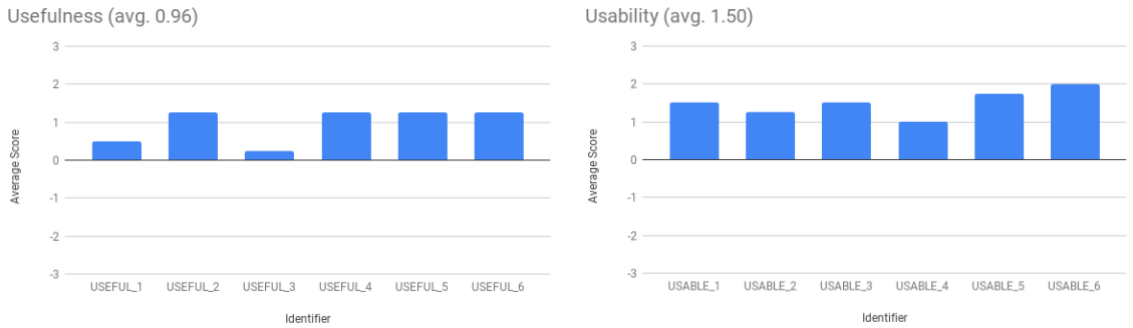
Other remarks made by participants were that they found the artifact practical and well-structured. They also particularly liked the fact that it forced them to think about the rationale behind decisions, partially thanks to making pairwise decisions, and avoid inconsistent decisions. Points of improvement focussed on the tooling that was not mature yet, as well as that it could take quite some time to apply the decision-making framework for a large project. Participants also foresaw cases in which part of the framework could also be used to make specific choices when needed. For example, in an existing system, certain decisions may have already been made. When for instance a system's communication mechanism and service interconnection implementation are already determined, these can be filled in. A subsequent

choice on the service discovery decision can then use the input from these already made decisions.

To sum up, the practical observations from this case study were:

- Making pairwise comparisons using the AHP was received favourably and seen as supportive in making more informed decisions;
- The overview of and dependencies between decisions showed no shortcomings to the participants;
- Documenting the reasoning behind a certain pairwise comparison could be useful for establishing design-rationale;
- Certain requirements can be more or less important when comparing short- and long-term implications to decisions made;
- At times, participants were uncertain about their rankings of requirements. It was indicated that this was likely to change with more practice;
- Sometimes, there can be dependencies with challenges in other categories;
- The artifact was seen as practical and well-structured;
- Participants liked that they were 'forced' to think about the rationale behind their decisions;
- The tooling was seen as not ready for use in practice yet;
- The use case of using part of the framework when a system needs to be changed and certain decisions are already set was also foreseen by participants.

The participants were asked to fill out the same survey as the one used in the first case study. The aggregated results are shown in Figure 42. It can be seen that – on average – the participants ranked all three of the investigated indicators of the artifact favourably. The decision quality indicator stands out most, since this shows a high improvement over that in the first case study. The lowest scores in the usefulness category were assigned to questions about whether the artifact could help participants accomplish their tasks more quickly and be more productive in their job. The lower scores here likely arise from the fact that decisions took quite some time during the case study, and therefore not many actual decisions could be made within the time available. The average score that participants gave about perceived practicality was 1.5, so the outcomes were seen as applicable in practice. Again, no technical shortcomings inhibiting practicality of the framework were found during the case study.



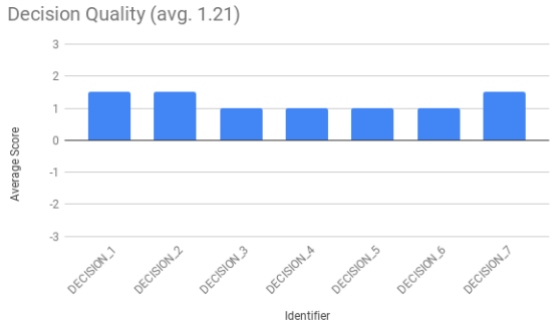


Figure 42 - Survey Results Case Study 2

8.5 Discussion and Conclusions

With the case studies complete, their results can be analysed. In this section, the knowledge questions set for validating the artifact are answered based on these results. Given the limited number of practitioners that participated in these case studies, no purely statistical analysis can be done. To assess the expected *usefulness*, *usability* and *decision quality* for decision-makers, the data gathered through the surveys in these case studies is used. Graphs summarising the outcomes for these indicators and their average scores have already been shown when discussing the individual case studies in Figure 34 and Figure 42. Refer to Appendix E for combined views of these indicators in both case studies. Histograms are also included to show the spread of answers given by participants. These give more insight beyond the mere average scores per indicator, such as insight in the spread of individual responses. These histograms have been normalised, meaning that they show relative frequencies. This denotes the proportion of responses that were assigned a certain value. The sum of the heights equal 1 for each data series. Note that the question scale used in the decision-quality related questions was different from the other indicators. This asked about agreement with statements rather than likeliness that a situation would be true in practice. The number of options and coding of scores was identical though. Nevertheless, an individual score regarding usefulness or usability can probably not be directly compared to a score for decision quality.

The first and main question to be answered is about functional correctness: can the artifact be used to support decision-making in a microservice architecture? During the case studies, participants were indeed able to make decisions using the framework about several microservice-related challenges. These decisions were based on real-world scenarios, and only the number of variables involved were limited due to time constraints. Participants were not limited in the actual requirements and alternatives to be considered in their decision-making. Furthermore, no indications were found that the overview of challenges and their relations were incorrect. Participants understood the structure and steps involved and found these well-structured and compatible with their software development process. Given these outcomes, it is expected that the decision-making methodology of the artifact can indeed be used to support decision-making in a microservice architecture. A notable limitation to this generalisation is that not all challenges could be addressed in the two case studies. Even though the process of decision-making is the same for all these challenges, no empirical data is available yet to validate whether the findings in these case studies hold true for all challenges within the scope of the artifact.

In the case studies, the average scores given to questions about the perceived *usefulness* of the artifact were 1.00 and 0.96, respectively. This corresponds with the answer option of *'somewhat likely'*. In terms stated by Davis [82], this would mean that the participants thought

it is somewhat likely that using the artifact would enhance their job performance. The observations from the case studies seem to be in line with this score. Participants indicated that they thought the artifact could indeed support discussions about managing challenges in a microservice architecture and had favourable opinions about the pairwise decision-making and structured approach. On the other hand, decisions did in many cases take a considerable amount of time. This likely inhibited the perceived usefulness, thus contributing to the score on this subject not being higher.

The average scores regarding *usability* were somewhat higher; 1.29 and 1.50, respectively. Again, this most closely corresponds with the *'somewhat likely'* answer option but is leaning a bit more towards *'quite likely'* than the scores for usefulness. Therefore, looking at the definition given by Davis [82], it can be said that on average participants thought it is somewhat likely that using the artifact would be free from effort. The perception of usability was likely helped by the fact that participants recognised the clear steps in the decision-making process and the fact that the process is meant to be iterative as this closely resembled their day-to-day work activities. Participants also indicated that after some practice, they would likely be able to use the artifact in a real-world scenario. The biggest inhibitor to the usability score was tooling, as many participants indicated that more mature tools to guide the decision-making process would likely improve it.

The indicator that showed most difference between the two case studies is *decision quality*. The results of the first case study were somewhat disappointing in this regard, with an average score, yielding an average score of 0,00. As described, this was expected to be caused by unclarity about the selection and prioritisation of requirements during this case study. The artifact was changed to include more guidance on how to select the right architecturally significant requirements, and how to assign scores in pairwise comparisons. The expectation was that this would improve decision quality. The second case study – conducted with these clarifications in place – showed an improvement in the perceived decision quality with an average score of 1.21. Besides this, whereas in the results from case study 1 more than half of the responses given regarding decision-quality were negative or neutral, all scores given in case study 2 are neutral or positive. Furthermore, in the second case study there was also more consistency in the answers, showing a sample variance of 0.69 as compared to 2.74 in case study 1. This means that the average reported for case study 2 is more indicative of the dataset than that for case study 1, which showed much dispersion of answers. The more favourable score from case study 2 is no definitive proof that the additions to the artifact have improved the perceived decision quality by participants on their own. The score can be indicative of *some* improvement in this regard. It must be noted though that, even though the case study execution was kept as consistent as possible between the two case studies, the actual case used for decision-making did differ. This resulted in different decision being made, which may have also caused a difference in decision quality scores. Nevertheless, the observation that in case study 1 the requirements selection and prioritisation had to be redone and in case study 2 this was not needed, while in case study 2 the additions to the artifact aimed at improving these steps were in place, can be a sign of improvement. A fair conclusion to be drawn than is that on average, the participants were neutral to somewhat confident about the quality of the decision outcomes. Scores on perceived practicality were also low in the first case study, but better in the second one, with averages of -0.25 and 1.5, respectively. This difference can likely also be attributed to the uncertainty in the requirements selection and prioritisation in the first case study. During both case studies, no technical shortcomings to the framework were found. This also gives a favourable view of the usability in practice of the decision-making framework.

In section 8.1 the knowledge goals to be answered through the case studies were defined. These were as follows:

- Can the artifact be used to support decision-making in a microservice architecture?
- To what extent do decision-makers find the artifact *useful*?
- To what extent do decision-makers find the artifact *usable*?
- How confident are decision-makers about the artifact's *decision outcomes*?
- To what extent do decision-makers find the artifact and its outcomes *usable in practice*?

As discussed in this section, the observations and results from the two case studies give reason to believe that the artifact can indeed be used to support decision-making in a microservice architecture. Participants on average found the artifact somewhat useful, and somewhat to quite usable. They were on average neutral to somewhat confident about the decision outcomes and practicality, with indications to believe that the changes to the artifact to improve on this yielded more favourable results. No technical shortcomings that would inhibit the decision-making frameworks use in practice were found.

It is expected that these findings can be generalised to real-world use of the artifact, because of the case studies' similarity to conditions of practice. The case studies were set up to be as similar to the real world as possible, and real scenarios were used as cases to be studied. Participants were merely limited in the number of variables involved, such as requirements and alternatives to be considered, not in their choice of these. As said before, little control was exerted over the treatment; the role of the researcher, materials and tooling was to educate participants about the decision-making framework and show its intended use, but not restrict them or intervene in their actual use during decision-making. In the case studies, not all identified microservice challenges could be considered. However, given that the process of decision-making is the same for all these challenges, and participants indicated no technical shortcomings to the overview of challenges, it is expected that the artifact is functionally correct for real-world scenarios.

