FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Applying the Normalized Systems Theory on Microservice Architecture
Student:	Bc. Vincenc Kolařík
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2019/20

Instructions

- Analyze the current state-of-the-art design methods and industrial applications of microservice architecture.
- Discuss compliance of the currently used methods with Normalized Systems theory (NS) and explore possibilities for improvements using NS. Address the most important challenges encountered in the industry.
- Then formulate guidelines for designing microservices based on the results from the previous topic.
- Discuss the results and demonstrate them on a case study.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague September 29, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Applying the Normalized Systems Theory on Microservice Architecture

Bc. Vincenc Kolařík

Department of Software Engineering
Supervisor: Ing. Robert Pergl, Ph.D.

January 10, 2019

Acknowledgements

Thanks to my family and especially to my girlfriend, Tamara, for all their patience and care. Thanks to my thesis supervisor, Robert, for everything he taught me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 10, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kolařík, Vincenc. *Applying the Normalized Systems Theory on Microservice Architecture*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Práce se zabývá schopností evolvability aplikací postavených na architektuře mikroslužeb. Aplikuje pravidla definovaná teorií Normalizovaných systémů a diskutuje jejich dopad. Práce zevrubně zkoumá doménu architektury mikroslužeb a identifikuje její klíčové architektonické aspekty. Aspekty jsou dále zkoumány pomocí teorie Normalizovaných systémů. Výsledkem je soubor pravidel pro dosažení evolvability aplikací využívajících mikroslužeb. Vhodnost navrhovaných pravidel je poté diskutována.

Klíčová slova architektura mikroslužeb, evolvabilita, mikroslužby, softwarová architektura, teorie normalizovaných systémů

Abstract

The thesis is engaged in evolvability of applications build using microservice architecture. It applies the rules defined by the Normalized Systems theory and discusses its impacts. The thesis thoroughly analyses the domain of microservice architecture and identifies the key architectural aspects. The aspects are examined using the Normalized Systems theory and results in a set of guidelines to achieve evolvability of the microservice applications. The viability of proposed guidelines is then discussed.

Keywords evolvability, microservice architecture, microservices, normalized systems theory, software architecture

Contents

List of Figures

List of Tables

Introduction

Combining Normalized Systems and microservices is like making two worlds collide. The virtuous academia, which favors sound theories and logical proofs, crashes with the vibrant and restless software development industry, which prefers trial and error to a formulation of a hypothesis.

Normalized systems specify how to build *evolvable* systems — systems that will be able to absorb changes with least effort for an infinite amount of time.

The community around microservices does not use the term evolvability. Nevertheless, in the industry where microservices are widespread, business agility is the key to success. There is clear evidence that frequent software releases are correlated with good business performance.[1] Companies like Spotify and Netflix invest tremendous effort to improve their agile software development, focusing on shortening lead time. The technology and retail behemoth Amazon.com deploys new code to production systems every 11.7 seconds on average.[2] All of this is an obvious indication of a need for evolvability.

Both worlds are utilizing similar concepts — high modularity, strict separation of concerns, high cohesion and low coupling — just to name some. However, the way the tools are used vary.

The goal of this work is to streamline the best practices of the industry using a solid scientific approach. For academia, this topic is not so enriching. The greatest achievement this work can bring is just another proof that the theories were right. However, that is a dream come true of every researcher, isn't it?

[1] https://puppet.com/resources/whitepaper/2017-state-of-devops-report
[2] https://www.youtube.com/watch?v=dxk8b9rSKOo

Chapter 1

Goals and Approach

1.1 Goals

According to a thesis assignment the following goals were set:

- Analyze the current state-of-the-art design methods and industrial applications of microservice architecture (MSA).
- Identify key features and the most significant challenges encountered in the industry.
- Discuss compliance of the currently used methods with Normalized Systems theory (NS) and explore possibilities for improvements using NS.
- Formulate guidelines for designing microservices based on the results from the previous chapter, discuss them and demonstrate them on a case study if reasonable.

