MICROSERVICE REFERENCE ARCHITECTURE FOR DERIVING APPLICATION ARCHITECTURES

UYGULAMA MİMARİLERİ ELDE ETMEK İÇİN MİKROSERVİS REFERANS MİMARİSİ

MEHMET SÖYLEMEZ

ASSOC. PROF. DR. AYÇA KOLUKISA TARHAN

Supervisor

Submitted to

Graduate School of Science and Engineering of Hacettepe University

as a Partial Fulfillment to the Requirements

for the Award of the Degree of Doctor of Philosophy

in Computer Engineering

2023

ABSTRACT

MICROSERVICE REFERENCE ARCHITECTURE FOR DERIVING APPLICATION ARCHITECTURES

MEHMET SÖYLEMEZ

Doctor of Philosopy, Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ayça KOLUKISA TARHAN

Co-Supervisor: Prof. Dr. Bedir TEKİNERDOĞAN

January 2023, 161 pages

Microservice architecture (MSA) is an architectural style for distributed software systems, which promotes the use of fine-grained services with their own lifecycles. Several benefits of MSA have been reported in the literature, including increased autonomy and modularity, flexible configuration, easier development, easier maintenance, and increased productivity. Therefore, many practitioners leverage this architectural style either to break their existing big monolithic applications into small pieces or to start their new projects, in order to level up the agility of the development process and increase the autonomy of services. On the other hand, there are many concerns that the practitioners have to deal with, due to MSA's distributed nature and design principles to consider. Therefore, it is still challenging for the practitioners to handle these concerns and come up with application architecture, and unfortunately, there is no comprehensive study yet to address this issue. To fill this gap, in this thesis, we propose a novel reference architecture together with an approach to derive an application architecture from it, as the keys to successfully building microservice-based applications. To this end, we first identify what kind of challenges are there in MSA adoption and then we follow a domain-driven software architecture design approach to identify basic features of MSA. We provide a domain model by using feature diagrams including the common and variant features of MSA. Leveraging the challenges and family feature model of MSA, we apply the architecture design process to design the reference architecture by using architectural viewpoints. Finally, after designing the reference architecture, we carry out a multiple case study to evaluate the proposed reference architecture.

Keywords: Microservice architecture, reference architecture, software architecture, application architecture, architecture adoption, case study

ÖZET

UYGULAMA MİMARİLERİ TÜRETMEK İÇİN MİKROSERVİS REFERANS MİMARİSİ

Mehmet SÖYLEMEZ

Doktora, BİLGİSAYAR MÜHENDİSLİĞİ Bölümü

Tez Danışmanı: Doç. Dr. Ayça KOLUKISA TARHAN

Eş Danışman: Prof. Dr. Bedir TEKİNERDOĞAN

Ocak 2023, 161 sayfa

Mikro hizmet mimarisi (MHM), kendi yaşam döngüleriyle birlikte küçük boyuttaki hizmetlerin kullanımını destekleyen, dağıtılmış yazılım sistemleri için bir mimari stildir. Literatürde MHM'nin artan özerklik ve modülerlik, esnek yapılandırma, daha kolay geliştirme, daha kolay bakım ve artan üretkenlik dâhil olmak üzere çeşitli faydaları bildirilmiştir. Bu nedenle, birçok uygulayıcı, geliştirme sürecinin çevikliğini yükseltmek ve hizmetlerin özerkliğini artırmak amacıyla mevcut büyük monolitik uygulamalarını küçük parçalara ayırmak veya yeni projelerine başlamak için bu mimari stili kullanır. Öte yandan, MHM'nin dağıtık yapısı ve dikkate alınması gereken tasarım ilkeleri nedeniyle uygulayıcıların ele alması gereken birçok ilgi vardır. Uygulayıcılar için bu ilgileri ele almak ve bir uygulama mimarisi oluşturmak hâlihazırda zordur ve ne yazık ki, henüz bu konuyu adresleyen kapsamlı bir çalışma literatürde yer almamaktadır. Bu boşluğu doldurmak için, bu tezde, mikro hizmet tabanlı uygulamaları başarılı bir şekilde oluşturmanın anahtarı olarak, kapsamlı bir referans mimari ve ondan bir uygulama mimarisi türetmek için bir yaklaşım öneriyoruz. Bu amaçla, önce MHM'nin benimsenmesinde ne tür zorlukların olduğunu tespit ediyoruz ve ardından MHM'nin temel özelliklerini belirlemek için etki alanına dayalı bir mimari tasarım yaklaşımı izliyoruz. MHM'nin ortak ve değişken özelliklerini içeren özellik diyagramlarını kullanarak bir etki alanı modeli sağlıyoruz ve ardından, MHM'nin zorluklarından ve aile özellik modelinden yararlanarak mimari bakış açıları tabanlı bir referans mimariyi tasarlamak için, mimari tasarım sürecini uyguluyoruz. Son olarak, referans mimarisini tasarladıktan sonra, önerilen referans mimarisini değerlendirmek için çoklu vaka çalışması yürütüyoruz.

Anahtar Kelimeler: Mikro hizmet mimarisi, referans mimari, yazılım mimarisi, uygulama mimarisi, mimari benimseme, durum çalışması

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my supervisor Assoc. Prof. Dr. Ayça Kolukısa Tarhan and my co-supervisor Prof. Dr. Bedir Tekinerdoğan for their inexhaustible support, patience, friendly encouragement and valuable guidance.

I am grateful to the thesis supervising committee members Prof. Dr. Ali Doğru and Assoc. Prof. Dr. M. Kayhan İmre for their advices and helpful discussions throughout my thesis study. I also thank Assoc. Prof. Dr. Altan Koçyiğit and Assist. Prof. Dr. Murat Aydos for their review and suggestions during my thesis defense.

I would like to thank to the companies, where I conducted case studies, for their contributions and support.

I would like to express my deep gratitude to my father and mother. I have always felt to be privileged of having such a family.

I would like to give my special thanks to my beloved wife Seda for her endless love, support and encouragement throughout my life. Last but not least, I would like to express my huge thanks to my beloved son, Deniz and my beloved daughter, İpek for being such a good kids always cheering me up.

January 2023, Ankara

TABLE OF CONTENTS

ABSTRACT i

ÖZET iii

ACKNOWLEDGMENTS v

TABLE OF CONTENTS vi

LIST OF FIGURES ix

LIST OF TABLES xi

LIST OF ABBREVIATIONS xii

1. INTRODUCTION 1

2. BACKGROUND 5

- 2.1. Microservice Architecture 5
- 2.2. Related Work 8
- 2.3. Arcitectural Views 9
- 2.4. Domain Analysis 10
- 2.5. Domain Driven Design (DDD) 11

3. CHALLENGES AND SOLUTION DIRECTIONS OF MSA 12

- 3.1. Research Methodology 12
- 3.2. Overview of Selected Studies 15
- 3.3. Identified Challenges and Solution Directions 18
  - 3.3.1. RQ1. What are the identified challenges in the MSA domain? 18
    - 3.3.1.1. Service Discovery 22
    - 3.3.1.2. Data Management and Consistency 22
    - 3.3.1.3. Testing 23
    - 3.3.1.4. Performance Prediction, Measurement and Optimization 24
    - 3.3.1.5. Communication and Integration 24
    - 3.3.1.6. Service Orchestration 25
    - 3.3.1.7. Security 26

3.3.1.8. Monitoring, Tracing and Logging (MTL) 27
3.3.1.9. Decomposition 27
3.3.2. RQ2. What are the identified solution directions? 28
3.3.2.1. Service Discovery 30
3.3.2.2. Data Management and Consistency 31
3.3.2.3. Testing 32
3.3.2.4. Performance Prediction, Measurement and Optimization 34
3.3.2.5. Communication and Integration 36
3.3.2.6. Service Orchestration 37
3.3.2.7. Security 46
3.3.2.8. Monitoring, Tracing and Logging 48
3.3.2.9. Decomposition 53
3.4. Summary 54
4. METHODOLOGY 56
5. FEATURE DRIVEN CHARACTERIZATION OF MSA 58
5.1. Research Methodology 58
5.2. Characterization Framework 60
5.2.1. Data Management and Consistency 62
5.2.2. Communication Style 64
5.2.3. Service Orchestration 65
5.2.4. Decomposition 67
5.2.5. Service Mesh and Sidecar Pattern 68
5.2.6. Observability 68
5.2.7. Provisioning and Configuration Management 69
5.2.8. Security 70
5.2.9. Testing 71
5.2.10. Resilience and Fault Tolerance 72
5.3. Survey of MSA 73
5.4. Analysis of Existing Key Cloud Providers 76
6. REFERENCE ARCHITECTURE 79
6.1. Family Feature Model 80

6.2. Decomposition View 82
6.3. Layered View 83
6.4. Deployment And Service Oriented Architecture View (Combined View) 84
6.5. Method To Derive Application Architecture From Reference Architecture 88

7. VALIDATION BY MULTI-CASE STUDY 90

7.1. Case Study Protocol 90
7.2. Case Study – Transportation Management System 92
7.2.1. Application Feature Model.. 93
7.2.2. Decomposition View.. 94
7.2.3. Layered View.. 96
7.2.4. Deployment & Service Oriented Architecture View 96
7.3. Case Study – Remote Team Management System........................ 97
7.3.1. Application Feature Model.. 98
7.3.2. Decomposition View.. 99
7.3.3. Layered View.. 100
7.3.4. Deployment & Service Oriented Architecture View 101
7.4. Discussion On Results.. 101
7.4.1. How Effective Is The Adopted Microservice Reference Architecture? 101
7.4.2. How Practical Is The Adopted Microservice Reference Architecture? 103
7.5. Limitations And Threats To Validity 104

8. CONCLUSION.. 107

9. BIBLIOGRAPHY.. 111

APPENDIX 118

APPENDIX 1 – Study Quality Assesment Checklist.............................. 118
APPENDIX 2 – List of Primary Studies.. 121
APPENDIX 3 – Related Publications – Journal Articles 131

RESUME 133

LIST OF FIGURES

Figure 2.1. Reference model for MSA as adapted from [28] 7
Figure 3.1. Activities under the SLR protocol.. 12
Figure 3.2. Year-wise distribution of the number of primary studies............................ 15
Figure 3.3 Quality metric results for the primary studies 17
Figure 3.4. Visual Summary of Identified Problems 21
Figure 4.1. Development Methodology of Microservice Reference Architecture 56
Figure 5.1. Research methodology 59
Figure 5.2. Top-level feature diagram of MSA. 60
Figure 5.3. Feature diagram of data management and consistency 63
Figure 5.4. Feature diagram of communication style 65
Figure 5.5. Feature diagram of service orchestration 66
Figure 5.6. Feature diagram of decomposition 68
Figure 5.7. Feature diagram of observability 69
Figure 5.9. Feature diagram of security 70
Figure 5.10. Feature diagram of testing 72
Figure 5.11. Feature diagram of resilience and fault tolerance 72
Figure 6.1. Microservice Reference Architecture – Decomposition View..................... 83
Figure 6.2. Microservice Reference Architecture – Layered View 84
Figure 6.3. Microservice Reference Architecture – Deployment & Service Oriented Architecture View 87
Figure 6.4. Method to derive application architecture from reference architecture 88
Figure 6.5. Three scenarios of using reference architecture to derive an application architecture adopted from Kassahun [66] 89
Figure 7.1. Decomposition view for Transportation Management System 95
Figure 7.2. Layered view for Transportation Management System 96
Figure 7.3. Deployment & Service Oriented Architecture view for Transportation Management System 97
Figure 7.4. Decomposition view for Remote Team Management System 100
Figure 7.5. Layered view for Remote Team Management System 100

Figure 7.6. Deployment & Service Oriented Architecture view for Remote Team Management System 101
Figure 7.7. Interview results for Transportation Management System.........................102
Figure 7.8. Interview results for Remote Team Management System..........................103

LIST OF TABLES

Table 3.1. Publication Sources Searched 13
Table 3.2. Quality Assessment Checklist 14
Table 3.3. Studies by research methods 16
Table 3.4. Primary Studies with Identified Problems of MSA 18
Table 3.5. Solution Directions for the Identified Challenges in MSA 28
Table 5.2. Mapping features with MSA technologies 74
Table 5.3. Feature-based service comparison: AWS, Google Cloud, and Microsoft Azure 77
Table 6.3 Explanation of each component in Deployment and Service Oriented Architecture View 85
Table 7.1. Case study design 90
Table 7.2. Questionnaire used for qualitative data analysis 91
Table 7.3. Application Feature Domain Model 93
Table 7.4. Application Feature Domain Model 98
Table 7.5. Threats to construct validity and countermeasures 106
Table 8.1. SWOT Analysis of our approach 109

LIST OF ABBREVIATIONS

2PC	Two Phase Commit
ABE	Attribute Based Encryption
ACID	Atomicity, Consistency, Isolation, Durability
ACO	Ant Colony Optimization
ACM	Association for Computing Machinery
ADAL	Accelerated Distributed Augmented Lagrangian
API	Application Programming Interface
AWS	Amazon Web Services
BAC	Backup, Availability, Consistency
BASE	Basically Available, Soft State, Eventually Consistent
BDD	Behavior Driven Development
BE	Backend
BFF	Backend for Frontend
C&C	Component & Connector
CAP	Consistency, Availability and Partition Tolerance
CI&CD	Continuous Integration & Continuous Delivery
CAS	Central Authentication Service
CQRS	Command and Query Responsibility Segregation
CPU	Central Processing Unit
DB	Database
DDD	Domain Driven Design
DevOps	Development and Operations
DPDK	Data Plane Development Kit
ECR	Energy Consumption Rate
ERP	Enterprise Resource Planniung
ESMS	Elastic Scheduling for Microservices
FaaS	Function as a Services
GMAT	Graph-based Microservice Analysis and Testing
HA	High Availability
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure

IBM	International Business Machine
ICN	Information Centric Networking
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IMAP	Internet Message Access Protocol
I/O	Input / Output
IP	Internet Protocol
IT	Information Technology
JWT	JSON Web Token
LXC	Linux Containers
LwIP	Lightweight IP
MAPE	Mean Absolute Percentage Error
MEC	Mobile Edge Computing
MFRL-CA	Method Based on Correlation Analysis
MicADO	Microservice Architecture Deployment Optimizer
MQ	Message Queue
MSA	Microservice Architecture
MTL	Monitoring Tracing Logging
MVC	Model View Controller
NFV	Network Function Virtualization
ORM	Object Relational Mapping
RBAC	Role Based Access Control
RCA	Root Cause Analysis
REST	REpresentational State Transfer
RQ	Research Questions
SLR	Systematic Literature Review
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SRE	Site Reliability Engineering
SSO	Single Sign-on
TCP	Transmission Transfer Protocol
TLS	Transport Layer Security
VM	Virtual Machine

SWOT	Strengths, Weaknesses, Opportunities, Threats

1. INTRODUCTION

Microservice Architecture (MSA) is an architectural style that encourages practitioners to develop loosely coupled and highly cohesive services [1]. Compared to other architectural styles such as Service Oriented Architecture (SOA) and Monolithic Architecture, it stands out by putting emphasis on autonomous services that basically allow practitioners to have independently deployable services [2]. Monolithic Architecture is another and the most traditional architectural style in which the application is built as a single operating unit [2]. Moreover, all the functionality in the codebase is deployed together [3]. MSA can be considered as a variant of the SOA, which includes a collection of loosely coupled services [4]. It adheres to the separation of concerns principle relying on clear boundaries of services [5]. However, SOA depends on heavy-weight middleware such as enterprise service buses and it breaks the autonomy of services [6].

In MSA, services are small in size, autonomously developed, independently deployable, and decentralized, while the protocols are lightweight. The following attributes are considered to be present in an MSA: (1) It lends itself to a software development methodology based on continuous delivery. This means that modifying a tiny portion of the application just necessitates rebuilding and redeploying one or a few services. (2) It follows business-driven development ideas such as fine-grained interfaces (to create independently deployable services) and Domain-Driven Design (DDD). The idea is for teams to be able to bring their services to life without relying on others. Because service developers do not need to care about the service's users and do not push their modifications on them, loose coupling lowers all forms of dependencies and complications. Accordingly, MSA gives importance to autonomous and lightweight services [6]. MSA has been in more demand because it minimizes the disadvantages that come with SOA. MSA can be deployed, developed, tested, and operated independently.

It is crucial to design microservices as fine-grained services [2,4,7] that should adhere to single responsibility principle by encapsulating their data. Furthermore, well-defined abstraction and interfaces should be used in communicating with other services. Having fine-grained services allows software agility to go up with independent development, deployment, versioning, and scaling [6]. Decomposing domain into services organized around business capability is the most popular way to obtain fine-grained services [2]. There are also some patterns to decompose the domain systematically. Domain-Driven Design (DDD) pattern could be used to define bounded context and domain models in order to decompose domain by building around business capability [8,9].

Autonomous services bring many benefits such as continuous delivery, being independently deployable, and improved scalability [2]. MSA adopts automation in continuous delivery as well as testing and deployment. These are the key benefits of MSA [6]. Besides, as a result of its lightweight nature, containerization and communication become also lightweight. Accordingly, it is easy to manage changes and extend the system according to new coming requirements. Furthermore, practitioners take an advantage of the microservices to be able to develop in different programming languages and technologies. Therefore, there exists a freedom to use the most appropriate technology or language to satisfy the user's needs [10].

Having improved scalability is another ability to scale autonomous services independently. According to the growing amount of work, system should handle this work by adding resources to the system [11]. Apart from that, it is expected that availability and reliability will be improved thanks to the autonomous nature of MSA. Moreover, application architecture is expected to be designed to adopt failure isolation and tolerance principles to meet availability and reliability requirements. These are the non-functional requirements that users can encounter while using the system, and they should be handled properly to ensure customer satisfaction.

MSA promises software development firms increased agility because it is more open to changing requirements and related use cases and technologies than monolithic applications [2]. Design, development, and infrastructure automation processes can be handled successfully with MSA. Infrastructure automation decreases manual effort in building, deploying and operating microservices. On the other hand, decentralized governance and data management allow services to be independent[2]. Thanks to important benefits, several important vendors such as Amazon, Netflix, LinkedIn, and Spotify have implemented their applications using MSA [2,12].

Despite the advantages mentioned above, there are also many concerns that practitioners have to deal with due to MSA's distributed nature and design principles to consider. Orchestration of microservices, defining optimal boundary of microservices, ensuring data consistency and distributed transaction management, versioning, and distributed tracing are the main challenges to address during development of MSA based applications [10,13–18]. After our comprehensive state-of-the-art survey [19] and systematic literature review [20], we reached the conclusion that it is still challenging for practitioners to handle these concerns, and to come up with an application architecture. Unfortunately, there is no comprehensive study yet to address this issue. In order to fill this gap, we consider that a comprehensive reference architecture is the key to successfully building microservice-based applications.

Therefore, this study aims to propose a reference architecture of microservices and a method to derive an application architecture from it. To this end, we adopt a domain-driven architecture design approach to identify basic features of MSA. As a result of this step, the domain model is provided by using feature diagrams including the common and variant features of MSA. After that, an architecture design process is applied to design the reference architecture by using architectural viewpoints. Finally, multiple case study is conducted to assess the proposed reference architecture after it has been designed. As a result, the following are the study's contributions:

- A domain-driven architectural design approach is described, and it is utilized to create a microservice reference architecture.
- A reference architecture is created using the architecture design approach to derive microservice application architecture.
- An industry multiple-case study is used to validate both the technique and the reference architecture.

The rest of this thesis is organized as follows. Section 2 presents the background including a detailed explanation of MSA, related work, and architecture design alternatives. Section 3 explains the research methodology. Sections 4 and 5 respectively presents our Systematic Literature Review (SLR) study and Feature Characterization Framework of MSA study. Section 6 explains development method of Microservice Reference Architecture. The results of multiple case study are explained in Section 7. Finally, Section 8 concludes the thesis.

2. BACKGROUND

2.1. Microservice Architecture

MSA was firstly described by Lewis and Fowler in their famous article [2]. It has managed to get a lot of attention in the academy, mostly in the industry from that day on. Lewis and Fowler defined MSA as “an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often a HyperText Transfer Protocol (HTTP) resource Application Programming Interface (API)”. This is not the only definition of MSA. For example, Newman defines microservices as “small autonomous services that work together, modeled around business domain” [1]. Even though there are many definitions of microservices, all of them emphasize their small nature and autonomy.

The main reason why MSA is popular and used by many companies and individual practitioners is to accelerate the software development process. This is the expected result of its autonomous nature. The autonomy of the service emerges from the decomposition of the complex domain into smaller subdomains and components, which can be developed, deployed, versioned, tested, and managed independently [21]. With MSA, it is expected to have high cohesive and loosely coupled services by applying the separation of concerns principle. At this point, it is very important to determine the boundaries of each microservice. Microservices are organized around business capabilities, which simplifies identifying service boundaries. The most famous approach of identifying boundaries is Business-Driven Development. It is an approach to identify bounded context and to decompose a domain into subdomains [8,9].

MSA also provides great benefits in adapting to new requirements and change management. In Monolithic Architecture, changes require the whole system to be rebuilt and completely redeployed while with MSA, only the affected services are rebuilt and deployed independently. Microservices are highly maintainable and testable [22]. This gives great agility to a software system. It is easier to change the business direction according to customer needs. It also allows practitioners to select the most appropriate technology for the customer’s needs. It eliminates the long-term commitments to the technology stack since each microservice can be implemented in a different technology stack that best fits its functional and non-functional requirements.

Another advantage of MSA is providing an infrastructure where microservices, as the units of scaling, have the ability to be scaled independently according to their requirements and the interests that they face [6]. Scalability is not only adapting a growing amount of work by adding some resources but also operating the system efficiently while preserving the quality [23,24]. Besides, practitioners can make decisions about each service independently. In other words, different scaling policies can be applied to microservices depending on their runtime metrics and states. All these abilities enable the applications to be highly scalable and available.

The fact that MSA includes loosely-coupled services allows the system to be more fault tolerant [25]. Generally, specific and relevant services are affected by a failure and only those services need to be rebuilt and deployed. This situation prevents the whole system from being unavailable. Accordingly, we have a more reliable architecture against any failure.

Many companies, such as Amazon, Netflix, LinkedIn, and Spotify, have started to use MSA in their projects [2,12,26]. All of these companies follow the basic model for MSA, as shown Figure 2.1. This model is structured by some crucial building blocks, such as main business services, infrastructural services, discovery mechanisms, and communication infrastructure. Each block must be isolated from other blocks and communicate with them using a lightweight protocol. Therefore, it is easy for them to evolve over time according to the needs of the business or technology.

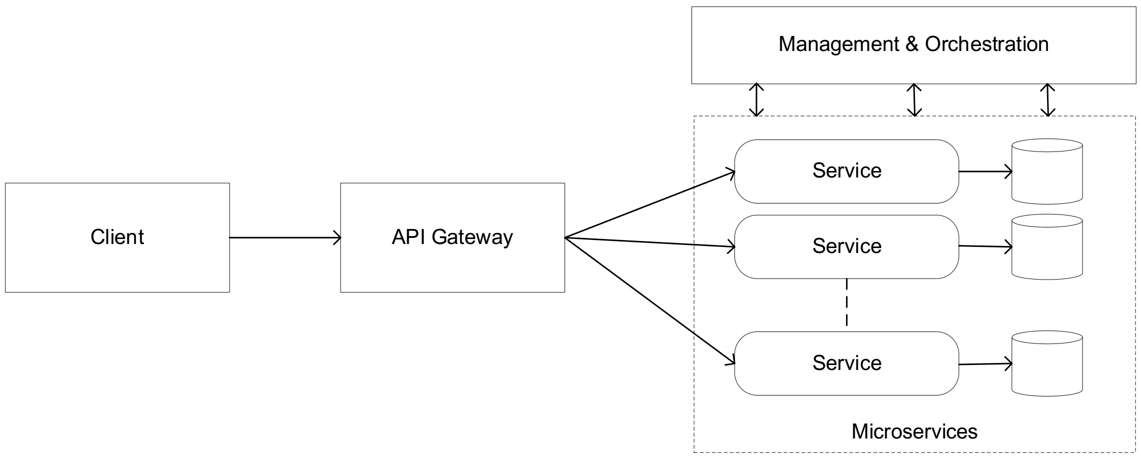


Figure 2.1. Reference model for MSA as adapted from [27]

Systems are always open to changes as scenarios evolve and requirements change. This is an expected behavior in software development process. However, managing these changes better has become much more sustainable and applicable with MSA. Since every microservice is a small business process and represents a small aspect of business functionality, it is easy to adapt to new changes [28]. However, all these conveniences are the outcome of an evolutionary process. This process starts with determining the boundaries of microservices and shaping them around the business capability and continues with the creation of DevOps practices and the evolution of the organization accordingly. The next step is to have an elastic infrastructure and automate that infrastructure to the possible extent by creating Continuous Integration & Continuous Delivery (CI&CD) processes. With the automated infrastructure, many advanced deployment techniques can be used, and projects become ready for using MSA [29].

Despite the advantages listed above, it is still difficult for software teams to implement MSA in distributed projects, and for practitioners to guide teams to successful MSA adoptions. The notion of MSA is complicated in terms of distributed service, identification, management, and maintenance, which is one of the key reasons for its complexity. As a result, successful MSA adoption necessitates a thorough awareness of the issues and potential solutions.

2.2. Related Work

Few studies have addressed the architectural aspect of MSA-based development. Some come with a general reference architecture, while others come with an architecture that focuses on the specific aspects of MSA or the particular domains. In this section, we provide an overview of the current studies.

Yu et al. [30] discussed the key characteristics of MSA. They proposed a reference architecture with main building blocks and key components. They also emphasized some common issues and solution alternatives while building microservice-based applications, such as the uncertainty in business ownership and communication problems. However, they are a bit far from today's concerns and issues because with the increase in the usage rate of microservices, more diverse and more critical concerns have started to emerge. Aside from that, even though this study provides general guidance by involving many buildings blocks such as service API registry, API proxy, etc., it does not propose a systematic guidance to help practitioners choose the best-fit components for each concern while building microservices.

Baylov and Dimov [31] proposed a reference architecture for self-adaptive microservice systems. They focused on five basic components in their study. These were Service Consumer, Service Registry, Service Provider, Service Instance, and Adaptation Registry. Interactions of components with each other and the purpose for which they exist were also explained in the reference model. The authors also defined how the reference architecture they proposed could be used through an example application. However, this study is lacking sufficient guidance for MSA based application development as well as evidence for the verification of the reference architecture.

Aksakalli et al. [32] have proposed a model-driven architecture that offers automated deployment alternatives for MSA based systems. The purpose of the architecture is to minimize the execution cost and also the communication cost between microservices, and to use cloud resources efficiently. The architecture consists of five main components. These are microservice data exchange metamodel, microservice definition and communication metamodel, microservice infrastructure metamodel, microservice runtime execution configuration metamodel, and finally, microservice deployment metamodel. The proposed architecture is implemented using genetic and minimum nodes algorithms in an example application and the alternative results of deployment are shared. This study proposes a model specific to deployment only, and the results are discussed along with the application of the proposed architecture.

In addition to academic studies, giant technology companies have also developed reference architectures, but mostly focused on their own technologies or services [33–36]. These reference architectures have addressed many critical building blocks in the system. In addition, sample application architectures have been shared with the reference architecture on how to use it within their own services or technologies. Although these sample architectures cannot be used fully in more complex architectures, they are considered to be useful for new starters. However, since these architectures are more technology-oriented and most of the services provided by giant technology providers are managed, practitioners could have trouble in comprehending the logic under the hood.

Based on the literature analysis presented in this section, we observe that the studies mostly focused on the use of MSA for particular concerns. Additionally, in few existing studies, the reference architecture was not dealt with comprehensively, and thus has not reached sufficient maturity to serve as a guide for developing MSA based application architectures.

2.3. Arcitectural Views

Software architecture is defining the structures of computing system composing the elements and relationships among them [37,38]. Software architecture is a very important tool to detect and prevent difficulties that may arise during the development phase of the system, and to make the system more maintainable and reliable. In addition, it is an essential artifact of software development process, as the decisions taken at this stage will also affect the entire software development process of the system [39].

Architectures are shaped by addressing stakeholder concerns. Stakeholders are individuals, teams or organizations that take the lead with their knowledge, concern for the design and development of the system, and play an important role in determining all kinds of requirements of the system [37]. Considering the concerns of the stakeholders, using different architectural views is the method generally followed in the definition of software architecture. An architectural view is a representation that describes the system components and their relationships from a particular point of view [38]. Thus, the views that fit the concerns of the stakeholders more closely are defined, and an architecture that appeals to all the stakeholders is obtained. Also, each stakeholder defines or analyses the architecture using the views of his/her own interest.

There are multiple approaches to documenting software architectures. It has gained importance that the architectures are reusable and easy to maintain, and how the architecture will be represented. The latest approach that emerged for this purpose is Views and Beyond [38]. In this approach, each view consists of different styles. In general, views are divided into four main styles that are Module, Component and Connector, Allocation, and Hybrid. Each view is meant to address architecture from different viewpoints, and also addresses different concerns. Each style is also divided into different sub-styles, and there are 17 sub-styles in total. Module style focuses on implementation details, while Component and Connector style deals with interactions of software components. Allocation style, on the other hand, addresses how to allocate software components. In our study, we have leveraged each type of styles to document not only microservice reference architecture but also application architectures. Sub-styles that have been leveraged in each style are explained in detail in Section 6 and 7.

2.4. Domain Analysis

Domain analysis is used to determine the needed knowledge. Domain analysis is the systematic method of obtaining and storing domain information to aid the engineering design process. Domain scoping and domain modeling are the two most basic processes in domain analysis. Domain scoping specifies the domain's scope as well as the knowledge sources required to determine the core ideas. Domain modeling seeks to express domain knowledge in a way that may be reused. One of the methodologies for domain modeling that may be employed is feature modeling [40]. In this method, feature models are used to depict domain models, which may be used to convey common and variable aspects of a product or system, as well as the connections between variable features. There are four fundamental feature 'types' in a feature diagram: (1) must have/must contain features, (2) optional features that can have/or not include components, (3) alternative features (XOR) that must include one of the potential components, and (4) and/or features that must include at least one of the components.

2.5. Domain Driven Design (DDD)

Domain Driven Design (DDD) was first proposed by Eric Evans. Its main purpose is to provide a software development solution for complex needs by deeply connecting the core business concepts of the application to a model. This theory consists of the following three items [8]:

- Place the primary focus of the project on the main domain and domain logic.
- Fit complex designs into a model.
- Initiate a collaboration between technical and domain experts to further explore the conceptual basis of the problem.

DDD requires new skills, discipline and a systematic approach. DDD is not a technology or methodology. DDD provides a framework of practice and terminology for making design decisions that focus and accelerate software projects dealing with complex domains [8]. DDD is critical to microservice architecture. While developing software for a large and complex area, it is possible to divide this area into different sub-domains and to draw the boundaries of microservices from there correctly, with the perspectives provided by DDD.

3. CHALLENGES AND SOLUTION DIRECTIONS OF MSA

3.1. Research Methodology

The second step of this thesis aims to identify the state of the art of MSA and describe the challenges in applying MSA and the corresponding solution directions. To this end, a systematic literature review (or systematic review) was applied following the guidelines by Kitchenham and Charters [41]. The basic activities of the review are shown in Figure 3.1. The SLR starts with defining the research questions followed by a definition of the search strategy and the identification of the study selection and elimination criteria. Subsequently, study quality assessment criteria are defined and the data extraction form is developed. Once these steps are ready, the data synthesis method is developed.

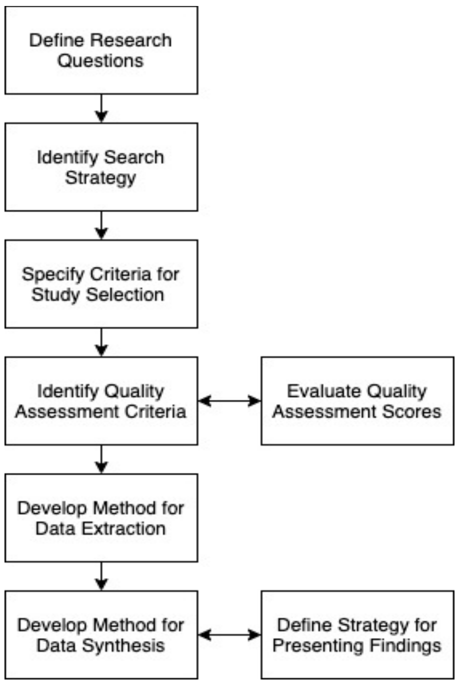

Define Research Questions
Identify Search Strategy
Specify Criteria for Study Selection
Identify Quality Assessment Criteria
Evaluate Quality Assessment Scores
Develop Method for Data Extraction
Develop Method for Data Synthesis
Define Strategy for Presenting Findings

Figure 3.1. Activities under the SLR protocol

The research questions (RQs) of our SLR are given below:

RQ1. What are the identified challenges of microservice architectures?