The case studies and the answers to the knowledge questions presented in this chapter serve as the answer to RQ-5: *How can the designed framework's fitness for purpose best be validated?*

Besides this, there were some qualitative insights that arose when running the case studies:

- Allowing each participant to give their own rankings and then consolidating was seen as helpful to reduce biases present in collective decision-making;
- Making pairwise comparisons using the AHP was received favourably and seen as supportive in making more informed decisions and evoking discussions;
- The outcome of the decision-making process can possibly be used to document and justify the decisions made – establishing design-rationale;
- The tooling used during the case study was seen as too complicated and could be more 'hands-on';
- The artifact was seen as practical and well-structured;
- The use case of using part of the framework when a system needs to be changed and certain decisions are already set was also foreseen by participants.
- Participants liked that they were 'forced' to think about the rationale behind their decisions.

These insights, together with the outcomes of the surveys and the observations done during the case studies can be used to further improve upon the artifact in the future. Most often, the need for better tooling was mentioned as inhibiting the decision-making framework's potential to be shown. Nevertheless, participants were convinced that many parts of the framework would be useful for helping them manage challenges when developing microservice architectures.

9 DISCUSSION

9.1 Implications and Contributions

The main goal of this research was to design a decision-making framework to better manage the design challenges related to communication between, integration and management of microservices in the design of a microservice software architecture. To accomplish this, microservice challenges in literature and practice as well as decision-making methodologies for software architecture design have been researched. The combination of these two in the subsequently designed decision-making framework have implications for both organisations and future research.

Insights in microservice challenges for academia and practice

The topic of microservices has been evolving continuously and in a fast pace throughout the past few years. Many definitions, opinions and descriptions of the concept and its characteristics exist in both academia and practice. Finding clear and practical information on the challenges involved when implementing a microservice architecture can be hard and overwhelming for practitioners. In practical sources, much is written about all kinds of new tools and software solutions that will solve all kinds of challenges around microservices. Countless tools try to tackle as many problems as they can and are aiming to become the new 'go-to' solution to be used by organisations aiming to develop a microservice architecture. These are all solution-driven descriptions, though, and by only basing architectural decisions on such descriptions, unexpected challenges can be encountered down the road. The overview of challenges from academic literature can be used to give practitioners a clear and consistent insight into what they can expect to come across when designing such a system. Having a reference that originated from academic work can be valuable in this situation.

Conversely, the academic field to an extent seems to be lacking some practical insight into microservice challenges. Numerous academic works exist that discuss certain specific challenges, but only few exist that aim to summarise challenges and explain what this means for practice. Furthermore, the academic field researching microservices seems to be lagging behind due to the industry moving rather rapidly. No longer are software engineering problems related to microservice always first discussed in research, after which the outcomes can be translated into software solutions. Nowadays, the reverse is often true; new solutions are sometimes being used widely in practice, only to have academia catch up and try to understand what happened. The overview of challenges from literature as well as practice is useful in introducing more practical insights into the field of microservices research.

Combining the fields of decision-making and microservices

Decision-making is a well-researched topic in academia. Many tools and frameworks exist that describe general decision-making techniques such as the aforementioned AHP, as well as works aimed at software engineering specifically. However, no frameworks were found that were specifically tailored to help make decisions on microservice challenges. In combining the somewhat theoretical field of decision-making with the more practical field of microservice research, a decision-making framework has been created that builds upon a strong theoretical foundation but can easily be used in practice. Added to this is the inclusion of guidelines for certain microservice challenges. Not every challenge could be captured as a simple decision between alternatives, though they might influence other decisions.

Practical views on decision-making on microservice challenges

Throughout this research, the aim was to include practical insights with academic findings on the topics discussed. The research was conducted in collaboration with Thales Netherlands B.V., that had a desire to improve on the process of deciding on and managing challenges in a microservice architecture. The contributions from practice in present research helped shed a real-world light on motivations to conduct this research, identifying challenges and decision-making techniques. For example, the decision to not consider dependencies between all challenges when making decisions but rather divide them up in categories originated from feedback by practitioners. This decision was not made lightly, as it would fundamentally change the decision-making process. During these discussions the real goal for the decision-making framework also became clear; it was not to try and replace the process, but rather facilitate the discussion on microservice challenges. This switched the process from a tools-first focussed one to one that was specifically designed to help improve decision-makers' day-to-day work, not to replace it.

9.2 Research Quality

To reflect on the quality of the design research documented in this thesis, guidelines by Hevner et al. [103] are used. The authors propose seven guidelines that quality design research should incorporate. The guidelines, description and execution are shown in Table 14. When comparing these guidelines with present research, it can be seen that all have been fulfilled. Weak parts that can be identified are mainly the limited number of participants in the case studies used for validation, and the fact that only marginal changes to the artifact have been made within the scope of this research. Future changes are anticipated, but not included. The next step to improve on this is to implement the artifact in practice, evaluate upon this and make changes accordingly. Nevertheless, all other guidelines have been adhered to, suggesting that the research overall is reliable.

Table 14 - Guidelines for Desing Science - Adopted from [103]

Guideline	Description	Execution
1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method or an instantiation.	An artifact in the form of a decision-making framework for managing microservice challenges was developed.
2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.	The motivation for this research in part directly originates from a business, and the research benefits businesses by helping manage technical challenges.
3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.	Various methods of validation including interviews on the challenges and single case mechanism experiments for the artifact were conducted.
4: Research Contributions	Effective design science research must provide clear and verifiable contributions in the areas of the design artifact,	Practical and theoretical insights on microservices and decision-making were generated, along with a

	design foundations and/or design methodologies.	decision-making framework that can be used in practice.
5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.	The DSM by Wieringa was followed to structure the research. This methodology guided the design and evaluation of the artifact.
6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.	The validated design represents a first step in an iterative process of improvement. It is acknowledged that real-world use is needed to further assess its fitness for purpose, and possible future changes are also considered.
7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.	The descriptions and representations of the artifact were made to be communicated effectively to both academics and professionals involved with decision-making in microservices.

9.3 Validity and Reliability

When talking about research validity, Gregor and Hevner give a clear description: “validity means that the artifact works and does what it is meant to do; that it is dependable in operational terms in achieving its goals” [104]. Wieringa [4] describes different types of validity that are involved in design science research. Those applicable to this research and in particular its validation are *construct validity, descriptive validity, internal validity* and *external validity*.

“A conceptual framework is a set of definitions of concepts, often called constructs” [4]. Construct validity is defined by Wieringa as “the degree to which the application of constructs to phenomena is warranted with respect to the research goals and questions” [4]. The constructs of the conceptual framework can be subject to certain threats to their validity. Wieringa discusses several of these. A first threat is inadequate definition; constructs should be clearly defined to be able to distinguish instances of a concept from those that are not. Effort has been put in defining the concepts involved in the conceptual framework based on related literature. For instance, a meta-model was included in the artifact to clearly show the different entities and their interactions during decision making. Through these measures, inadequate definition is avoided as much as possible. Construct confounding is another threat; in which instances of use cases can be ambiguous. An example to help aid in this is the inclusion of usage requirements to the artifact. These describe the cases in which it should be applicable, to rule out cases that it is not. These also help in defining the population to which the findings can be generalised. Another possible threat is that of mono-operation bias; in which the indicators defined for a concept do not fully capture it. This was avoided as much as possible; by for instance measuring indicators on usability, usefulness and decision quality in the case studies by using measurement questions from academic research. This way,

participants would not only be directly asked to comment on these indicators, but its value is determined by multiple different measurements. Mono-method bias can also be at play here, when for instance all indicators are measured in the same way. This is why besides the surveys in the case studies, also qualitative feedback and details about the decision-making process were gathered to find out if the views from both are aligned. That is not to say that qualitative feedback may not introduce other biases, but the combination of both should mitigate this as much as possible. With these measures against threats to construct validity in place, the aim is to make the constructs as valid as possible.

Descriptive validity is relevant for assessing the support for descriptive inferences made in this research. Checklists that Wieringa [4] proposed for designing the validation methodology can be looked back at after execution of the research to assess the descriptive validity of the inferences. For instance, questions like whether prepared data represents the same phenomena as the 'raw' results, as well as whether scientific peers would be able to make the same interpretations from the data presented. The measurements and descriptions of the outcomes have been prepared with descriptive validity in mind. In the interviews on challenges, for example, participants were asked whether the interpretations of their answers matched their beliefs (member checking). Another example is that during the case study no data was removed, results were discussed qualitatively to give more context for interpretation (triangulation), and the statistical procedures applied to data were straight-forward and could be reproduced by others.

Internal validity refers to the support of any abductive inferences done in this research. Wieringa [4] describes that the single case mechanism case studies that were conducted can support abductive inference. Again, threats to internal validity are listed. A main concern is possible sampling influence that can introduce biases in the results. For example, there can be a selection effect by which subjects participating in a case study can act differently just because they have been chosen to participate. Besides random selection of subjects, it is hard to control this effect. No direct indications were found during the case studies that participants altered their behaviour, but there is also no evidence to definitively say that this cannot have been the case. It was found that participants were not hesitant to voice any concerns during the case study, indicating that they were not likely to be less critical to satisfy the researcher. Treatment must also be controlled, by for instance randomly selecting subjects, or at least as random as possible. Practitioners were asked about possible case studies and the teams involved, but after selection of the case studies, no deliberate selection between team members were made; all were asked and free to participate. The experimental set up was also made to resemble working conditions from practice closely, as to reduce any responses based on the set up alone. It is possible that subjects have responded to the novelty of the artifact under study in these case studies. While no direct indications were found for this, there is a chance that participants responded more favourably to the artifact just because it was novel. Influences by the experimenter were also avoided as much as possible by being aware of the possible threats of experimenter expectation – bias through hope for a particular experiment outcome – and experimenter compensation – when experimenters treat subjects differently based on their interaction with the treatment. Awareness of these threats helped in the experimenter not blindly making the mistakes described.

External validity comes into play when generalising about findings beyond the cases in which it has been tested. Wieringa describes requirements for this [4]. Validation cases were chosen that fulfilled the usage requirements outlined in the artifact design. When searching for cases, practitioners were explicitly asked for real-world cases that were currently being considered. This was done to ensure they were a representative sample of the target population, thus

helping support analogous inference. The treatment was also kept as similar to the intended use in practice as possible, by for instance only limiting the number of requirements and alternatives considered, but not the decision-making process and options itself. One notable inhibitor for validity is the limited number of participants during validation, which makes for less certainty about the generalisability of the findings. The artifact behaved as intended during the case studies. By controlling as many factors as possible to create an environment similar to the intended environment for use in practice, generalisations are supported as well as possible.