1.2 Approach

Microservices architecture is an enterprise architecture (EA) pattern. EA could be analyzed in enormously broad context, spanning from business – IT alignment, through SW development methodologies to a choice of programming languages. This section discusses boundaries for this thesis to make the topic reasonably narrow without overlooking the quintessence of microservice architecture.

1.2.1 Organizational Aspects

Microservices are often valued for their positive effects on organizations and teams of engineers creating them. Development of end-to-end features and operational responsibility fosters DevOps culture [2]. Code ownership and

cross-functional teams cultivate team spirit and nurture motivation of developers [3, 4, 5].

There's also a sociological observation called *Conway's law* that states:

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. [6]

The bi-directional influence between software architecture and the organization that creates it is a remarkable topic, and it would be unwise to ignore it while running a business. Despite that, it will be ignored in this thesis, as it would broaden its scope excessively. The NS literature [7, 1] has the same attitude and avoids discussion how the organization influences the software (SW) and vice versa.

1.2.2 Performance Aspects

The Normalized Systems theory describes a set of laws which, if applied strictly, guarantee a system to be free from combinatorial effects, i.e., to be indefinitely evolvable. Until now, the industrial software projects [8, 9] are focused on the evolvability of the SW artifact and does not value any other non-functional requirement as much as evolvability.

On the other hand, the notion of a microservice (MS) came into being due to the need for performance and scalability [5, 10]. The evolvability aspect is never more important that those two mentioned. Therefore this thesis will respect this order and will not sacrifice performance and scalability for the sake of evolvability.

1.2.3 User Interface, Client Applications, etc.

Successful Internet applications have tenths of GUIs: web browser front-end (FE), desktop apps, mobile apps, watch apps and desktop widget. Due to the heterogeneity of the human-facing interfaces, they are out of scope of this thesis.

1.2.4 General Rules

In an effort to make this work conceptually coherent, there were defined essential principles with which this thesis is created:

1. Avoid discussion how organization influences the SW and vice versa.
2. Do not sacrifice performance and scalability for the sake of evolvability.
3. Focus on the internal architecture of the back-end (BE).

1.3 Thesis Structure and Tasks

To fulfill the goals of this thesis the following finer-grain task list was defined. The final structure of this work is derived from this list.

1. Perform a literature review on MS and on application of NS to MS and/or related architecture styles and patterns

   ⇒ chapter 3: *Microservice Architecture Literature Review*

2. Analyze the microservice architecture

   - Extract the essential concepts of state-of-the-art design of MS
   - Identify key concerns in MS design and implementation

   ⇒ chapter 4: *Analysis of the Microservice Architecture*

3. Examine compliance of microservice architecture to Normalized Systems theory

   - Discuss the architectural patterns and principles using Normalized Systems theory
   - Apply the *Design Theorems for Stable Software* [11] to the essential concepts from previous chapter

   ⇒ chapter 5: *Towards Stable Microservice Architecture*

4. Summarize design guidelines for MSA

   - Provide concise and comprehensive overview of formulated guidelines
   - Demonstrate guidelines on suitable case study

   ⇒ chapter 6: *The Stable Microservice Architecture*

5. Summarize successes and failures

   ⇒ chapter: *Conclusion*

Chapter 2

Theoretical Background

2.1 Introduction to Normalized Systems

Normalized Systems theory (NS) is a theoretical framework designed to engineer systems (in its broad meaning) to be able to absorb a set of anticipated changes in an infinite period. The ability to absorb changes — *evolvability* — is the essential property of studied systems. The theory formulates a set of *rules of evolvability* backed by formal proofs (more in section 2.1.2).

2.1.1 Ongoing Research

The theory is being developed at the University of Antwerp, the department Management Information Systems of the faculty Applied Economics. Due to its success, the authors, Jan Verelst and Herwig Mannaert, have established the Normalized Systems Institute for further applied research in the SW industry.