RQ2. What are the proposed solution directions?

The studies published between January 2014 (which is the date when MSA was first defined by Lewis and Fowler And February 2022 were included in the SLR. The electronic digital libraries included in the search were (in alphabetical order): ACM Digital Library, IEEE Xplore, Science Direct, Springer, and Wiley Inter Science (see Table 3.1).

Table 3.1. Publication Sources Searched

Source	**# Studies Initially Retrieved**	**# Studies After Applying Exclusion/Quality Criteria**
IEEE Xplore	233	48
ACM	755	12
Springer	1619	10
Science Direct	978	11
Wiley	174	4
Total	3842	85

Journal papers, conference papers, workshop papers and books were considered as potential search items. We used both automatic and manual search. Automatic search was performed by defining search strings using the APIs of the corresponding search databases. This was complemented with a manual search in which we used snowballing techniques. For selecting the primary studies, the following query was used:

(("micro service" OR "microservice" OR "micro-service") AND

("challenge" OR "obstacle" OR "difficulty" OR "difficulties" OR "problem"))

We identified and used the exclusion criteria listed below in order to eliminate the studies that were irrelevant for the purpose of this SLR:

EC 1: Studies with abstracts/titles that do not discuss MSA

EC 2: Studies with abstracts/titles that do not bring an approach to MSA

EC 3: Studies without a full text

EC 4: Duplicate studies retrieved from different digital libraries

EC 5: Studies that are not in English

EC 6: Studies that do not explicitly discuss challenges of MSA

EC 7: Studies that relate to MSA but are experience and survey papers

EC 8: Studies that present the application of MSA and do not critically reflect on MSA concepts

The application of the exclusion criteria resulted eventually in 85 papers of the 3842 papers that were initially selected.

The subsequent step included the quality assessment of the resulting primary studies, for which we used the quality checklists as defined in [42]. For the quality assessment, we used accordingly the checklist in Table 3.2. For the assessment scale we adopted a three-point scale (i.e. yes = 1, somewhat = 0.5, no = 0). The scores for the assessment of the primary studies are provided in Appendix 1.

Table 3.2. Quality Assessment Checklist

No	Assessment Question
Q1	Is the aim of the study defined clearly?
Q2	Are the scope, context and experimental design of the study stated clearly?
Q3	Does the study report have implications for research and/or practice?
Q4	Are the variables used in the study evaluation seem to be valid and reliable?
Q5	Are the measures used in the study explicit and aligned with the research purpose?
Q6	Is the research process documented in an understandable manner?
Q7	Are the main findings of the study presented clearly in terms of validity and reliability?
Q8	Is there an explicit statement of the limitations of the study?

The SLR followed with the detailed analysis and data extraction of the full-text of the 85 primary studies. The quality evaluation was included in this SLR as part of the data analysis, therefore the review data was preserved in the same format. To develop the data extraction, form a number of pilot primary studies were used and after a number of iterations, the final data extraction form was provided based on consensus between myself and my supervisors.

In the final step of the SLR, the data synthesis, a qualitative and quantitative analysis was independently performed on the data that was extracted from the primary studies. We discussed and selected suitable visual representations to support the synthesis process.

3.2. Overview of Selected Studies

The list of primary studies that were identified by this SLR is given in Appendix 2. In Figure 3.2., we present the distribution of the primary studies by year. From the figure we can observe a growing interest in the studies since 2015, following the year that MSA was proposed.

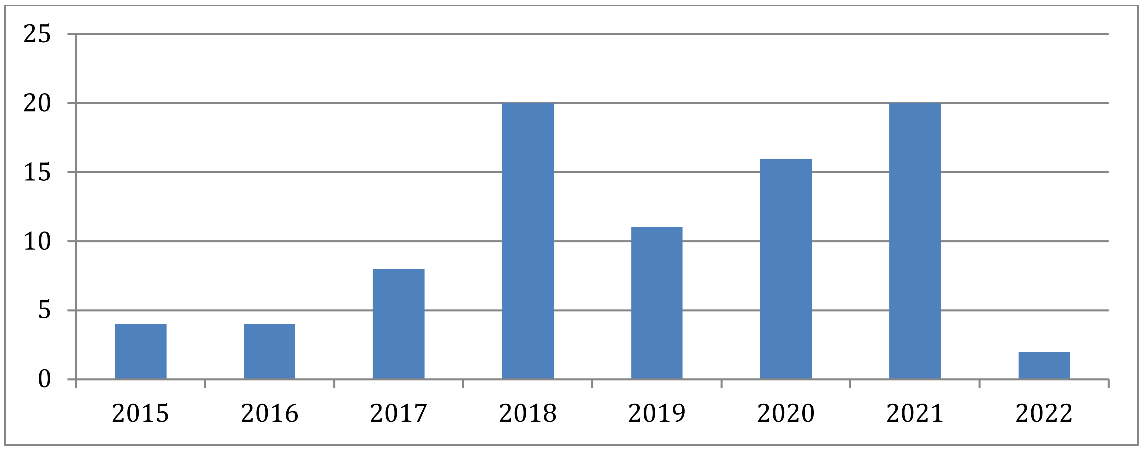


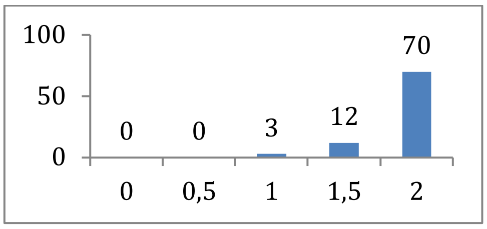
Figure 3.2. Year-wise distribution of the number of primary studies

We have also analyzed the research methods employed in the primary studies to investigate the strength of evidence in these. Table 3.3. presents the adopted research methods in 85 primary studies. As shown in the table, six different types of research methods were searched in the review. From the table we can observe that the majority of the primary studies are based on a single case study.

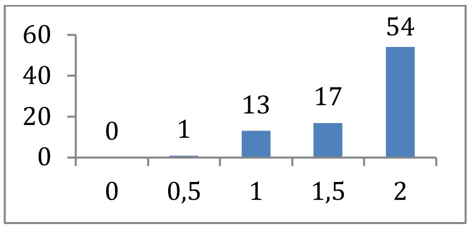
Table 3.3. Studies by research methods

Adopted Research Method	**Study Labels**	**# Studies**	**Percentage**
Descriptive or Not described	A, B, I, J, P, S, Y, AG, AH, AJ, AL, AQ, BE, BL	14	16.48 %
Single-case	D, F, G, K, L, M, N, O, T, U, V, X, Z, AA, AB, AC, AD, AE, AF, AI, AM, AN, AP, AR, AT, AW, AX, AY, BF, BG, BH, BI, BJ, BK, BM, BN, BO, BQ, BR, BS, BT, BV, BW, BX, CA, CB, CC, CE, CG	49	57.64 %
Multiple-case	C, E, H, Q, R, W, AK, AO, AS, AU, AV, AZ, BA, BB, BC, BD, BP, BU, BY, BZ, CD, CF	22	25.88 %
Experiment	-	0	0 %
Benchmarking	-	0	0 %
Survey	-	0	0 %

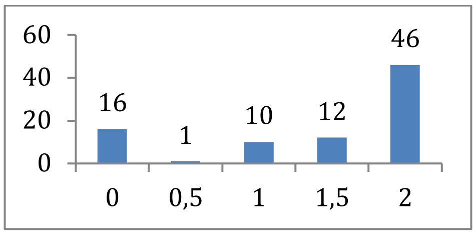
The result of the quality assessment using the quality checklist of Table 3.2. is shown in Figure 3.3. We wanted to look at rigor, credibility, relevance, and reporting quality when it came to methodological quality.



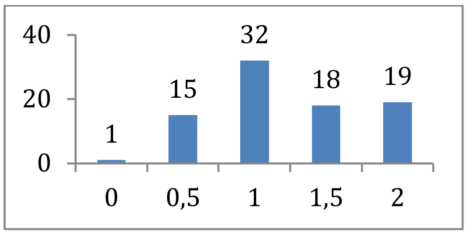
(a) Reference reporting quality



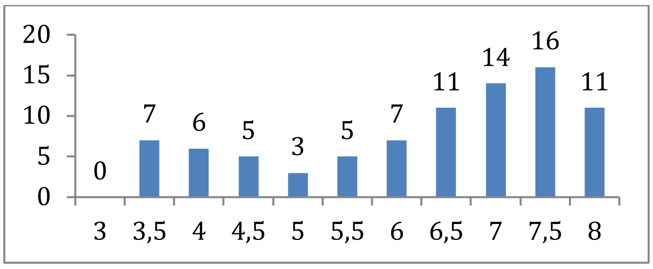
(b) Relevance quality



(c) Rigor quality



(d) Credibility of evidence in the studies



(e) Overall quality of the primary studies

Figure 3.3 Quality metric results for the primary studies

From this quality assessment it was concluded that majority of the primary studies (82.3%) are good with respect to reporting quality, and 63.5% of the studies (54 studies) were directly relevant to the field. Considering rigor of the research methods we can observe that 58 of the primary studies (%68.2) properly present the validity of their findings. In terms of rigor, forty-six studies demonstrate top quality. Nineteen research received the highest level for credibility of evidence, with reasonably robust and relevant results and conclusions. As a result of the quality scores for reporting, relevance, rigor and credibility of evidence, we can state that 59 studies (69.4%) with scores equal or greater than 6 are relatively good, eleven studies being high quality. On the other hand, 26 studies with scores less than 6 are identified as being poor quality. As a result, the majority of the reviewed studies are assessed to be good.

3.3. Identified Challenges and Solution Directions

The results obtained in relation to the research questions are outlined in this section. The data extracted from the primary studies are summarized with findings, separately for each question.

3.3.1. RQ1. What are the identified challenges in the MSA domain?

Table 3.4 depicts a summary of the nine problems discovered. The labels of the primary studies are listed in the first column, the publication dates in the second column, and the discovered faults (P1 to P9) in the research are listed in the remaining columns. At the right of the table is an explanation of the issues. Figure 3.4 shows a graphic representation of the problems that have been identified. The issues arising from the primary research are discussed in the sub-sections that follow.

Table 3.4. Primary Studies with Identified Problems of MSA

P1	Service Discovery
P2	Data Management and Consistency
P3	Testing
P4	Performance Prediction, Measurement and Optimization
P5	Communication and Integration
P6	Service Orchestration
P7	Security
P8	Monitoring, Tracing and Logging
P9	Decomposition

		Identified Challenges								
Study	Year	P1	P2	P3	P4	P5	P6	P7	P8	P9
A	2015			X						
B	2015			X						
C	2015		X							
D	2015							X		
E	2016				X					
F	2016						X			
G	2016			X						

H	2016			X						
I	2017	X								
J	2017	X								
K	2017				X					
L	2017						X			
M	2017	X								
N	2017						X			
O	2017						X			
P	2017								X	
Q	2018						X			
R	2018					X				
S	2018		X							
T	2018							X		
U	2018									X
V	2018							X		
W	2018						X			
X	2018								X	
Y	2018						X			
Z	2018								X	
AA	2018				X					
AB	2018									X
AC	2018							X		
AD	2018								X	
AE	2018						X			
AF	2018	X								
AG	2018			X						
AH	2018		X							
AI	2018					X				
AJ	2018							X		
AK	2019				X					
AL	2019			X						
AM	2019						X			
AN	2019						X			
AO	2019						X			
AP	2019									X
AQ	2019						X			
AR	2019				X					
AS	2019								X	
AT	2019						X			
AU	2019						X			
AV	2020						X			

AW	2020						X			
AX	2020						X			
AY	2020						X			
AZ	2020								X	
BA	2020						X			
BB	2020				X					
BC	2020	X								
BD	2020						X			
BE	2020						X			
BF	2020						X			
BG	2020								X	
BH	2020						X			
BI	2020						X			
BJ	2020								X	
BK	2020						X			
BL	2021								X	
BM	2021						X			
BN	2021								X	
BO	2021								X	
BP	2021						X			
BQ	2021								X	
BR	2021								X	
BS	2021								X	
BT	2021							X		
BU	2021								X	
BV	2021								X	
BW	2021						X			
BX	2021						X			
BY	2021				X					
BZ	2021						X			
CA	2021						X			
CB	2021						X			
CC	2021									X
CD	2021						X			
CE	2021								X	
CF	2022				X					
CG	2022								X	
TOTAL : 85		5	3	6	8	2	33	6	18	4

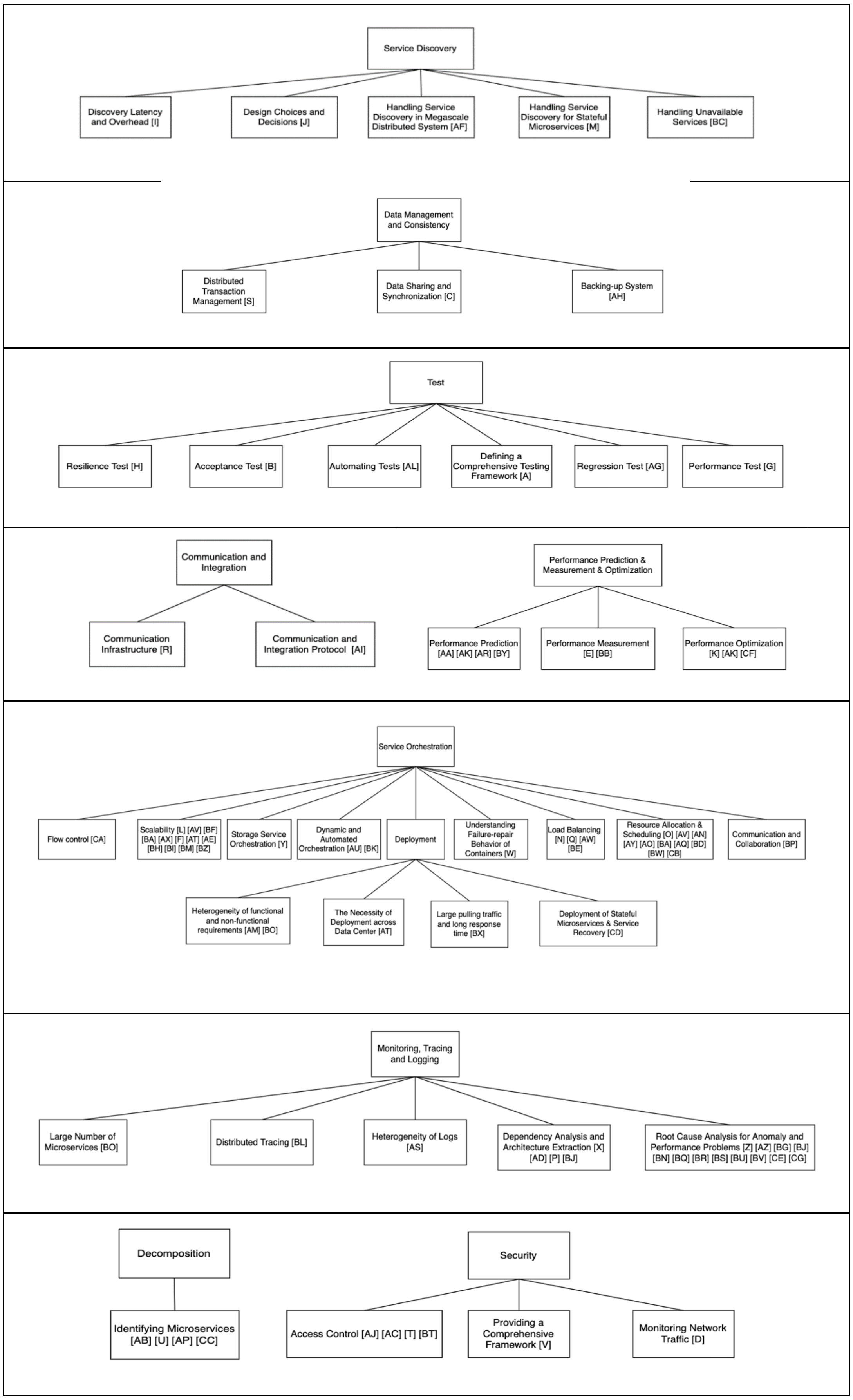

Service Discovery
Discovery Latency and Overhead [I]
Design Choices and Decisions [J]
Handling Service Discovery in Megascale Distributed System [AF]
Handling Service Discovery for Stateful Microservices [M]
Handling Unavailable Services [BC]
Data Management and Consistency
Distributed Transaction Management [S]
Data Sharing and Synchronization [C]
Backing-up System [AH]
Test
Resilience Test [H]
Acceptance Test [B]
Automating Tests [AL]
Defining a Comprehensive Testing Framework [A]
Regression Test [AG]
Performance Test [G]
Communication and Integration
Communication Infrastructure [R]
Communication and Integration Protocol [AI]
Performance Prediction & Measurement & Optimization
Performance Prediction [AA] [AK] [AR] [BY]
Performance Measurement [E] [BB]
Performance Optimization [K] [AK] [CF]
Service Orchestration
Flow control [CA]
Scalability [L] [AV] [BF] [BA] [AX] [F] [AT] [AE] [BH] [BI] [BM] [BZ]
Storage Service Orchestration [Y]
Dynamic and Automated Orchestration [AU] [BK]
Deployment
Understanding Failure-repair Behavior of Containers [W]
Load Balancing [N] [Q] [AW] [BE]
Resource Allocation & Scheduling [O] [AV] [AN] [AY] [AO] [BA] [AQ] [BD] [BW] [CB]
Communication and Collaboration [BP]
Heterogeneity of functional and non-functional requirements [AM] [BO]
The Necessity of Deployment across Data Center [AT]
Large pulling traffic and long response time [BX]
Deployment of Stateful Microservices & Service Recovery [CD]
Monitoring, Tracing and Logging
Large Number of Microservices [BO]
Distributed Tracing [BL]
Heterogeneity of Logs [AS]
Dependency Analysis and Architecture Extraction [X] [AD] [P] [BJ]
Root Cause Analysis for Anomaly and Performance Problems [Z] [AZ] [BG] [BJ] [BN] [BQ] [BR] [BS] [BU] [BV] [CE] [CG]
Decomposition
Identifying Microservices [AB] [U] [AP] [CC]
Security
Access Control [AJ] [AC] [T] [BT]
Providing a Comprehensive Framework [V]
Monitoring Network Traffic [D]

Figure 3.4. Visual Summary of Identified Problems

3.3.1.1. Service Discovery

In a distributed architecture such as microservice architectures, the discovery of microservices is one of the primary challenges. The challenges for service discovery relate to the design, implementation and quality concerns. At the design level designing the service discovery is considered to be a challenge due to the multiple various service discovery mechanisms such as client-side, server-side, hybrid service discovery. The proper decision needs to be made based on the various different requirements and quality concerns. Often multiple different design alternatives can be identified, and it is not easy to derive a feasible alternative. Implementing service discovery is directly dependent on system size and selected design so the most important criteria for the implementation are high availability and scalability. A misguided design selection and subsequent development will affect the system's availability and scalability. During the operation time or run-time, discovering the proper services requires the corresponding orchestration which needs to be aligned with the required quality of service parameters. One important quality factor is the latency of the discovered and triggered service.

3.3.1.2. Data Management and Consistency

Data Management and Consistency is another challenge of MSA because of its distributed nature. The challenges related to data management and consistency are more about distributed transaction management, but it is also about backing up the system and data integration. Architects and developers often choose database per service pattern to achieve distributed transaction management and MSA also favors decentralized data management. Although this pattern comes up with a lot of advantages like loosely coupled services, independently deployable and scalable services, the management of distributed transaction is really tough work. Backing up the entire application decomposed into microservices consists of some trade-offs, so it is not possible to handle backing up the entire system besides providing availability and consistency at the same time. Therefore, it is another challenging point for practitioners to decide which ones are more important according to the system design. In some MSA’s, there is not a mature mechanism for data sharing and synchronization. Microservices can operate on each other’s data without a coherent architecture and then it makes the system more complex.

It should be handled these sharing and synchronization operation without operating on each other’s data.

3.3.1.3. Testing

Testing plays a critical role for a system being ready before going live and for developers moving forward confidently. However, for MSA it is a challenging task to satisfy testing activities due to the distributed nature of MSA. Each microservice lives in a distributed environment and can be developed using different technologies, languages, and infrastructures which as such provides additional complexity for the testing process.

Testing resilience capabilities of MSA is identified as an important challenge. Resiliency enables systems to handle failure cases properly, but in a distributed architecture it is not easy to achieve this because microservices architecture is composed of a set of services that operates together and thus are prone to frequent changes. They should be loosely coupled and autonomous for being resilient as well.

Another challenging issue are performance tests. Non-functional requirements such as throughput and response time are important performance parameters for software systems. It is needed to have performance tests to measure these kinds of performance parameters to trace the system properly and prevent any failure. In distributed environments, however, measuring these quality parameters is not as easy as in a monolithic architecture because of the diversity and number of microservices.

Regression testing is needed to ensure that the system is still running after newly added feature or after a bug that has been resolved. Regression testing for MSA is not trivial, since all the test activities need to be handled in an agile way, they must be automated and included in the continuous delivery process.

Another issue is related to acceptance tests. They are as a set of test activities that must be run to ensure customer satisfaction and create robust systems, but this requires high maintenance costs to be handled in the microservice world because of agility of MSA.

In addition, defining a comprehensive testing framework has emerged as a tough work in recent years because it consists of many sub-challenging points like self-validation of interfaces, unit validation and integration validation of services. Besides, each validation should be automated to provide continuous and agile deployments.

The last one is automating tests. It is directly related to define or reuse a framework to run tests automatically. The proper testing flow needs to be prepared and run from time to time depending on decisions on test plan and at each testing cycle, it needs to be ensured that it is not affected system's reliability, and it is not an easy work.

3.3.1.4. Performance Prediction, Measurement and Optimization

Performance is a key quality factor for the systems employing MSA, because it is addressed at different levels for the stages of system design, implementation and operation. Usually, it is beneficial to estimate the performance of a software system before it is implemented because it may be very difficult or costly to change the system afterwards. After the implementation, performance measurement and optimization become important in order to satisfy the quality of MSA-based system requirements.

3.3.1.5. Communication and Integration

Communication and Integration is another challenging point that has emerged as a result of distributed architecture. Even if microservices communicate with a more lightweight protocol, it is still difficult to ensure that the communication infrastructure is reliable and the protocol to be used for communication and integration can handle complex workflows. The most important criteria for both challenges are reliability and durability, if these criteria are not met, the proper operation and reliability of the system will be affected, possibly it will cause cascading failures in the system.

3.3.1.6. Service Orchestration

Service orchestration is a concept that contains deployment, scalability, scheduling, management and networking of microservices. Although container orchestration tools that address many of these concepts have also been developed in industry, there are also studies that suggest solutions for challenging points in each area. The challenges for Service Orchestration relate to scalability, dynamic and automated orchestration, storage service orchestration, deployment, load balancing and scheduling.

Adapting containers' resources dynamically according to the changing requirements makes MSA-based systems highly available. However, it is a challenging issue to trace the changing requirements of containers and make the necessary adjustments according to usage of resources over time.

Providing persistent storage among different containers is another challenging point. With the increasing usage of deployment for stateful applications in the cloud, the need to address issues and challenges with persistent storage for containers are emerged. This problem includes many sub challenging points like multi-protocol support and storage service orchestration for volume management. They need to be handled by designing comprehensive solution to overcome workloads and resource problem.

Deployment is also a big challenge for practitioners. Although deployment processes gain a big momentum with containers, it has still some challenges. These challenges are mostly related to planning and configuring deployment. Besides, decentralized deployments are newly emerged challenges due to the necessity of deploying across data centers. Finally, heterogeneity of functional and non-functional requirements of microservices pushes practitioners to find an optimal deployment model to satisfy all these requirements for each service.

Load balancing plays a critical role in effectively distributing incoming requests to the backend servers. The most important criteria for the load balancing are availability because requests that are not effectively distributed will cause the system to stop responding so it will affect system availability.

Auto-scalability is another challenging point for container orchestration because in distributed environments, services need to be monitored and automatically adjusted resources according to changing loads to keep application predictable, resilient and available. These requirements bring a lot of challenges for scalability.

Other challenging point is resource allocation and scheduling. It is an important activity to organize and manage chain of services and schedule the available resources to effectively use. Besides, reducing total traffic cost and delay are important criteria for scheduling. Misguided scheduling directly affects the availability and reliability of system.

Understanding failure-recovery behavior of containers is the last challenge of orchestration. It needs comprehensive analysis on how the containers run from availability and reliability point of view. Moreover, deployment configurations often need to be reviewed for effective analysis and if needed it should be improved.

3.3.1.7. Security

With the development of MSA and distributed architecture, security has started to be more important topic because each microservice exposes new entry point to both internal and external side. This situation brings with it a lot of security problems as long as it is not addressed.

The first problem is access control mechanism in MSA. With the access control, the resource that is intended to be accessed is either restricted or not allowed. However, applying access control in the MSA is difficult to apply because of a distributed nature.

Furthermore, building a comprehensive framework to establish security between microservice is a tough work to achieve because it is very difficult to integrate and use non-easy to use and non-lightweight frameworks comfortably.

Another important issue is monitoring of the network traffic and running some security rules defined according to requirements.

3.3.1.8. Monitoring, Tracing and Logging (MTL)

Monitoring, tracing and logging is an important activity to ensure that systems are able to satisfy availability, performance and reliability concerns. However, this important activity consists of several challenging points related to identifying strong coupled services, root cause of anomalies and performance problems, and heterogeneity of logs.

The logs from different microservices might be heterogeneous so understanding and traversing the logs emerges a challenging point. If the trace cannot be established among the logs, the ability to monitor of system is directly affected. Thus, practitioners will not make proper decisions for troubleshooting.

It is critical to identify these problems and take quick action as soon as possible. Otherwise, the system's availability, reliability and fault tolerance will be directly affected.

Expected behavior for the MTL process is that trouble spots are detected, and the system is made more available, scalable, reliable and fault tolerant by taking quick actions or making changes in the design if necessary.

3.3.1.9. Decomposition

Decomposition allows us to have autonomous services organized around business capability services. it is necessary to separate the system into suitable pieces functionally and obtaining high cohesive and loosely coupled services are expected as a result of decomposition. The challenging point encountered first after deciding to go with MSA is to determine the right size of business capability and if it cannot succeed properly, MSA will not be an advantage and it might cause many problems in terms of mainly scalability, performance, availability and reliability.

3.3.2. RQ2. What are the identified solution directions?

When addressing the challenges of MSA, many studies provide the related solution directions together with the challenges.

Table 3.5. provides a summary of the solution directions for the identified problems in Table 3.4. From what we observe from the Table 3.5, the solution directions are inherently varied based on the identified challenges. Design heuristics and design abstractions, algorithm implementation, adoption of other paradigms and ways to realize system-wide quality management are some of the solution directions. Each challenge's solution directions will be discussed individually below.

Table 3.5. Solution Directions for the Identified Challenges in MSA

Primary Challenge	**Solution Direction**
P1. Service Discovery	- Client-Side Service Discovery. Study J - Server-Side Service Discovery. Study J - Service Registry. Study J - ICN Based Service Discovery. Study I - Static - Dynamic Service Description. Study AF - Stateful Routing Mechanism. Study M
P2. Data Management and Consistency	- Multi Agent-based Framework. Study S - BAC Theorem for Backing up Data. Study AH - Solution Framework. Study C
P3. Testing	- Reusable BDD based Acceptance Test Architecture. Study B - A Flow for Regression Test. Study AG - An Architecture and Framework to automate performance test. Study G - A Framework for Testing the Failure-handling Capabilities. Study H - Validation Framework. Study A - Automation Testing Framework. Study AL
P4. Performance Prediction, Measurement and Optimization	- Simulation Model. Study AA - Performance Model and Prediction Method. Study AK - Performance Prediction Model. Study AR - Performance Analytical Model. Study E

	- An approach for the quantitative assessment of microservice architecture deployment configuration alternatives. Study BB - Performance Degradation Prediction Framework. Study BY - A Model-driven Approach for Continuous Performance Improvement. Study CF
P5. Communication and Integration	- Reference Architecture and Orchestrator Language. Study AI - High-performance Userspace Networking Solution. Study R
P6. Service Orchestration	- An Extendable Solution for Autoscaling. Study L - Database-is-the-service Pattern. Study F - Workflow Scheduling Algorithm. Study AV - Autoscaling Research Pipeline. Study BF - Ant Colony Algorithm for Microservice Scheduling Optimization. Study AN - A Novel Scheduling Strategy. Study BA - A Lightweight and Flexible System for Autoscaling. Study AX - A Generic Architecture and Implementation for Automated Orchestration. Study AU - Configuration Models and Tool for Analyzing the Availability. Study W - Storage Service Orchestrator Framework. Study Y - A Monitoring based Architecture for Managing Deployment. Study AM - Decentralized Orchestrator. Study AT - Process Definition for an Elasticity Controller. Study AE - Decentralized Load Balancing Algorithm. Study N - Overload Control Method. Study Q - A Hybrid Approach Combining Client-side and Server-side Load Balancing. Study BE - Queue-based Chain-oriented Load Balancing Method. Study AW - A Novel Fair Weighted Affinity-based Scheduling Approach. Study O - A Novel Scheduling Framework for Kubernetes. Study AQ - Dynamic Microservice Scheduling Algorithm for Mobile Edge Computing. Study AY - Resource Allocation Optimization Approach. Study AO - Many-Objective Genetic Algorithm Scheduler. Study BD - A Novel Formula and Model for Determining the Thresholds of Total Resource Consumption. Study BH - Autoscaling Research Pipeline. Study BI - Using Declarative Business Processes for Service Orchestration. Study BK - RL agent based intelligent autoscaling model. Study BM - A Decision Framework to Select Right Microservice Collaboration Pattern – Study BP - Elastic Scheduling Algorithm – Study BW - Layered Container Structure for Microservice Deployment – Study BX - Autoscaling Framework for Microservice chain – Study BZ - Dynamic Flow Control Algorithm – Study CA - Microservice Rescheduling Framework – Study CB - A Kubernetes Controller for Managing Availability – Study CD
P7. Security	- An approach that provides authentication and decentralized role-based authorization. Study T - A Platform for Identity and Access control of microservices. Study AC - Access Control Optimization Model. Study AJ - Prototype Layered Security Framework (hardware, virtualization, cloud, communication, application, and orchestration). Study V - An Approach for Handling Security as Security-as-a-service . Study D - Extended Role-based Access Control Model. Study BT

P8. Monitoring, Tracing and Logging	- An Approach and Tool for Generating Service Dependency Graph. Study X - A Tool for Generating Service Causal Graph. Study Z - An Approach for Analyzing Architecture. Study AD - A Tool for Handling Traversing Distinct Type of Logs. Study AS - A Tool for Architecture Recovery. Study P - A Root Cause Analysis Framework for Detecting Anomalies. Study AZ - An Execution Trace based Root Cause Location Method. Study BG - A Graph-based Trace Analysis Approach – Study BJ - An Offline Approach to Distributed Tracing – Study BL - A Four Layered Framework for Detection and Diagnosis of Faulty Microservices – Study BN - In-kernel Transparent Monitoring Service – Study BO - Microservice Fault Detection Method Based on Correlation Analysis – Study BQ - A Fault Model based Root Cause Localization Framework – Study BR - An Anomaly Detection Method based on Semi-supervised Learning – Study BS - A Root Cause Localization Approach - Study BU - Lightweight Spectrum-based Performance Diagnosis Tool – Study BV - An Anomaly Detection Approach with execution Trace Comparison – Study CE - An Agent Based Monitoring Platform to Detect Anomalies and Unexpected System Dependencies – Study CG
P9. Decomposition	- A Conceptual Methodology for Deciding Right Size of Microservices. Study AB - A Functional Decomposition Approach. Study U - A Dataflow-driven Decomposition. Study AP - A Dependency capturing and clustering based Microservice Identification Approach. Study CC

3.3.2.1. Service Discovery

The authors [I] propose a new approach that uses information-centric networking (ICN) to find a solution to latency and overhead problems of service discovery mechanism. They make service discovery process in MSA easier by using information-centric network concepts. Since it is possible to offer a simple discovery process that decreases the number of service name record, name-based routing and hierarchical naming are used.

Study [J] presents multiple decision guidance models that can be used when deciding on an eligible microservice infrastructure. In this paper, there are multiple decision guidance model with own design options addressing the fault tolerance and service discovery area. For each of the models, they provide specific infrastructure technologies to implement design options.