As for the question of how *reliable* present research is, we must look at the extent to which the operations of the study can be repeated with the same results [105]. The research followed Wieringa's DSM [4] as design research protocol. The steps taken in this process have been described as detailed as possible. The fact that practical insights from the case study largely originate from Thales Naval does somewhat inhibit repeatability. While for many practical insights it is expected that they also occur in similarly sized and structured organisations, every organisation is different. It is expected that if this research were to be conducted in collaboration with another company, similar general findings will likely be found.

As Wieringa [4] states; constructs and inferences are never totally valid, since science is fallible. This is true for all described types of validity. No major threats to the types described have been found. Nevertheless, case study research can be limited in especially construct validity because the measurements are inherently subjective to an extent [105]. Instruments such as multi-method measurements and using multiple measurements per indicator have been put into place to limit subjectivity. Still, the number of participants was a notable limiting factor for validity. Overall, care has been taken to have as little threats to research validity and reliability as possible. It is expected that the general findings are supported well through the conducted research, and that these are replicable in future research.

9.4 Future Work

In this thesis, one particular decision-making framework for managing microservice challenges related to communication, integration and management has been designed and validated. This leaves multiple opportunities for future research to be conducted. First and foremost, other categories of microservice challenges can be researched to investigate whether they can also be incorporated in the design of the artifact. While it is expected that many challenges can be addressed in similar ways as described in this research, there may be exceptions. Furthermore, the research field on the topic of microservices is still under constant development because only recently it has gained much attention, and its current popularity. Therefore, it can be expected that in the future, additions to the overview of challenges may need to be made. Conversely, as technology advances, some challenges may become easier to solve in the near future. The rapid development of the microservices field can open up new research opportunities just as quickly as it evolves.

Besides this, the artifact has been validated for its fitness for purpose. The design was based on a base methodology selected in comparison with other decision-making methodologies to minimise the involved expected difficulties. The validation suggests that the artifact may be used in practice. However, it has not yet been compared to other decision-making approaches. Its design is tailored specifically to suit the identified microservice challenges, but there may also be a way to incorporate said challenges in existing general decision-making methodologies. Research that compares the use of the current artifact with that of pre-existing methodologies in combination with microservices could help further identify strengths and weaknesses of the artifact. This then gives opportunities for future development of the artifact.

Furthermore, when using the decision-making framework, decision-makers end up with decisions on what alternatives to choose for certain challenges, and information about the guidelines available in practice to help solve them. This information then still needs to be translated to an actual microservice architecture design before it can be implemented. The focus of the artifact is on finding out how to solve challenges on a conceptual level but is not involved with translating these findings to eventually end up with a working system. Future research should focus on the incorporation of these decisions in software architecture design and their implications.

Wieringa also describes the scaling up approach of research. This is illustrated in Figure 43. One goal is to "test an increasingly realistic model of the artifact under increasingly realistic conditions of practice" [4]. This so-called case-based inference can gain plausibility when research conditions increasingly approach practice. Sample-based inference on the other hand, can be used to generalise towards the intended population. The different types of experiments shown can be used to 'scale up' the research on a topic. In this research, single-case mechanism experiments were employed. This makes for the recommendation to include expert opinion, tactical action research or statistical difference-making experiments in future research to approach conditions of practice and the intended population for more robust inferences.

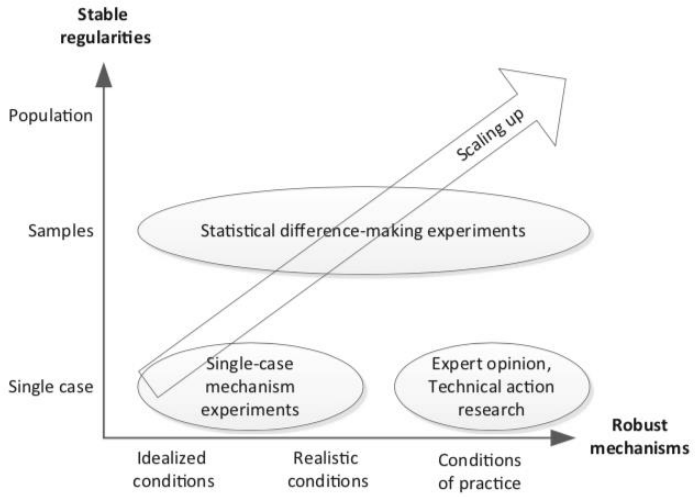


Figure 43 - Scaling Up Approach in Design Science Research - Adopted from [4]

9.5 Further Recommendations

A large inhibiting factor to realising the artifact's true potential in practice was found to be the tooling used to support the decision-making. Given the constraints to the case studies and the organisational context, tooling was chosen that best fit with this. However, during the case studies it turned out that using this tooling was rather cumbersome, and that while it worked for running the case studies by the researcher, it would be unsuitable for use in practice. In the search for tooling, many alternatives were found that supported the AHP decision-making process. Many of these were in the form of specialised SaaS products. While offline tools exist that could possibly be used with success after training to become familiar with them, these generally did not seem as easy to use and learn as the online offerings. It is therefore strongly recommended that organisations aiming to use the decision-making framework in practice, make use of one of the SaaS offerings by various vendors.

A possibly highly valuable outcome of making decisions using the artifact designed in this research, is that its decision-outcomes and pairwise comparisons used to arrive at these decisions can be used for establishing the design rationale behind these decisions. This way, the outcomes can become part of a business' organizational memory. Wijnhoven [106] describes different functions of such memory. The investigated and chosen alternatives for challenges in this decision-making methodology can function as *know-how* information about what solutions are used in practice. The comparisons made and the reasoning behind them can serve as *know-why*; why certain decisions were made and what the reasoning behind them was. Furthermore, application of the framework can help in externalising tacit memory – e.g. software architects' experience – into explicit memory when design rationale is systematically documented.

10 CONCLUSIONS

10.1 Research Questions

The motivation to conduct this research originated from both an academic ambition to further investigate and aid in mitigating challenges when designing a microservice architecture, as well as a practical need to improve this process. The main technical research problem set during this research was:

> *How to design a decision-making framework that gives confidence in its results, and satisfies effort and practicality requirements so that decision-makers can better manage the design challenges related to communication between, integration and management of microservices in the design of a microservice software architecture?*

For this problem to be addressed, several open descriptive knowledge questions were defined. The first question aimed at investigating microservice challenges in academia:

RQ-1 What common design challenges related to communication between, integration and management of microservices can be found in academic literature?

Through a structured literature review of 40 selected academic works, a total of 42 challenges in 8 categories have been identified. Individual challenges occurrences ranged from 1 to 13 times. No challenges were disregarded for being uncommon. These were then mapped to the topics of communication, integration and management, resulting in 15 challenges being proposed to be used in the artifact design. Challenges ranged from purely technical considerations to high-level management-related questions. A practical view on these challenges was then gathered through interviews aiming to answer the second question:

RQ-2 What common design challenges related to communication between, integration and management of microservices can be found in practice?

A total of seven practitioners at Thales Naval were interviewed to understand their view on microservice challenges. In general, they recognised the challenges found in literature, though for their organisation some challenges did receive more attention than they did in academia. The categories of challenges that were considered as hardest to manage were also identified, which showed to largely concur with the communication, integration and management-related challenges that were emphasised. Minor changes to the challenge overview were made based on these interviews; most notably the splitting of one challenge into two to better be able to address different aspects of it. With a set of challenges as input, the next question was:

RQ-3 What are the dependencies between the identified challenges and what possible alternatives and guidelines are available as solutions?

Through searching academic works and sources from practice, challenges were characterised, their dependencies shown, and possible decision alternatives and guidelines were documented. It must be noted that these descriptions are not exhaustive, nor intended to – given that the field of microservices is under constant development. This overview of challenges and their dependencies can be used in the artifact design as reference for which challenges to consider, and in what order. After this, the treatment design could start. The decision-making framework to be designed would build upon existing work in the decision-making field of research. Therefore, the question was:

RQ-4 What decision-making methodology for selecting between design alternatives can serve as conceptual foundation for the framework to be designed?

A comparison of decision-making methodologies for selecting between software architecture alternatives was made to find the most suitable methodology to be used as theoretical foundation for the artifact design. The *ArchDesigner* methodology by Al-Naeem et al. [87] was selected for this, after which the artifact design commenced. The design was made to best fulfil the requirements set at the start of this research. To validate whether the goals set for the design were accomplished, validation research was done to answer the last question:

RQ-5 How can the designed framework's fitness for purpose best be validated?

Through two case studies in the form of single-case mechanism experiments, this validation was done. Two different cases were considered, and in each case study conditions of practice were replicated as closely as possible. In both case studies, four practitioners participated. The outcome of the first case study suggested that the artifact was somewhat usable and useful, but decision-quality was lacking. Additions to the artifact in the form of more guidance on what requirements to select and how to weigh them when comparing were made. The second case study showed minor improvements in usability and usefulness, but a significant improvement in decision-quality. Though it cannot be said definitively that the increase decision-quality rating was due to the changes to the artifact, it is expected that a fair conclusion would be that on average, the participants were neutral to somewhat confident about the quality of the decision outcomes.

10.2 Key Contributions and Findings

During this research, the aim was to base the artifact's on academic literature as well as insights from practice. The research was conducted at Thales naval to ultimately better align academic and practical views on the topics at hand. Through answering the RQs set in this research, and the design of the artifact itself, several contributions to academia and practice have been made:

- *Scientific:* A previously not found overview of microservice challenges has been constructed through literature research.
- *Scientific:* Knowledge on decision-making methodologies for software architecture design has been applied to a practical case in the field of microservices, providing insights in the behaviour of such methodologies in this context.
- *Practical:* Academic literature has been used to characterise microservice challenges in a clear and consistent way, providing insights in what challenges can be encountered when designing a microservice architecture, as well as possible decision alternatives and guidelines to consider.
- *Practical:* A previously non-existent decision-making framework to be used for managing microservice challenges in practice has been designed based on an academic foundation and was validated in practice.

Through several statements, the most important findings of this research are discussed next.

Microservice challenges are numerous and touch upon all parts of an organisation.

Through the conducted literature research and subsequent interviews with practitioners, it became clear that using a microservice architecture comes with many challenges that are not straight-forward to solve. It can be hard to find out which ones to consider, because they are

so various. Challenges range from service discovery to organisational structure and culture, and from memory consumption to the development process used.

Not all microservice challenges have clear alternatives to decide between for managing it; sometimes guidelines are necessary.

Microservices are a rather opinionated subject; practical sources that lead to their recent popularity are vocal about characteristics that microservice architectures are supposed to possess. In line with this, for certain challenges certain guidelines are given to solve it in a microservices-fashion. For other challenges, clear and distinct alternatives are available to choose between. This is different for each challenge, but both types of challenges may influence each other. It is therefore important to consider both when designing a microservice architecture.