Although the theory originated in a narrow field — software development, it has been generalized and now is being applied to wide range of various disciplines: from business process modeling through legal documents to first thoughts on civil engineering. [11]

Due to the focus of this thesis, Normalized Systems theory will be explained on the domain of software development.

2.1.2 Essential Principles

The authors of Normalized Systems theory state the contemporary IT problems are manifestations of fundamental flaws in currently used SW development methodologies. The Achilles' heel is the evolvability — adding new features to existing code base generates *combinatorial effects* (or *instabilities* in newer literature, e.g. [1]), that lead to a growth of overall system complex-

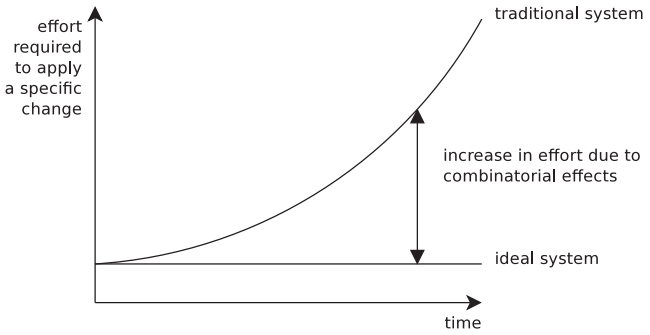

Figure 2.1: Combinatorial effects explained

ity (see fig. 2.1 [7]). Such effects cause to increase the cost of future changes and decrease overall software quality.[7]

Initial idea was first uttered by Manny Lehman in 1980:

> As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it. [12]

In spite of that, Normalized Systems theory strive to fulfill the dream articulated by Douglas McIlroy in 1968:

> Expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, [the user] should be able safely to regard components as black boxes. [13]

NS assume that a change introduced to a system is a natural and unavoidable phenomenon. Therefore all rules and principles are designed to accommodate that fact. The theory defines a set of guidelines on how to engineer the software architecture as a structure of highly independent modules, which can be added, removed or changed separately. Sufficiently granular architecture will suppress all combinatorial effects during evolution of the system.

The normalized design theorems require a strict separation of data and actions manipulating that data. Such segregation might be controversial as it contradicts the essence of object oriented programming (OOP), which postulates that data and related actions belong into one entity — a *class*.

2.2 Design Theorems of Stable Software

This section introduces the four rules of software evolvability — *the design theorems of stable software*, which are the building blocks of NS theory. An experienced software developer will recognize these rules as they originate from a heuristic design knowledge. The value added by NS is the theoretical

proofs which promote these practical experiences to defensible theorems. (The proofs are omitted for brevity, but can be found in [7, 1].)

2.2.1 The Four Theorems

The NS theory can prove the system is free from instabilities if and only if the system complies to all the design theorems. Therefore the following postulate is set as an ultimate goal:

> An evolving information system should not have *instabilities* (*combinatorial effects*): a bounded amount of additional functional requirements cannot lead to an unbounded amount of additional (versions of) software primitives. [1]

2.2.1.1 Separation of Concerns

> A processing function can only contain a single task in order to achieve stability. [1]

This theorem implies the identification and separation of every single *task*. Correct separation of tasks will induce *separation of concerns* in the big picture.

Separation of concerns is a widely used *best practice* among software architects. However, it is very vaguely formulated. Current manifestations in software development include for example *multi-tier architectures* (e.g., MVC, MVVM) or use of an *integration bus* for inter-process communication.

2.2.1.2 Data Version Transparency

> A structure that is passed through the interface of a processing function needs to exhibit version transparency in order to achieve stability. [1]

Data version transparency is an instrument to cope with an addition or removal a *data field* in entity. It implies encapsulation of the data fields. Wrapping the data entity allows co-existence of various versions of such entity.

An example from OOP: data version transparency can be easily achieved by using exclusively the *0-parameter constructor* for instantiation and *accessor methods* for the attribute access (e.g., a POJO). In that case, all internal data fields are hidden and addition of a new one does not cause processing method to fail.