Study [AF] provides a novel solution architecture to solve scalability and workload problems of service discovery in mega scale systems. The authors focus on the idea that service description data can be broken into dynamic and static properties. They propose an architecture subdivided into two independent, interconnected processing levels for static and dynamic query parts. Both processing levels consist of interconnected peers which allow to scale the registry dynamically.

Study [M] proposes a mechanism to optimize service discovery operation in stateful microservices. Scalability and efficient usage of infrastructure resources are the main aspects of this study. The authors claim that an efficient and scalable routing mechanism is needed to figure these problems out. The proposed model has been validated with two experiments and it has been observed that there was an increase in scalability and a decrease in usage infrastructure resources.

Study [BC] argues the unavailable states of services, useless and faulty interactions topics. The authors indicate that there should be synchronization mechanism to support microservices communications. Hence, they offer a framework called Synchronizer. This framework achieves collecting health/state information of microservices by using distributed registries. This framework has been validated with multiple use cases and according to result it brings effectiveness to synchronization among microservices.

To sum up, the corresponding solutions to the service discovery are based on the mainly scalability and workloads problems. To cope with these problems, identified primary studies propose either a decision model helping to choose the best option within the existing solutions or novel model for handling service discovery problems.

3.3.2.2. Data Management and Consistency

Study [S] focuses on management of distributed transactions. In this study, a multi agent-based framework is proposed to coordinate distributed transactions of the system. This solution is based on agents associated particular microservice, eventual consistency, SAGA pattern [43] and semi-orchestrated asynchronous model and provides decoupled autonomous layer to application to simplify the microservice interactions.

Study [AH] introduces backup, availability, and/or consistency (BAC) theorem to be used in backing up microservice. This theorem indicates that practitioners have to pick up two out of three items which are backup, availability or consistency. This theorem inspired by the Consistency, Availability and Partition Tolerance (CAP) theorem [44] claims that it is not possible to satisfy both availability and consistency at the same time in backing up microservices.

Study [C] provides Synapse framework supporting independent services to share data with each other through clean APIs. This study addresses the problem of complex service groups that do not have a consistent, manageable structure, operate on each other's data. Synapse provides a transparent data propagation layer by using Model-View-Controller (MVC) framework and Object/Relational Mappings (ORMs). Synapse has been implemented for Ruby-on Rails and shown that it provides good performance and scalability.

In conclusion, the primary studies specified that data consistency, data backup, and data synchronization are challenging to handle. For distributed transaction management, the identified solution proposes multi agent-based framework. For backing up, the identified solution proposes a novel BAC theorem. Finally, a solution comprehensive framework is proposed to be used in data synchronization.

3.3.2.3. Testing

In Study [AG], the authors propose an automated method of running regression test. They focus on the software reliability challenge. They claim that regression test is an essential step of continuous delivery process and in order to ensure reliability, automated method of regression test plays a critical role. Authors define the process of how to run regression test automatically and place it in continuous delivery.

Study [B] presents reusable automated acceptance testing architecture to handle maintainability and reusability of the application. This study encourages developers to use Behavior-Driven Development (BDD) acceptance test more frequently. The authors claim that this architecture will minimize issues with integration maintenance cost.

Study [A] presents an analysis of existing cloud application test methods and defines the characteristics of MSA. Based on this analysis, they propose a validation methodology of microservice systems. This methodology consists of microservice unit validation and integration validations of microservice systems.

Study [H] focuses on the problem of testing resiliency of MSA-based applications. They present Gremlin that is a framework for assessing the failure-handling capabilities of microservices. This framework is based on the idea that is about manipulating interservice messages at the network layer while designing and executing tests. This framework has been validated by multiple case studies and their results show that this framework helps uncovering bugs in failure-recovery code and is suitable to MSA-based systems.

Study [G] addresses the problem of performance tests by checking the needs about non-functional requirements like response time and throughput. In this paper, authors propose a new framework that make performance tests run automatically. This solution is hooked on the HTTP and can be built comfortably. It consists of two main aspects which are a methodology allowing external applications to access the test parameters and a mechanism for using the methodology. The main feature of this proposal is to place the test methodology to each service. According to tests results, the mean of the average response time is decreasing compared to the one without the framework.

Study [AL] argues the capabilities of testing framework to ensure reliability and quality of applications. In order to overcome the lack of capabilities of testing framework, authors decided to provide the automation testing of the microservice with the help of integrated structure that includes the adaptation layer, data layer, test case layer, execution layer, analysis layer and management layer. As a result of this study, an automation testing framework is proposed to ensure reliability and quality and ease test data generation.

To sum up, testing MSA is an essential step to deliver an application because there are a lot of points that needs to be tested. Due to having a distributed nature, it can be complex and difficult to manage. In order to overcome these challenges, an automation testing solution and comprehensive testing framework are proposed. Furthermore, testing failure handling capabilities and microservice unit validation and functional integration validation approaches are proposed to develop resilient and reliable application.

3.3.2.4. Performance Prediction, Measurement and Optimization

Study [AA] focuses on dynamic workloads problem and to address it, the authors propose an adaptive performance simulation approach. They measure the performance of applications with a queue-based model and then estimate the response time by modifying the parameters of the performance model of the application. To validate this approach, microservice based application and simulated workloads are set up. It has been observed according to experiment result that this approach on performance simulation gives better results in terms of response time compared to other existing methods.

Study [AK] investigates the factors to decrease performance overhead for microservices. Therefore, the authors suggest three-layer performance model and prediction method and it is built upon performance optimization and modeling. They carried out both experimental and simulation tests to validate performance model. It is evaluated that this method provides significant advantages to enhance the performance of microservices.

Study [AR] points out the challenge of predicting the workload capacity of microservices. The authors suggest a performance prediction model to address this challenge, and a tool called Terminus was prepared to estimate the capacity of each microservice with respect to different deployments. To evaluate this model, an experiments environment is set up, which consists of 4 microservices. Experiment results show that it gives good result to predict capacity with Mean Absolute Percentage Error (MAPE) less than 10%.

Study [K] indicates that the size of a microservice directly impacts its performance and availability. This paper proposes an approach providing workload-based feature clustering for deployment to improve the performance of an MSA. This approach uses genetic algorithm for clustering. To leverage this approach, they have created Microservice Architecture Deployment Optimizer (MicADO), an open-source tool and this approach has been applied in a case study on an Enterprise resource planning (ERP) system. The case study results show that there is a meaningful improvement in performance of the system.

Study [E] focuses on the scalability, manageability and performance issues. This paper points out that these issues have become more remarkable with the MSA getting popular. This paper proposes a performance analytical model for what-if analysis and capacity planning. Finally, two experiments have been conducted to validate this approach by using the performance metrics like response time and probability of request rejection. It has been seen that what-if analysis and capacity planning for MSA could be applied for minimum cost and time.

Study [BB] offer an approach for assessing scalability and performance on different microservice deployment configurations quantitatively. Besides, a domain-based metric for each alternative is defined and can be used for making decision on which one is well-suited. This approach has been evaluated by extensive experiments. The authors note that the domain-based metric for one of the environments is a function that does not increase the number of CPU resources. Also, they strongly recommend that it is necessary to have and execute performance engineering activities to modify by adding resources to deployment configuration in auto-scaling cloud environments.

Study [BY] points out that there is a lack of predicting performance degradation and its root cause. Although some approaches aim to predict performance degradation, they do not address its root cause. This paper proposes a framework to detect its root cause as well. This framework called SuanMing can predict root causes for potential performance degradation. Further, its aim is to prevent performance degradation before it occurs. To validate this approach, the authors evaluated their framework in two MSA-based systems. Evaluation results confirmed its accuracy of over 90% on predicting performance degradation.

Study [CF] indicated that the number of studies addressing performance problems of MSA-based systems is limited. In this study, the authors propose a model-driven approach for continuous performance enhancements by defining some dedicated metamodels. This study provides refactoring actions that enable performance improvements by taking advantage of the relationship between the monitored data and the architectural model. This approach has been used on two MSA-based systems to validate its feasibility.

In summary, the challenges about this topic are related to performance prediction, measurement and optimization. With the wide use of microservice applications, it has become possible to create a wider solution set for performance problems. There are studies that address performance issues directly, as well as studies that address other problems such as scalability and load balancing and provide benefits at the point of performance. These studies have been evaluated in their own categories.

3.3.2.5. Communication and Integration

Study [AI] focuses on complex microservice data flows and communication. To contribute on the solution of this problem, event driven lightweight platform called Beethoven for microservice orchestration is proposed. The platform is formed of reference architecture and an orchestration language. To prove its practicality, an example application has been implemented.

Study [R] focuses on network pressure increase because of inter-microservice communication. The networking of containerized microservice is inefficient. This paper proposes high-performance user space networking solution for containerized microservices called DockNet and provides a master-slave threading model to decouple execution and management. This model uses Data Plane Development Kit (DPDK) and customized Lightweight IP (LwIP) as the high-performance data plane and TCP/IP stack. Thus, in order to improve network performance, a robust and fast channel between microservices is built. Various experiments are conducted to validate it and as a result of these experiments, DockNet delivers over 4.2 ×, 4.3 ×, 5.5 × higher performance compared to existing networking solutions.

To sum up, with the widespread use of MSA, the need for communication and integration between microservices has become a challenging point. However, there are not enough studies proposing solutions. We identified only two studies figuring out some solutions to challenging part of communication. These studies come up with solutions about complex microservice data flows and network performance.

3.3.2.6. Service Orchestration

Study [L] focuses on the auto-scalability issue and provides a solution called Elascale for managing resources according to workload and application states. However, there is a need for collecting and analyzing performance metrics to manage the scalability of the system. For this purpose, Elasticsearch is used. In this paper, the authors offer architecture and the initial implementation of Elascale. Elascale consists of auto-scalability and monitoring-as-a-service components. Thanks to the monitoring-as-a-service feature, the application stack is monitored and if necessary, scale in or out process is applied. Besides, Elascale is an extendable solution so if desired, a new scaling algorithm can be added.

Study [F] addresses complexity problem of microservices communication and scalability issues. This study proposes to place the business logic in the database to reduce complexity and obtain more scalable services. Its goal is combining services with data. The proposed model has been validated by conducting proof of concept study and experimental results show that there is an increase in terms of performance.

Study [AV] investigates the task scheduling and auto-scaling challenges in clouds. The authors noted that existing algorithms are not compatible with a two-layer structure consisting of virtual machines and containers. Therefore, the authors recommend an

Elastic Scheduling for Microservices (ESMS) approach with a workflow scheduling algorithm and a statistics-based strategy to find out the best-suited configuration under a continuous workload. To validate this approach, many simulation base experiments have been conducted. Experiments results show that ESMS reduces the cost.

The study [BF] argues how to provide auto-scalability efficiently to reduce costs and energy usage and the authors stated that a good solution will bring a significant increase in performance. Hence, they aim to build an autoscaling system using past service experiences. To this end, they focus on which microservice needs to be scaled for performance improvements. Finally, they propose a pipeline for auto-scaling and also an evaluation of a hybrid sequence and a supervised learning model. According to the experimental result, using a supervised model is so useful on which microservices should be scaled up more.

Study [AN] addresses the container resource scheduling challenge. The authors indicated that handling the container resource scheduling problem in an effective will decrease the cost and increase the cluster performance. Hence, a multi-objective optimization model with a novel ant colony algorithm for the container-based microservice scheduling is proposed. They aim to improve the metrics related to computing and storage by the proposed ant colony algorithm. To validate this approach, an experiment was conducted and its result shows that the proposed ant colony algorithm for optimization gives good results in terms of load balancing and cluster service reliability.

Study [BA] focuses on utilizing the computing resources challenge. To address this challenge, container-aware application scheduling strategy is proposed in this paper. The proposed strategy has multiple capabilities composed of using appropriate lightweight containers with minimum deployment cost and heuristic-based auto-scaling policy for optimizing computing resources. According to evaluation results, proposed method shows significant improvement compared to existing studies in terms of processing cost, processing time, resource utilization.

Study [AX] indicates that autoscaling is an important mechanism to manage workload. Computing resource is a key concept for autoscaling because when the workload increases in the system, it should be used in an effective way not to decrease performance. To this end, in this paper, a novel system named Microscaler is proposed to automatically identify the services that need scaling by collecting and analyzing metrics in the application stack and scale them to manage workload properly. The experimental results in a microservice benchmark show that Microscaler gets a better result than state-of-the-art methods in terms of optimum service scale and achieves an average of 93% accuracy in determining the service needed in scaling.

Study [AU] analyzes how an orchestration mechanism is integrated to microservice based cloud applications without making much reengineering. This paper suggests a generic architecture and initial implementation called MICADO to support service orchestration. Also, an implementation of this architecture is provided to show its usage and how the scalability of Data Avenue file transfer application can be improved. The authors claim that scaling up and down application cluster is made with MicADO effectively.

Study [W] addresses the issues about failure-repair behavior of the containers. In this study, the authors propose different configuration models inspired by Google Kubernetes to deploy software as a container. To make container availability analysis, non-state-space and state-space analytic models are developed. These configuration models are defined by a fail-response and migration service. Besides in this paper, an open-source tool is developed by using these models. It helps system administrators to monitor and evaluate containerized system availability.

Study [Y] focuses on the needs for supporting the stateful application workloads by providing persistence storage. The authors propose the Cloud-Native storage service orchestration platform based on the IBM Ubiquity framework. This platform provides solutions for persistent storage among different container orchestrators and supporting multi-protocol access for Volume management within the storage systems The authors give the results of the effectiveness of the Cloud-Native storage service orchestration platform by preparing a prototype implementation. They estimate that the proposed framework will be useful for MSA-based systems.

Study [AM] addresses the challenge of heterogeneity of functional and non-functional requirements of microservices. It is important to satisfy all these requirements to overcome these challenges. The authors aim to support the deployment of microservices based on monitoring. To this end, an architecture is proposed to manage the deployment involving several cloud providers and to find the best deployment plan. Evaluation results show that this approach provides a solution within expected time interval.

Study [AT] identifies that there is a lack of deployment across data centers because most of the study has worked on the deployment to clusters in data centers. Therefore, a more dynamic approach is necessary to handle the deployment of applications in the edge computing paradigm. To this end, the authors recommend a fully distributed and decentralized orchestrator for containerized microservices, which is called DOCMA. To validate this approach, an experiment was conducted and its result shows that DOCMA has the required ability for orchestration of microservices.

Study [AE] focuses the problem of facing unpredictable workloads. The microservice-based application must match as closely as possible to the request to respond quickly and keep costs to a minimum. This paper proposes a novel heuristic adaptation process including two mechanisms that complement each other. While first mechanism balances load intensity by scaling containers according to capability of process, latter one manages additional containers to handle unpredictable workload changes. The experiment results show that this method manages unpredictable workloads successfully.

Study [N] addresses the load balancing issues and proposes a simple algorithm for decentralized load balancing system for microservices inside container used to implement a task executing in a cloud. It can provide better performance compared to the existing centralized container orchestration systems.

Study [Q] addresses the problem of overload control for large-scale microservice-based applications. Authors propose an overload control scheme designed for MSA, called DAGOR. It monitors the load status of each microservice in real-time and distributes the load between the related services when overload is detected. DAGOR has been used in the messaging application for five years. According to experience and experiment results, DAGOR achieved high success.

Study [AW] focusses on the latency because of long-chain microservices. Generally, a request is processed by many microservice called chain and these microservices chains are in a competition to use resources. The authors noticed that there is not enough study to handle the competition between microservices chains. In this study, a queue-based and chain-oriented load balancing method is proposed. With this method, it is claimed that this method decreases the latency of the long chain. Their evaluations also show that it could decrease the latency of long chains.

Study [BE] addresses the load balancing issue. The authors stated that load balancing is the most important mechanism for availability and scalability and there are some techniques such as client-side and server-side to implement load balancing in a system. However, in order to benefit each method's advantages, they consider combining them. To this end, they propose a hybrid model to leverage advantages of both sides.

Study [O] points out the scheduling problem of microservices, especially in multiple clouds. The authors believe that there is an alternative way showing decreases overall turnaround time in contrast to the standard biased greedy scheduling algorithm. For this purpose, they propose affinity-based scheduling approach and compare it with the standard biased greedy algorithm. The proposed approach achieves a big improvement.

Study [AY] addresses total network delay and network price issues. Also, the authors noted that increasing energy efficiency is an important task in an edge platform. They aim to minimize network delay and price and improve energy efficiency by designing a novel approach. To this end, they propose a dynamic microservice scheduling algorithm for mobile edge computing (MEC) and evaluate the computational complexity of the scheduling algorithm. According to simulation results, it has been observed that the microservice scheduling framework improves the performance metrics based on total network delay, energy consumption rate (ECR), failure rate, average price, satisfaction level.

Study [AO] points to increase energy consumption and low service performance with MSA. The authors stated that since resource allocation should be handled efficiently, unlike the current studies not focusing on optimization issues for such chain-oriented service provisioning, they focus on the resource allocation optimization problem. They aim to optimize end-to-end response time and resource usage. To this end, the three-stage scheme is proposed to improve the metrics mentioned above. According to the evaluation, their approach provides a better result than benchmarking algorithms on load balancing and energy consumption.

Study [AQ] works on a novel scheduling framework for Kubernetes. The authors aim to introduce a solution providing improvement for locally main tasks. For this purpose, they propose a hybrid-state scheduler with for the unscheduled jobs. To validate this approach, they carried out an analysis of their approach's capabilities and evaluation results show that it will overcome problems of their existing solution in their clusters such as collocation interference, priority preemption, high-availability and baseline scheduling problem.

Study [BD] handles scheduling issues in terms of some concerns like availability, reliability, resource utilization, scalability and power consumption. It is noted that current scheduler solutions do not cover all of these concerns. However, the authors claimed that these concerns should be handled together in a scheduling approach to take better results. To this end, they propose a Many-Objective Genetic Algorithm Scheduler (MOGAS) to handle all these concerns. According to comparison results with Ant Colony Optimization (ACO)–based scheduler, it gives better results in distributing tasks equally and reducing power consumption.

Study [BH] focuses on the problem of determining the accurate resource consumption thresholds to scale applications properly and to ensure high availability. It is also stated that lower thresholds could cause many problems where the services become unavailable against the load. For this purpose, authors propose a model for calculating total resource consumption of containers by using mathematical formulas based on Gaussian functions and they managed to calculate the upper threshold values. They use a research project to validate calculated value being the minimum number of containers to deal with the load.

Study [BI] addresses a challenge of auto-scaling MSA or IoT-based systems. It is also stated that in order to enhance our system availability and reduce cost and energy usage, auto-scaling should be handled in an effective and efficient way. Hence, the authors aim to design a prototype auto-scaling system for MSA-based web applications. As a part of their study, they have developed a pipeline to be used to auto-scale microservices by experimenting with a hybrid sequence and supervised learning model to validate and endorse scaling solutions.

Study [BK] focuses on the difficulty of orchestrating microservices when business processes expand across multiple microservices. Therefore, this study proposes using declarative business processes to coordinate and orchestrate microservices from a data flow perspective. To validate their recommendations, they used the Beethoven platform introduced in Study [AI] and demonstrated the usability of this environment for microservice orchestration along with their proposed method.

Study [BM] provides an approach for container-level scalability. Since most of the cloud applications tend to be containerized every day and are expected to provide near real-time response especially in real-time applications, scalability is becoming a real challenge for this kind of application. The threshold values for autoscaling are getting important to ensure scalability efficiently. Kubernetes suggests some techniques for setting thresholds but setting the right values is still a big challenge. To this end, the authors introduce an intelligent autoscaling system including two modules. The first is in charge of identifying resource demands through a generic autoscaling algorithm and the second one is responsible for identifying the autoscaling threshold values by using reinforcements learning agents. To validate their results, they conducted an experiment, and experiment results show its efficiency compared to the default autoscaling paradigm. Up to 20% enhancements in response time have been measured.

Study [BP] addresses the challenge of selecting the right communication and collaboration pattern for microservices. There are two well-known patterns in the literature right now which are choreography and orchestration. To address this challenge, the authors propose a decision framework to help solution architects to consider key factors and goals. Further, they provide a weighted scoring method to select the most convenient pattern. The requirements of three case studies (Danske Bank, LGB Bank and Netflix) were reviewed and evaluated to demonstrate this framework's usability. According to the results of their evaluation, a hybrid approach using both patterns has been suggested.

Study [BW] focuses on the challenges of scheduling and autoscaling. The authors claimed existing algorithms had some trouble on streaming workloads and the two-layer structures consisting of virtual machines and containers. Therefore, they propose an Elastic Scheduling algorithm to overcome these challenges. This algorithm handles task scheduling and auto-scaling which is based on Variable-Sized Bin Packing Problem (VSBPP) together. With the conducted experiments, the proposed algorithm has been validated that proposed algorithm improves success ratio and cost.

Study [BX] points out that remote registry-based images could cause increased pulling traffics and startup time latency. To solve these issues, the authors come up with an idea of layer sharing deployment for microservices. Since containers are generally implemented as multi-layered structures, they claim that common layers can be shared between microservices. For this purpose, they propose an Accelerated Distributed Augmented Lagrangian (ADAL) based algorithm to be used by servers and registries. Experiment results show that it reduces the microservice startup time by 2.20 times on average.

Study [BZ] addresses an issue of performance degradation when traffic increases. The authors claim that existing approaches of autoscaling do not pay enough attention to the microservice chain and performance degradation issues. This study proposes an autoscaling framework for microservice chains. It includes two modules. The first is responsible for collecting samples from microservices and training a latency model using the GNN. The second is responsible for identifying the number of microservice instances through the GNN model. Their evaluation results demonstrate pHPA effectiveness with reduced latency and improved resource usage.

Study [CA] notes that the flow control rules are generally adjusted and applied manually. Besides, it is also noted that availability is really critical for MSA-based systems and it should be handled with some concepts like fault tolerance and flow limiting. To improve the availability of a system, the authors claim that flow control rules should be handled dynamically. To this end, they propose a dynamic flow control algorithm. The algorithm works on monitoring data and current flow and determines the flow-limiting thresholds. Evaluation results show that automatic flow control mechanisms obtain better results in terms of performance compared to traditional static methods.

Study [CB] proposes a microservice rescheduling framework to address performance degradation and response time challenges. The authors point out that response time is one of the most important keys for Quality of Service. Hence, runtime adaptations and rescheduling should be handled carefully. They stated that existing works lack handling the effect of configuration parameters of container-based microservices. The proposed solution makes some periodic monitoring and then rescheduling activities are triggered based on threshold-based rules. Experiment results demonstrate that with the proposed framework, a significant reduction of up to 13.97 % in the average response rate was achieved.

Study [CD] focusses on the availability issues on MSA-based systems. It is pointed out that availability is still a problem while migrating legacy application to MSA even if microservices will be running on Kubernetes, which is a popular service orchestration platform. The authors stated that repair actions of Kubernetes cannot satisfy the High

Availability (HA) requirements. Hence, they propose an approach in which automatic service redirection to healthy microservices and application state replication can be achieved by adding service recovery to the repair actions of Kubernetes. Their experiments results show that their solution brings an improvement in terms of response time.

To sum up, there are many kinds of studies addressing almost all the concerns about deployment, scheduling, auto-scalability, load-balancing and orchestration. These are the most important areas in service orchestration. Besides, these studies use some best practices and technologies implemented by some big vendors. Thus, it allows these studies to be used on wide-spread application area.

3.3.2.7. Security

Study [AJ] focuses on the limitations of access control technologies in the microservice environment. This paper suggests an access control optimization model based on Role-based Access Control (RBAC). This model enhances the Attribute Based Encryption (ABE) model being one of the most common cryptographic mechanisms, in which existing RBAC users can directly access the ABE encrypted data in microservices. It has several advantages compared to ABE, which are improving the expression ability of access policies, the security and operational efficiency of microservices, and reducing the computational cost.

Study [V] investigates the microservices security topic and tries to identify taxonomy of security issues. While making this research, Docker Swarm and Netflix security decisions are also investigated. This paper claims that microservice security requires a layered security solution consisting of hardware, virtualization, cloud, communication, application, and orchestration. In this paper, prototype framework for microservice security is described and a case study is conducted. The case study result shows the performance overhead of the security is around 11%.

Study [D] focuses on two problems of microservice security. First, network complexity complicates monitoring the security. Second, due to trusting among microservices, if any microservice fails, it may affect entire application. In this paper, the authors propose a design for security-as-a service for microservices-based cloud applications and they implement a flexible monitoring and policy enforcement infrastructure for network traffic by adding a new API primitive FlowTap for the network hypervisor. Effectiveness analysis results show that the proposed solution is able to tackle various monitoring scenarios.

Study [AC] addresses the problems of authentication and authorization in 5G platform. The authors come up with a solution based on specifically for identity and access control of microservices. The proposed solution has been implemented in the Network Function Virtualization (NFV) based platform called SONATA. It encourages using well-known techniques and simple designs for identity and access control and favors Role-Based Access Control.

Study [T] focuses on the problems of the authenticity and confidentiality of microservice calls. This paper criticizes of the HTTP based approach used for microservice an API calls and transport layer security (TLS) providing only link level channel security. In order to prevent these security problems, this paper comes up with a solution consisting of authentication with password and key pair and decentralized role-based authorization.

Study [BT] points out that the security of access control becomes challenging as the system grows because it causes more access points to be handled for security. The authors propose an extended version of Role-based Access Control (RBAC) called Hierarchical Trust RBAC. This model enables security managers to detect unauthorized access to sensitive information and identify verification. They also conducted a case study to show feasibility of their model. Case study results showed that it provides faster and more flexible access to sensitive information.

In summary, secure microservices are tightly dependent to our MSA design. Software architects should design MSA by taking into account the security concern since it might be tough work to provide secure system later. These studies propose solutions to identified problems, but there is no study proposing a model about how to design MSA to ensure security.

3.3.2.8. Monitoring, Tracing and Logging

Study [X] focuses on the managing complex dependency relationships between microservices. This paper proposes an approach called Graph-based Microservice Analysis and Testing (GMAT) that automatically prepare Service Dependency Graph (SDG). It allows us to analyze, visualize and trace the dependency relationships between microservices. Besides, it allows detecting anomalies by watching service invocation chains. Experiments results show that GMAT is capable of managing complex dependency relationships for MSA-based systems.

Study [Z] addresses the problem of complex interactions, identifying abnormal services. Hence, the authors present a novel system called Microscope to efficiently generate a service causal graph and extract the causes of performance problems. Experimental evaluations show that Microscope has a good result and it is also claimed that it is better than most recent technology solutions.

Study [AD] takes attention that extracts component relations from just static sources are not enough for the accurate result because component relationships might arise at runtime. Extracting component relations is important to detect design drawbacks or potential architectural improvements. In order to overcome these issues, the authors offer an approach to extract and analyze the architecture of an MSA-based software system according to not only static service information but also aggregated runtime information. They have conducted an experiment to evaluate an approach. The results show that this approach is useful for detecting design drawbacks and improving the design.

Study [AS] focuses on the problem of heterogeneity of logs. In other words, each microservice can create logs in different format and it causes a heterogeneity for logs. It is a tough work to understand and interpret these logs to make right decision for the system. Therefore, this paper suggests a novel approach based on REpresentational State Transfer (REST) architecture style. Two case studies have been made to validate an approach and evaluate an implementation of this approach called MetroFunnel. The assessments results indicated that it is successful in traversing logs and reducing the size of collected data.

Study [P] addresses the importance of high decoupling among microservice because authors realized that there is a lack of highlighting microservice communications. Hence, an architecture recovery tool called MicroART is presented in this study to show communications among microservices. This tool consists of 4 main components which are Docker Analyzer, Github Analyzer, Log Analyzer and Model Log Analyzer to be able to generate the models. The authors indicated that it can be used by software architects for analysis, documentation and architectural reasoning.

Study [AZ] investigates the root cause of anomalies in the application and it is stated that it can be a complicated and time-consuming job because a lot of communications need to be investigated. In this work, the root cause analysis framework is recommended which is graph-based. In order to show the effectiveness of this framework Grid'5000 testbed has been used to deploy three different architectures and then some anomalies were injected into these architectures. The evaluation result shows that this approach is more effective than a machine learning method ignoring the relationship between elements.

The study [BG] focuses on anomaly identification and its fundamental cause in MSA. The majority of Root Cause Analysis (RCA) investigations, according to the authors, focus on data monitoring, data reliance among services, and invocation data. However, they apply invocation chain anomaly analysis to solve the RCA problem in this study. They used a robust principal component analysis and a single indication anomaly detection approach to create the algorithm. They tested their algorithm on three batches of test data from the 2020 International AIOps Challenge and a batch of sample data.

They received a high score based on the organizers' scoring criteria, and their system performed well, with more accuracy than several other typical anomaly detection methods.

Study [BJ] points out the challenges experienced in traceability analysis in MSA base systems. Since it is a complex and dynamic environment, analyzing to investigate any problem can be challenging. This is mostly due to the fact that there is too much trace data and it is difficult to obtain the necessary information to detect the real problem. Therefore, the authors recommend a graph-based approach for trace analysis. The strength of their proposed method is that it provides efficient processing and storage, as well as a powerful access mechanism by combining graph database and real-time analytics database. They have conducted an experiment to validate their approach and the results of the experiment have confirmed its efficiency and effectiveness in diagnosing the problem.

Study [BL] focusses system-wide challenges of observability. The distributed and heterogeneous nature and tendency to decentralize responsibility are the factors that complicate the observability of MSA-based systems. In this study, the challenges of providing observability in MSA-based systems are emphasized and an offline approach that performs distributed tracing is proposed. With this method, it is recommended to model microservices as observable execution paths, so an abstraction is provided to generate realistic trace data again.

Study [BN] proposes a novel layered diagnosis framework including service response layer, timing constraints, causality analysis and ranking algorithm for detecting faulty microservices. The authors indicated that as system size grows, detecting faulty microservices in a complex environment would get challenging. Thus, they claim that their framework could be a solution to this problem. They also carried out a case study to validate their approach. Experimental results show that it managed to achieve 89% specificity and 77% recall.

Study [BO] provides a monitoring solution called Kmon for MSA-based systems. The authors aim to monitor the complex microservice environment and internal states of microservices in an effective way with their proposed solution. This solution collects indicators by breaking into three categories: TCP request data, topology level and the other indicators related to CPU, memory and block I/O, etc. To validate the proposed solution, the authors conducted an experiment. Experiment results show that it has little effect on response time and low CPU usage.

Study [BQ] points out the challenge of detecting faulty microservices and root cause localization. For this purpose, the authors propose a method called Microservice Fault Root Cause Location Method Based on Correlation Analysis (MFRL-CA). In this method, a microservice fault propagation graph is built by collecting the correlation between historical fault data and dependent call data to reduce the time consumption of detecting faulty services. They carried out an experiment to show their approach effectiveness and the results show that this method effectively managed to detect faulty services and their root cause.

Study [BR] proposes a root cause localization framework called ModelCoder. In this study, the authors have introduced some concepts to figure out the root cause localization problem and developed the framework upon these concepts. The first one is a concept for building dependency graphs between microservices. The second one is a formulization for root cause localization problem based on the graph built in the first step. Finally, a fault model called ModelCoder is built on these two concepts. They evaluated ModelCoder on a real-world system and the results show that ModelCoder is able to detect faulty root nodes within 80 seconds on average.

Study [BS] also points out the root cause localization problem and aims to detect microservice failures in an effective way. For this purpose, the authors come up with a method for detecting microservice failures by using a semi-supervised learning model and dynamic sliding window methods. To evaluate their model, they used public data and the results showed that the model had good performance and the accuracy of anomaly detection and root-cause location was close to 100%.

Study [BU] addresses availability issues caused by service anomalies. The authors stated that existing approaches were limited in terms of inefficient traversing mechanism of service dependency graph and detecting anomalies process also could result in failure. To this end, they propose a highly efficient root cause localization approach based on dynamically constructed service call graphs. Experimental results and the result of being used in Alibaba showed it obtained good results in terms of accuracy and efficiency.

Study [BV] aims to address the root cause of performance issues. The authors claim that complex communication among services makes the system performance unpredictable and hard to trace and detect the root cause of performance issues. Therefore, they propose a tool called T-Rank. It uses tracking data and combines them with a tracing chain. Further, it provides a ranked suspicious list of the containers based on the spectrum algorithm. As a result of their experiment with the data collected from a real-world MSA-based system, T-Rank is feasible to be used in MSA base system thanks to its high accuracy and low resource cost.