Merely expressing optimal alternatives numerically in decision-making is not always trusted in practice.

A further main insight that had a large impact in the artifact design was that practitioners indicated that they would not easily trust a single ranking or score to determine a definitive set of alternatives to be used in a microservice architecture. Creating a decision-tree like structure can also become cumbersome quickly and can reduce insight and thus confidence in the results of decision-making. By employing a decision-making process structured in categories and focussing on finding the best few alternatives for each decision, a final discussion can take place to assemble these pieces into a complete solution. This sanity-check as it was called by practitioners was in their view of vital importance in providing confidence in the framework's outcomes.

A decision-making framework need not always replace a way of working; it can also be used for guidance in existing work practices.

Throughout this research, practitioners involved with decision-making and software architecture design were vocal about the role that the decision-making framework should take; it should be used to help support discussions on how to manage microservice challenges. This approach of putting practice before tools turned out to be viable, since the designed artifact with this in mind was received favourably by practitioners. In qualitative feedback, the general consensus was the fact that it was practical and could easily fit in with their current day-to-day work was an advantage to them.

Insights gathered during the decision-making process can be just as valuable as its outcomes.

Feedback by practitioners during the case studies suggests that by documenting the reasoning and considerations made during the decision-making process, valuable information on why certain decisions were made and on what assumptions weights given in comparisons are based can be gathered. By doing this in a structured way, it was seen as possibly valuable input for establishing design rationale behind a microservice architecture design.

BIBLIOGRAPHY

[1] N. Dragoni *et al.*, “Microservices: yesterday, today, and tomorrow,” pp. 1–17, 2016.
[2] M. Fowler and J. Lewis, “Microservices,” 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: 20-Feb-2018].
[3] O. Zimmermann, “Microservices tenets: Agile approach to service development and deployment,” *Comput. Sci. - Res. Dev.*, vol. 32, no. 3–4, pp. 301–310, 2017.
[4] R. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. 2014.
[5] P. J. M. Verschuren and J. A. C. M. Doorewaard, “Het ontwerpen van een onderzoek.” Utrecht : Lemma, 1995.
[6] SIPRI, “The SIPRI Top 100 arms-producing companies, 2008, SIPRI Fact Sheet,” 2010.
[7] TechTarget, “API management,” 2018. [Online]. Available: https://searchmicroservices.techtarget.com/definition/API-management. [Accessed: 12-Sep-2018].
[8] CITO Research, “Cloud-based API Management: Harnessing the Power of APIs,” 2015.
[9] Thales Group, “TACTICOS - Worlds’ favourite Combat Management System - The best got better,” 2015.
[10] D. Falessi, G. Cantone, R. Kazman, and P. Kruchten, “Decision-making techniques for software architecture design,” *ACM Comput. Surv.*, vol. 43, no. 4, pp. 1–28, 2011.
[11] S. Lauesen, “Software requirements styles and techniques,” *Neurosurgery clinics of North America*, vol. 20, no. 2. pp. 179–86, 2009.
[12] H. Heerkens and A. Van Winden, *Geen Probleem*. Nieuwegein: Van Winden Communicatie, 2012.
[13] S. Newman, *Buiding Microservices*. O’Reilly Media Inc., 2015.
[14] Gartner, “Hype Cycle Research Methodology.” [Online]. Available: https://www.gartner.com/en/research/methodologies/gartner-hype-cycle. [Accessed: 21-Nov-2018].
[15] Gartner, “Hype Cycle for Application Architecture, 2015.” [Online]. Available: https://www.gartner.com/doc/3102217/hype-cycle-application-architecture-. [Accessed: 21-Nov-2018].
[16] Gartner, “Hype Cycle for Application Architecture, 2016.” [Online]. Available: https://www.gartner.com/doc/3392818/hype-cycle-application-architecture-. [Accessed: 21-Nov-2018].
[17] Gartner, “Hype Cycle for Application Architecture, 2017.” [Online]. Available: https://www.gartner.com/doc/3763463/hype-cycle-application-architecture-. [Accessed: 21-Nov-2018].
[18] Gartner, “Hype Cycle for Application Architecture, 2018.” [Online]. Available: https://www.gartner.com/doc/3886164/hype-cycle-application-architecture-. [Accessed: 21-Nov-2018].
[19] C. Richardson, “Microservices,” 2017. [Online]. Available: http://microservices.io/. [Accessed: 12-Apr-2018].
[20] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” *Linux J.*, vol. 2, no. 239, 2014.
[21] B. Schmaus, “Deploying the Netflix API – Netflix TechBlog – Medium,” 2013.
[22] M. Fowler and J. Lewis, “Microservices Guide.” [Online]. Available: https://martinfowler.com/microservices/. [Accessed: 20-Mar-2018].
[23] M. Fowler, “Microservice Trade-Offs,” 2015. [Online]. Available: https://www.martinfowler.com/articles/microservice-trade-offs.html. [Accessed: 07-Aug-2018].
[24] D. Moher, A. Liberati, J. Tetzlaff, and D. G. Altman, “Guidelines and Guidance Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement.”
[25] R. B. Schwartz and M. C. Russo, “How to Quickly Find Articles in the Top IS Journals,” *Commun. ACM*, vol. 47, no. 2, 2004.
[26] C. Pahl and P. Jamshidi, “Microservices: A Systematic Mapping Study.”
[27] L. Bass, I. M. Weber, and L. Zhu, *DevOps : a software architect’s perspective*. .

[28] J. Gray, “A conversation with Werner Vogels,” *ACM Queue*, vol. 4, no. 4, pp. 14–22, 2006.
[29] M. E. Conway, “How do Committees Invent?,” *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.
[30] Thales Group, “Naval Mission Solutions - Guaranteed mission system performance with Thales on Board,” 2012.
[31] Thales Group, “TACTICOS Combat Management System - Exploiting the Full DDS Potential,” 2006.
[32] Thales Group, “Tacticos | CMS | Thales Group,” 2018. [Online]. Available: https://www.thalesgroup.com/en/tacticos-combat-management-system. [Accessed: 20-Feb-2018].
[33] Object Management Group, “DDS Specification,” 2015. [Online]. Available: https://www.omg.org/spec/DDS/About-DDS/. [Accessed: 23-Mar-2018].
[34] ADLINK Technology IST, “Vortex OpenSplice.” [Online]. Available: http://www.prismtech.com/vortex/vortex-opensplice. [Accessed: 23-Mar-2018].
[35] ADLINK Technology IST, “Data Distribution Service.” [Online]. Available: http://www.prismtech.com/vortex/technologies/data-distribution-service. [Accessed: 26-Mar-2018].
[36] A. Corsaro, “OpenSplice DDS - A Gentle Introduction,” 2009.
[37] OSGi Alliance, “OSGi Architecture.” [Online]. Available: https://www.osgi.org/developer/architecture/. [Accessed: 07-Nov-2018].
[38] H. Bossenbroek and R. Van Hees, “The INAETICS architecture - Introducing INAETICS,” 2015.
[39] B. Kool, “Integrated Mission Management, een onderzoek door prototyping,” *Marineblad*, pp. 11–15, Oct-2017.
[40] R. van (Robertus J. M. Tulder, *Skill sheets : an integrated approach to research, study and management*. Pearson, 2012.
[41] M. L. George, *Lean Six Sigma for Service*. 2003.
[42] J. W. Wittwer, “PICK Chart for Lean Six Sigma,” 2017. [Online]. Available: https://www.vertex42.com/ExcelTemplates/PICK-chart.html. [Accessed: 24-Oct-2018].
[43] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, “Cloud Container Technologies: a State-of-the-Art Review,” *IEEE Trans. Cloud Comput.*, 2017.
[44] A. Sill, “The Design and Architecture of Microservices,” *IEEE Cloud Comput.*, 2016.
[45] Cloud Native Computing Foundation, “CNCF Cloud Native Interactive Landscape.” [Online]. Available: https://landscape.cncf.io/. [Accessed: 19-Oct-2019].
[46] S. Hassan and R. Bahsoon, “Microservices and their design trade-offs: A self-adaptive roadmap,” in *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, 2016.
[47] J. P. Gouigoux and D. Tamzalit, “From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture,” in *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, 2017, pp. 62–65.
[48] C. Richardson, *Microservices Patterns*. Manning Publications, 2017.
[49] E. Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software: Amazon.de: Eric J. Evans: Fremdsprachige Bücher,” vol. 7873, no. 415, p. 529, 2003.
[50] K. Lew and S. Narayanan, “Lessons from Building Observability Tools at Netflix,” 2018. [Online]. Available: https://medium.com/netflix-techblog/lessons-from-building-observability-tools-at-netflix-7cfafed6ab17. [Accessed: 20-Nov-2019].
[51] Y. Yu, H. Silveira, M. Sundaram, Y. Yale, H. Silveira, and M. Sundaram, “A microservice based reference architecture model in the context of enterprise architecture,” *2016 IEEE Adv. Inf. Manag. Commun. Electron. Autom. Control Conf.*, pp. 1856–1860, 2016.
[52] A. R. Sampaio *et al.*, “Supporting microservice evolution,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 539–543.
[53] J. Postel, “DOD standard transmission control protocol,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 10, no. 4, pp. 52–132, 1980.
[54] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, “Assuring the Evolvability of Microservices: Insights into Industry Practices and Challenges,” pp. 1–11, 2019.
[55] M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges When Moving from Monolith to Microservice Architecture,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7703, pp. 189–203, 2012.