2.2.1.3 Action Version Transparency

> A processing function that is called by another processing function, needs to exhibit version transparency in order to achieve stability. [1]

Analogously to previous theorem, various versions of data entities need to co-exist in single system.

This can be achieved by using wrapper functions in procedural programming or by using polymorphism in OOP.

2.2.1.4 Separation of States

> Calling a processing function within another processing function, needs to exhibit state keeping in order to achieve stability. [1]

It is a formalization of instinctive *avoiding the transition to an undefined state*. When a state is kept for every call of a processing function, the whole system behaves as a deterministic state machine. This eliminates the need for complicated recovery from undefined error states.

An example of a manifestation of this design theorem is a database transaction mechanism. The commit (rollback) action guarantees atomic transition from one defined state to another.

2.2.2 Impacts on Software Development

The postulate in section 2.2 implies all the *design theorems for stable software* must be consistently followed. The SW artifact must be free of instabilities at compile time, deployment time and run time. It requires the code base to be entirely error free, which is not an easy task to achieve.

Other inconveniences are inevitably encountered during the development of SW according to NS. For example the *data* and *action version transparency* rules also imply a lot of *boilerplate* (non-logic) code (e.g., wrapper classes, accessor methods). For a human programmer writing code complying with NS is annoying and frustrating.

On the other hand, NS presents a set of *software design patterns* (described in [7]) which could be easily produced via code generation. The generated boilerplate code skeleton is then enriched with custom code containing business logic and algorithms. In case the skeleton itself is introduced to a change, the custom code is *harvested* and then *re-injected* back to the fully re-generated skeleton. [14]

The code generation approach remedies the developers' disgruntlement for sure. However, it requires the development of sophisticated tooling to get the fully NS-compliant software working. The only known implementation — the *NSX Expanders*[3] — is capable of production of remarkably granular, yet still *monolithic* applications.

This thesis is an attempt to broaden the application of Normalized Systems theory to a domain of distributed software architectures.

[3] https://normalizedsystems.org/tools/

2.3 Introduction to Software Architecture

As time goes by, the size and complexity of a software system grow, the design questions soon grow beyond algorithms and data structures. The new problem of the overall system design emerges.

When the problem is untreated, applications soon become tightly coupled, brittle and increasingly difficult to change. Even experienced team of developers without a vision resort to the prosaic layered architecture pattern also known as the *n-tier architecture*, creating implicit layers by separating source-code modules into packages. A result of this practice often is a collection of poorly organized source code, modules and components lack clear borders, responsibilities, and relationships with each other. This primeval architecture style is mockingly called a *the big ball of mud* [15].

2.3.1 Formal Definition

Obviously, the topic of software architecture (SA) is frequently discussed. There are many conferences organized by respected organizations (e.g., IEEE International Conference on Software Architecture[4], O'Reilly Software Architecture Conference[5]), many people bear a job title *Software Architect* or *Solutions Architect*[6]. Despite all that, there are many definitions of *software architecture*, and none of them is considered universal.

For example, the IEEE Computer Society defines software architecture as:

> [Software] Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. [16]

One of the pioneers of software architecture, Len Bass, defines this complex discipline as:

> The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. [17]

After introducing these two definitions, the problem of SA seem to be highly abstract and difficult. It is clear there's a need of an instrument, that will help break down this complex domain. Such an instrument is featured in the next chapter.