The authors [CE] propose an anomaly detection approach for MSA-based systems. They stated that existing approaches do not have the required skills to detect faulty services accurately. Therefore, they propose an anomaly detection approach. In this approach, first, execution traces are collected across microservices, then anomaly degree of traces is calculated and then differences between traces are analyzed to locate the components causing anomalies. According to their evaluation results, this approach achieves high precision and recall in detecting anomalies.

Study [CG] provides an agent-based monitoring platform by monitoring not only internally developed services but also externally developed services with the help of sidecar containers. Agents are responsible for monitoring incoming and outgoing network traffic and also system state by reading kernel data. Prototype evaluation results show that their solution has a similar performance as Prometheus, but also, they offer some functionalities focused on multi-vendor service integration.

In summary, it is important to monitor the environment after developing microservices so these studies in this part are generally focus the monitoring architecture model, extract dependencies and anomaly detection. There is a lack of a powerful tool with integration API with other 3rd party software among these studies.

3.3.2.9. Decomposition

Study [AB] focuses on deciding right size of microservices and provides a conceptual methodology to decompose business capability based on domain driven design principles. To evaluate the usage of this methodology, a case study is conducted on the weather information dissemination domain. Evaluation results show that weather information dissemination system is partitioned into different microservices successfully.

Study [U] proposes a systematic approach using functional decomposition and based on functional requirements. This approach aims to build high cohesive and low coupled decomposition. To evaluate this approach, they have compared microservices implementations by three independent teams. Evaluations results show that it achieves to identify microservices much faster.

The authors [AP] stated that decomposition process is so challenging task and it should be supported with an approach, so they suggest a dataflow-driven decomposition approach to handle decomposition problem of MSA. They aim to obtain independently deployable and scalable microservices, so they defined a four-step decomposition procedure consisting of business requirement analysis, building fine-grained Data Flow Diagrams, extracting dependencies between processes and finally identifying microservices by clustering processes. They conducted a case study to validate this approach and it has been observed that microservice candidates are determined by taking coupling and cohesive constraints into consideration.

Study [CC] proposes an approach for identifying microservices by analyzing dependencies between business processes thanks to control, data and semantic models.

Further, it also provides a clustering method to identify potential microservices. To validate this approach, the authors carried out a case study. The results of the case study demonstrate its doability. Besides, it also achieves better results than existing approaches in terms of microservice identification.

In summary, the better we decompose the business capability into microservices, the more powerful microservices we have so we can say that this challenge is the primary among other challenges. Despite of this fact, we could not find enough studies to work on this topic deeply.

3.4. Summary

We conducted a systematic literature review following the guidelines of Kitchenham et al [45]. The main purpose of the SLR was to identify relevant challenges and solution directions. For this purpose, we conducted a comprehensive study and selected 85 as primary studies from 3736 papers. We have carefully applied selection and elimination criteria in order to catch the most appropriate studies for our SLR study. As a result of our study, we could explore nine problem categories. We have observed that each study has addressed one or more problems and explained their solution to problems in their study. Quality factors like as dependability, availability, scalability, and performance are at the root of many of these issues. We've raised concerns about quality as a result of the difficulties we've uncovered. This SLR might be used in future research to emphasize the importance of quality issues in MSA. It might also point in the right way for identifying whatever quality problems have yet to be addressed directly. However, just because no in-depth research has been done on these quality problems does not mean they are irrelevant to MSA. Therefore, this observation could typically initiate further research on the quality concerns in MSA.

We have observed that with the usage of cloud computing, cost of resources has emerged as an important topic, so optimization of resource usage and performance and scheduling problems have become crucial. Besides, it has been observed that the challenges of service orchestration and monitoring have been covered in many more studies in recent years and detailed and comprehensive solutions have been presented on those areas. We see this as an expected consequence of any system development process. Since as the systems get bigger and more complex and the need for scalability increases, the need for monitoring starts to occur in those systems and in parallel, the orchestration needs increase. This is also the case in the development process of MSA-based systems. We consider these challenges as newly recognized challenges as a result of the growth and complexing of MSA-based systems.

The main threats to validity [46] of this SLR are related to publication and selection bias, and also to data extraction and synthesis. The publication bias is about the likelihood of the researchers to publish positive results rather than negative ones, which is beyond our control and remains as an open issue for future work. We carefully identified and applied the inclusion/exclusion criteria during the screening and review of the primary studies. Subjectivity in setting the criteria and picking the primary research, on the other hand, might have jeopardized the study's validity. To eliminate bias in the inclusion/exclusion criteria, we initially chose a random group of ten papers as recommended Zhang et al. [47] and defined the selection criteria recommended. I conducted the evaluation and selection of the primary studies, which were then randomized and reviewed by the my supervisors. Any difference in the selection of the primary studies was discussed in detail and a final decision was reached per study. After the primary studies were evaluated and selected, the relevant data for a pilot set of primary studies were extracted using a data extraction sheet and taking informative notes on it. The pilot data extraction was then reviewed by the supervisors and conflicts were resolved again by discussions until a common understanding was reached. Regarding the data synthesis, we applied a systematic grouping of the extracted data on the sheet. We evaluated and debated the problem categories and their justification in meetings, so the categories we established might be deemed to encompass the major issues. Some issues, on the other hand, might be classified as sub-categories of the core categories. We used feature models to draw attention to these.

4. METHODOLOGY

In this section, we describe the process we followed to develop the reference architecture for microservices. As illustrated in Figure 4.1, the process begins with MSA vendor analysis as explained in Section 5. To design the reference architecture, it is important to analyze the components, services, and architectures of MSA offered by the three major cloud providers. We also examine challenges, opportunities, emerging technologies, and new trends in MSA.

MSA is growing in popularity and several companies, including Amazon, Google, and Microsoft, offer architectures and services to facilitate the implementation of MSA in the cloud and the migration of monolithic architectures to MSA. Our review of the literature shows that several reference architectures have been proposed, but they are either conceptual in nature or do not fully address all aspects of modeling MSA.

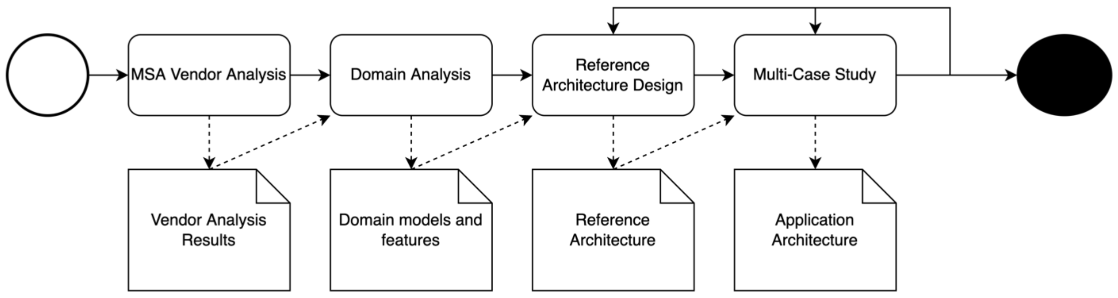


Figure 4.1. Development Methodology of Microservice Reference Architecture

The MSA Vendor Analysis has been followed by Domain Analysis [19] as explained in Section 5. It is an activity to store domain knowledge for the engineering team to use in architecture development. Domain analysis comprises two primary activities: domain scoping and domain modeling. Identifying the information sources and domains is what domain scoping is about. Domain modeling aims to represent domain knowledge in a reusable format. One methodology for domain modeling is feature modeling [40]. The domain model is represented using feature models that can be used to show common and variable features of a product or system, and the dependencies among variable features. A feature diagram has four basic feature types: (1) mandatory features which are so-called must have/must include, (2) optional features which can have/or not components, (3) alternative features (XOR) in which case it must include one of the possible components, and (4) and/or features that at least one component should be included in.

After the domain analysis, the reference architecture of MSA using viewpoints has been designed as described in Section 6. The architecture design can be represented using architecture design viewpoints. Several architectural viewpoints are defined to address different stakeholder concerns. In this study, we have adopted the layered view, decomposition view, and deployment & service-oriented architecture view. Reference architectures are generally designed by a group of varied organizations or by a company that serves many different clients. The reference architecture should meet the following characteristics to be beneficial [48]: Understandable to all stakeholders, accessible and read/seen by most of the companies, handling significant domain concerns, having satisfactory quality, acceptable, current, maintained, and offering value to the business.

Architecture design has been followed by a multiple-case study. Case study is an approach to evaluate the artifacts of the reference architecture [49]. The feedback from the case study can be used to enhance the reference architecture. Two case studies have been conducted to apply our reference architecture. Their implementations with recommendations, lesson-learned and evaluations are shared in Section 7.

5. FEATURE DRIVEN CHARACTERIZATION OF MSA

5.1. Research Methodology

This study aims to facilitate the design and development stages of applications to be developed with MSA and to serve as a guide. For this purpose, it is intended to identify the features in the MSA and to classify the technologies according to these features. In this way, the decision-making process will be accelerated, and it will be determined which factors the technology choice depends on. In order to achieve this goal, we have determined the following research questions:

RQ1—What are the current key MSA approaches in the state of the art?

RQ2—What are the key features of these MSA approaches in RQ1?

RQ3—What are the current implementation approaches for the MSA features in RQ2?

RQ4—What are the common and different features of the selected MSA vendor's approaches?

We developed and applied a research methodology shown in Figure 4.5 to reliably analyze all of the published work involved in this study. This protocol starts by performing domain analysis for MSAs and components; then, a characterization framework is developed according to this domain analysis. Domain analysis is the systematic process for analyzing and modeling the corresponding domain knowledge necessary for the engineering process. Domain analysis includes two key sub-steps of domain scoping and domain modeling. In the domain scoping process, the scope of the investigated domain is defined. In the domain modeling step, the domain knowledge is modeled for further reuse [9]. In this thesis, we use feature diagrams, which is one of the approaches for domain modeling [40]. Feature diagrams represent the common and variant features of a domain or system.

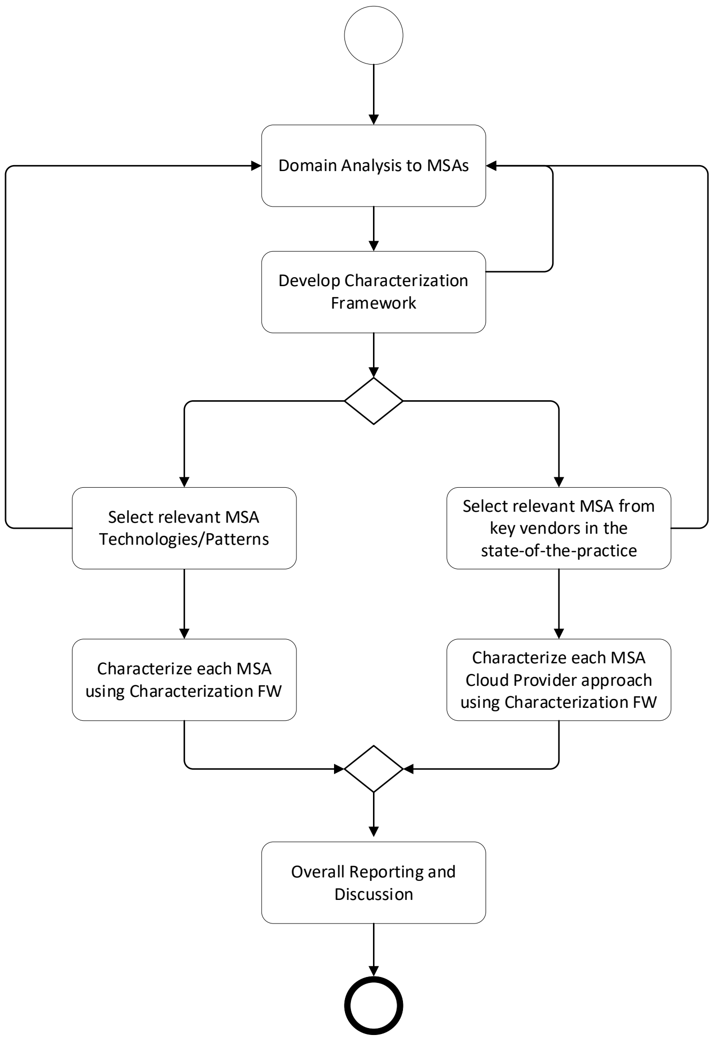

Domain Analysis to MSAs
Develop Characterization Framework
Select relevant MSA Technologies/Patterns
Select relevant MSA from key vendors in the state-of-the-practice
Characterize each MSA using Characterization FW
Characterize each MSA Cloud Provider approach using Characterization FW
Overall Reporting and Discussion

Figure 5.1. Research methodology

This process is followed iteratively because, in the meantime, the missing points in the characterization framework can be completed by returning to domain analysis again. Then, to validate our characterization framework, the studies that suggest technology and patterns from both the key providers and MSA area are handled separately and the related technologies are structured according to the characterization framework developed. While selecting and evaluating related MSA technologies and key vendors' infrastructure, the characterization framework can be updated again by going back to the domain analysis phase. Finally, we will eventually present a general evaluation of the work done.

5.2. Characterization Framework

We followed a bottom-up approach to classify studies on the MSA. As a result of this process, the characterization framework emerged. Figure 4.6 introduces the feature diagram of MSA, which represents the common and variant features as provided by the solutions. Table 4.6 defines the features of MSA. It has many features, with sub-elements being optional, obligatory, or having AND/OR and XOR relationships. Each top-level feature, together with the sub-elements, will be evaluated and discussed in detail in the following sub-section.

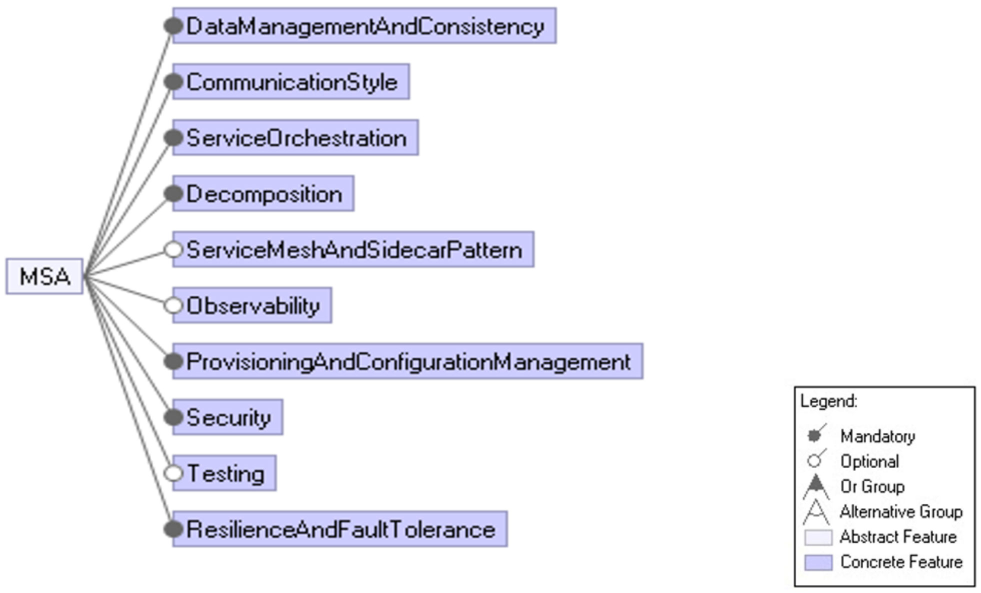


Figure 5.2. Top-level feature diagram of MSA.

Table 5.1. Description of the top-level features of the feature diagram for MSA

1	Data Management and Consistency	Data Management and Consistency highlights ensuring the quality of the distributed data management and consistency between microservices. Moreover, it tries to answer what kind of techniques exist to tackle data management and consistency.
2	Communication Style	Communication Style pays attention to the importance of communication style because it is one of the most complicated parts of microservices. So, it is crucial to find out what kind of communication method exists to provide a stable communication channel between microservices and outside.

3	Service Orchestration	Service Orchestration is the most comprehensive one, addressing lots of critical concerns, such as auto-scaling, service discovery, resource management, load balancing, container availability, and deployment. It focuses on the methods and concepts to handle all of these concerns.
4	Decomposition	Decomposition is the most basic stage of the design of microservices. It directly affects the further detail designs and development activities. It is concerned with which practices we can use while dividing our domain model to microservices.
5	Service Mesh and Sidecar Pattern	Sidecar Pattern is a preceded pattern for the service mesh, and the Service Mesh is usually built on this pattern. Sidecar pattern and service mesh infrastructure is a dedicated infrastructure layer for communication among services and providing resiliency and fault tolerance.
6	Observability	Observability is an important item that ensures the sustainability of the system. In distributed systems, it is critical to obtain information about the general performance of the system and the status of each block of the system and to take appropriate action according to this information or to avoid problems that will force the system.
7	Provisioning and Configuration Management	Provisioning is the process of setting up the system infrastructure. In this process, the necessary resources for the system and users must be managed. These management operations can be achieved with various specialized tools. Configuration Management, on the other hand, is a process that takes charge after provisioning and is used to ensure that our system remains in the desired and consistent state.
8	Security	Security stands on two headings, which are authentication and authorization. In microservice-based systems, since a system consists of many small parts, it must be designed very differently from the one that is designed for monolithic application. Being authenticated and being authorized for many services are the main topics for this feature.
9	Testing	In the MSA, although the fact that a system consists of smaller services increases the testability and maintenance capability of the system to a great extent, it is necessary to develop structures suitable for the distributed architecture in order to test use cases that spread on many services.
10	Resilience and Fault Tolerance	Resilience and Fault Management is the concept for admitting that failures always happen and the system is designed for failures.

5.2.1. Data Management and Consistency

The relationship between the data layer and services creates different alternative situations in a distributed architecture because the design is shaped according to preferences. When a monolithic application is allocated to microservices, it begins to separate in transactions, which means that local transactions, which were previously in the monolithic, are now being handled as distributed between services. There are different approaches here.

The first and more primitive of these is to manage distributed transactions with a shared database. Each service can access data owned by other services using local Atomicity, Consistency, Isolation, Durability (ACID) transactions. While this situation enables distributed transactions to be handled more easily and to make queries that require joining from different tables more easy, it causes many disadvantages. These are coupling creation in run time, different services needing different requirements from the same database during development, and changes to affect all services [22].

Another method is to have a *database for each service*. There are many advantages over a shared database in this more common alternative, where microservices are literally decoupled. Each service uses the database that best suits its needs and, since the dependency between services is removed, loosely coupled services are obtained, and this situation makes deployment activities more independent.

As shown in Figure 5.3, there are some operations that need to be handled if a database per service pattern is selected. First of all, the business transactions spanning multiple services need to be managed and data consistency must be provided. In this case, since it is important to have a highly available system, one needs to choose availability, as specified in the CAP theorem [50], and consider the consistency eventually. This situation corresponds with the Base Availability, Soft State, Eventually Consistency (BASE) database types, which is proposed by eBay for supporting faster reaction to possible inconsistencies by dismissing synchronization [51]. It is a database design methodology which favors availability over consistency of operations [52].

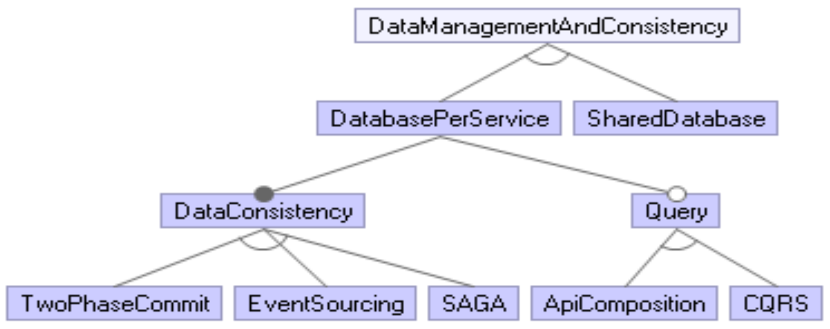


Figure 5.3. Feature diagram of data management and consistency

For *data consistency*, there are three alternatives. *Two-phase commit (2PC)* pattern is the traditional and only synchronous solution recommended for the management of distributed transactions. This pattern consists of prepare and commit phases and ensures that the data in the entire service are consistent at any given time. According to its setup, in case of failure, writing operations are blocked and availability is compromised. *SAGA* is another alternative for distributed transaction management. It is asynchronous and is used to ensure eventual data consistency without ACID operations when spanning multiple services [22]. When necessary, SAGA carries out compensatory actions at different stages to take it back when any business rule is violated. Each local transaction causes the start of the new local transaction by publishing a new domain event and, at the same time, compensatory functions are executed to undo local transactions as needed [53]. SAGA is difficult to implement due to the reasons of implementing these compensatory actions, the developers implementing these compensatory actions, and the difficulty of managing and debugging these processes. Furthermore, a microservice needs to update its business entity and transmit the message atomically to avoid data integrity and possible bugs. This situation is possible with some improvement of the solution brought by SAGA. With the *event sourcing* pattern, the atomicity problem is avoided. It stores all states of the business entity in order in the event store. State changes and message delivery are performed atomically on the business entity and a new state event store is added for each state change. In this way, a more reliable transaction infrastructure is provided. Moreover, the status of the business entity at any time can be determined by queries made over the event store [54].

Queries requiring different microservices have become difficult with the existence of distributed transactions. Because most queries are obtained by joining operations over data of more than one service, to overcome this situation, a structure that makes separate queries from each service and combines them can be considered. The *API composer* pattern recommends this. The results of the original query are calculated by firstly dividing the queries into the required services and then composing the results from each service. However, this situation often causes in-memory problems due to the excess of in-memory joins [22]. Another solution is the *command query responsibility segregation (CQRS)* pattern. With the CQRS pattern, queries are made over a view database that is registered to domain events and shaped according to the type of queries, thus making handling of complex queries easier [55].

5.2.2. Communication Style

Communication in a microservices architecture is one of the most challenging points due to its distributed nature. It directly affects the availability and resiliency of the systems. In the MSA, we can examine the communication in two headings, *intra-microservice communication* and *inter-microservice communication.*

As shown in Figure 5.4, the first and most complex of the two is *communication between services.* Services can communicate with each other through a *sync communication* infrastructure, but, with this architecture, both the client and the server must be available to sustain the communication. Moreover, there is a tight runtime coupling among services. Communication can be sustained without the need for any message brokers, but services need to know each other's locations, which brings extra complexity. Furthermore, an external request generally needs collaboration between services, which might cause blocking of the system for a long time and some problems in availability and resource usage of the system. However, these concerns can be eliminated with *async messaging*. Availability and resource management improves, and runtime coupling becomes loose. The presence of a message broker can be counted as a challenging point. In addition, communication management is more complex, too. Sometimes, *domain-specific protocol* can be used in the communication between services; although this type of usage is limited, it can be preferred in an appropriate use case, such as SMTP or IMAP [22].

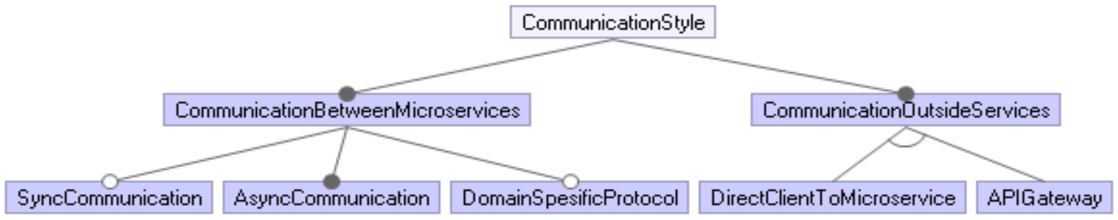

CommunicationStyle
CommunicationBetweenMicroservices
CommunicationOutsideServices
SyncCommunication
AsyncCommunication
DomainSpesificProtocol
DirectClientToMicroservice
APIGateway

Figure 5.4. Feature diagram of communication style

As shown in Figure 5.4, some patterns are recommended to make the communication of microservices outside of them healthier. For example, with the *API Gateway pattern*, all requests coming from outside are transferred to the appropriate services inside through this structure, and the services respond to this request by communicating with each other [56]. Different APIs can be created for each type of client. It is called Backend for Frontend (BFF) by SoundCloud [57]. Moreover, it can translate external requests into protocols used across microservices. Since the location information of the services changes dynamically, the outside world does not need to know this location information thanks to API Gateway. This structure can be thought of as the only door opening to the outside world and isolates the system inside. Security concerns can be addressed here. For example, in a scenario where HyperText Transfer Protocol Secure (HTTPS) is used when talking to the outside world, it will be sufficient for the services inside to talk with the HTTP protocol because the inside can be considered safe after the API Gateway. Some cross-cutting concerns, such as SSL, could be handled in API Gateway so internal microservices are lightweight and simplified [58]. Another solution is that *each client communicates directly with microservices*, but this method is a primitive method and its usage area is very limited. None of the benefits that come with API Gateway can be achieved with this pattern.

5.2.3. Service Orchestration

This concept, which can be referred to as service orchestration or container orchestration, automates the management, scaling, deployment, and networking of microservices. The application provides great assistance in deploying to different environments, without the need for a new design, to orchestrate the services. In this context, service orchestration is a concept that addresses many different concerns.

As shown in Figure 5.5, *auto-scaling* is one of them, and, by monitoring our application, it automatically adjusts the capacity according to the incoming load and keeps the system highly available and steady [59,60]. Within the auto-scaling configurations, the system can scale horizontally or vertically. It also provides a manual scaling feature to be used in some cases.

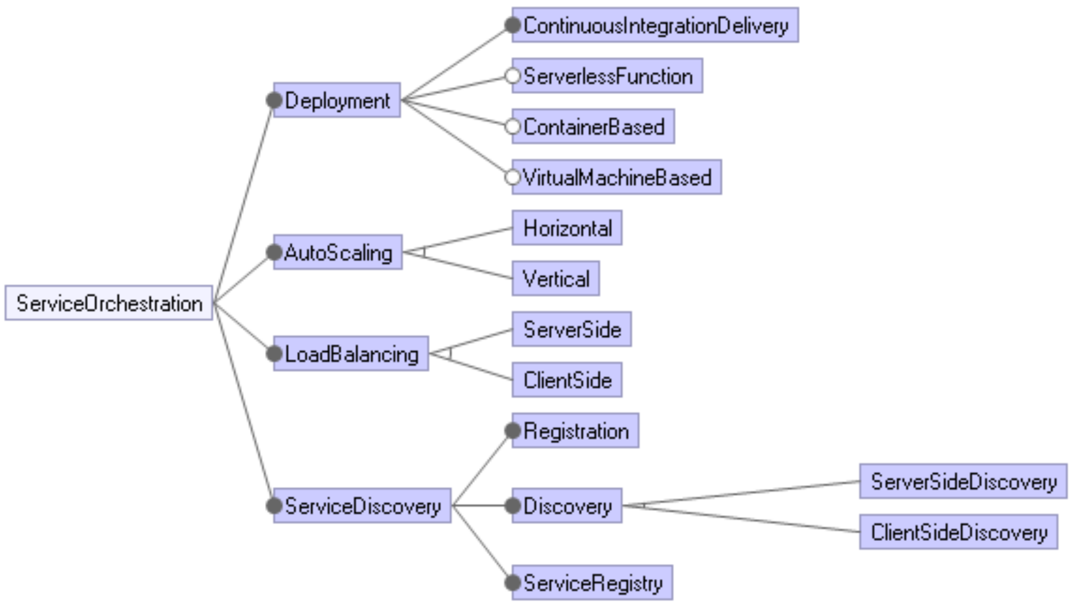


Figure 5.5. Feature diagram of service orchestration

Another concern is *load balancing*. It is used to distribute the traffic coming to the system efficiently. It also provides high availability and reliability by sending incoming requests only to the servers that are standing [61]. It works in harmony with the new server, adding and removing operations when necessary. In this way, the system will be more scalable and flexible. Different variations depend on where the load balancing setup is carried out. For example, in server-side load balancing, the client does not interfere with the load balancing process and its request is distributed efficiently to the appropriate servers on the server side; but, in client-side load balancing, the client takes over the load balancing job. After querying which servers are suitable or not from a structure, such as a service registry, it distributes the load effectively.

Service discovery, on the other hand, is an indispensable structure in the distributed architecture. Thanks to this structure, the changing services, whose location information is dynamic, become able to discover after they complete service-registration. Here, similar to the client- and server-side load balancer distinction, there is either a registry-aware client mechanism or a structure that requires the request from the client to be directed to our services via a registry-aware router.

Independent deployment is one of the most important skills aimed at and acquired by MSA. In this way, the *CI and CD* pipelines of our services are separated. An advanced deployment setup is created with automated infrastructure. In this way, fast delivery is ensured. While deploying, one can prepare an application for deployment with the help of *containers*. Thus, the containers are isolated from each other and encapsulated in the technology stack used while the services are built. Moreover, the services can be easily scaled up and down. Another method is to deploy the services using *virtual machine* (VM). Compared to containers, resource usage is high in VM. The container-based method has become a de facto for deploying the services at the moment and it is a lot more portable. Another deployment method is *serverless deployment*. It emerged as a result of the spread of microservices and cloud environments. With this deployment method, the user simply writes the code and uploads a provider that provides a serverless infrastructure. After that, it is completely up to the provider. Many headings, such as scalability, deployment, and operating system, are completely managed by the provider. Moreover, *serverless is the deployment* and development method, which is developed to implement the Function as a Service (FaaS) category of cloud computing services.

5.2.4. Decomposition

One can develop systems, which are large in terms of business rule and domain, with MSA. Hence, the aim is to develop the system in smaller applications and achieve continuous delivery and deployment. In addition, each microservice is developed faster and more easily. However, determining the boundaries of these small applications is not an easy task and needs to be carried out carefully. Moreover, the aim is to create loosely coupled, highly cohesive, and autonomous services. In addition, tools can be more cross-functional in this way.

The method used to design an application as smaller services are either *decompose by business* capability or *decompose by subdomain*, as shown in Figure 5.6. In decomposition by business capability, services are concentrated around business capability; while using DDD [62] principles in the decomposition by subdomain, they are concentrated on subdomains and use cases related to these subdomains.

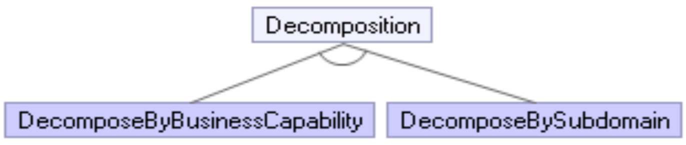


Figure 5.6. Feature diagram of decomposition

5.2.5. Service Mesh and Sidecar Pattern

Before the service mesh ecosystem was introduced, sidecar proxies had emerged and started to be used. *Sidecar proxies* encapsulate service discovery, communication protocols, load balancing, and fault tolerance mechanism to abstract them from the developer [29]. With the service mesh structure built on the sidecar proxy pattern, a fully integrated service-to-service communication infrastructure is provided and the security, reliability, and observability features are managed by the platform layer [6].

5.2.6. Observability

The large and complex nature of modern systems, dynamic infrastructure, and monitoring the health of these systems and taking the necessary actions as a result of this monitoring reveal the importance of observability.

As shown in Figure 5.7, *monitoring* collects information about a system by communicating with services. In order for this need to continue uninterruptedly, the system must have a scalable infrastructure and it must be easy to query the collected information. Monitoring focuses on runtime metrics created by the applications themselves and related measurements, such as CPU, memory, I/O, etc., which are the infrastructural metrics of the system. *Distributed tracing* in a system, on the other hand, is where requests are spread over multiple services and each service responds to this request by communicating with different layers. It follows the behavior of the application while responding to this request and whether it is experiencing any problems by assigning an external request ID to each request and recording it. In *log aggregation*, it is ensured that the logs coming from all these services are collected in a central service and can be queried and analyzed from there. In addition, by creating alerts for specific logs to be examined, developers are notified when such logs occur. *Exception tracking*, on the other hand, concentrates on exceptions and records the exceptions that occur in the system. With the help of the recorded data, various inquiries and informing the developers using alerts are provided when necessary. In this way, with a central exception tracking infrastructure, developers are prevented from working continuously with the same error because historic data are provided for the relevant error type and the user knows that the error has been solved before. *Audit logging*, on the other hand, records the information that the system users performed on the system and the stages they went through.

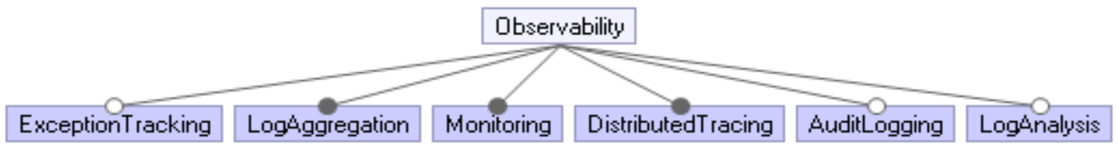


Figure 5.7. Feature diagram of observability

5.2.7. Provisioning and Configuration Management

Provisioning and configuration management has become a hot topic with the increasing interest in distributed systems and MSA. As shown in Figure 5.8, they should be established in each mature MSA.