[56] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, “Open Issues in Scheduling Microservices in the Cloud,” *IEEE Cloud Comput.*, vol. 3, no. 5, pp. 81–88, 2016.
[57] S. Dustdar and W. Schreiner, “A survey on web services composition,” *Int. J. Web Grid Serv.*, vol. 1, no. 1, pp. 1–30, 2005.
[58] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, “Web services composition: A decade’s overview,” *Inf. Sci. (Ny).*, vol. 280, pp. 218–238, 2014.
[59] M. Fowler, “CQRS.” [Online]. Available: https://martinfowler.com/bliki/CQRS.html. [Accessed: 21-Oct-2019].
[60] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, “Gremlin: Systematic Resilience Testing of Microservices,” in *Proceedings - International Conference on Distributed Computing Systems*, 2016, vol. 2016-Augus, pp. 57–66.
[61] Netflix, “GitHub - Netflix/chaosmonkey: Chaos Monkey is a resiliency tool that helps applications tolerate random instance failures.” [Online]. Available: https://github.com/Netflix/chaosmonkey. [Accessed: 21-Oct-2019].
[62] “Principles of Chaos Engineering.” [Online]. Available: https://principlesofchaos.org/?lang=ENcontent. [Accessed: 21-Oct-2019].
[63] M. Garriga, “Towards a Taxonomy of Microservices Architectures,” in *SEFM 2017 Workshops*, 2016, vol. 9763, pp. 203–218.
[64] D. Namiot and M. Sneps-Sneppe, “On Micro-services Architecture,” 2014.
[65] G. Hohpe and B. Woolf, *Enterprise integration patterns : designing, building, and deploying messaging solutions*. Addison-Wesley, 2004.
[66] K. B. Long, H. Yang, and Y. Kim, “ICN-based service discovery mechanism for microservice architecture,” in *International Conference on Ubiquitous and Future Networks, ICUFN*, 2017, pp. 773–775.
[67] C. Rotter, J. Illes, G. Nyiri, L. Farkas, G. Csatari, and G. Huszty, “Telecom strategies for service discovery in microservice environments,” in *Proceedings of the 2017 20th Conference on Innovations in Clouds, Internet and Networks, ICIN 2017*, 2017, pp. 214–218.
[68] J. Stubbs, W. Moreira, and R. Dooley, “Distributed Systems of Microservices Using Docker and Serfnode,” in *Proceedings - 7th International Workshop on Science Gateways, IWSG 2015*, 2015, pp. 34–39.
[69] OpenAPI Initiative, “OpenAPI Specification | Swagger.” [Online]. Available: https://swagger.io/specification/. [Accessed: 19-Oct-2019].
[70] RAML Workgroup, “raml-spec/raml-10.md at master · raml-org/raml-spec · GitHub.” [Online]. Available: https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/. [Accessed: 19-Oct-2019].
[71] W3C, “Web Services Description Language (WSDL) Version 2.0 Part 0: Primer.” [Online]. Available: https://www.w3.org/TR/wsdl20-primer/. [Accessed: 19-Oct-2019].
[72] SmartBear Software, “The Best APIs are Built with Swagger Tools | Swagger.” [Online]. Available: https://swagger.io/. [Accessed: 19-Oct-2019].
[73] Google, “Protocol Buffers | Google Developers.” [Online]. Available: https://developers.google.com/protocol-buffers. [Accessed: 19-Oct-2019].
[74] P. Clements *et al.*, *Documenting software architectures : views and beyond*. Addison-Wesley, 2003.
[75] P. Kruchten, H. Obbink, and J. Stafford, “The Past, Present, and Future for Software Architecture,” *IEEE Softw.*, vol. 23, no. 2, pp. 22–30, Mar. 2006.
[76] J. F. Maranzano, S. A. Rozsypal, G. H. Zimmerman, G. W. Warnken, P. E. Wirth, and D. M. Weiss, “Architecture Reviews: Practice and Experience,” *IEEE Softw.*, vol. 22, no. 2, pp. 34–43, Mar. 2005.
[77] P. Kruchten, “Mommy, Where Do Software architectures Come from?,” 1999.
[78] Fitzgerald, “The Transformation of Open Source Software,” *MIS Q.*, 2006.
[79] W. Engelsmana, D. Quartelc, H. Jonkersa, and M. van Sinderen, “Extending enterprise architecture modelling with business goals and requirements,” *Enterp. Inf. Syst.*, vol. 5, no. 1, pp. 9–36, 2011.
[80] J. Kontio, “OTSO: A Systematic Process for Reusable Software Component Selection,” 1995.
[81] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America, “A general model of software architecture design derived from five industrial approaches,” *J. Syst. Softw.*, 2007.
[82] F. D. Davis, “Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology,” *MIS Q.*, vol. 13, no. 0, pp. 319–340, 1989.

[83] C. S. Spetzler, H. Winter, and J. Meyer, *Decision Quality Value Creation from Better Business Decisions*. 2016.
[84] M. Svahnberg, C. Wohlin, L. Lundberg, and M. Mattsson, “A Quality-Driven Decision-Support Method For Identifying Software Architecture Candidates,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 13, no. 05, pp. 547–573, 2003.
[85] M. Moore, R. Kaman, M. Klein, and J. Asundi, “Quantifying the value of architecture design decisions: lessons from the field,” *25th Int. Conf. Softw. Eng. 2003. Proceedings.*, pp. 557–562, 2003.
[86] T. Gilb, *Competitive Engineering - A handbook for systems engineering, requirements engineering, and software engineering using planguage*. Butterworth-Heinemann, 2005.
[87] T. Al-Naeem, I. Gorton, M. A. Babar, F. Rabhi, and B. Benatallah, “A quality-driven systematic approach for architecting distributed software applications,” in *Proceedings of the 27th international conference on Software engineering - ICSE ’05*, 2005.
[88] R. Keeney and H. Raiffa, *Decisions with Multiple Consequences: Preferences and Value Tradeoffs*. Cambridge: Cambridge University Press, 1976.
[89] T. L. Saaty, “Decision making with the analytic hierarchy process,” *Int. J. Serv. Sci.*, vol. 1, no. 1, p. 83, 2008.
[90] C. E. Lindblom, “The science of ‘muddling through,’” in *Performance Based Budgeting*, 2018.
[91] H. Mintzberg, D. Raisinghani, and A. Theoret, “The Structure of ‘Unstructured’ Decision Processes,” *Adm. Sci. Q.*, vol. 21, no. 2, p. 246, 1976.
[92] I. L. Janis, “Groupthink,” *Psychology Today Magazine*, pp. 84–90, 1971.
[93] D. Kahneman, P. Slovic, and A. Tversky, “Judgment under uncertainty: Heuristics and Biases Amos,” *Sci. new Ser.*, vol. 185, no. 4157, pp. 1124–1131, 1974.
[94] G. DeSanctis and R. B. Gallupe, “Foundation for the Study of Group Decision Support Systems,” *Manage. Sci.*, 1987.
[95] Z. PÓLKOWSKI, “Online Gdss With the Ahp Method To Facilitate Decision Making.,” *Online Gdss Z Metod. Ahp Dla Ułatwienia Pod. Decyz.*, no. 72, pp. 50–61, 2014.
[96] K. D. Goepel, “Implementation of an Online Software Tool for the Analytic Hierarchy Process – Challenges and Practical Experiences,” pp. 1–20, 2017.
[97] K. D. Goepel, “Implementing the Analytic Hierarchy Process as a Standard Method for Multi-Criteria Decision Making in Corporate Enterprises – A New AHP Excel Template with Multiple Inputs,” *Proc. Int. Symp. Anal. Hierarchy Process*, vol. 2, no. 10, pp. 1–10, 2013.
[98] C. Glur, “GitHub - gluc/ahp: Analytical Hierarchy Process (AHP) with R.” [Online]. Available: https://github.com/gluc/ahp. [Accessed: 21-Mar-2019].
[99] C. Glur, “AHP File Format.” [Online]. Available: https://cran.r-project.org/web/packages/ahp/vignettes/file-format.html. [Accessed: 21-Mar-2019].
[100] L. Chen, M. A. Babar, and B. Nuseibeh, “Characterizing architecturally significant requirements,” *IEEE Softw.*, vol. 30, no. 2, pp. 38–45, 2013.
[101] R. B. Grady and R. B., *Practical software metrics for project management and process improvement*. Prentice Hall, 1992.
[102] “ISO - ISO/IEC 25010:2011 - Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.” [Online]. Available: https://www.iso.org/standard/35733.html. [Accessed: 17-Oct-2019].
[103] A. R. Hevner, S. T. March, J. Park, S. Ram, and S. Ram, “Design Science in Information Systems Research,” *MIS Q.*, vol. 28, no. 1, pp. 75–105, 2004.
[104] S. Gregor and A. R. Hevner, “Positioning and presenting design science research for maximum impact,” *MIS Q. Manag. Inf. Syst.*, vol. 37, no. 2, pp. 337–355, 2013.
[105] R. K. Yin, *Case Study Research Design and Methods, Third Edition, Applied Social Research Methods Series, Vol 5 2002*. 2003.
[106] F. Wijnhoven, *Managing dynamic organizational memories: Instruments for knowledge management*. 1999.

APPENDICES

APPENDIX A – SELECTED PUBLICATIONS

ID	Publication
1	M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, “Performance evaluation of microservices architectures using containers” in Proceedings - 2015 IEEE 14th International Symposium on Network Computing and Applications, NCA 2015, pp. 27–34, 2016.
2	A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture” 2016.
3	A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices Migration Patterns” no. 1, pp. 1–21, 2015.
4	I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, “Serverless computing: Current trends and open problems” Research Advances in Cloud Computing, pp. 1–20, 2017.
5	L. Baresi, M. Garriga, and A. De Renzis, “Microservices identification through interface analysis” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (J. E. B. Schulte S. De Paoli F., ed.), vol. 10465 LNCS, pp. 19–33, Springer Verlag, 2017.
6	D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, “Multi-objective scheduling of micro-services for optimal service function chains” in IEEE International Conference on Communications, 2017.
7	N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: yesterday, today, and tomorrow” pp. 1–17, 2016.
8	F. Dudouet, A. Edmonds, and M. Erne, “Reliable cloud-applications” in Proceedings of the 1st International Workshop on Automated Incident Management in Cloud - AIMC ’15, pp. 1–6, 2015.
9	C. Esposito, A. Castiglione, and K. K. R. Choo, “Challenges in Delivering Software in the Cloud as Microservices” IEEE Cloud Computing, vol. 3, no. 5, pp. 10–14, 2016.
10	M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, “Open Issues in Scheduling Microservices in the Cloud” IEEE Cloud Computing, vol. 3, no. 5, pp. 81–88, 2016.
11	A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, “Migrating Enterprise Legacy Source Code to Microservices: On Multi-Tenancy, Statefulness and Data Consistency” IEEE Software, pp. 1–1, dec 2017.
12	M. Garriga, “Towards a Taxonomy of Microservices Architectures” in SEFM 2017 Workshops, vol. 9763, pp. 203–218, 2016.
13	B. Götz, D. Schel, D. Bauer, C. Henkel, P. Einberger, and T. Bauernhansl, “Challenges of Production Microservices” Procedia CIRP, vol. 67, pp. 167–172, 2018.
14	J. P. Gouigoux and D. Tamzalit, “From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture” in Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings, pp. 62–65, 2017.
15	C. Guerrero, I. Lera, and C. Juiz, “Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications” The Journal of Supercomputing, pp. 1–28, 2018.
16	S. Haselböck, R. Weinreich, and G. Buchgeher, “Decision Guidance Models for Microservices Requirements and Use Cases” Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems - ECBS ’17, pp. 1–10, 2017.
17	S. Hassan and R. Bahsoon, “Microservices and their design trade-offs: A self-adaptive roadmap” in Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016, pp. 813–818, 2016.
18	V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. Reiter, and V. Sekar, “Gremlin: Systematic Resilience Testing of Microservices” in Proceedings - International Conference on Distributed Computing Systems, vol. 2016-Augus, pp. 57–66, 2016.
19	M. Kalske, N. Mäkitalo, and T. Mikkonen, “Challenges When Moving from Monolith to Microservice Architecture” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7703, pp. 189–203, 2012.
20	H. Kang, M. Le, and S. Tao, “Container and microservice driven design for cloud infrastructure DevOps” in Proceedings - 2016 IEEE International Conference on Cloud Engineering, 1 2 IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016, pp. 202–211, 2016.
21	N. Kratzke and P. C. Quint, “Investigation of impacts on network performance in the advance of a microservice design” in Communications in Computer and Information Science (C. J. M. M. V. Ferguson D. Helfert M., ed.), vol. 740, pp. 187–208, Springer Verlag, 2017.
22	K. B. Long, H. Yang, and Y. Kim, “ICN-based service discovery mechanism for microservice architecture” in International Conference on Ubiquitous and Future Networks, ICUFN, pp. 773–775, IEEE Computer Society, jul 2017.
23	G. Mazlami, J. Cito, and P. Leitner, “Extraction of Microservices from Monolithic Software Architectures” in Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017, pp. 524–531, 2017.
24	D. Namiot and M. Sneps-Sneppe, “On Micro-services Architecture” 2014.
25	C. Pahl and P. Jamshidi, “Software architecture for the cloud A roadmap towards controltheoretic, model-based cloud architecture” in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9278, pp. 212–220, 2015.