[4] http://icsa-conferences.org/
[5] https://conferences.oreilly.com/software-architecture/
[6] https://www.glassdoor.com/

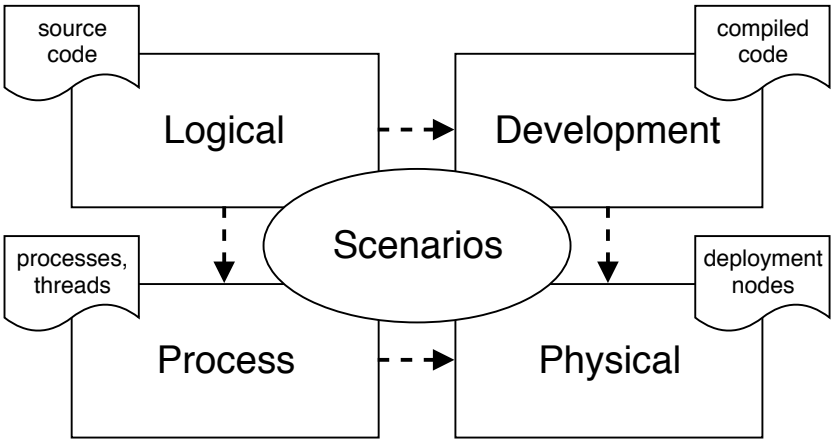

Figure 2.2: 4+1 View Model of Architecture

2.4 4+1 View Model of Architecture

The *4+1 View Model* was designed by Philippe Kruchten as a tool *describing the architecture of software-intensive systems, based on the use of multiple, concurrent views.* [18]

Architects may use the four views to systematically depict the miscellaneous SW elements and the fifth view — scenarios (or *use cases* in contemporary terminology) which *illustrate* or *animate* the static system.

The view model is *generic*. It does not prescribe any particular notation for the different perspectives, or any set of available *architectural patterns* or *styles*, hence allowing the multiple styles coexist in one system.

2.4.1 Views Description

- **Logical view** describes the elementary code fragments produced by developers — classes, funtions, configuration files, etc.
  - **Components:** class (OOP), function (FP)
  - **Connectors:** association, inheritance, composition
  - **Stakeholders:** end-user
  - **Concerns:** functionality
- **Development view** depics the artefacts of a packaged runnable code. It describes modules (e.g., a JAR file) and components (e.g., a WAR archive or executable JAR) components and dependencies among them.
  - **Components:** compiled SW artefacts
  - **Connectors:** compile-time dependencies

  - **Stakeholders:** developer, operations engineer
  - **Concerns:** modularity, code reuse, portability, deployable artefact boundaries

- **Process view** represents components at runtime.
  - **Components:** processes
  - **Connectors:** inter-process communication (e.g., REST/SOAP/RPC call)
  - **Stakeholders:** developer, operations engineer
  - **Concerns:** performance, availability, fault-tolerance, data integrity

- **Physical view** describes how the running deployed artifacts are mapped to *deployment nodes*.
  - **Components:** nodes (i.e. containers, VMs, physical machines)
  - **Connectors:** network interfaces
  - **Stakeholders:** operations engineer
  - **Concerns:** scalability, performance, availability

- **Scenarios** are detailed recipes describing actions across the whole application. This view is redundant to the previous four — therefore marked as *+1* — but serves as a validation mechanism for the whole architectural vision.
  - **Components:** step-by-step scenarios
  - **Connectors:** use-case dependencies
  - **Stakeholders:** end-user, developer, QA engineers
  - **Concerns:** understandability

2.4.2 Relations between views

The four views are not fully independent or orthogonal. Elements of one view must relate to elements in other views, otherwise, the model has some inconsistencies (e.g., non deployed code artifacts, not utilized or inaccessible VMs). Directions of those relationships — *mappings* — between the views are shown in fig. 2.2 using the black arrows.

Although every software architecture could be viewed from all of those four viewpoints, it is not always necessary to draw all of them to describe the architecture sufficiently. For example, a simple web application running on a single machine with an embedded database does not need a physical view, as it would depict just one machine hosting one process. On the opposite side, systems with millions of lines of code may require logical view diagrams

containing thousands of classes and packages. This view would require a high level of abstraction to be understandable. The abstractions could become almost identical to the development view, and therefore render the logical view redundant.

The scenarios are always useful since they contain information about the purpose of the application — the business value.