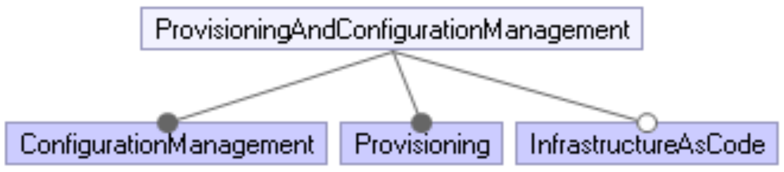


Figure 5.8. Feature diagram of provisioning and configuration management

Within the scope of *provisioning*, operations such as introducing the information technology (IT) infrastructure and then managing the resources needed by this infrastructure are handled. In addition, providing this infrastructure to the service of the system and users is also one of the provisioning activities. There are four subtypes: service, user, server, and network, coming after provisioning. It is a process to maintain systems and software, which ensures that systems remain in the desired state. With any *configuration management* tool, we can separate and manage the system into related groups or modify the basic configuration center, prioritize some actions, and automate processes, such as updating the system and expanding new settings [63]. *Infrastructure as code* can be considered as the next step. With this feature, one can program infrastructure by writing code and configure it the way it is wanted. In other words, one writes code to automate the infrastructure and run it. The idea behind this approach is that the systems and devices used to run the software themselves can be treated like software [64].

5.2.8. Security

In an MSA, security is actually gathered under two main headings as in all other systems, as shown in Figure 14. These are authorization and authentication. As shown in Figure 5.9, both are concepts to be addressed. However, handling these processes in MSA can create a more complex structure compared to the monolithic architecture. There are some best practices and patterns for this.

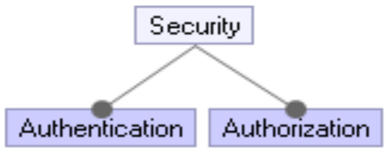


Figure 5.9. Feature diagram of security

For *authentication*, a token is usually given to the user by performing an identity check through a structure that is developed into a communication task between the external world and the internal world, such as the API Gateway. This token contains the information that the user has authenticated to the system and what his/her permissions are. Thanks to this structure, the user can authenticate from a single point to a structure with many services. This eliminates the disadvantage that many services have relations with the outside world. Moreover, in this way, the services inside will have the convenience of talking to each other with HTTP instead of HTTPS as an example. In this case, however, it should not be forgotten that the API Gateway is a centralized failure point and the design for failure should be carried out accordingly. Another option, for each microservice to run, is having its own authentication and authorization processes locally, as opposed to global management. This situation requires the requests to be authenticated separately for each microservice and the complexity increases. However, each service can use different authentication and authorization methods according to preference and, at this point, a more fine-grained mechanism should be designed.

5.2.9. Testing

With the spread of software development with MSA, the need for revising some approaches used in monolithic applications and adding new approaches has arisen, because, now, this is an environment where each microservice can be deployed individually and perhaps developed by different teams.

As shown in Figure 5.10, there are various types of test able to be used in MSA. We can test whether the system shows the expected behavior from end to end, with the *end-to-end* test, as in monolith applications. However, this approach is very difficult to manage because the test boundaries are too large and tests are very fragile. We can test smaller parts with *integration tests*. For example, it can be detected with this approach whether there is an error in the communication interfaces between different layers, such as the data layer and service layer. With the *consumer-driven contract test*, we concentrate on communication between services and, in the communication of the two services, it is tested whether the waiting of the service that will consult a message can be met by the service that will produce the message. In the *service component test,* which is another type of test, the components to be tested are isolated from the remaining parts of the system by using test doubles and are tested by manipulating through internal interfaces.

This enables each tested item to be tested in more detail. Finally, in *chaos engineering*, to ensure the stability of the system under all kinds of conditions, the system's responses are examined by leaving the system to deal with various failure conditions in the production environment, and thus the reliability of the system is tested. To sum up, it is of great importance to use the mentioned test approaches together and in harmony for the systems to reach high test coverage.

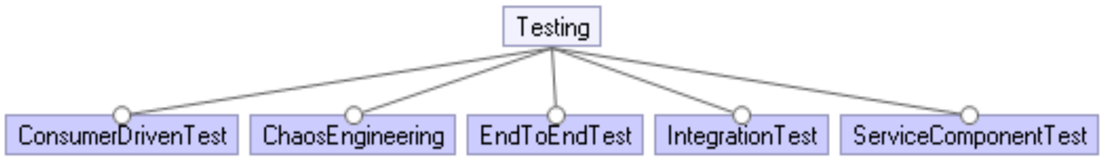


Figure 5.10. Feature diagram of testing

5.2.10. Resilience and Fault Tolerance

In monolithic applications, any failure had the potential to completely down the application. However, in case of failure that can be experienced with the MSA, it provides an opportunity to compensate for this situation without affecting the overall application. It is necessary to admit that there will always be failures in the system, and designs that address failure situations should be made to quickly avoid such failures or to reduce the number of failures. For this, failure scenarios should be determined as much as possible, locations that may cause a single point of failure should be identified, and our designs should be arranged to avoid cascading failure in case of a failure. As shown in Figure 5.11, there are many ways to ensure resiliency. It is highly recommended to use as many patterns as possible.

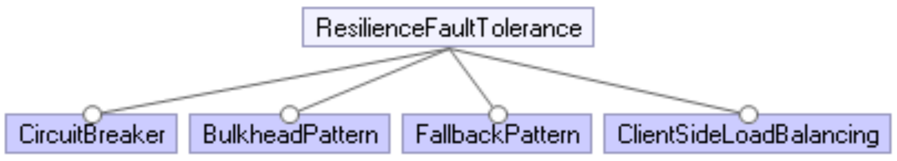


Figure 5.11. Feature diagram of resilience and fault tolerance

Client-side load balancing is often used in some scenarios as it eliminates a single point of failure and distributes the responsibility for load balancing and is easier to scale than server-side load balancing. Service instances query and cache information of health services from service discovery. In calls to be made by the service to other services, service information is received from the service discovery and communication is provided in a way that the load will be distributed equally. If one of the services responds late or gives an error, the load balancing mechanism detects this problem and removes it from the service repository and prevents cascading failures and system downtime. If service discovery does not respond, client services can use the information in their own cache copy. In *circuit breaker pattern,* if a client faces several problems over the call to another service, it stops communicating with the relevant service, and thus prevents cascading failure in the system. *Fallback pattern* is another approach for handling failure cases. In this pattern, as a result of the detection of a problematic request, it prevents the occurrence of a large problem that will affect the system in general by giving an alternative response to the client. In *bulkhead pattern,* by separating and isolating both suitable components and data from each other, it is ensured that the problems encountered in any group will not affect those in the other group.

5.3. Survey of MSA

With the development of microservices and distributed architecture, the need for practitioners to develop common solutions to problems arose. Due to these needs, many products have been or are still being developed by various companies or communities. Knowing what purpose these developed solutions serve and where they are located in the MSA enable us to design the architecture more comfortably and to make our architecture more robust. Therefore, we think that showing the feature set we have determined in the field of microservices to match the relevant technologies will benefit practitioners greatly.

As is seen in Table 5.2, common solutions have been developed by various companies or open-source communities for many features. We observe that no technology has been developed for some feature sets and they are more design-oriented feature sets. In other words, for these feature sets, it is recommended to apply the design decisions specified in the feature instead of a solution. For example, database per service or shared database, which are two different patterns in data management and consistency, is entirely a design decision about how you will position the data layer.

Table 5.2. Mapping features with MSA technologies

Feature	Technology/Product/Service
Testing/Chaos Engineering	Chaos Monkey Chaos Toolkit Simian Army
Testing/Service Component Test	Spring Cloud Contract Test
Resilience and Fault Tolerance/Circuit Breaker	Netflix Hystrix Resilience4j
Communication Style/API Gateway	Nginx Netflix/Zuul Spring Cloud Gateway
Communication Style/Domain Spesific Protocol	SMTP IMAP
Communication Style/Async Communication	Apache Kafka Rabbit MQ Active MQ
Observability/Log Analysis	Kibana Datadog LogDNA
Observability/Distributed Tracing	Zipkin Datadog OpenCensus Sentry LogDNA
Observability/Monitoring	Prometheus Graphite Grafana InfluxDB Zabbix
Observability/Log Aggregation	Kibana Datadog LogDNA
Observaility/Exception Tracking	Sentry
Provisioning and Configuration Management	Ansible Chef Puppet SaltStack

Provisioning and Configuration Management/Infrastructure as Code	Terraform
Security/Authentication	CAS Spring Security SSO
Security/Authorization	JWT Spring Security
Decomposition/Decompose by Subdomain	Domain Driven Design
Service Orchestration	Kubernetes Apache Mesos + Marathon Docker Swarm
Service Mesh and Sidecar Pattern	Istio Linkerd Envoy Redhat Openshift
Deployment/CI & CD	Jenkins CircleCI Travis DroneCI Gitlab CI Bamboo
Deployment/Container	Docker LXC
Deployment/Virtual Machine	VMWare VirtualBox
Load Balancing/Server-side	Nginx Zuul Eureka
Load Balancing/Client-side	Ribbon Client
Service Discovery/Service Registry	Eureka Zuul Consul Apache Zookeeper
Service Discovery/Server-side	Eureka Zuul Consul Apache Zookeeper
Service Discovery/Client-side	Ribbon Client

We also observe that some features are taken together, and solutions are proposed accordingly. Such solutions suggest a more comprehensive solution for one or more features. For example, provisioning and configuration management are two separate activities that can be considered as a continuation of each other. One cannot be thought of without the other. Therefore, it makes sense to propose a solution that handles these two features together while recommending a solution. Instead of learning and using multiple technologies and integrating them, it is a more preferred way for developers to use the ready-made solution. In such cases, if these solutions are suitable for all sub-features of the relevant parent feature, these solutions are shown at the parent level. If it does not fit all children, these technologies are shown in the feature diagram for each feature. It is also observed that some solutions may be not only for siblings, but also for features in different feature families. For example, the solutions suggested in load balancing and service discovery are taken together for both features, and solutions addressing these two features are produced.

5.4. Analysis of Existing Key Cloud Providers

In parallel with the development of the distributed architecture and MSA, cloud computing is also improving. Cloud providers develop managed services for the difficulties brought by the distributed architecture and make them ready to be used in related architectures. Users configure and use the relevant services and are not interested in many quality indicators because the services give warranties on many basic quality indicators and developers focus more on domains and business rules. However, it should be decided by considering the cost of the usage of cloud services.

In this section, the services provided by Amazon AWS, Google Cloud Platform, and Microsoft Azure, which are the three most preferred cloud providers today, have been examined and which solutions are available for which functions are presented.

As is seen in Table 5.3, services are offered by cloud providers for many features. These services are provided as managed by the cloud provider, so you can start using the services by making the necessary configurations. The services have the ability to work in harmony with each other and can be easily configured as interoperable. In this respect, using these products will give us speed. Another important point is the fact that services are developed parallel to each other by all three providers. You can find the equivalent of each service in another provider. Here, which one will be selected can be concluded by making some more detailed evaluations within the scope of performance, usability, use cases, and cost analysis. The absence of a direct solution for some features indicates that it must be a feature that needs to be programed and designed, that is, it cannot be fully managed by the cloud provider. They are more conceptual and design-oriented features. However, by bringing together more than one service, the design criteria recommended within the scope of these features can be met.

Table 5.3. Feature-based service comparison: AWS, Google Cloud, and Microsoft Azure

Feature		**AWS**	**Google Cloud**	**Microsoft Azure**
Communication Style	Async Communication	AWS MQ AWS SNS AWS SQS AWS Kinesis	Google Dataflow Google Pub/Sub	Azure Queue Storage Azure Service Bus Azure Event Grid Azure Event Hubs
	API Gateway	AWS API Gateway	Google Apigee	Azure API Management
Service Orchestration		AWS ECS AWS EB AWS EKS	Google Cloud Run Google App Engine Google Kubernete Engine	Azure Container Instances Azure App Service Azure EKS
Service Orchestration	Deployment/CI and CD	AWS CodeDeploy AWS CodeBuild AWS CodePipeline	Google Cloud Build	Azure Devops
	Deployment/Serverless Function	AWS Lambda AWS Step Function	Google Cloud Function Google Cloud Composes	Azure Durable Azure Functions
	Deployment/Container	AWS Fargate AWS EKS	Google Cloud Run Google Kubernete Engine	Azure Container Instance Azure Kubernete Service
	Deployment/VM	AWS EC2	Google Compute Engine	Azure VM

	Auto-scaling	AWS EC2 Auto-scaling	Google Computer Engine Auto-scaling	Azure Virtual Machine Scale Set
	Load Balancing/Server-side	AWS ELB	Google Cloud Engine Load Balancing	Azure Load Balancing
	Service Discovery/Server-side	AWS Route 53 AWS Cloud Map AWS ELB	Google Cloud DNS Google Cloud Engine Load Balancing	Azure DNS Azure Load Balancing
Service Mesh and Sidecar Pattern		AWS AppMesh	Google Anthos Service Mesh	Azure Service Fabric Mesh
Observability	Log Analysis	AWSElasticsearch Service AWS Redshift AWS Quicksight AWS Athena	Google Elasticsearch Service Google BigQuery	Azure Elasticsearch Service Azure PowerBI Azure Data Lake Analytics
	Exception Tracking	AWS CloudWatch	Google Cloud Debugger Google Cloud Trace	Azure Application Insights Azure Monitor
	Log Aggregation	AWS CloudWatch	Google Cloud Logging	Azure Application Insights Azure Monitor
	Audit Logging	AWS CloudTrail AWS Config	Google Audit Logs Google Cloud Asset Inventory	Azure Monitor
	Distributed Tracing	AWS X-Ray	Google Cloud Debugger Google CloudTrace	Azure Monitor
	Monitoring	AWS CloudWatch	Google Cloud Monitoring	Azure Monitor
Provisioning and Configuration Management		AWS CloudFormation	Google Cloud Deployment Manager	Azure Resource Manager Azure VM extensions Azure Automation
Security		AWS Cognito	Google Firebase Authentication	Azure Active Directory B2C

6. REFERENCE ARCHITECTURE

A Reference Architecture is a representation of the architecture of a system or collection of systems that serves as a guide for developing application architecture. It captures the key principles, elements, and relationships of the architecture, and helps to ensure that the architecture is maintainable, scalable, and aligned with the needs of the stakeholders.

After the development of the family feature model with domain analysis, the next step is to develop the reference architecture. The reference architecture has been built on domain knowledge that was formed as a result of domain engineering activities. The most important factor that increases the usability of the reference architecture is that the reference architecture is documented with the viewpoints which address the concerns of all stakeholders. We chose Views and Beyond by Clements et al. [38], which is a well-defined framework for documenting software architecture. In the Views and Beyond (V&B) approach, there are numerous defined styles (views) that address different concerns of stakeholders. To select the appropriate styles for our study, we conducted a survey with 50 practitioners with experience in software architecture design and distributed architecture, working in various positions at different software companies (as shown in Table 6.1.). All 17 viewpoints of the V&B approach were presented to the participants.

Firstly, we explained each view with detailed information to the participants and then asked them to what extent they were concerned with each defined view in Views and Beyond. Participants chose one of the answers for each view in the survey. The first option "d" indicated that detailed information was requested by the participant, the second option "s" indicated that some detailed information was requested by the participant, and the third option "o" indicated that overview information was requested by the participant. Table 6.1 presents the survey results for the views with at least one "s" or "d", where d: detailed information, s: some detailed information, o: overview information.

Table 6.1. Survey results for selecting architectural design styles

Main Stakeholder	Layered View (Style)	Decomposition View (Style)	Deployment View (Style)	Service-Oriented Architecture View (Style)
Software Architect	d	d	d	d
Software Developer	s	d	s	d
DevOps Team member	d	d	d	d
QA	o	d	o	o
Project Manager / CTO / Engineering Manager	o	d	o	o

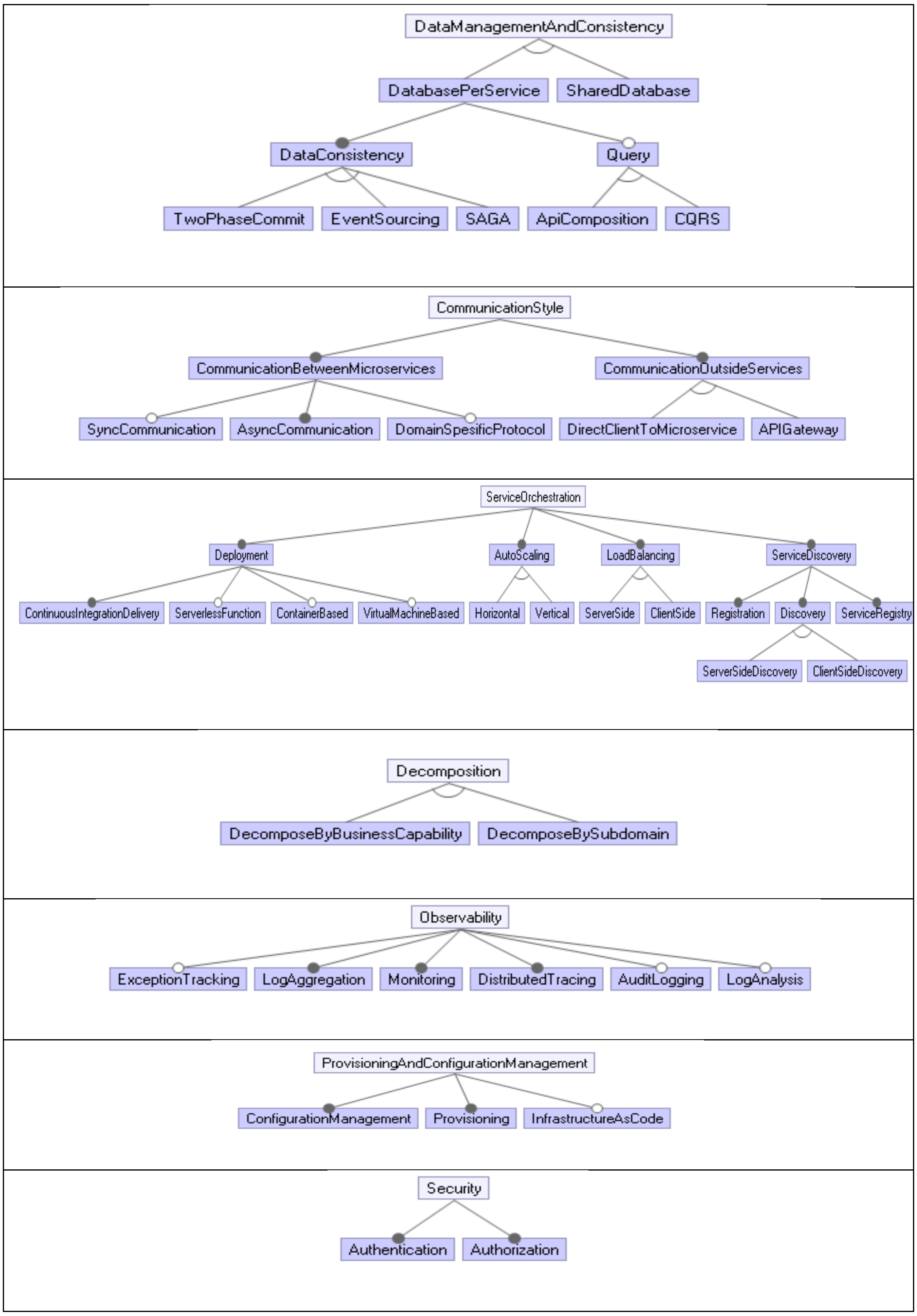
We analyzed the architectures described by key vendors. These architectures were generally not defined using architectural views and were specific to the vendor. Therefore, it was important to identify and make visible the components and relationships that are necessary in a vendor- and platform-agnostic microservice architecture. Through our vendor analysis, we examined the architectures of three major vendors (AWS, Google Cloud and Microsoft Azure) in the field of MSA and determined which solutions address which problems.

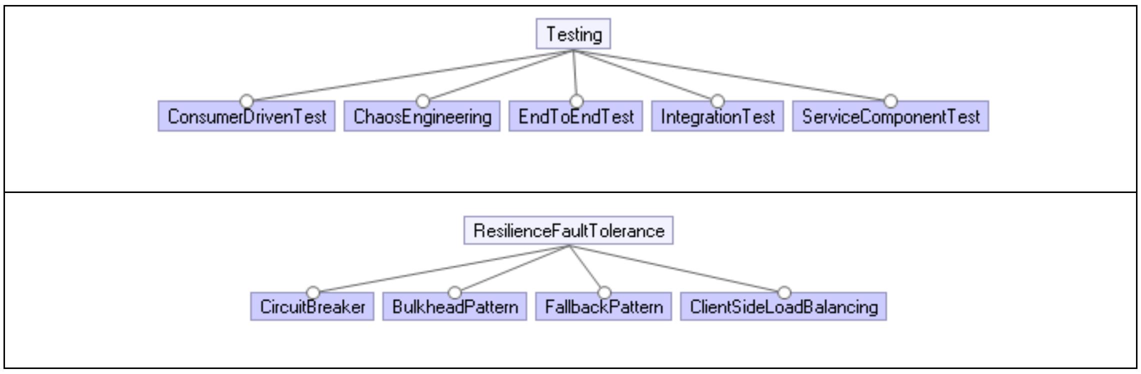
Additionally, we used the output of the domain analysis process. Through our research of sources in multi-vocal literature and interviews with experts in the field, we completed our domain analysis and modeled the domain using the feature-driven approach. We incorporated all of these outputs into the design of the Reference Architecture.

6.1. Family Feature Model

In our previous study [19], we proposed a Feature Driven Model as shown in Table 6.2, which resulted from our domain analysis. Including this model in the Reference Architecture as a guide for determining which components should make up the application architecture can be a useful practice for practitioners. This way, the components that are necessary in the system as a whole and the relationships between them can be defined first, and the application feature model can be generated from the family feature model.

Table 6.2. Family Feature Model of MSA





6.2. Decomposition View

Decomposition view expresses responsibility segregation of the system. It relies on separation of concerns principle. Responsibilities are separated across modules and each module has its own sub-modules. It belongs to Module Style of "Views and Beyond" approach. It is also a guide for other views because it contains valuable information about the overall structure of the system. In our proposed method, the decomposition view is obtained based on the features in the family feature model.

The Family Feature Model can address features as implementation units, but some features may also be considered as design patterns and have no direct implementation unit counterpart. Additionally, multiple features may come together to form an implementation unit.

The decomposition view of the microservice reference architecture is shown in Figure 6.1. Each colored part in the figure represents the main modules. Different modules under the same color are sub-modules of the relevant main module. In this case, decomposition view represents all the sub-modules.

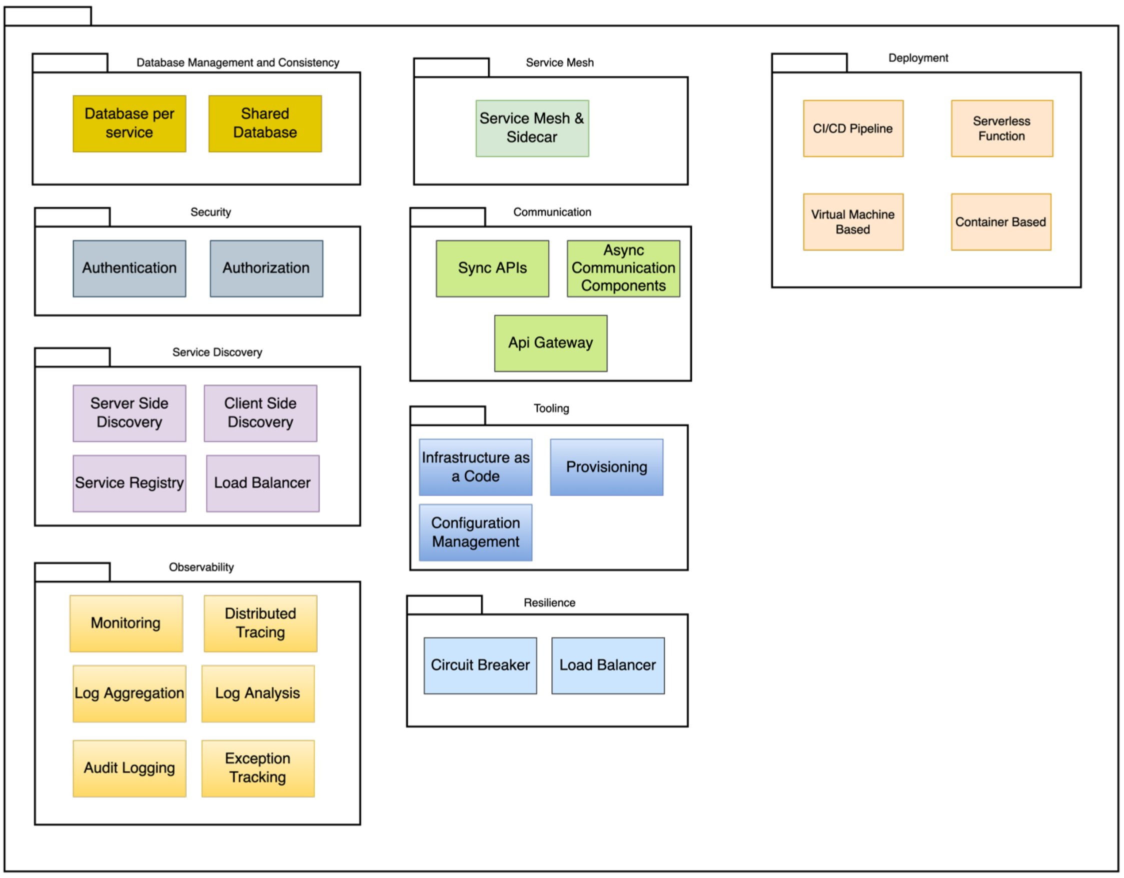


Figure 6.1. Microservice Reference Architecture – Decomposition View

6.3. Layered View

Layered architecture is another modular style. A layered view is created by ordering the modules specified in the decomposition view according to the layers. Layers are logical groups that can help practitioners create and communicate their architecture. Also, each layer provides a cohesive set of services [38].

Figure 6.2 presents the layered view of the microservice reference architecture. It composes several layers and each layer is responsible for different concerns. As seen in the figure, the layered view consists of many sidecar layers, for example Testing, and Authorization & Authentication. This means that these modules can be associated with one or more of the horizontal modules, so it is represented vertically. The first request to a distributed system developed with a microservice architecture is met with the Communication with outside layer, and then this request is forwarded to the appropriate service as a result of service discovery and load balancing processes. In business rules spread over many services, communication between services is done with an approach that takes place under the Communication between microservices module. At this time, since instances of a microservice could be found in many nodes in a distributed environment, and when it is thought that they are constantly dynamic, it becomes important to distribute the load equally. With the help of client-side service discovery, it is tried to distribute the load equally to microservices, and a more available and resilient system is obtained. The service registry knows which service has which IP (Internet Protocol) and port information. Both types of service discovery use the registry and learn which services are standing by which instances and what is the IP and port information. Considering that microservices may have different local transactions for a use case, one or more of the techniques in the Data Management and Consistency layer are used to provide data consistency handling all the local transactions. Other sidecar layers are responsible for the jobs defined in Section 5.

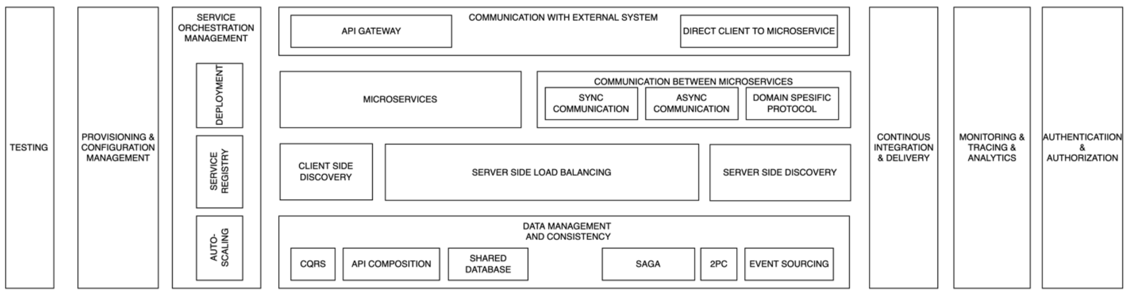

TESTING
PROVISIONING & CONFIGURATION MANAGEMENT
SERVICE ORCHESTRATION MANAGEMENT
DEPLOYMENT
SERVICE REGISTRY
AUTO-SCALING
COMMUNICATION WITH EXTERNAL SYSTEM
API GATEWAY
DIRECT CLIENT TO MICROSERVICE
MICROSERVICES
COMMUNICATION BETWEEN MICROSERVICES
SYNC COMMUNICATION
ASYNC COMMUNICATION
DOMAIN SPESIFIC PROTOCOL
CLIENT SIDE DISCOVERY
SERVER SIDE LOAD BALANCING
SERVER SIDE DISCOVERY
DATA MANAGEMENT AND CONSISTENCY
CQRS
API COMPOSITION
SHARED DATABASE
SAGA
2PC
EVENT SOURCING
CONTINOUS INTEGRATION & DELIVERY
MONITORING & TRACING & ANALYTICS
AUTHENTICATIION & AUTHORIZATION

Figure 6.2. Microservice Reference Architecture – Layered View

6.4. Deployment And Service Oriented Architecture View (Combined View)

Decomposition and Service Oriented Architecture View is a combined view. The Decomposition view is in the Allocation style, while the Service Oriented Architecture view is in the Component & Connector (C&C) style. They are styles that are frequently used in combination with distributed systems [38]. While the layered and decomposition views focus on the modular decomposition of a software system and how these modules interact with each other in terms of layers view, the deployment view deals with how these modules are allocated to the hardware of the computing platform [38]. These modules are native to the C&C style [38]. In distributed and service-based architectures, since each service can be deployed independently and also distributed, it can also be represented as separate independent deployable components (units) as shown in Figure 6.3.

Service Oriented Architecture style is under the C&C style that defines the relationships between distributed services with the help of interfaces they provide. While the components in the distributed architecture provide services for other components, other components are also responsible for consuming the services provided to them.

Figure 6.3 shows the combined view of deployment view and service-oriented architecture view. As seen from the figure, each component can be deployed independently to the identified machine in the cloud or on-premise. Each component is a deployable component and some of them are deployed to the given component as shown in the figure. On the other hand, services communicate with each other through the Application Programming Interfaces (API) they provide and ensure the integrity of the system. Table 6.1 provides more detailed information about each component and its relations.

Table 6.3 Explanation of each component in Deployment and Service Oriented Architecture View

1	API Gateway	It is a single-entry point for all clients and a place between clients and backend services. It is responsible for transmitting client requests to the backend services. It usually acts as a load balancer and forwards the request to the appropriate backend services. In addition, it performs authentication and authorization controls on the incoming request, and if the request is not secure, it cancels the forwarding request to the backend services. The benefit of this is, for example, when communicating with the outside over HTTPS, HTTP can be used inside. Also, metrics are collected from the incoming request by performing some tracing and logging operations on API gateway. If rate limiting is desired, it can be done via API gateway. But the most critical point to consider when using it is the single point of failure situation. If a single instance is deployed, this risk is too high. In order to rule out this risk, a load balancer that will understand the load coming to API gateways can be included in the system, and if the API

		gateway is down, a new instance must take over. It is not the only method for microservices to communicate with outside. If desired, clients can directly communicate with Backend (BE) services, but this is not a recommended method because all clients have access to all services, creating a security vulnerability and considering that BE services are dynamic, and IP and port information are also not fixed. Such cases are the limitations of this method.
2	Authentication & Authorization	It is responsible for authenticating and authorizing users to reach the backend services.
3	Load Balancer	It is used for distributing the requests to backend services which is considered healthy by looking up Service Registry. In order to eliminate the single point of failure, a new instance of load balancer must take over in the case of failure. The load balancer is a very important component to increase the availability and scalability of the system. It can be used as two different deployable components with API Gateway, or it can be used without API Gateway in some cases. In this case, it is possible that the concerns addressed in the API gateway will be moved to the backend services, but this is not a recommended approach.
4	Service Registry	It is a key value database containing network information of services. The API it provides is used by many components as shown in Figure 4.
5	Service Orchestrator	It is the most comprehensive component, addressing lots of critical concerns, such as auto-scaling, service discovery, resource management, load balancing, container availability, resiliency and deployment. It focuses on the methods and concepts to handle all of these concerns by the help of the agents in the application servers.
6	Service Orchestrator Agent	It is for caring about the state of the services and sharing this information with the Service Orchestrator.
7	Messaging Platform	It is used in asynchronous communication between microservices. It manages messaging by providing APIs for producing messages and consuming messages by other microservices. Information on which node will consume the messages can be obtained from the Service Registry. Successfully handling the cases where it is important to consume the messages in the same order as they are produced and trying to consume the message again in case of failure are the subjects of this module.