26	C. Pahl, P. Jamshidi, and O. Zimmermann, “Architectural Principles for Cloud Software” ACM Transactions on Internet Technology, vol. 18, no. 2, pp. 1–23, 2018.
27	C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, “Microservices in Practice, Part 1: Reality Check and Service Design” IEEE Software, vol. 34, no. 1, pp. 91–98, 2017.
28	C. Rotter, J. Illes, G. Nyiri, L. Farkas, G. Csatari, and G. Huszty, “Telecom strategies for service discovery in microservice environments” in Proceedings of the 2017 20th Conference on Innovations in Clouds, Internet and Networks, ICIN 2017, pp. 214–218, 2017.
29	A. R. Sampaio, H. Kadiyala, B. Hu, J. Steinbacher, T. Erwin, N. Rosa, I. Beschastnikh, and J. Rubin, “Supporting microservice evolution” in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 539–543, Institute of Electrical and Electronics Engineers Inc., nov 2017.
30	D. Savchenko and G. Radchenko, “Microservices validation: Methodology and implementation” in CEUR Workshop Proceedings, vol. 1513, pp. 21–28, 2015.
31	J. Stubbs, W. Moreira, and R. Dooley, “Distributed Systems of Microservices Using Docker and Serfnode” in Proceedings - 7th International Workshop on Science Gateways, IWSG 2015, pp. 34–39, 2015.
32	Y. Sun, S. Nanda, and T. Jaeger, “Security-as-a-service for microservices-based cloud applications” in Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015, pp. 50–57, 2016.
33	D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation” IEEE Cloud Computing, vol. 4, pp. 22–32, sep 2017.
34	J. Thönes, “Microservices” 2015.
35	G. Toffetti, S. Brunner, M. Bl, J. Spillner, and T. M. Bohnert, “Self-managing cloud-native applications : design , implementation , and experience” Future Generation Computer Systems, vol. 72, pp. 165–179, 2016.
36	T. Ueda, T. Nakaike, and M. Ohara, “Workload Characterization for Microservices Limited Distribution Notice Workload Characterization for Microservices” Proceedings of the 2016 IEEE International Symposium on Workload Characterization, IISWC 2016, pp. 85–94, 2016.
37	M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud” in 2015 10th Colombian Computing Conference, 10CCC 2015, pp. 583–590, 2015.
38	M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, “Infrastructure Cost Comparison of Running Web 3 Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures” in Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016, pp. 179–182, 2016.
39	Y. Yu, H. Silveira, M. Sundaram, Y. Yale, H. Silveira, and M. Sundaram, “A microservice based reference architecture model in the context of enterprise architecture” 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), pp. 1856–1860, 2016.
40	O. Zimmermann, “Microservices tenets: Agile approach to service development and deployment” Computer Science - Research and Development, vol. 32, no. 3-4, pp. 301–310, 2017

APPENDIX B – LITERATURE REVIEW DIAGRAMS

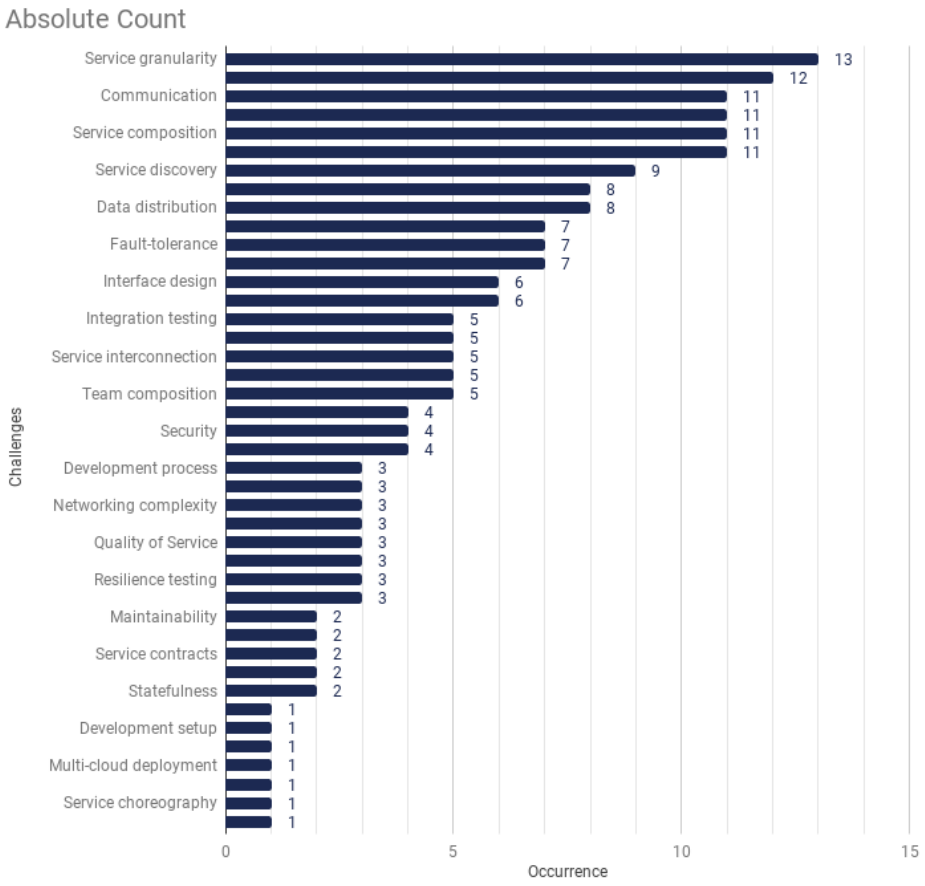


Figure 44 - Absolute Occurrence of Challenge Keywords

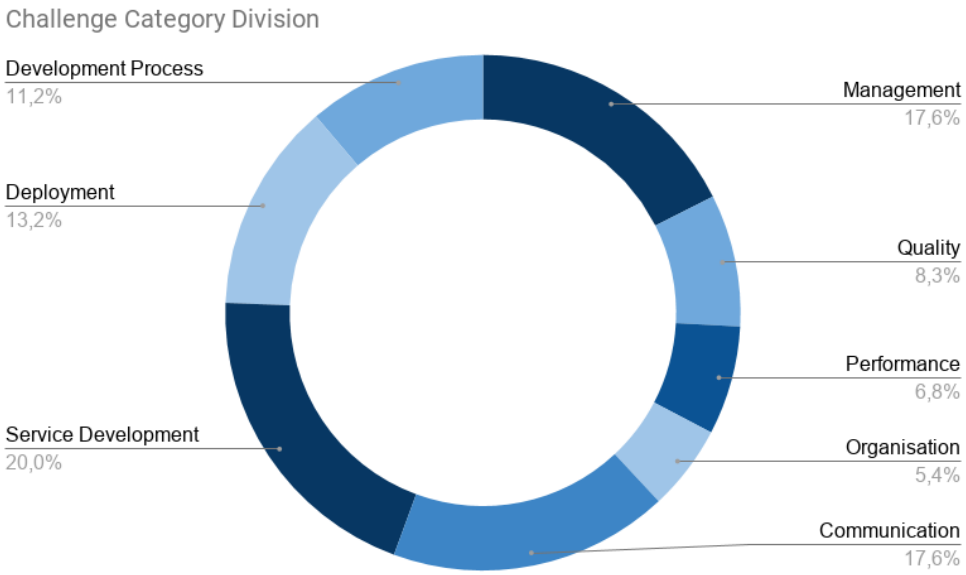
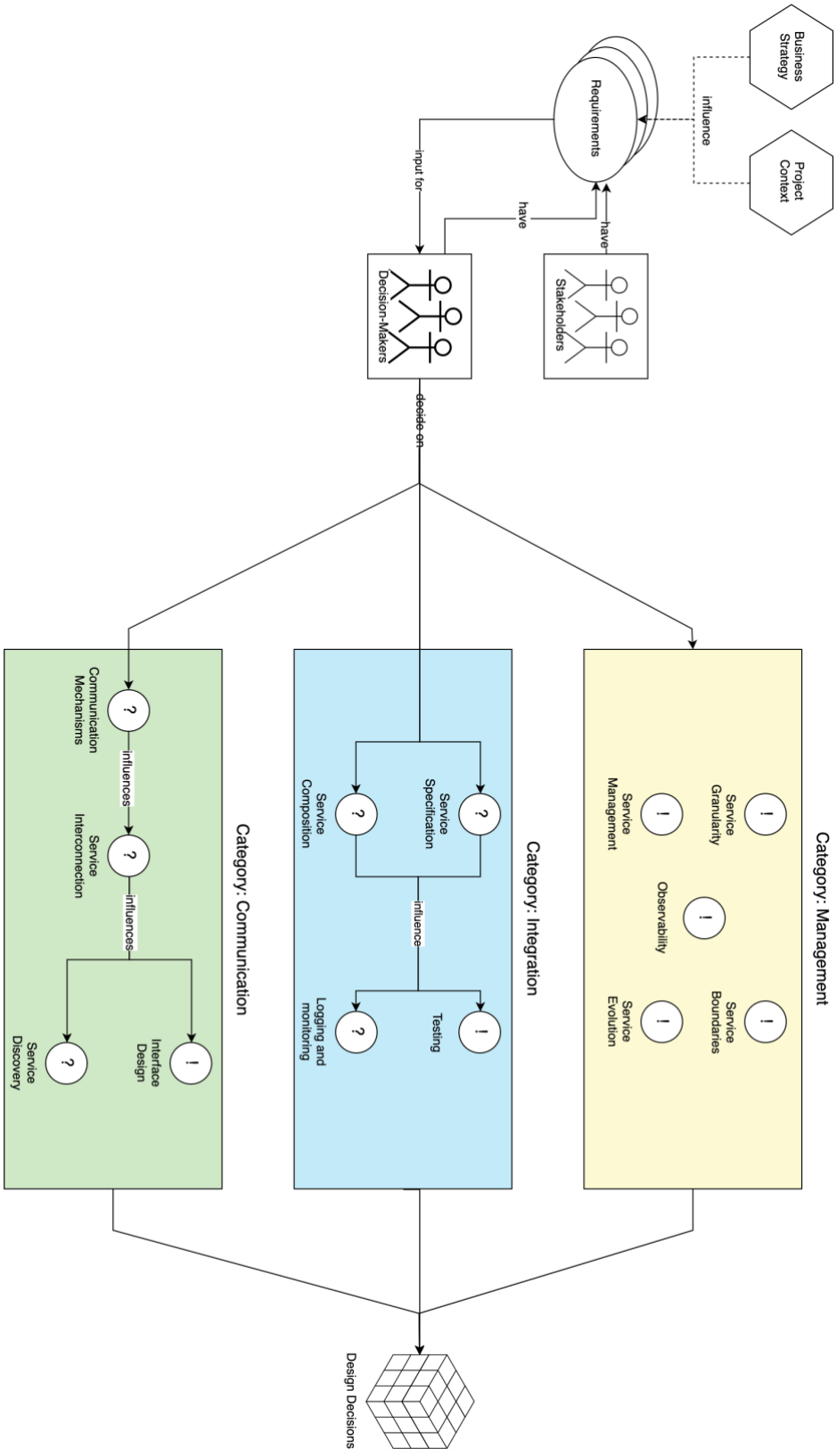