2.5 Architectural Styles

The term *architectural style* is in similar manner used in civil engineering. Building were designed in *renaissance*, *functionalism* or *brutalism* styles. All buildings of the particular style were different, yet they shared the same materials, had similar properties (e.g., aesthetics, hygiene), and they were build to fulfill the same ideals.

That is surprisingly close to how it applies to software — the applications of particular style are built using the same set of elements and relations between them (e.g., classes and their composition), they exhibit the same properties (e.g., layering, modularity) and are build to fulfill the same non-functional requirements (e.g., high availability, rapid evolvability).

There is a often-cited definition of architectural style by David Garlanand and Mary Shaw published in 1994:

> An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. [19]

Architectural styles are usually characterized only from a single view. For example, a Model-View-Controller architecture (MVC) application is defined in logical view as a trio of interacting components, each with its defined responsibility and function. MVC does not say how the application should be deployed — it does not prescribe anything for the other views.

Next two sections provide a brief introduction to some of the architectural styles that are used and compared in further chapters.

2.5.1 Layered Architecture

Layered architecture organizes code elements SW into stacked modules — layers. Each layer has a properly defined purpose and responsibilities. Elements in those layers can interact only with elements from layer right above or right below. A layer can only depend on a layer right below.

This architecture style is sometimes called the *n-tier architecture*, where *n* can be replaced with an integer. A frequently used instance is the *3-tier architecture*:

- **presentation layer** handles user-interaction code
- **business logic layer** contains the core functions and algorithms
- **persistence layer** handles database transactions

Such architecture elegant in its simplicity, and this principle may be applied to any of the four views.

However, major drawbacks arise for advanced applications. Imagine a logical view of a simple web application. What if a customer running the website wants to add a mobile application to his portfolio? In the style of layered architecture, the code of the mobile app would belong to the presentation layer, causing the web FE and mobile app code to be mixed. If it would be put to a separate layer, the mobile app would have to interact with the business logic layer through the the web FE.

2.5.2 Hexagonal Architecture

The *Hexagonal Architecture* was first proposed in 2007 by Alistair Cockburn, one of the co-authors of *The Agile Manifesto*[7].

It aims to overcome limits of the layered architecture. The business logic is isolated in the centre (*Core Application* in Hexagonal Architecture) and exposes interfaces called *ports*.

Ports have two directions:

- **inbound** handle the invocations of the business logic from the outer world, usually an application programming interface

[7] https://agilemanifesto.org/

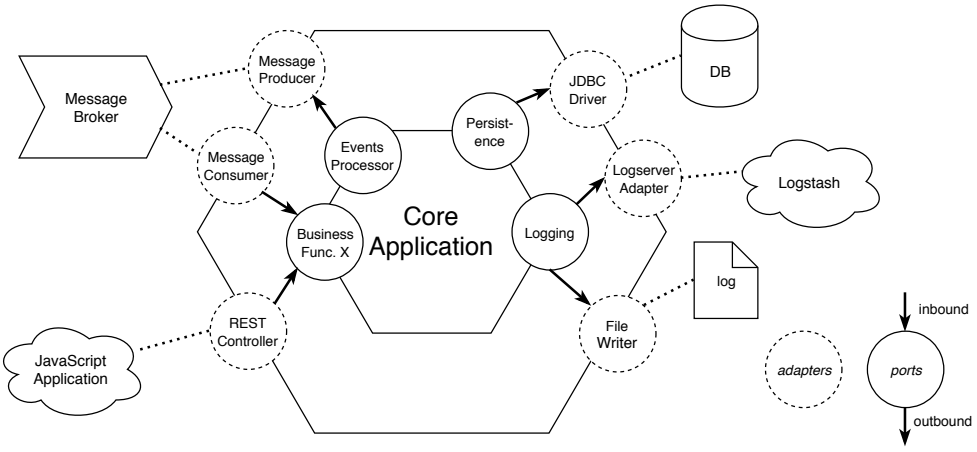

Figure 2.3: Hexagonal Architecture

- **outbound** is how the business logic interacts with external systems, e.g. a data persistence, logging, message publishers

Between a port and an external system is an *adapter*, which bridges the business logic port with the specific technology (e.g., a data access object). The core application does not depend on the adapters. Therefore it is easy to swap, e.g., a generic ORM for a custom optimized DAO.