8	Caching	It is high-speed storage of a subset of data. It is used for many purposes, especially by microservices, but could be used by other components.
9	Microservices	They are small, loosely coupled and independent services organized around business capability. They are deployed to the application servers. They can produce or consume messages for interaction with other microservices. Besides that, they can use Service Registry to distribute the synchronous requests to other services.
10	Application Server	It is a server that hosts microservice applications
11	Monitoring & Analytics Agent	It is used for collecting information about the modules where it runs. It is responsible for sharing this information with the Monitoring and Analytics Server.
12	Monitoring & Analytics Server	It is responsible for storing all kinds of logs, metrics, traces, events etc. Besides, visualizing and querying the logs are also among its responsibilities. It also provides some services to set up alerting mechanisms.
14	Database Server	It is a server that runs database application.

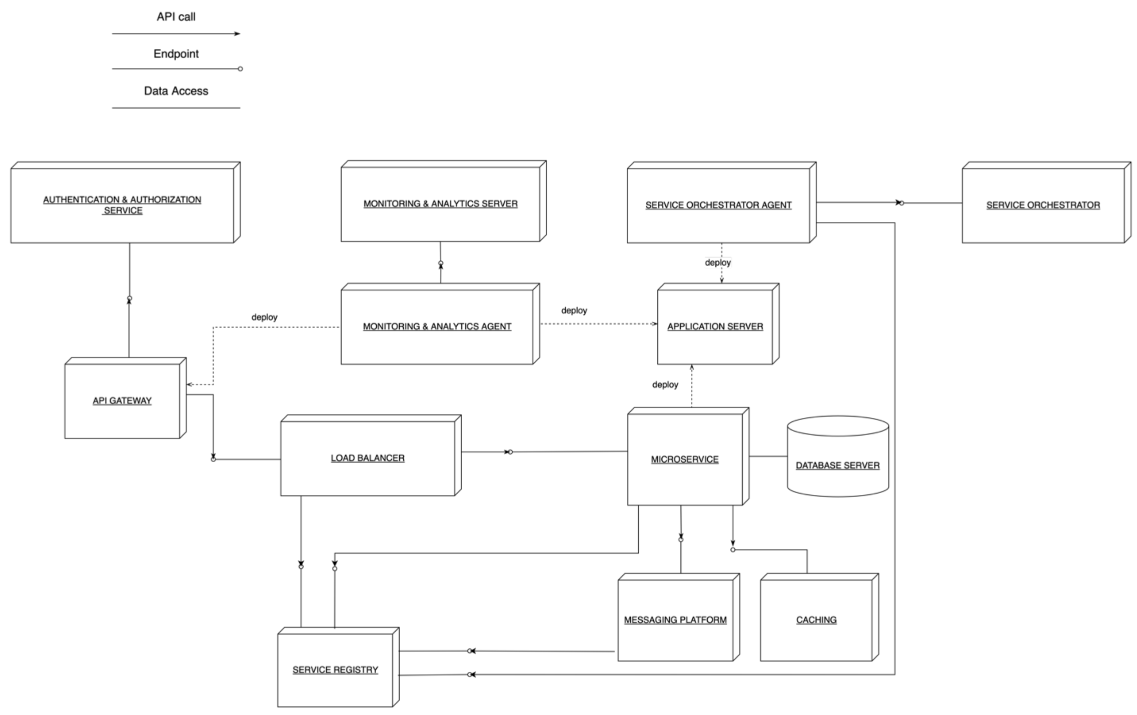


Figure 6.3. Microservice Reference Architecture – Deployment & Service Oriented Architecture View

6.5. Method To Derive Application Architecture From Reference Architecture

Application architecture is implemented by using the family feature model and microservice reference architecture, which are the result of domain engineering activities explained in the previous sections. Figure 6.4 represents the method to derive an application architecture from the reference architecture. While creating the application architecture, the reference architecture and family feature model can be updated according to the feedback received. Before starting to develop the application architecture, the requirements collected from stakeholders are analysed and the relevant features are selected from the family feature model in order to derive the application feature model. The next step is to develop the application architecture addressing software components that correspond to the features in the family feature model. Here, application architecture is derived using the views (styles) in the reference architecture.

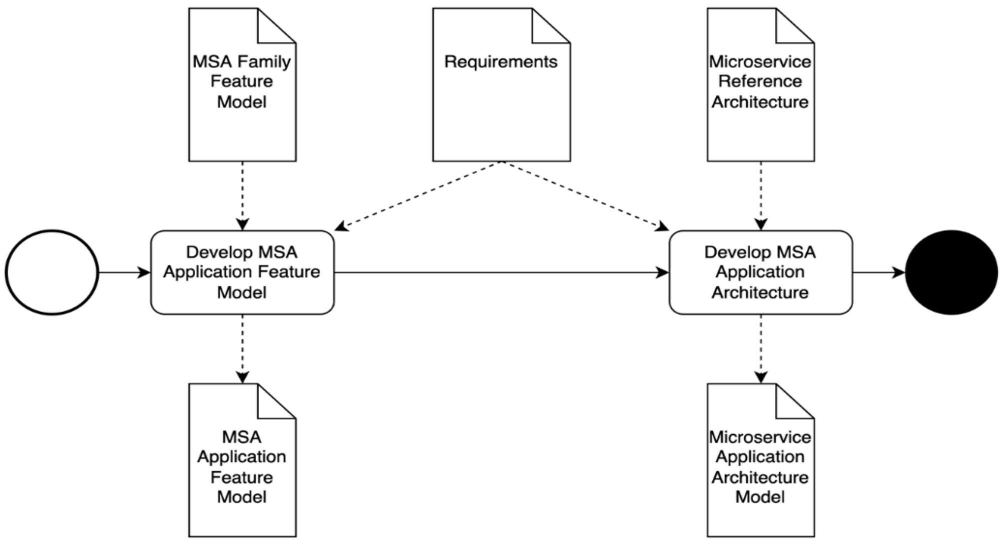

MSA Family Feature Model
Requirements
Microservice Reference Architecture
Develop MSA Application Feature Model
Develop MSA Application Architecture
MSA Application Feature Model
Microservice Application Architecture Model

Figure 6.4. Method to derive application architecture from reference architecture

There are three potential circumstances for applying a reference architecture, according to Kassahun [65], as shown in Figure 6.5. The reference architecture is used as a starting point for deriving an application architecture. Practitioners can add a new module for a new feature that they concern about, if they cannot find the corresponding module in the reference architecture. If there is already a module that fits their needs, they can either use it as it is, or they can change it according to their concerns and needs by decomposing it or combining it with other modules, etc. Finally, the application architecture is obtained.

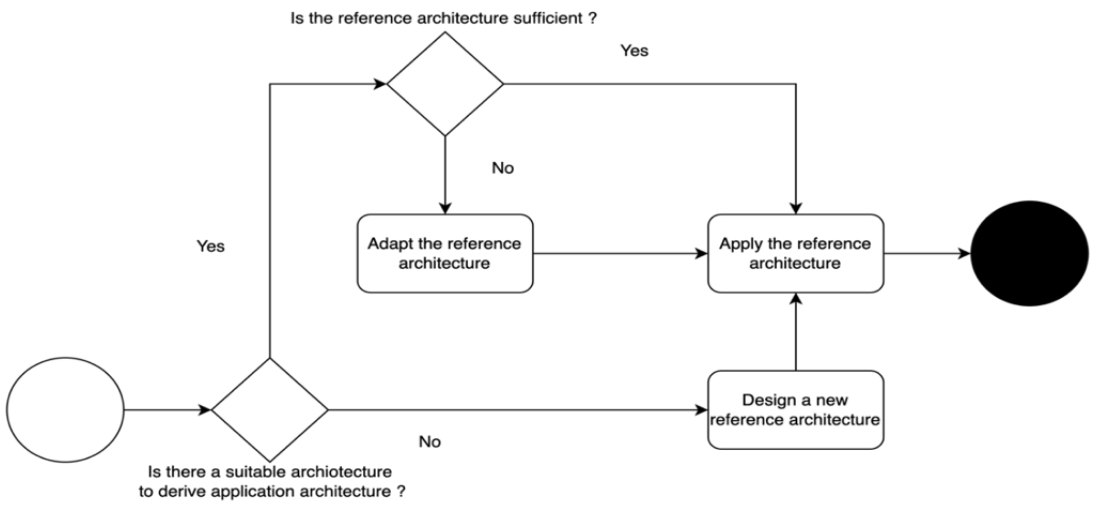

Is the reference architecture sufficient ?
Yes
No
Yes
Adapt the reference architecture
Apply the reference architecture
Design a new reference architecture
No
Is there a suitable archiotecture to derive application architecture ?

Figure 6.5. Three scenarios of using reference architecture to derive an application architecture adopted from Kassahun [65]

7. VALIDATION BY MULTI-CASE STUDY

In this section, microservices reference architecture is evaluated using case study research. In this evaluation, our architecture designs based on viewpoints and feature diagrams are used. Section 7.1 describes the case study protocol. Section 7.2 and 7.3 present the results from the two cases respectively.

7.1. Case Study Protocol

We used the case study empirical evaluation technique described by Runeson and Höst [66] to validate our approach. The steps in the procedure are as follows: (1) case study design, (2) data collection preparation, (3) data collection execution on the examined case, (4) data analysis, and (5) reporting. The design of the multiple-case study is shown in Table 10.

Table 7.1. Case study design

Case study design activity	(For the two cases in the multiple-case study)
Goal	Assessing the effectiveness and practicality of the reference architecture
Research Questions	How effective is the adopted microservices reference architecture? How practical is the adopted microservices reference architecture?
Data Collection	- Observation of the process and systems - Meetings - Semi-structured interviews were used to obtain both indirect and direct data (a mixture of open-ended and closed-ended questions)
Data Analysis	Qualitative data analysis using Radar Charts

We also have adopted holistic and multiple-case design as suggested by Yin [49]. The two cases within the multiple-case design have emerged in two different software companies and have been prospective studies in which the companies intended to migrate their monolithic applications to MSA. As previously stated, the goal of the multiple-case study evaluation is to aid in the design and development process. The purpose of this case study is to assess the architecture designs that have been established, as well as the development process. Data is collected by interviewing the architects, developers and Site Reliability Engineering (SRE) team members and by observing the design and development process.

Before starting the interviews, the reference architecture was reviewed by another researcher with seven years of MSA experience and provided in its final form. Subsequently, all interviews in the case study were conducted by myself and this researcher.

We use radar charts to examine the findings of both situations using a qualitative data analysis technique. Both direct and indirect data analysis were employed. We performed semi-structured interviews with architects, developers, and SRE team members for direct data analysis, using the predetermined questions in Table 7.2. The questions contained a 5-point Likert scale for possible responses (ranging from "1: strongly disagree" to "5: strongly agree"). In addition, for each question, an additional explanation has been sought.

Table 7.2. Questionnaire used for qualitative data analysis

Questions	
1	What is your opinion on the provided application architecture?
2	Do you think that the provided recommended application architecture is of high quality?
3	Do you think that the reference architecture is of high quality?
4	Is the method and the reference architecture sufficient to derive the application architecture?
5	Do you think that the method is practical?
6	Will you use the method again?
7	Do you think that the application of the method can provide a competitive advantage to the organization?
8	Has the usage of the method enhanced your knowledge on MSA?
9	Do you have any suggestions for improving the method?
10	Do you have any suggestions for improving the feature models?
11	Do you have any suggestions for improving the reference architecture?

Both cases in the multiple-case study research have followed the steps listed below:

- An initial interview with architects, SRE team members and developers was planned first. The purpose of this interview was to get a sense of the participants' early ideas and experiences with MSA.
- In the second phase, we provided a brief presentation on the proposed method's purpose. We also briefly detailed how the approach works as well as the ultimate result.
- In the third phase, we applied the approach to both cases with the development team with several meetings and application architecture was developed incrementally by holding multiple meetings with the development team. As the last step, it was reviewed and finalized by the SRE team and the researchers. (elaborated in Sections 7.2 and 7.3).
- The researchers assessed the architecture design that emerged from the method's application to the prospective cases in the fourth phase.
- In the fifth phase, the researchers conducted a post-interview to determine the method's effectiveness and practicality.
- In the sixth phase, the researchers analyzed data from the initial interview and the post interview. The assessment was completed separately but reviewed together to determine the lessons learned.

7.2. Case Study – Transportation Management System

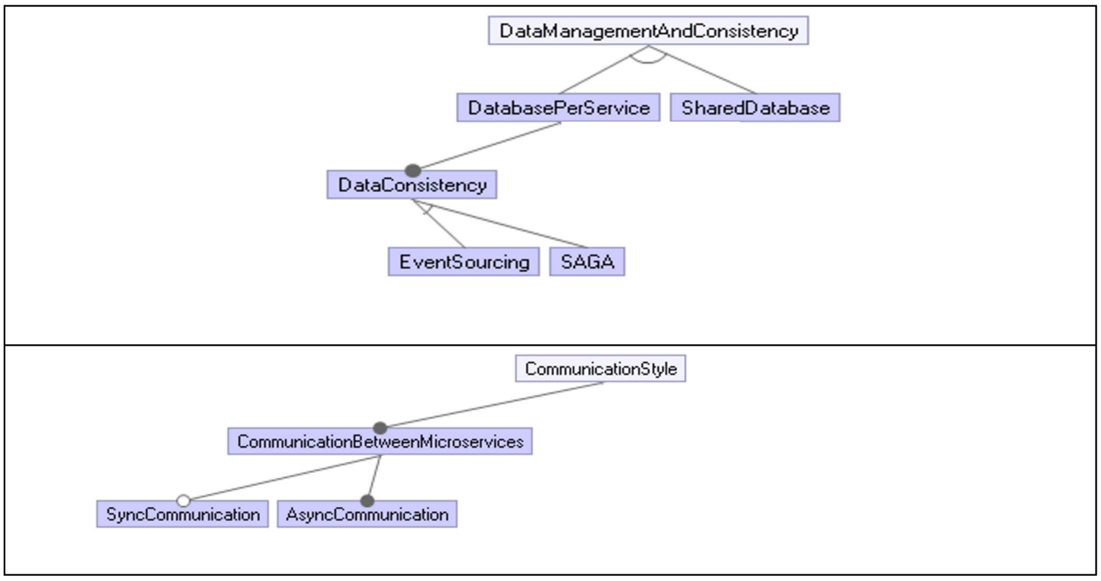
With the changes that Coronavirus brought to our lives, many activities are now carried out online. At the beginning of these activities comes the ones that require transportation. Increased transportation demands of companies have increased the interest in this sector and made it more competitive. Despite the increasing workload, companies that provide transportation services want to continue their services effectively and to better monitor and manage their processes. This situation, in turn, has increased the interest in transportation management systems. Companies want to use easy-to-use systems that best suit their concerns and manage their processes efficiently. They also want to choose the most suitable solution for them financially.

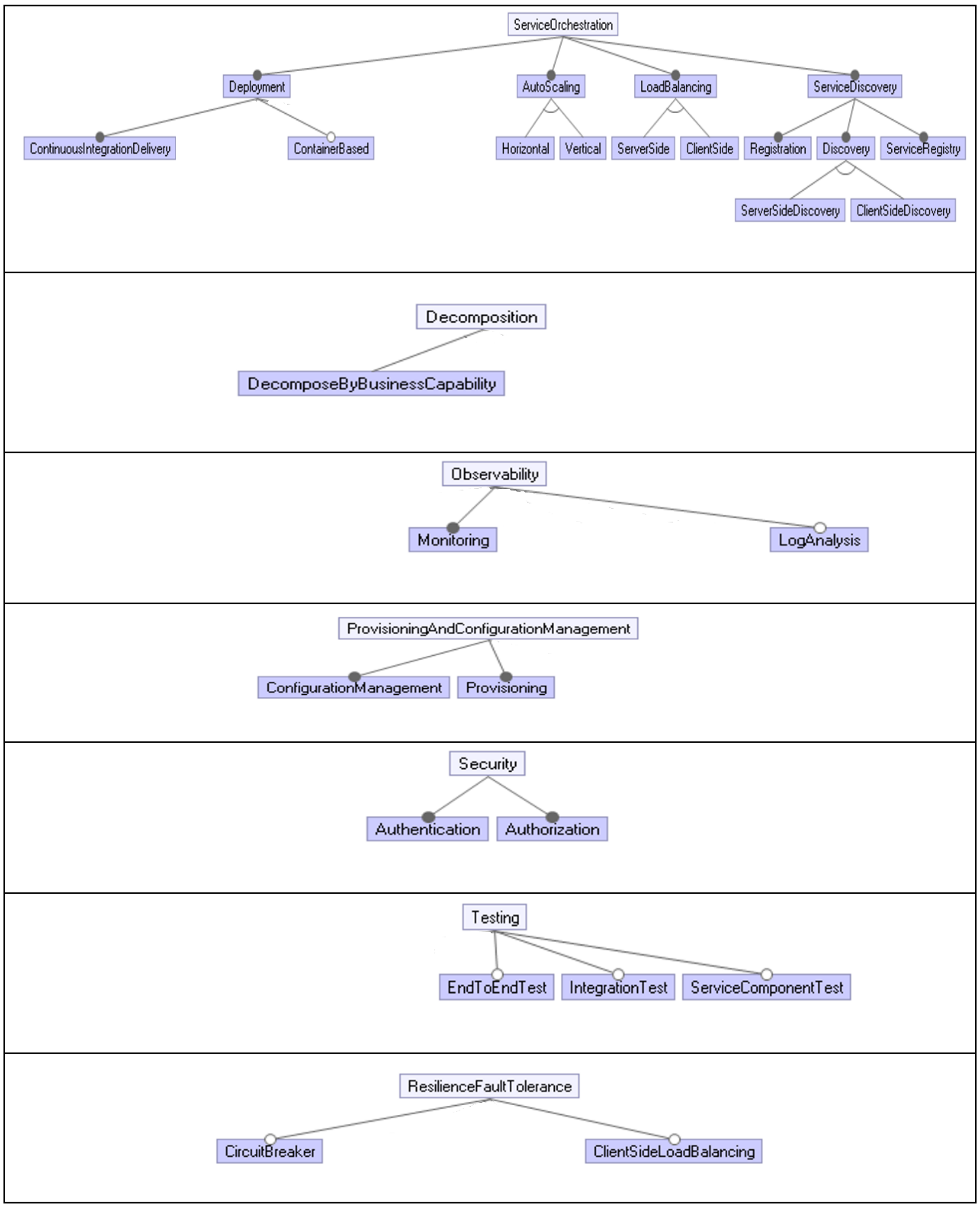
We have served our reference architecture to a software company that wants to develop a solution for all these demands. This startup company, which has a team of 15 members, develops various mobile and web-based products. The development team consists of 11 people. There are 2 SREs, 1 architect, 1 designer, 1 business analyst, and 6 developers. They have an average of six years of software development experience. They have decided to deploy their services to AWS and use its services. This was not a problem for us since our reference architecture is platform independent. The application architecture obtained after applying the case study protocol is presented in the sub-sections below.

7.2.1. Application Feature Model

This application feature model shown in Table 7.3. is derived by picking the features required for this specific case from the family feature model of MSA presented in our prior study [19] and summarized in Section 5.

Table 7.3. Application Feature Domain Model





7.2.2. Decomposition View

The decomposition view is based on this feature model as we explained how to derive the decomposition view from the family feature model in Section 6.1. Hence, only the decomposition view will be shown here.

Definitions there and information such as what the common and variable features are critical information for deriving the decomposition view. Figure 7.1 shows the derived decomposition view of the transportation management system.

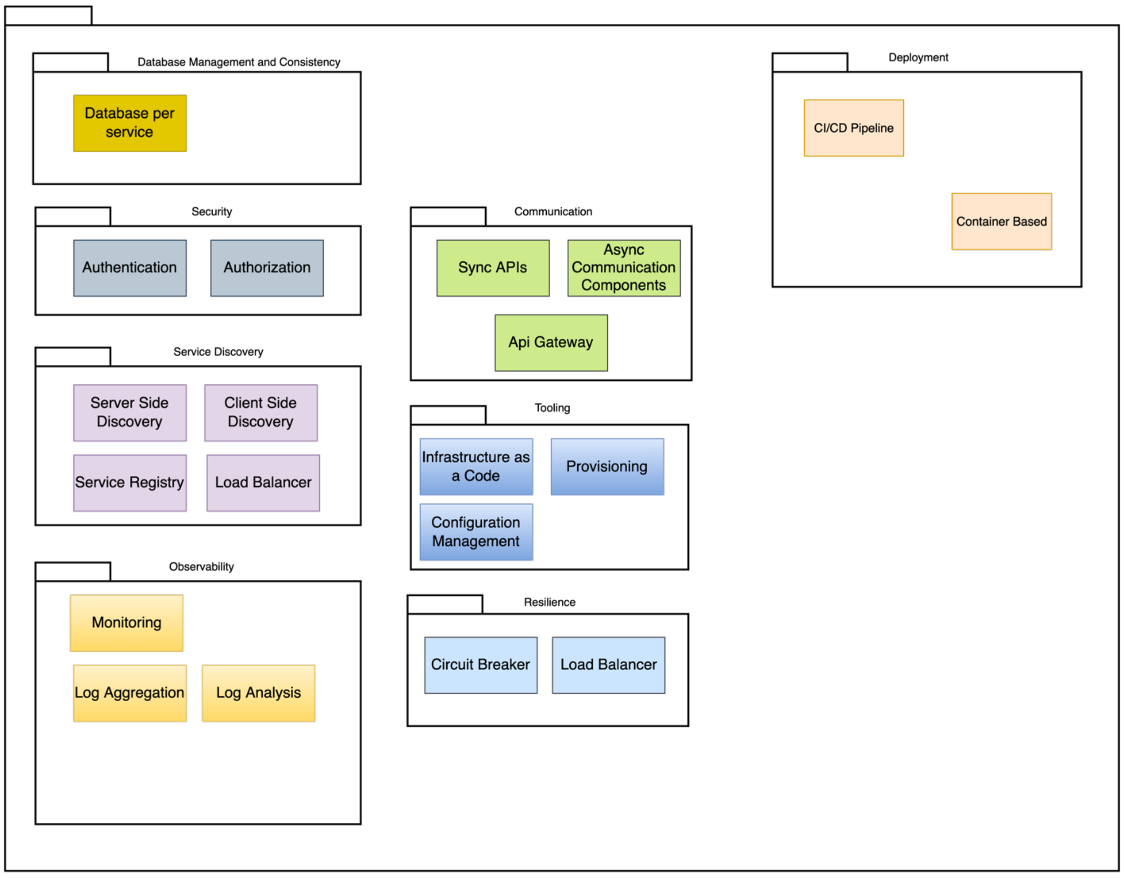


Figure 7.1. Decomposition view for Transportation Management System

This is how the architecture is shaped according to the initial needs of the stakeholders. As their understanding about the system progresses, they will be able to include other components in the design. At the moment, they did not have a need for Query, so it will not be implemented; but other than that, Database per Service will be used for data management and consistency. Monitoring and log analysis will be adapted for monitoring activities. Infrastructures will be configured for load balancing service discovery and auto-scaling. The modules used in other parent modules are also shown in Figure 7.1.

7.2.3. Layered View

This layered view, like the decomposition view, is adapted from the reference architecture’s layered view diagram shown in Figure 6.2. The layered view of transportation management system is depicted in Figure 7.2. Here, the decomposition view's modules are spread among the layers in the layered view.

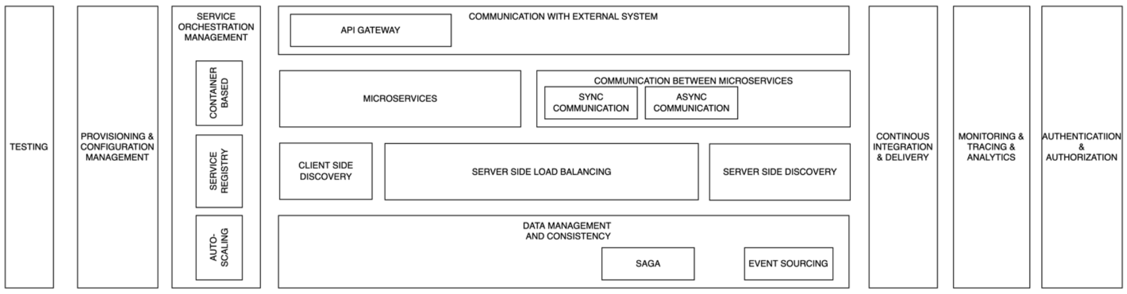

TESTING
PROVISIONING & CONFIGURATION MANAGEMENT
SERVICE ORCHESTRATION MANAGEMENT
CONTAINER BASED
SERVICE REGISTRY
AUTO-SCALING
API GATEWAY
COMMUNICATION WITH EXTERNAL SYSTEM
MICROSERVICES
COMMUNICATION BETWEEN MICROSERVICES
SYNC COMMUNICATION
ASYNC COMMUNICATION
CLIENT SIDE DISCOVERY
SERVER SIDE LOAD BALANCING
SERVER SIDE DISCOVERY
DATA MANAGEMENT AND CONSISTENCY
SAGA
EVENT SOURCING
CONTINOUS INTEGRATION & DELIVERY
MONITORING & TRACING & ANALYTICS
AUTHENTICATIION & AUTHORIZATION

Figure 7.2. Layered view for Transportation Management System

7.2.4. Deployment & Service Oriented Architecture View

The deployment and service-oriented architecture view of the transportation management system is shown in Figure 7.3. As seen from the figure, almost every deployable unit/module in the reference architecture is also included in the application architecture. Here, a specialized deployable unit is not preferred as an API gateway. AWS costs are the most important factor in reaching this decision. Instead, the load balancer will take over this responsibility. In addition, each microservice is responsible for performing authentication and authorization using JWT tokens.

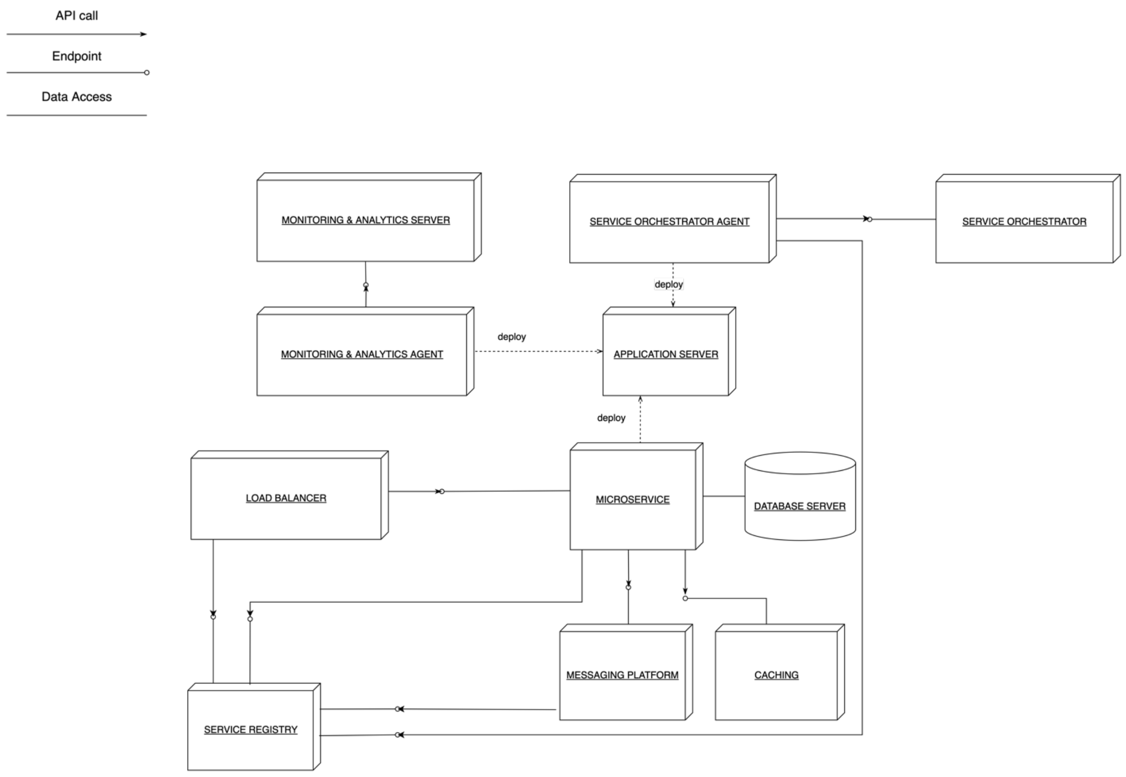


Figure 7.3. Deployment & Service Oriented Architecture view for Transportation Management System

7.3. Case Study – Remote Team Management System

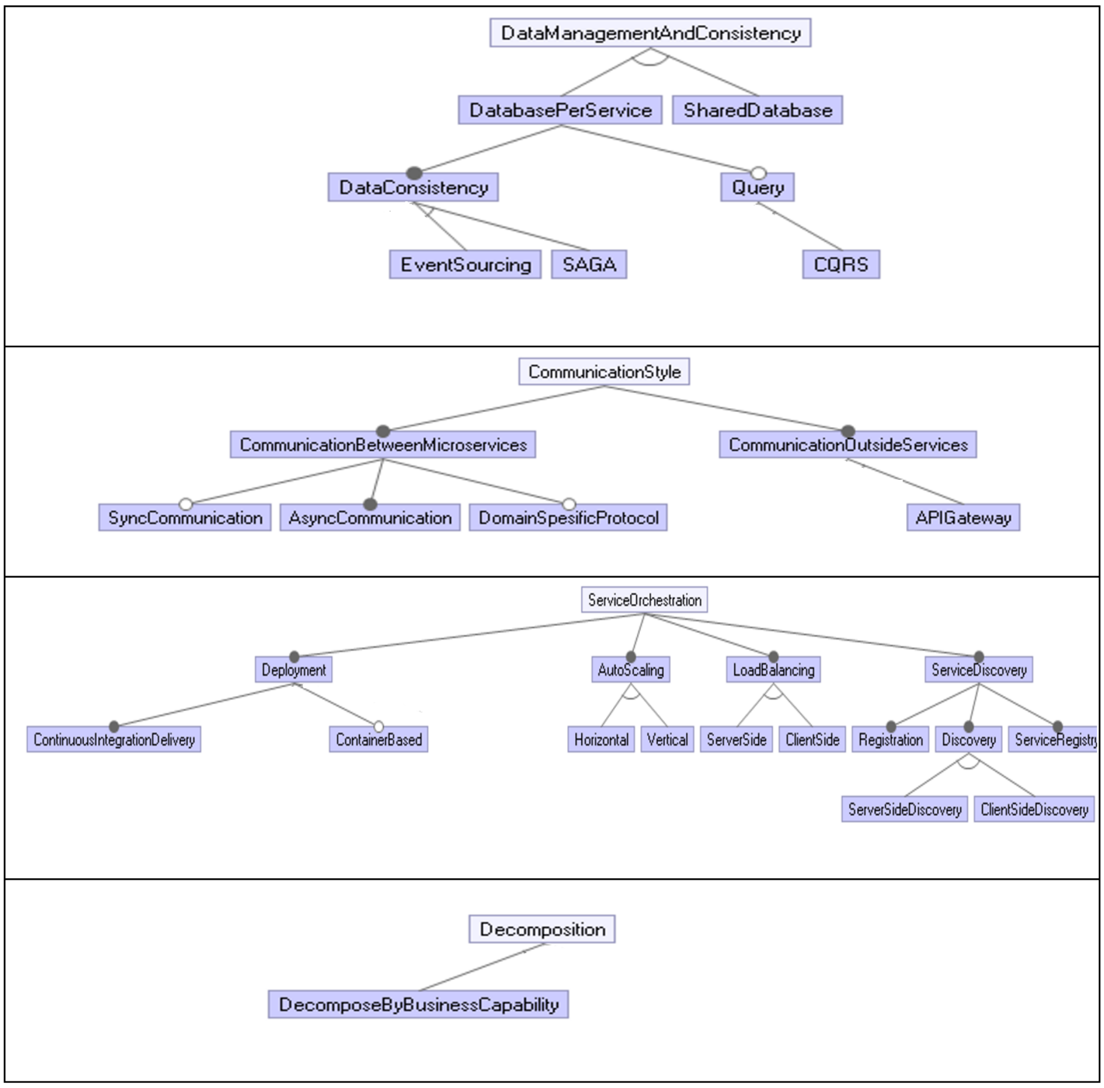
Similar to the transportation management, remote team management has also become a widespread need with the pandemic. Some companies encourage working remotely, while others set up their teams entirely from the people who can work remotely even in different continents. The number of applications developed as a solution to this need is increasing every day and this industry is becoming competitive. It is expected that the solution to be developed will provide efficient answers to many concerns like time management, project management, reporting etc., and will facilitate the daily work of the employees.