Figure 45 - Division of Challenge Categories based on Total Count

APPENDIX C – DECISION-MAKING MODEL



APPENDIX D – CASE STUDY QUESTIONNAIRE

Usefulness

Scale: extremely unlikely, quite unlikely, somewhat unlikely, neither, somewhat likely, quite likely, extremely likely; coded -3 through 3

- **USEFUL_1:** Using the framework in my job would enable me to accomplish tasks more quickly
- **USEFUL_2:** Using the framework would improve my job performance
- **USEFUL_3:** Using the framework in my job would increase my productivity
- **USEFUL_4:** Using the framework would enhance my effectiveness on the job
- **USEFUL_5:** Using the framework would make it easier to do my job
- **USEFUL_6:** I would find the framework useful in my job

Usability

Scale: extremely unlikely, quite unlikely, somewhat unlikely, neither, somewhat likely, quite likely, extremely likely; coded -3 through 3

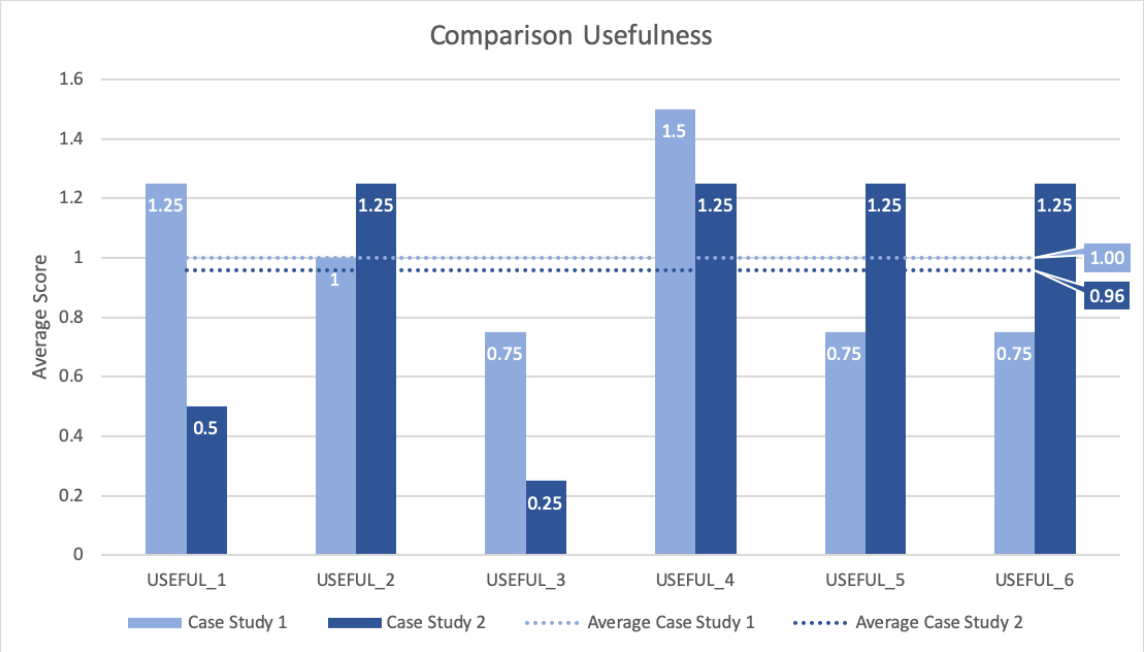
- **USABLE_1:** Learning to operate the framework would be easy for me
- **USABLE_2:** I would find it easy to get the framework to do what I want it to do
- **USABLE_3:** My interaction with the framework would be clear and understandable
- **USABLE_4:** I would find the framework to be flexible to interact with
- **USABLE_5:** It would be easy for me to become skilful at using the framework
- **USABLE_6:** I would find the framework easy to use

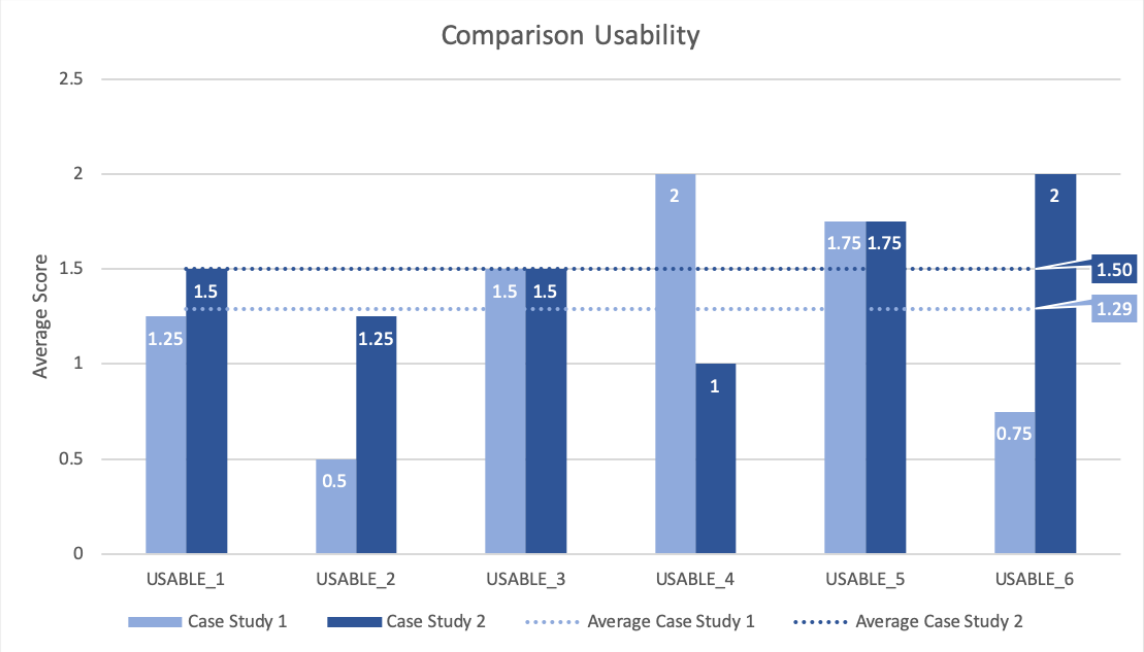
Decision Quality

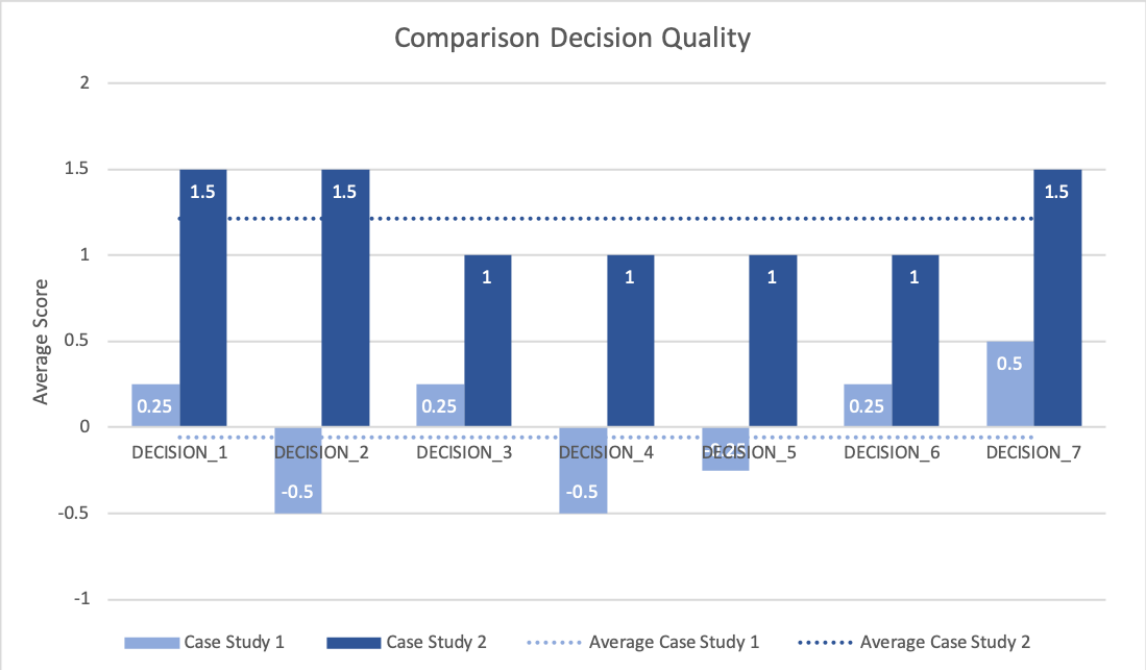
Scale: -3 - fully disagree - through 3 - fully agree