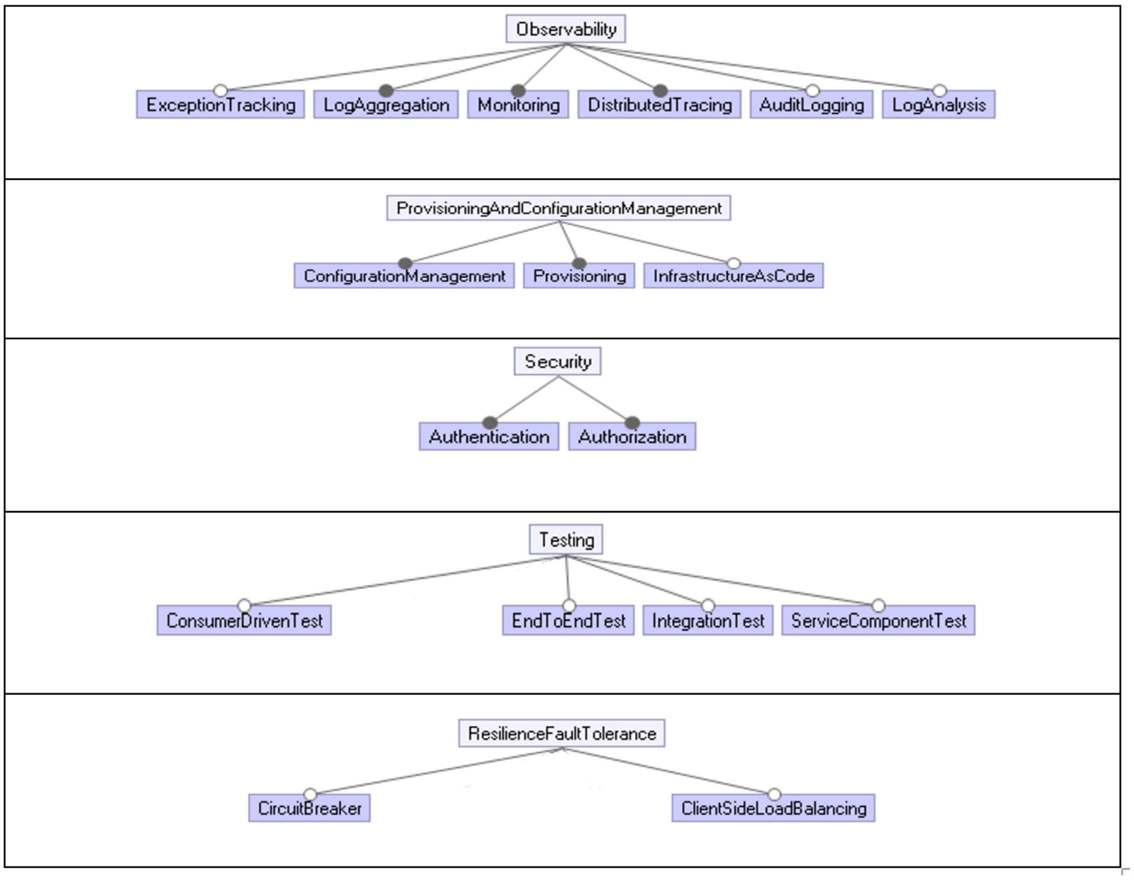
We have served our reference architecture to a software company that wants to develop a solution for managing teams remotely and efficiently. This startup company, which has a team of 11 members, develops various web-based products. The development team consists of 9 people. There are 1 SREs, 1 architect, 1 designer, 1 business analyst, and 5 developers. They have an average of eight years of software development experience. They have decided to deploy their services to Google Cloud and use its services. The application architecture obtained after applying the case study protocol is presented in the following sub-sections.

7.3.1. Application Feature Model

The selected feature set for the Remote team management case from the Family Feature Model is shown Table 7.4. As shown Table 9, CQRS is implemented to satisfy intense query needs.

Table 7.4. Application Feature Domain Model





7.3.2. Decomposition View

Figure 7.4 shows the decomposition view of the remote team management system. Database per Service for data management and consistency is used. All monitoring modules will be used for monitoring activities. Infrastructures will be configured for load balancing, service discovery and auto-scaling. Besides, instead of through manual process, managing and provisioning of infrastructure will be handled through code. The modules used in other parent modules are also shown in Figure 7.4.

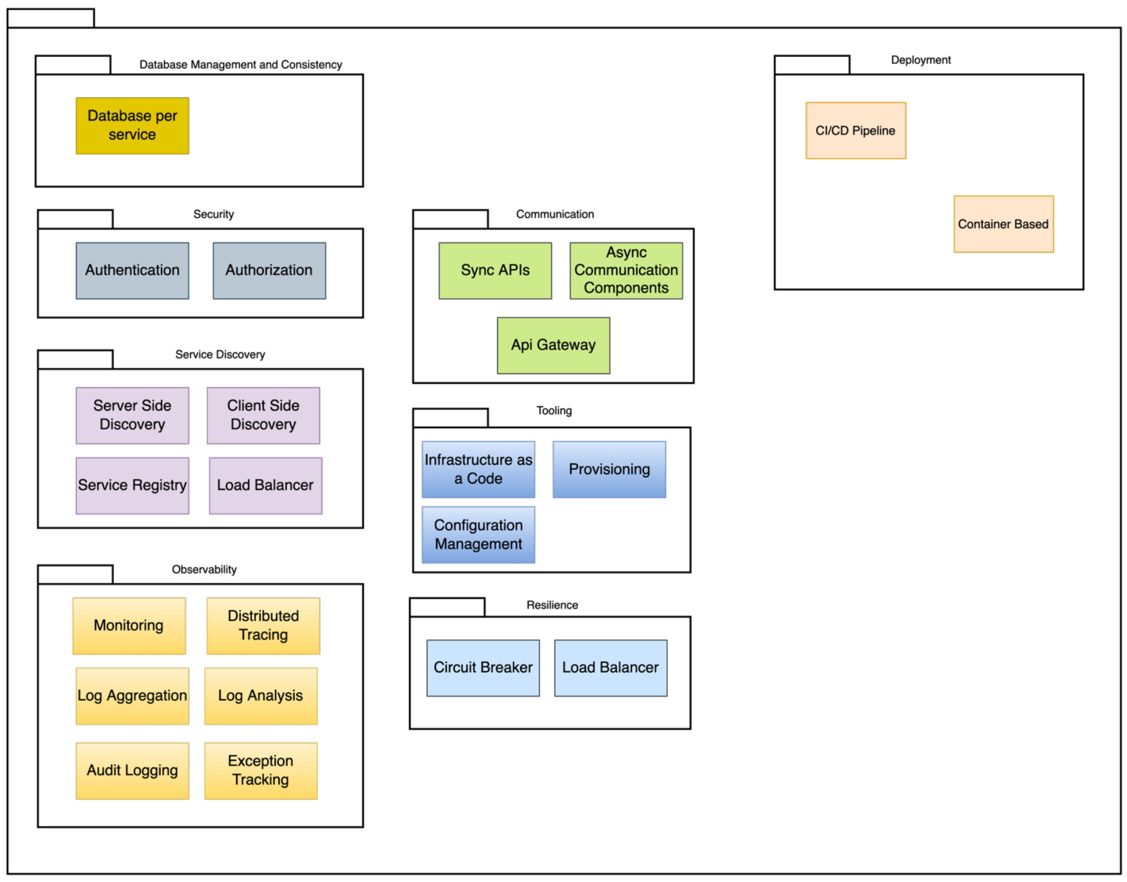


Figure 7.4. Decomposition view for Remote Team Management System

7.3.3. Layered View

The layered view of remote team management system is shown in Figure 7.5. It is customized from the layered view of the reference architecture given in Section 6.3 Similar to the previous case, the layer view of the modules in the decomposition view is given in the figure.

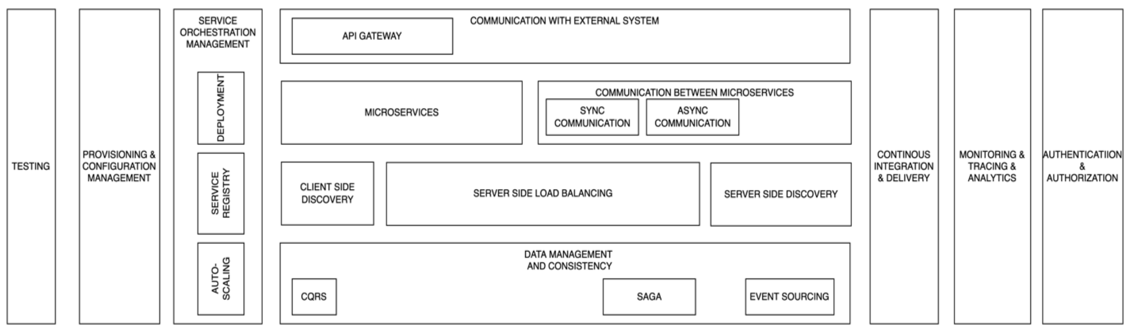

TESTING
PROVISIONING & CONFIGURATION MANAGEMENT
SERVICE ORCHESTRATION MANAGEMENT
DEPLOYMENT
SERVICE REGISTRY
AUTO-SCALING
API GATEWAY
COMMUNICATION WITH EXTERNAL SYSTEM
MICROSERVICES
COMMUNICATION BETWEEN MICROSERVICES
SYNC COMMUNICATION
ASYNC COMMUNICATION
CLIENT SIDE DISCOVERY
SERVER SIDE LOAD BALANCING
SERVER SIDE DISCOVERY
DATA MANAGEMENT AND CONSISTENCY
CQRS
SAGA
EVENT SOURCING
CONTINOUS INTEGRATION & DELIVERY
MONITORING & TRACING & ANALYTICS
AUTHENTICATIION & AUTHORIZATION

Figure 7.5. Layered view for Remote Team Management System

7.3.4. Deployment & Service Oriented Architecture View

Almost every deployable unit/module in the reference architecture is also included in the application architecture for this case. Since load balancer capability of API Gateway will be used here, it is not separately deployable. In addition, authentication and authorization will be carried out through the API gateway. In addition, the caching module will be implemented due to the need for services to access various information quickly. Figure 7.6 presents the Deployment & Service Oriented Architecture view of remote ream management system faster.

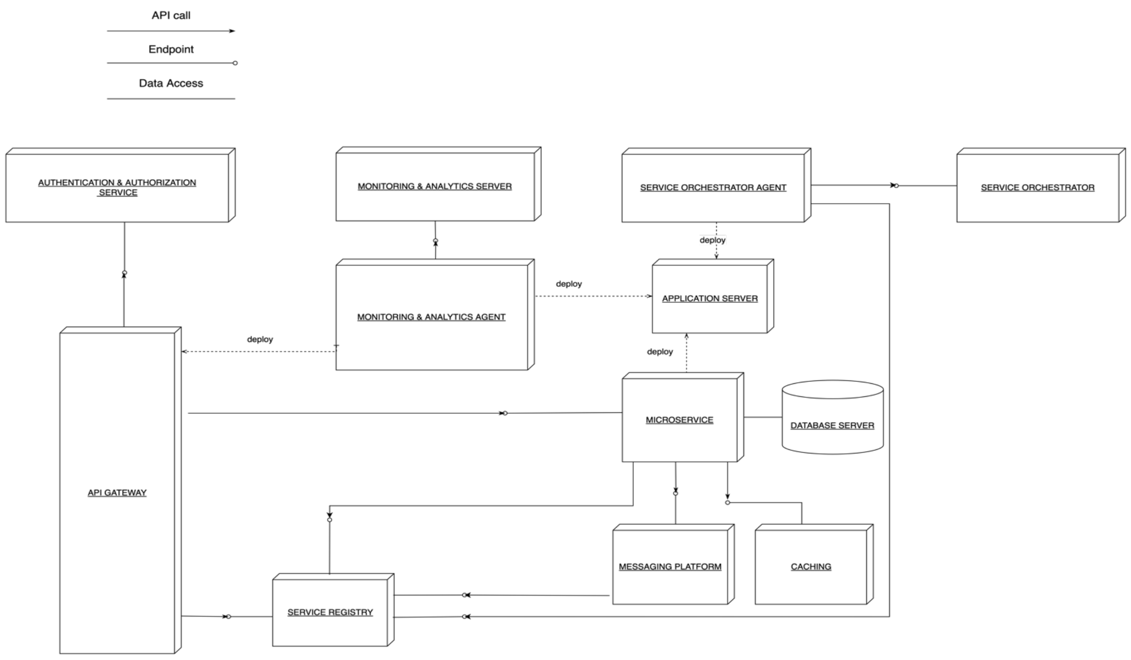


Figure 7.6. Deployment & Service Oriented Architecture view for Remote Team Management System

7.4. Discussion On Results

7.4.1. How Effective Is The Adopted Microservice Reference Architecture?

In the previous sections, we have explained how the reference architecture can be used to create application architecture in two different cases within a multiple-case study design. We used the research questions that we defined in the case study protocol to evaluate the results of the prospective cases. Below are the interview results of the first case and the second case, respectively, using the radar chart. In the interviews, we used the predefined questions that we have given in Table 7.2. During the interviews, we also asked if there was anything else the team members wanted to point out aside from their answers to the questions. The answers and comments were positive and good scores were obtained for all the predefined questions.

As seen in Figure 7.7, the scores of 4 and above in the answers to questions 1 to 3 for the first case study show that our reference architecture was effective for the given case. We also received very good feedback from the answers to questions 8 to 11. The interviewees were able to adapt the application architecture they obtained to the AWS environment in a short time and stated that the architecture provided sufficient infrastructure and guidance. They said that in the future, architectural changes might be made according to the needs, in case they could follow the reference architecture and application architecture development methodology again. However, they stated that it would be beneficial to enhance the reference architecture by adding different views in the future according to changing needs of the industry. They also stated that the application architecture was adapted comfortably, the learning curve was reduced, and the concepts and methods to be followed were well defined, which are the biggest benefits of the proposed approach.

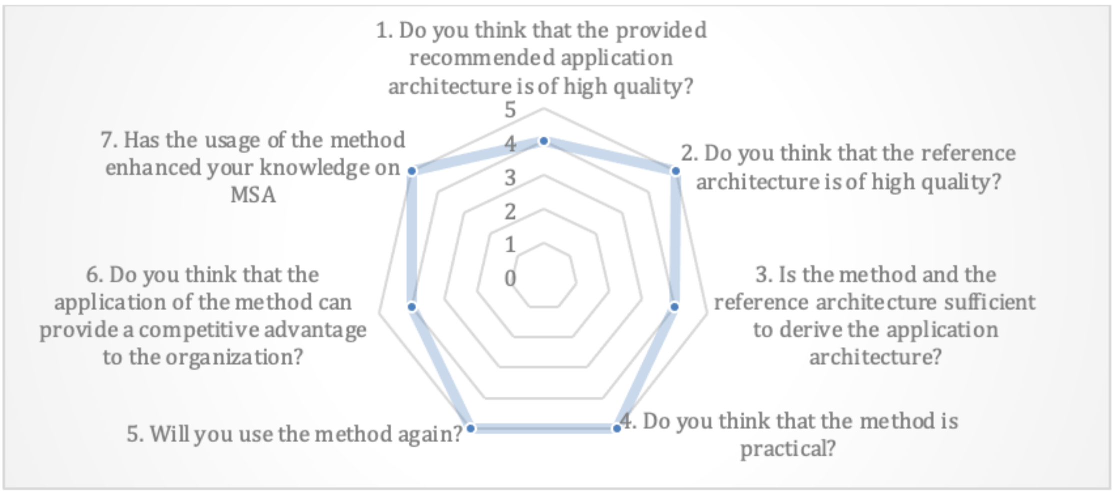


Figure 7.7. Interview results for Transportation Management System

For the second case, the results were similar as seen in Figure 7.8, where only the scores of 4 and above were rated. Unlike the first case, the architecture had been developed but had not been implemented yet. Therefore, the interviewees stated that they could apply the same approach again if any changes would be needed while implementing. The developers and architects participating in the survey stated that they could easily follow how the application architecture would be derived, and that they would start to implement the system accordingly. Additionally, they pointed out that the approach was easy to understand and gave lots of details that they would take into account while implementing.

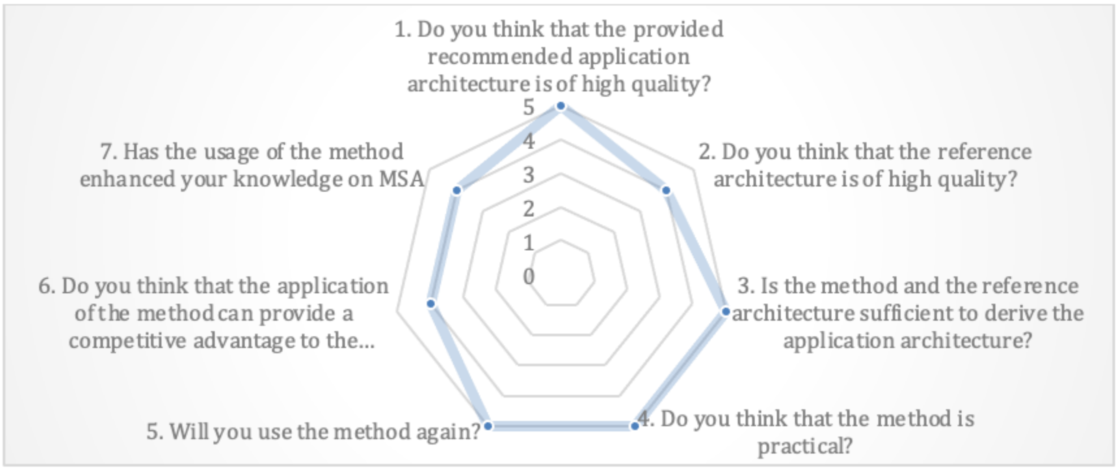


Figure 7.8. Interview results for Remote Team Management System

7.4.2. How Practical Is The Adopted Microservice Reference Architecture?

We mostly used questions from 4 to 7 and open-ended questions to improve the practicality of the approach. The responses we received were only the scores 4 or above, as shown in Figure 7.7 and Figure 7.8, and we received very positive feedback on the practicality and the ease of use of the method in both case companies.

For the first case, architects and developers stated that the family feature diagram was a quick guide about possible components and prevented wasting time searching for possible solutions. They also pointed out the fact that the reference architecture and the application architecture development approach accelerated the decision-making process and created awareness of the concepts in the MSA. They stated that the approach enabled them to make some decisions early against the changes that might adversely affect the architecture in the future.

Also for the second case, the interviewees found the approach useful and practical and they stated that it gave the team confidence for the work to be done. They also had some suggestions in terms of improving the reference architecture in the future. They suggested that giving some widely used template application architectures, in addition to the reference architecture, would enable the architecture to be implemented much faster.

7.5. Limitations And Threats To Validity

The reference architecture is meant to be general enough to be used to create various application architectures. Nonetheless, the reference architecture, like other reference architectures, does not give all the information. Similarly, the reference architecture would not cover the modules of system that required highly specific features that were not foreseen beforehand. We also focused on showcasing the reference architecture as well as the approach for creating a concrete application architecture. This seems to be handy and practical. We do not claim, however, that the reference architecture is complete, and we believe that more research is needed to enhance and develop it.

Despite the fact that we demonstrated the effectiveness and practicality of our approach in two cases in different companies, it can be taken as the base to conduct further cases. In the multiple-case study, we have not concentrated on developing the entire system due to elaborating on the design phase, which is a very critical phase of software development, and to the confidentiality reasons. However, a study focusing on implementation is considered as a future study.

The feature diagrams and reference architecture are both easily extensible. A more comprehensive reference architecture and family feature diagram can be obtained by addressing the concerns that may arise in the future, or some modules that do not affect the architecture much but are important, such as Testing, can be handled in more detail.

Another point is that the reference architecture can be expanded by adding different architectural views according to the needs that may arise in the future.

To validate our approach, we followed a systematic, multiple-case study research method. Every empirical study faces a number of possible risks to its validity. We will briefly explore them for our multiple-case study research and detail each threat with its mitigation technique(s).

Construct validity refers to the extent to which the operational measures under investigation accurately reflect the researchers' intentions and the study objectives [49]. Reference architecture development approach as shown Figure 4.1 ensures the construct validity to some extent. Table 7.5. lists the different threats to construct validity with the countermeasures to minimize or eliminate their undesired effects.

Internal validity refers to the existence of a causal link between treatment and response. The existence of a reference architecture development approach and the confirmation of feedbacks through interviews made in rounds can be considered an advantage.

External validity refers to the ability of generalization of study. Despite the fact that choosing cases from different companies and vendors for external validity can be considered an advantage, it is not enough; there is a need for more work and for improvement of suggestions.

Table 7.5. Threats to construct validity and countermeasures

Threat	**Countermeasures**
Interviewees' incorrect perceptions of the questions' descriptions	For the questions and answers, we used the ideas given by Kitchenham and Pfleeger [67]. We included thorough explanations to guarantee that each person's understanding of the questions is unique.
Incorrect understanding of descriptions of replies by interviewees, as well as incorrect answer selection	It might be difficult to tell the difference between "Strongly Agree" and "Agree". We defined each scale in detail for each Likert-scale question to mitigate this treat.
The interviewees' incorrect understanding of the open-ended questions	We double-checked the interpretation of the questions with those who were interviewed to mitigate this threat.
Researchers' incorrect interpretation of the interviewed people's responses	Both researchers were present in the interview to establish observer triangulation, which mitigated this threat.
Participants' experience in MSA in the survey conducted for which V&B views were selected	We tried to select participants worked in at least one distributed architecture-based project in their professional career.

8. CONCLUSION

Microservice architecture (MSA) brings many advantages, such as agility and autonomy, in software development. There are several studies suggesting a reference architecture for MSA in the scientific literature, but these studies have remained at an abstract level and do not suggest a method about how one should derive an application architecture. In our study, we followed a systematic method to fill this gap.

First, we made a market analysis to define the reference architecture to be taken as a basis for the application architecture and determined the needs. Then, we carried out a systematic literature review and discussed the challenges in this area, and proposed solutions to these challenges. Afterwards, we adopted a domain-driven approach to define the family feature model of MSA, which represents common and variant features. As a final step, we represented and shared the reference architecture using different architectural views. Furthermore, we provided an architectural design method, specifying how the application architecture would be derived from the reference architecture, so that practitioners could easily derive their application architectures for different types of MSA-based applications using this approach.

Multiple-case study research was applied to demonstrate effectiveness and practicality of our approach. Two the cases within the multiple-case study design adopted the reference architecture regarding their specific application needs for transportation management system and remote team management system, respectively. According to the case-study results, both the reference architecture and the method proposed to derive the application architecture were revealed to be beneficial in deriving the concrete application architectures. Moreover, the results also showed that the overall approach was found practical and effective.

Moreover, the family feature model and the characterization of features, which is a crucial step of the reference architecture, by using the Table 5.3 we provided on a cloud provider basis, we believe that this can serve as a guide for companies that currently work with a cloud provider. In case of a change in cloud provider, it is possible to determine, through utilizing the Table 5.3 again as a reference, the solutions available for each feature on other cloud providers, and how these solutions should be used by paying attention to key considerations. This guidance needs to be further evaluated with more comprehensive studies. It is important to have in mind that it is not always necessary to have one-to-one mapping between different solutions from different vendors. It could be more feasible to map the functionality that covers each feature and build the new architecture by combining different solutions and also evaluating the tradeoffs between them.

Our proposed reference architecture is specifically designed for microservice architecture, however, we believe it can also be adapted for use in distributed systems, as microservice architecture is a specialized version of distributed architecture. By using the method that we have presented in Figure 6.5, we believe it is possible to develop a reference architecture for a broader range of distributed architectures. We understand that distributing a system is a complex task and it requires a lot of design decisions. Having a reference architecture that considers these decisions and provides guidance can help developers to navigate through the implementation of distributed systems and make the process more structured and manageable.

As stated in Figure 6.4, while deriving the application architecture from the reference architecture, some components of the reference architecture are either reused or the reference architecture is adapted. In the case studies we have done, it has been noticed that some variations may occur, especially in the deployment and service-oriented architecture view, and the reference architecture has been updated so that it can easily address these variants. As we mentioned in Section 3, this process is an opportunity to improve the reference architecture. It is a fact that with more case studies, the reference architecture can be further improved.

There are a few things to consider when deriving the application architecture from the reference architecture. First, it should be known for what purpose each component defined in the reference architecture will be used and what its variants are, and it should be decided accordingly – because many design choices are actually trade-offs. It is also highly recommended that the application architecture be documented according to all the views defined in the reference architecture, so that what components are required from different perspectives and how the process will be handled, and whether there is a missed point can be detected and corrected much earlier.

The use of the reference architecture for deriving more complex architectures and the evaluation of the implementation details are among the needs to be satisfied in the future, which also demonstrates the limitations of this study. In our upcoming studies, we plan to evaluate the applicability of the reference architecture in larger projects by focusing on these points.

Table 8.1 summarizes a SWOT (Strengths, Weaknesses, Opportunities, Threats) analysis of the proposed approach. Based on the SWOT analysis [68], it is determined that the proposed approach's strengths and opportunities outweigh its weaknesses and threats.

Table 8.1. SWOT Analysis of our approach

Dimension	**Explanation**
Strengths	Both architectural views and feature diagrams are defined. The method for deriving application architectures is provided. Easy tailoring of the reference architecture for creating application architectures is demonstrated by a multiple-case study.
Weakness	Further empirical studies are needed to understand how efficient the proposed model is during implementation phase. Runtime dependencies among components in each feature set can be variable and complicated; and, it is not addressed in this study.
Opportunities	There is an opportunity to evaluate implementation aspect of the application architectures. There is an opportunity to carry out further case studies to enhance the reference architecture.
Threats	MSA is quite a dynamic research domain for creating a reference architecture. Chosing Views & Beyond as a software architecture documentation approach can be considered a threat in relying on a specific documentation framework for generic reference architecture

The contributions of this study can be valuable to both practitioners and researchers who are working on MSA. Practitioners can use the reference architecture and the application architecture development method in order to evaluate, derive, or improve their own application architectures. Researchers, on the other hand, can take our contribution as a base to further assess or improve the reference architecture. As an example, our plans for future work include studying on implementation approaches from an application architecture and applying it on more complex architectures.

9. BIBLIOGRAPHY

[1] S. Newman, Building Microservices, 1st ed., O'Reilly Media, Inc., 2015.

[2] M. Fowler, J. Lewis, Microservices, https://martinfowler.com/articles/microservices.html, (Accessed September 19, 2022).

[3] S. Newman, Monolith to Microservices : Evolutionary Patterns to Transform Your Monolith., (2019) 272. https://books.google.com/books/about/Monolith_to_Microservices.html?id=ota_DwAAQBAJ (Accessed April 19, 2022).

[4] O. Zimmermann, Microservices Tenets, Comput. Sci. 32 301–310. https://doi.org/10.1007/s00450-016-0337-0.

[5] N. Josuttis, Soa in Practice: The Art of Distributed System Design, O'Reilly Media, Inc., 2007.

[6] P. Jamshidi, C. Pahl, N.C. Mendonca, J. Lewis, S. Tilkov, Microservices: The journey so far and challenges ahead, IEEE Softw. 35 (2018) 24–35. https://doi.org/10.1109/MS.2018.2141039.

[7] J. Thönes, Microservices, IEEE Softw. 32 (2015). https://doi.org/10.1109/MS.2015.11.

[8] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2004.

[9] B. Tekinerdoğan, M. Akşit, Classifying and Evaluating Architecture Design Methods, Software Architectures and Component Technology. (2002) 3–27. https://doi.org/10.1007/978-1-4615-0883-0_1.

[10] P.D. Francesco, P. Lago, I. Malavolta, Architecting with microservices: A systematic mapping study, Journal of Systems and Software. 150 77–97. https://doi.org/10.1016/j.jss.2019.01.001.

[11] A.B. Bondi, Characteristics of Scalability and Their Impact on Performance, Proceedings of the Second International Workshop on Software and Performance - WOSP '00. (2000). https://doi.org/10.1145/350391.

[12] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil, Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, in: 2015 10th Computing Colombian Conference (10CCC), 2015: pp. 583–590. https://doi.org/10.1109/ColumbianCC.2015.7333476.

[13] Microservice Decompositon: A Case Study of a Large Industrial Software Migration in the Automotive Industry | BIG TU Wien, https://www.big.tuwien.ac.at/publication/tuw-292039/ (Accessed April 20, 2022).

[14] Y. Wang, H. Kadiyala, J. Rubin, Promises and challenges of microservices: an exploratory study, Empir Softw Eng. 26 (2021). https://doi.org/10.1007/S10664-020-09910-Y.

[15] J. Ghofrani, D. Lübke, Challenges of Microservices Architecture: A Survey on the State of the Practice, http://ceur-ws.org/Vol-2072

[16] M. Viggiato, R. Terra, H. Rocha, M.T. Valente, E. Figueiredo, Microservices in Practice: A Survey Study, (2018). http://arxiv.org/abs/1808.04836

[17] C.M. Aderaldo, N.C. Mendonça, C. Pahl, P. Jamshidi, Benchmark Requirements for Microservices Architecture Research, Proceedings - 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering, ECASE 2017. (2017) 8–13. https://doi.org/10.1109/ECASE.2017.4.

[18] M. Bruce, P. Pereira, Microservices in Action, 2018.

[19] M. Söylemez, B. Tekinerdogan, A.K. Tarhan, Feature-Driven Characterization of Microservice Architectures: A Survey of the State of the Practice, Applied Sciences 2022, Vol. 12, Page 4424. 12 (2022) 4424. https://doi.org/10.3390/APP12094424.

[20] M. Söylemez, B. Tekinerdogan, A.K. Tarhan, Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review, Applied Sciences 2022, Vol. 12, Page 5507. 12 (2022) 5507. https://doi.org/10.3390/APP12115507.

[21] J. Ghofrani, A. Bozorgmehr, Migration to microservices: Barriers and solutions, Communications in Computer and Information Science. 1051 CCIS (2019) 269–281. https://doi.org/10.1007/978-3-030-32475-9_20.

[22] C. Richardson, Microservices Patterns, 2018.

[23] P. Jogalekar, M. Woodside, Evaluating the scalability of distributed systems, IEEE Transactions on Parallel and Distributed Systems. 11 (2000) 589–603. https://doi.org/10.1109/71.862209.

[24] G. Blinowski, A. Ojdowska, A. Przybylek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, IEEE Access. 10 (2022) 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803.

[25] M. Stefanko, O. Chaloupka, B. Rossi, The saga pattern in a reactive microservices environment, ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies. (2019) 483–490. https://doi.org/10.5220/0007918704830490.

[26] E.B.H. Yahia, L. Réveillère, Y.-D. Bromberg, R. Chevalier, A. Cadot, Medley: An event-driven lightweight platform for service composition, Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 9671 (2016) 3–20. https://doi.org/10.1007/978-3-319-38791-8_1.

[27] R. v. O'Connor, P. Elger, P.M. Clarke, Continuous software engineering—A microservices architecture perspective, Journal of Software: Evolution and Process. 29 (2017) e1866. https://doi.org/10.1002/SMR.1866.

[28] D. Shadija, M. Rezai, R. Hill, Towards an understanding of microservices, ICAC 2017 - 2017 23rd IEEE International Conference on Automation and Computing: Addressing Global Challenges through Automation and Computing. (2017). https://doi.org/10.23919/IConAC.2017.8082018.

[29] R. Benevides, Istio on Kubernetes, http://bit.ly/istio-kubernetes%0A, (Accessed September 12, 2022)

[30] Y. Yale, H. Silveira, M. Sundaram, A microservice based reference architecture model in the context of enterprise architecture, Proceedings of 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference, IMCEC 2016. (2017) 1856–1860. https://doi.org/10.1109/IMCEC.2016.7867539.

[31] K. Baylov, A. Dimov, Reference architecture for self-adaptive microservice systems, Studies in Computational Intelligence. 737 (2017) 297–303. https://doi.org/10.1007/978-3-319-66379-1_26/FIGURES/4.

[32] I.K. Aksakalli, T. Celik, A.B. Can, B. Tekinerdogan, A model-driven architecture for automated deployment of microservices, Applied Sciences (Switzerland). 11 (2021). https://doi.org/10.3390/APP11209617.

[33] Microservices architecture on AWS - Implementing Microservices on AWS, https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/simple-microservices-architecture-on-aws.html (Accessed April 26, 2022).

[34] Microservice Architecture Reference Architectures 2017 | Red Hat Customer Portal, (https://access.redhat.com/documentation/en-us/reference_architectures/2017/html/microservice_architecture/index (Accessed April 26, 2022).

[35] Microservices architecture design - Azure Architecture Center | Microsoft Docs, https://docs.microsoft.com/en-us/azure/architecture/microservices/ (Accessed April 26, 2022).

[36] Microservices architecture: Reference diagram - IBM Cloud Architecture Center, https://www.ibm.com/cloud/architecture/architectures/microservices/reference-architecture/ (Accessed April 26, 2022).

[37] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, 2003. http://books.google.fi/books?id=mdiIu8Kk1WMC.

[38] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley, Upper Saddle River, NJ, 2010. https://www.safaribooksonline.com/library/view/documenting-software-architectures/9780132488617/.

[39] S. Apel, D. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines, Feature-Oriented Software Product Lines. (2013). https://doi.org/10.1007/978-3-642-37521-7.

[40] B. Tekinerdogan, K. Öztürk, Feature-Driven Design of SaaS Architectures, (2013) 189–212. https://doi.org/10.1007/978-1-4471-5031-2_9.

[41] B. Kitchenham, S. Charters, Guidelines for performing Systematic Literature Reviews in Software Engineering, Engineering. 2 1051. https://doi.org/10.1145/1134285.1134500.

[42] B. Kitchenham, O.P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering – A systematic literature review, Inf Softw Technol. 51 7–15. https://doi.org/10.1016/j.infsof.2008.09.009.

[43] C. Richardson, Pattern: Saga, https://microservices.io/patterns/data/saga.html. (Accessed September 12, 2022).

[44] E. Brewer, CAP twelve years later: How the "rules" have changed, Computer (Long Beach Calif). 45 (2012) 23–29. https://doi.org/10.1109/MC.2012.37.

[45] B. Kitchenham, O.P. Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, Systematic literature reviews in software engineering – A systematic literature review, Inf Softw Technol. 51 (2009) 7–15. https://doi.org/https://doi.org/10.1016/j.infsof.2008.09.009.

[46] T. Dybå, T. Dingsøyr, Strength of Evidence in Systematic Reviews in Software Engineering, in: ESEM'08: Proceedings of the 2008 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ACM Press, New York, New York, USA, 2008: pp. 178–187. https://doi.org/10.1145/1414004.1414034.

[47] S. Tyszberowicz, R. Heinrich, B. Liu, Z. Liu, Identifying Microservices Using Functional Decomposition, in: Springer, Cham, 2018: pp. 50–65. https://doi.org/10.1007/978-3-319-99933-3_4.

[48] A Reference Architecture Primer, http://www.gaudisite.nl/ (Accessed September 27, 2022).

[49] R.K. Yin, Case Study Research: Design and Methods, SAGE Publications, 2009. https://books.google.com.tr/books?id=FzawIAdilHkC.

[50] E. Brewer, CAP twelve years later: How the "rules" have changed, Computer (Long Beach Calif). 45 (2012) 23–29. https://doi.org/10.1109/MC.2012.37.

[51] C. DeLoatch, S. Blindt, NoSQL databases: Scalable Cloud and Enterprise Solutions", University of Illinois at Urbana Champaign Thursday. (2012).

[52] D. Ganesh Chandra, BASE analysis of NoSQL database, Future Generation Computer Systems. 52 (2015) 13–21. https://doi.org/10.1016/j.future.2015.05.003.

[53] X. Limon, A. Guerra-Hernandez, A.J. Sanchez-Garcia, J.C. Perez Arriaga, SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture, in: 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), IEEE, 2018: pp. 50–58. https://doi.org/10.1109/CONISOFT.2018.8645853.

[54] M. Fowler, Event Sourcing, https://martinfowler.com/eaaDev/EventSourcing.html (Accessed September 12, 2022).

[55] C. Richardson, Command Query Responsibility Segregation (CQRS), https://microservices.io/patterns/data/cqrs.html (Accessed September 11, 2022).

[56] What is an API Gateway? | NGINX Learning, https://www.nginx.com/learn/api-gateway/ (Accessed September 11, 2022).

[57] BFF @ SoundCloud | ThoughtWorks, https://www.thoughtworks.com/insights/blog/bff-soundcloud (Accessed September 11, 2022).

[58] The API gateway pattern versus the direct client-to-microservice communication | Microsoft Docs, https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern (Accessed September 11, 2022).

[59] Microservices on AWS, AWS Whitepaper, https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices.html (Accessed 20 September, 2022)

[60] AWS Auto Scaling (Accessed 20 September, 2022)

[61] C. Liu, K. Li, K. Li, A Game Approach to Multi-Servers Load Balancing with Load-Dependent Server Availability Consideration, IEEE Transactions on Cloud Computing. 9 (2021) 1–13. https://doi.org/10.1109/TCC.2018.2790404.

[62] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2004.

[63] What is configuration management?, https://www.redhat.com/en/topics/automation/what-is-configuration-management (Accessed September 22, 2022).

[64] Infrastructure as Code: A Reason to Smile | ThoughtWorks, https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile (Accessed September 22, 2022).

[65] A. Kassahun, Aligning business processes and IT of multiple collaborating organisations, (2017). https://doi.org/10.18174/414988.

[66] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir Softw Eng. 14 (2009) 131–164. https://doi.org/10.1007/S10664-008-9102-8.

[67] B.A. Kitchenham, S.L. Pfleeger, Principles of survey research, ACM SIGSOFT Software Engineering Notes. 27 (2002) 20–24. https://doi.org/10.1145/511152.511155.

[68] T. Hill, R. Westbrook, SWOT analysis: It’s time for a product recall, Long Range Plann. 30 (1997) 46–52. https://doi.org/10.1016/S0024-6301(96)00095-7.

APPENDIX

APPENDIX 1 – Study Quality Assesment Checklist

	REPORTING		RELEVANCE		RIGOR		CREDIBILITY		
	Aim	Scope, Context & Design	Implications in practice & research	Validity & reliability of variables	Explicitness of measures	Adequacy of reporting	Creditability, validity & reliability	Limitations	
Primary Study	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Total**
A	1,0	1,0	0,5	0,5	0,0	0,0	0,5	0,0	3,5
B	1,0	1,0	0,5	0,5	0,0	0,0	0,5	0,0	3,5
C	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
D	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
E	1,0	1,0	1,0	1,0	1,0	0,5	1,0	0,0	6,5
F	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,0	3,5
G	1,0	1,0	1,0	1,0	1,0	0,5	1,0	0,0	6,5
H	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
I	1,0	1,0	0,5	0,5	0,0	0,0	0,5	0,0	3,5
J	1,0	1,0	1,0	0,5	0,0	0,0	1,0	1,0	5,5
K	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
L	0,5	1,0	1,0	1,0	0,5	1,0	1,0	0,0	6,0
M	0,5	0,5	1,0	1,0	1,0	1,0	1,0	0,0	6,0
N	1,0	1,0	1,0	1,0	0,5	1,0	1,0	0,0	6,5
O	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
P	1,0	1,0	0,5	0,0	0,0	0,0	0,5	0,5	3,5
Q	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
R	1,0	0,5	1,0	1,0	1,0	1,0	1,0	0,0	6,5
S	0,5	1,0	1,0	0,5	0,0	0,0	1,0	0,0	4,0
T	0,5	0,5	1,0	0,5	0,5	1,0	0,5	0,0	4,5
U	1,0	1,0	1,0	1,0	0,0	0,0	1,0	0,0	5,0
V	1,0	1,0	1,0	1,0	0,5	1,0	1,0	1,0	7,5
W	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
X	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
Y	1,0	1,0	1,0	0,5	0,0	0,0	1,0	1,0	5,5
Z	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5

AA	0,5	1,0	1,0	1,0	0,5	1,0	1,0	0,0	6,0
AB	0,5	1,0	1,0	0,5	0,0	0,0	1,0	0,0	4,0
AC	1,0	1,0	1,0	0,5	0,0	0,0	0,5	0,0	4,0
AD	1,0	0,5	1,0	0,5	0,5	1,0	1,0	1,0	6,5
AE	1,0	1,0	1,0	1,0	0,5	1,0	1,0	0,0	6,5
AF	1,0	0,5	1,0	0,5	0,5	1,0	1,0	0,0	5,5
AG	1,0	1,0	0,5	0,5	0,0	0,0	0,5	0,0	3,5
AH	1,0	1,0	1,0	1,0	0,0	0,0	1,0	0,0	5,0
AI	1,0	1,0	1,0	0,5	0,0	1,0	0,5	1,0	6,0
AJ	1,0	0,5	0,5	0,5	0,0	0,0	1,0	1,0	4,5
AK	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
AL	1,0	1,0	1,0	1,0	0,0	0,0	0,0	0,0	4,0
AM	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
AN	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
AO	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
AP	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
AQ	1,0	1,0	1,0	1,0	0,0	0,0	1,0	0,5	5,5
AR	1,0	1,0	1,0	1,0	0,5	0,5	1,0	0,0	6,0
AS	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
AT	1,0	1,0	1,0	1,0	0,5	0,5	1,0	1,0	7,0
AU	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
AV	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
AW	1,0	0,5	1,0	1,0	1,0	1,0	0,5	0,0	6,0
AX	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
AY	1,0	1,0	1,0	0,5	1,0	0,5	1,0	0,0	6,0
AZ	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
BA	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
BB	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
BC	1,0	1,0	1,0	0,5	1,0	1,0	1,0	0,5	7,0
BD	1,0	1,0	1,0	1,0	1,0	0,5	1,0	0,0	6,5
BE	1,0	1,0	1,0	1,0	0,0	0,0	0,5	0,0	4,5
BF	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
BG	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
BH	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0

BI	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
BJ	1,0	1,0	0,5	0,5	0,5	0,5	0,5	0,0	4,5
BK	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
BL	1,0	1,0	0,5	0,5	0,5	0,5	0,5	1,0	5,5
BM	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
BN	1,0	0,5	0,5	0,5	0,5	0,5	0,5	0,0	4,0
BO	1,0	1,0	0,5	0,5	0,5	0,5	0,5	0,0	4,5
BP	1,0	1,0	1,0	0,5	0,5	0,5	0,5	0,0	5,0
BQ	1,0	1,0	0,5	1,0	1,0	1,0	1,0	0,0	6,5
BR	1,0	1,0	0,5	1,0	1,0	1,0	1,0	0,0	6,5
BS	1,0	1,0	0,5	0,5	1,0	1,0	1,0	0,5	6,5
BT	1,0	0,5	0,5	0,5	0,0	0,5	0,5	0,0	3,5
BU	1,0	1,0	0,5	1,0	1,0	1,0	1,0	1,0	7,5
BV	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
BW	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
BX	1,0	1,0	0,5	1,0	1,0	1,0	1,0	0,0	6,5
BY	1,0	1,0	0,5	1,0	1,0	1,0	1,0	1,0	7,5
BZ	1,0	0,5	0,5	0,5	0,5	0,5	0,5	0,0	4,0
CA	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
CB	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5
CC	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,0	7,0
CD	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
CE	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
CF	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	8,0
CG	1,0	1,0	1,0	1,0	1,0	1,0	1,0	0,5	7,5

APPENDIX 2 – List of Primary Studies

A. D. I. Savchenko, G. I. Radchenko, and O. Taipale, “Micro-services validation: Mjolnirr platform case study,” in 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), May 2015, pp. 235–240, doi: 10.1109/MIPRO.2015.7160271.

B. M. Rahman and J. Gao, “A Reusable Automated Acceptance Testing Architecture for Micro-services in Behavior-Driven Development,” in 2015 IEEE Symposium on Service-Oriented System Engineering, Mar. 2015, pp. 321–325, doi: 10.1109/SOSE.2015.55.

C. N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh, “Synapse,” in Proceedings of the Tenth European Conference on Computer Systems - EuroSys ’15, 2015, pp. 1–16, doi: 10.1145/2741948.2741975.

D. Y. Sun, S. Nanda, and T. Jaeger, “Security-as-a-Service for Micro-services-Based Cloud Applications,” in 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), Nov. 2015, pp. 50–57, doi: 10.1109/CloudCom.2015.93.

E. H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, “Efficiency Analysis of Provisioning Micro-services,” in 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Dec. 2016, pp. 261–268, doi: 10.1109/CloudCom.2016.0051.

F. A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso, “The Database-is-the-Service Pattern for Micro-service Architectures,” Springer, Cham, 2016, pp. 223–233.

G. A. de Camargo, I. Salvadori, R. dos S. Mello, and F. Siqueira, “An architecture to automate performance tests on micro-services,” in Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services - iiWAS ’16, 2016, pp. 422–429, doi: 10.1145/3011141.3011179.

H. V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, “Gremlin: Systematic Resilience Testing of Micro-services,” in 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Jun. 2016, pp. 57–66, doi: 10.1109/ICDCS.2016.11.

I. K. B. Long, H. Yang, and Y. Kim, "ICN-based service discovery mechanism for micro-service architecture," in 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN), Jul. 2017, pp. 773–775, doi: 10.1109/ICUFN.2017.7993899.

J. S. Haselböck, R. Weinreich, and G. Buchgeher, "Decision guidance models for micro-services," in Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems - ECBS '17, 2017, pp. 1–10, doi: 10.1145/3123779.3123804.

K. S. Klock, J. M. E. M. Van Der Werf, J. P. Guelen, and S. Jansen, "Workload-Based Clustering of Coherent Feature Sets in Micro-service Architectures," in 2017 IEEE International Conference on Software Architecture (ICSA), Apr. 2017, pp. 11–20, doi: 10.1109/ICSA.2017.38.

L. H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: autoscaling and monitoring as a service," Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering. IBM Corp., pp. 234–240, 2017, Accessed: Jun. 11, 2019. [Online]. Available: https://dl.acm.org/citation.cfm?id=3172823.

M. N. H. Do, T. Van Do, X. Thi Tran, L. Farkas, and C. Rotter, "A scalable routing mechanism for stateful micro-services," in 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), Mar. 2017, pp. 72–78, doi: 10.1109/ICIN.2017.7899252.

N. M. Rusek, G. Dwornicki, and A. Orłowski, "A Decentralized System for Load Balancing of Containerized Micro-services in the Cloud," Springer, Cham, 2017, pp. 142–152.

O. 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 2017 IEEE International Conference on Communications (ICC), May 2017, pp. 1–6, doi: 10.1109/ICC.2017.7996729.

P. G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "MicroART: A software architecture recovery tool for maintaining micro-service-based systems," in Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings, Jun. 2017, pp. 298–302, doi: 10.1109/ICSAW.2017.9.

Q. H. Zhou et al., “Overload Control for Scaling WeChat Micro-services,” in Proceedings of the ACM Symposium on Cloud Computing - SoCC ’18, 2018, pp. 149–161, doi: 10.1145/3267809.3267823.

R. X. Luo, F. Ren, and T. Zhang, “High Performance Userspace Networking for Containerized Micro-services,” Springer, Cham, 2018, pp. 57–72.

S. X. Limon, A. Guerra-Hernandez, A. J. Sanchez-Garcia, and J. C. Perez Arriaga, “SagaMAS: A Software Framework for Distributed Transactions in the Micro-service Architecture,” in 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), Oct. 2018, pp. 50–58, doi: 10.1109/CONISOFT.2018.8645853.

T. K. Jander, L. Braubach, and A. Pokahr, “Defense-in-depth and Role Authentication for Micro-service Systems,” Procedia Comput. Sci., vol. 130, pp. 456–463, Jan. 2018, doi: 10.1016/J.PROCS.2018.04.047.

U. S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, “Identifying Micro-services Using Functional Decomposition,” Springer, Cham, 2018, pp. 50–65.

V. T. Yarygina and A. H. Bagge, “Overcoming Security Challenges in Micro-service Architectures,” in 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), Mar. 2018, pp. 11–20, doi: 10.1109/SOSE.2018.00011.

W. S. Sebastio, R. Ghosh, and T. Mukherjee, “An Availability Analysis Approach for Deployment Configurations of Containers,” IEEE Trans. Serv. Comput., pp. 1–1, 2018, doi: 10.1109/TSC.2017.2788442.

X. S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, “Using Service Dependency Graph to Analyze and Test Micro-services,” in 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Jul. 2018, pp. 81–86, doi: 10.1109/COMPSAC.2018.10207.

Y. A. Warke, M. Mohamed, R. Engel, H. Ludwig, W. Sawdon, and L. Liu, “Storage Service Orchestration with Container Elasticity,” in 2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC), Oct. 2018, pp. 283–292, doi: 10.1109/CIC.2018.00046.

Z. J. Lin, P. Chen, and Z. Zheng, “Microscope: Pinpoint Performance Issues with Causal Graphs in Micro-service Environments,” Springer, Cham, 2018, pp. 3–20.

AA. Y. Sun, L. Meng, P. Liu, Y. Zhang, and H. Chan, “Automatic Performance Simulation for Micro-service Based Applications,” Springer, Singapore, 2018, pp. 85–95.

AB. E. Bainomugisha, A. S. I. G. on S. Engineering, and ACM Digital Library., Partitioning Micro-services: A Domain Engineering Approach. ACM, 2018.

AC. D. Guija and M. S. Siddiqui, “Identity and Access Control for micro-services based 5G NFV platforms,” in Proceedings of the 13th International Conference on Availability, Reliability and Security - ARES 2018, 2018, pp. 1–10, doi: 10.1145/3230833.3233255.

AD. B. Mayer and R. Weinreich, “An Approach to Extract the Architecture of Micro-service-Based Software Systems,” in 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), Mar. 2018, pp. 21–30, doi: 10.1109/SOSE.2018.00012.

AE. F. Klinaku, M. Frank, and S. Becker, “CAUS : An Elasticity Controller for a Containerized Micro-service” in Companion of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE ’18, 2018, pp. 93–98, doi: 10.1145/3185768.3186296.

AF. K. Jander, A. Pokahr, L. Braubach, and J. Kalinowski, “Service Discovery in Megascale Distributed Systems,” Springer, Cham, 2018, pp. 273–284.

AG. M. J. Kargar and A. Hanifizade, “Automation of regression test in micro-service architecture,” in 2018 4th International Conference on Web Research (ICWR), Apr. 2018, pp. 133–137, doi: 10.1109/ICWR.2018.8387249.

AH. G. Pardon, C. Pautasso, and O. Zimmermann, “Consistent Disaster Recovery for Micro-services: the BAC Theorem,” IEEE Cloud Comput., vol. 5, no. 1, pp. 49–59, Jan. 2018, doi: 10.1109/MCC.2018.011791714.

AI. D. Monteiro, R. Gadelha, P. H. M. Maia, L. S. Rocha, and N. C. Mendonça, “Beethoven: An Event-Driven Lightweight Platform for Micro-service Orchestration,” Springer, Cham, 2018, pp. 191–199.

AJ. G. Fu, J. Sun, and J. Zhao, “An optimized control access mechanism based on micro-service architecture,” in 2018 2nd IEEE Conference on Energy Internet and Energy System Integration (EI2), Oct. 2018, pp. 1–5, doi: 10.1109/EI2.2018.8582628.

AK. L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance Modeling and Workflow Scheduling of Micro-service-based Applications in Clouds," IEEE Trans. Parallel Distrib. Syst., pp. 1–1, 2019, doi: 10.1109/TPDS.2019.2901467.

AL. Y. Wang, L. Cheng, and X. Sun, "Design and Research of Micro-service Application Automation Testing Framework," in Proceedings - 2019 International Conference on Information Technology and Computer Application, ITCA 2019, Dec. 2019, pp. 257–260, doi: 10.1109/ITCA49981.2019.00063.

AM. E. Fadda, P. Plebani, and M. Vitali, "Monitoring-aware Optimal Deployment for Applications based on Micro-services," IEEE Trans. Serv. Comput., pp. 1–1, Jul. 2019, doi: 10.1109/tsc.2019.2910069.

AN. M. Lin, J. Xi, W. Bai, and J. Wu, "Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Micro-service Scheduling in Cloud," IEEE Access, vol. 7, pp. 83088–83100, 2019, doi: 10.1109/ACCESS.2019.2924414.

AO. Y. Yu, J. Yang, C. Guo, H. Zheng, and J. He, "Joint optimization of service request routing and instance placement in the micro-service system," J. Netw. Comput. Appl., vol. 147, p. 102441, Dec. 2019, doi: 10.1016/j.jnca.2019.102441.

AP. S. Li et al., "A dataflow-driven approach to identifying micro-services from monolithic applications," J. Syst. Softw., vol. 157, p. 110380, Nov. 2019, doi: 10.1016/j.jss.2019.07.008.

AQ. O.-M. Ungureanu, C. Vlădeanu, and R. Kooij, "Kubernetes cluster optimization using hybrid shared-state scheduling framework," in Proceedings of the 3rd International Conference on Future Networks and Distributed Systems - ICFNDS '19, 2019, Accessed: Aug. 27, 2020. [Online]. Available: https://doi.org/10.1145/3341325.3341992.

AR. A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud micro-service applications," in ICPE 2019 - Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, Apr. 2019, pp. 25–32, doi: 10.1145/3297663.3310309.

AS. M. Cinque, R. Della Corte, and A. Pecchia, "Micro-services Monitoring with Event Logs and Black Box Execution Tracing," IEEE Trans. Serv. Comput., 2019, doi: 10.1109/TSC.2019.2940009.

AT. L. L. Jimenez and O. Schelen, “DOCMA: A decentralized orchestrator for containerized micro-service applications,” in Proceedings - 2019 3rd IEEE International Conference on Cloud and Fog Computing Technologies and Applications, Cloud Summit 2019, Aug. 2019, pp. 45–51, doi: 10.1109/CloudSummit47114.2019.00014.

AU. T. Kiss et al., “MiCADO—Micro-service-based Cloud Application-level Dynamic Orchestrator,” Futur. Gener. Comput. Syst., vol. 94, pp. 937–946, May 2019, doi: 10.1016/J.FUTURE.2017.09.050.

AV. S. Wang, Z. Ding, and C. Jiang, “Elastic Scheduling for Micro-service Applications in Clouds,” IEEE Trans. Parallel Distrib. Syst., vol. 32, no. 1, pp. 98–115, Jul. 2020, doi: 10.1109/tpds.2020.3011979.

AW. F. Wan, X. Wu, and Q. Zhang, “Chain-Oriented Load Balancing in Micro-service System,” in 2020 World Conference on Computing and Communication Technologies (WCCCT), May 2020, pp. 10–14, doi: 10.1109/WCCCT49810.2020.9169996.

AX. G. Yu, P. Chen, and Z. Zheng, “Microscaler: Cost-effective Scaling for Micro-service Applications in the Cloud with an Online Learning Approach,” IEEE Trans. Cloud Comput., pp. 1–1, Apr. 2020, doi: 10.1109/tcc.2020.2985352.

AY. A. Samanta and J. Tang, “Dyme: Dynamic Micro-service Scheduling in Edge Computing Enabled IoT,” IEEE Internet Things J., pp. 1–1, Mar. 2020, doi: 10.1109/jiot.2020.2981958.

AZ. Á. Brandón, M. Solé, A. Huélamo, D. Solans, M. S. Pérez, and V. Muntés-Mulero, “Graph-based root cause analysis for service-oriented and micro-service architectures,” J. Syst. Softw., vol. 159, p. 110432, Jan. 2020, doi: 10.1016/j.jss.2019.110432.

BA. S. N. Srirama, M. Adhikari, and S. Paul, “Application deployment using containers with auto-scaling for micro-services in cloud environment,” J. Netw. Comput. Appl., vol. 160, p. 102629, Jun. 2020, doi: 10.1016/j.jnca.2020.102629.

BB. A. Avritzer et al., “Scalability Assessment of Micro-service Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests,” J. Syst. Softw., vol. 165, p. 110564, Jul. 2020, doi: 10.1016/j.jss.2020.110564.

BC. A. De Iasio and E. Zimeo, “A framework for micro-services synchronization,” Softw. Pract. Exp., p. spe.2877, Aug. 2020, doi: 10.1002/spe.2877.

BD. M. Imdoukh, I. Ahmad, and M. Alfailakawi, “Optimizing scheduling decisions of container management tool using many-objective genetic algorithm,” Concurr. Comput. Pract. Exp., vol. 32, no. 5, Mar. 2020, doi: 10.1002/cpe.5536.

BE. M. Autili, A. Perucci, and L. De Lauretis, “A Hybrid Approach to Micro-services Load Balancing,” in Micro-services, Springer International Publishing, 2020, pp. 249–269.

BF. N. C. Coulson, S. Sotiriadis, and N. Bessis, “Adaptive Micro-service Scaling for Elastic Applications,” IEEE Internet Things J., vol. 7, no. 5, pp. 4195–4202, May 2020, doi: 10.1109/JIOT.2020.2964405.

BG. M. Jin et al., “An Anomaly Detection Algorithm for Microservice Architecture Based on Robust Principal Component Analysis,” IEEE Access, 2020, doi: 10.1109/ACCESS.2020.3044610.

BH. C. K. Rudrabhatla, “A Quantitative Approach for Estimating the Scaling Thresholds and Step Policies in a Distributed Microservice Architecture,” IEEE Access, vol. 8, pp. 180246–180254, 2020, doi: 10.1109/ACCESS.2020.3028310.

BI. N. C. Coulson, S. Sotiriadis, and N. Bessis, “Adaptive Microservice Scaling for Elastic Applications,” IEEE Internet of Things Journal, vol. 7, no. 5, pp. 4195–4202, May 2020, doi: 10.1109/JIOT.2020.2964405.

BJ. X. Guo et al., “Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis,” Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, vol. 20, 2020, doi: 10.1145/3368089.

BK. D. Monteiro, P. H. M. Maia, L. S. Rocha, and N. C. Mendonça, “Building orchestrated microservice systems using declarative business processes,” Service Oriented Computing and Applications, vol. 14, no. 4, pp. 243–268, Dec. 2020, doi: 10.1007/S11761-020-00300-2/FIGURES/12.

BL. D. Ernst and S. Tai, “Offline Trace Generation for Microservice Observability,” Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOCW, pp. 308–317, 2021, doi: 10.1109/EDOCW52865.2021.00062.

BM. A. A. Khaleq and I. Ra, “Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications,” IEEE Access, vol. 9, pp. 35464–35476, 2021, doi: 10.1109/ACCESS.2021.3061890.

BN. A. Bento et al., “A layered framework for root cause diagnosis of microservices,” 2021 IEEE 20th International Symposium on Network Computing and Applications (NCA), pp. 1–8, Nov. 2021, doi: 10.1109/NCA53618.2021.9685494.

BO. T. Weng, W. Yang, G. Yu, P. Chen, J. Cui, and C. Zhang, “Kmon: An In-kernel Transparent Monitoring System for Microservice Systems with eBPF,” Proceedings - 2021 IEEE/ACM International Workshop on Cloud Intelligence, CloudIntelligence 2021, pp. 25–30, May 2021, doi: 10.1109/CLOUDINTELLIGENCE52565.2021.00014.

BP. A. Megargel, C. M. Poskitt, and V. Shankararaman, “Microservices Orchestration vs. Choreography: A Decision Framework,” pp. 134–141, Dec. 2021, doi: 10.1109/EDOC52215.2021.00024.

BQ. Y. Chen, N. Chen, W. Xu, L. Lian, and H. Tu, “MFRL-CA: Microservice Fault Root Cause Location based on Correlation Analysis,” pp. 90–101, Dec. 2021, doi: 10.1109/DSA52907.2021.00018.

BR. Y. Cai, B. Han, J. Li, N. Zhao, and J. Su, “ModelCoder: A Fault Model based Automatic Root Cause Localization Framework for Microservice Systems,” 2021 IEEE/ACM 29th International Symposium on Quality of Service, IWQOS 2021, Jun. 2021, doi: 10.1109/IWQOS52092.2021.9521318.

BS. M. Li, D. Tang, Z. Wen, and Y. Cheng, “Microservice Anomaly Detection Based on Tracing Data Using Semi-supervised Learning,” 2021 4th International Conference on Artificial Intelligence and Big Data, ICAIBD 2021, pp. 38–44, May 2021, doi: 10.1109/ICAIBD51990.2021.9459100.

BT. C. Pasomsup and Y. Limpiyakorn, “HT-RBAC: A Design of Role-based Access Control Model for Microservice Security Manager,” pp. 177–181, Dec. 2021, doi: 10.1109/BDEE52938.2021.00038.

BU. D. Liu et al., “MicroHECL: High-efficient root cause localization in large-scale microservice systems,” Proceedings - International Conference on Software Engineering, pp. 338–347, May 2021, doi: 10.1109/ICSE-SEIP52600.2021.00043.

BV. Z. Ye, P. Chen, and G. Yu, “T-Rank: A lightweight spectrum based fault localization approach for microservice systems,” Proceedings - 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021, pp. 416–425, May 2021, doi: 10.1109/CCGRID51090.2021.00051.

BW. S. Wang, Z. Ding, and C. Jiang, “Elastic scheduling for microservice applications in clouds,” IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 1, pp. 98–115, Jan. 2021, doi: 10.1109/TPDS.2020.3011979.

BX. L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, “Exploring layered container structure for cost efficient microservice deployment,” Proceedings - IEEE INFOCOM, vol. 2021-May, May 2021, doi: 10.1109/INFOCOM42981.2021.9488918.

BY. J. Grohmann et al., “SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications,” ICPE 2021 - Proceedings of the ACM/SPEC International Conference on Performance Engineering, pp. 165–176, Apr. 2021, doi: 10.1145/3427921.3450248.

BZ. B. Choi, J. Park, C. Lee, and D. Han, “pHPA: A Proactive Autoscaling Framework for Microservice Chain,” 5th Asia-Pacific Workshop on Networking (APNet 2021), vol. 7, pp. 65–71, Jun. 2021, doi: 10.1145/3469393.3469401.

CA. Y. Li, Y. Zhang, Z. Zhou, and L. Shen, “Intelligent flow control algorithm for microservice system,” Cognitive Computation and Systems, vol. 3, no. 3, pp. 276–285, Sep. 2021, doi: 10.1049/CCS2.12013.

CB. C. T. Joseph and K. Chandrasekaran, “Nature-inspired resource management and dynamic rescheduling of microservices in Cloud datacenters,” Concurrency and Computation: Practice and Experience, vol. 33, no. 17, p. e6290, Sep. 2021, doi: 10.1002/CPE.6290.

CC. M. Daoud, A. el Mezouari, N. Faci, D. Benslimane, Z. Maamar, and A. el Fazziki, “A multi-model based microservices identification approach,” Journal of Systems Architecture, vol. 118, p. 102200, Sep. 2021, doi: 10.1016/J.SYSARC.2021.102200.

CD. L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, “A Kubernetes controller for managing the availability of elastic microservice based stateful applications,” Journal of Systems and Software, vol. 175, p. 110924, May 2021, doi: 10.1016/J.JSS.2021.110924.

CE L. Meng, F. Ji, Y. Sun, and T. Wang, “Detecting anomalies in microservices with execution trace comparison,” Future Generation Computer Systems, vol. 116, pp. 291–301, Mar. 2021, doi: 10.1016/J.FUTURE.2020.10.040.

CF. V. Cortellessa, D. di Pompeo, R. Eramo, and M. Tucci, “A model-driven approach for continuous performance engineering in microservice-based systems,” Journal of Systems and Software, vol. 183, p. 111084, Jan. 2022, doi: 10.1016/J.JSS.2021.111084.

CG. J. Moeyersons, S. Kerkhove, T. Wauters, F. de Turck, and B. Volckaert, “Towards cloud-based unobtrusive monitoring in remote multi-vendor environments,” Software: Practice and Experience, vol. 52, no. 2, pp. 427–442, Feb. 2022, doi: 10.1002/SPE.3029.

APPENDIX 3 – Related Publications – Journal Articles

M. Söylemez, B. Tekinerdogan, and A. Kolukısa Tarhan, “Feature-Driven Characterization of Microservice Architectures: A Survey of the State of the Practice,” Applied Sciences, vol. 12, no. 9, p. 4424, Apr. 2022, doi: 10.3390/app12094424.

M. Söylemez, B. Tekinerdogan, and A. Kolukısa Tarhan, “Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review,” Applied Sciences, vol. 12, no. 11, p. 5507, May 2022, doi: 10.3390/app12115507.