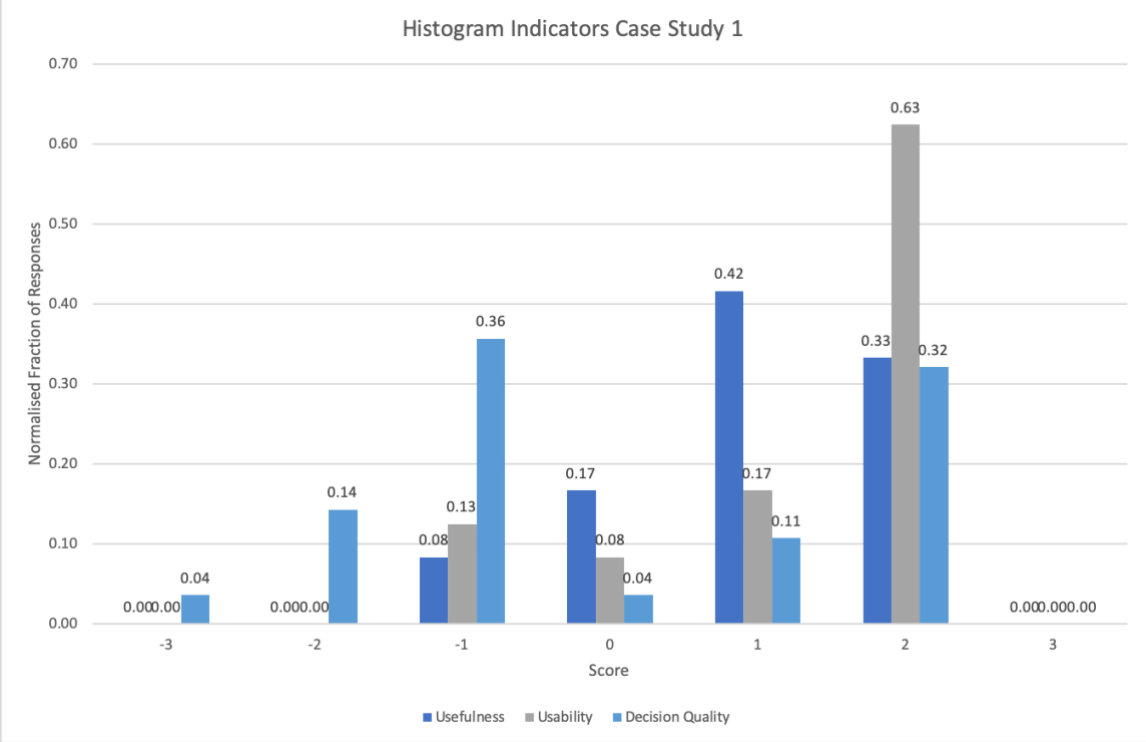
- **DECISION_1:** When making decisions, their purpose, perspective and scope were clear to me
- **DECISION_2:** When making decisions, I had appropriate and feasible options to choose between
- **DECISION_3:** When making decisions, I had enough information to inform the decision
- **DECISION_4:** When making decisions, I the values and trade-offs between options were clear to me
- **DECISION_5:** I feel that using the framework has given me more insight in the decision problems
- **DECISION_6:** I feel that relevant stakeholders could commit to the decisions made using the framework
- **DECISION_7:** I feel confident about the decision outcomes
- **DECISION_8:** I believe that the decision outcomes are easy to put into practice
- **DECISION_9:** I believe that the framework is useful for supporting decision-making conversations

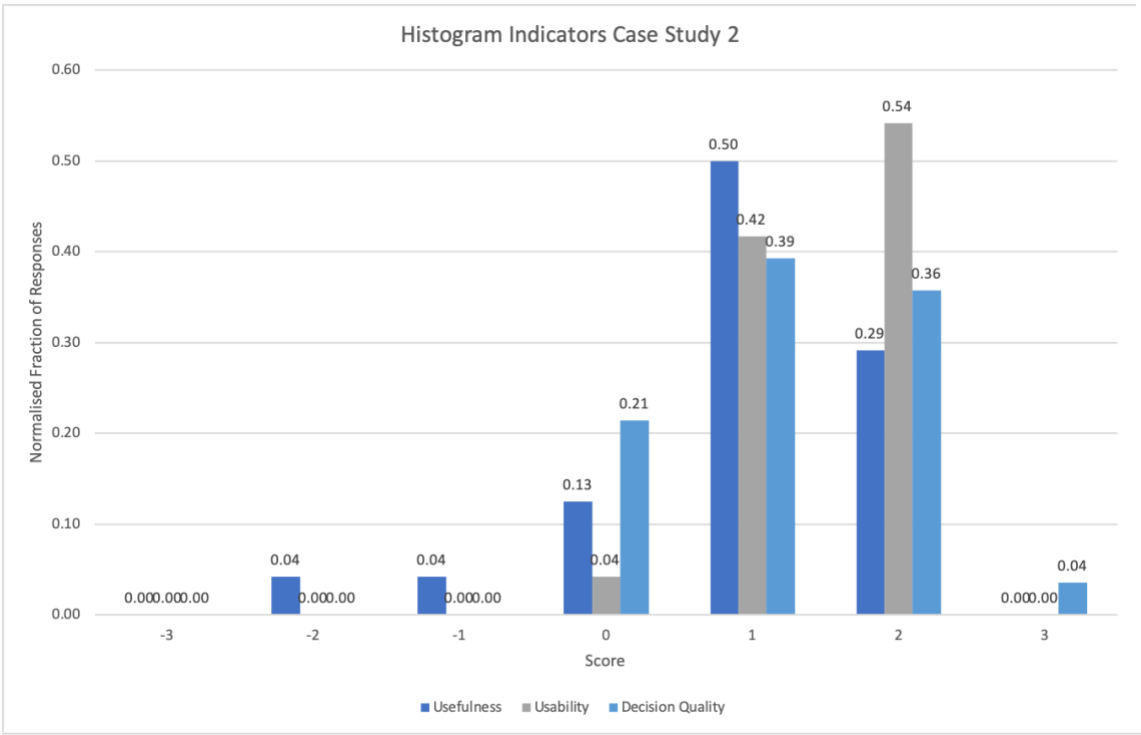
APPENDIX E – CASE STUDY SURVEY OUTCOME COMPARISONS

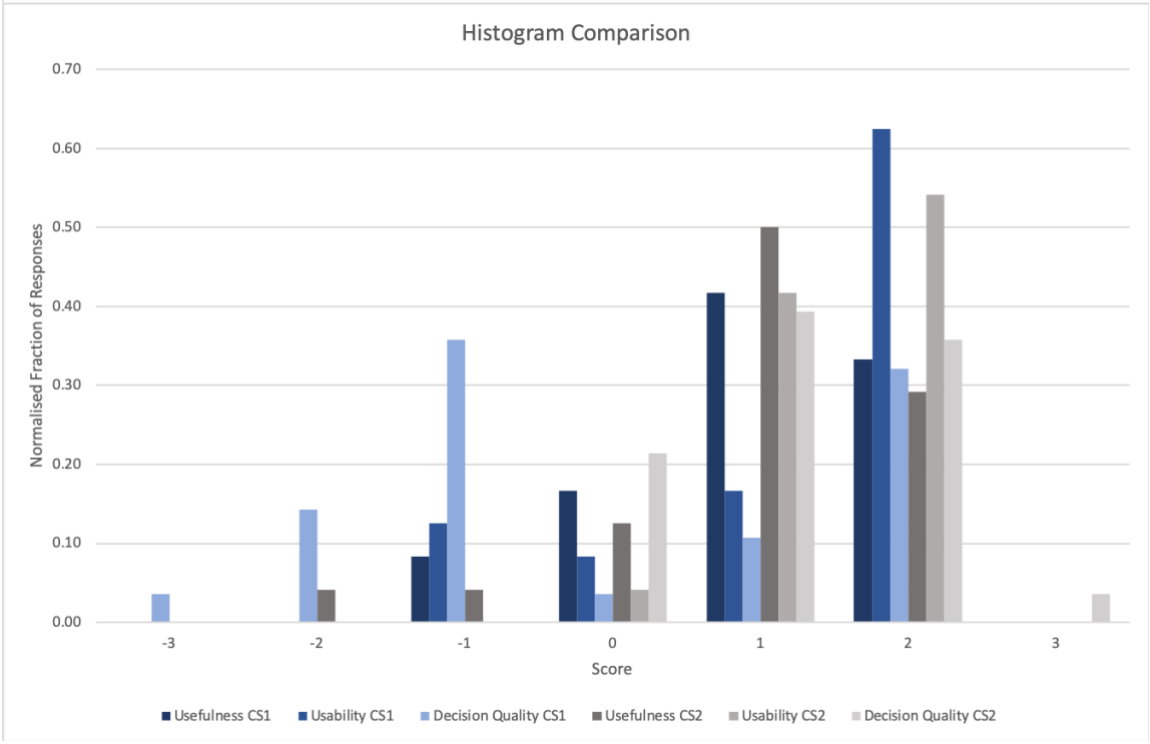





Comparison Decision Quality
Average Score
2
1.5
1
0.5
0
-0.5
-1
1.5
1.5
1
1
1
1
1.5
0.25
0.25
0.25
0.5
-0.5
-0.5
DECISION_1
DECISION_2
DECISION_3
DECISION_4
DECISION_5
DECISION_6
DECISION_7
Case Study 1
Case Study 2
Average Case Study 1
Average Case Study 2


Histogram Indicators Case Study 1
Normalised Fraction of Responses
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
0.04
0.14
0.36
0.13
0.08
0.17
0.08
0.04
0.42
0.17
0.11
0.63
0.33
0.32
0.00
0.00
0.00
0.00
0.00
0.00
0.00
-3
-2
-1
0
1
2
3
Score
Usefulness
Usability
Decision Quality


Histogram Indicators Case Study 2
Normalised Fraction of Responses
0.60
0.50
0.40
0.30
0.20
0.10
0.00
0.00 0.00 0.00
0.04
0.00 0.00
0.04
0.00 0.00
0.13
0.04
0.21
0.50
0.42
0.39
0.29
0.54
0.36
0.00 0.00
0.04
-3
-2
-1
0
1
2
3
Score
Usefulness
Usability
Decision Quality


Histogram Comparison
Normalised Fraction of Responses
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
-3
-2
-1
0
1
2
3
Score
Usefulness CS1
Usability CS1
Decision Quality CS1
Usefulness CS2
Usability CS2
Decision Quality CS2