More than one adapter may be bound to a single port (e.g., the same business logic action is invoked by a message received or by a REST call).

Decoupling using ports and adapters allows easy extendability without any changes introduced to the business logic. It also makes the core application testable without the need for the surrounding components (e.g., the whole application can be black-box tested without a need for DB)

2.5.3 Monolith Architecture

MSA in this thesis is often compared to a *monolith* or a *monolithic architecture*. This term is a polyseme — it is applied to more than one of the four architectural views and has a different meaning in each of them.

Originally it was used in context of the layered architecture (i.e., in the logical view). It was a synonym for the 1-tier logically unstructured code base [20, 21], usually exhibiting the same poor properties as the *big ball of mud* [15].

On the dawn of microservice architecture, it needed to be compared to the traditional single-process application — so the word monolith received a new meaning. In [22] it is defined as:

> A monolith is a software application whose modules cannot be executed independently.

In this context, *monolith* does not have the pejorative connotation, as it often represents the well structured, n-tier application (see Figure 4.1: *Monolith vs. Microservices Illustrated*).

In this thesis the word *monolith* is used exclusively with the second meaning.

Chapter 3

Microservice Architecture Literature Review

This chapter surveys the available academic literature and other sources of information on the topic of *microservice architecture*.

Section 3.2 investigates sources of information for the topics of *evolvability of microservice architecture* and *application of Normalized Systems theory to software architecture*. Then it compares the scope of this thesis to state-of-the-art academic research and industry publications.

The retrieved knowledge is synthesized and critically analyzed in chapter 4: *Analysis of the Microservice Architecture*.

3.1 Available Sources

Microservices as an architecture style started to emerge around year 2013 [3]. Since then, the research is lead by industry and professionals. Academia is was a long time reluctant to this field (see Table 3.1: *Number of articles including keyword microservices per year*).

Since a lot of know-how has been published on the Internet before the phenomenon reached scientific journals, many reviewed articles reference to those original ideas, e.g., [22, 23, 24, 25]. To track back the original sources and to follow the most recent trends, an unorthodox category of information sources is created — the community.

3.1.1 The Community

The microservice boom was kicked-off by practitioners — developers of big successful companies presenting their know-how on blogs, conference talks and even open-source SW code repositories.

It is almost impossible not to stumble upon the Martin Fowler's blog[8] and blogs of companies like Netflix[9] or Spotify[10]. Those are the entry-points to the heterogeneous web of cross-referenced documents, which contain fragments of the bleeding-edge expertise on the MSA — original ideas, success stories as well as failures and dead ends.

Due to the unstructured nature and temporality of the content, systematic review would require an unimaginable effort. Therefore it was believed that the community will expose and reference the most interesting ideas and experiences. The lack of systematic approach was balanced out by author's proficiency in this field, and his personal interest.

This category consists of various non-reviewed Internet sources. Trustworthiness of the content was assessed by cross-checking the information, reputation of the author(s), and to a lesser extent the publication medium.

Summary

The community provides an unstructured but exhaustive overview of all the aspects of MSA. With the exception of Martin Fowler's blog, community content misses overall visions. All of the sources also lack formal definitions, terminology or taxonomy. The community also prefers colloquial *story-telling* to an unambiguous formal language. Despite all of that, it was invaluable source of most recent ideas and inspiration.

3.1.2 Industry Literature

The professional literature publishing business offers vast numbers of books on this topic, so the long list had to be filtered. The criteria of the selection were:

- analysis of MSA in general (i.e., not limited to *testing MS* or *containerization of MS*)
- no focus to a particular technology (i.e., not limited to *building MS with Spring*)
- focus on the software architecture domain (i.e., not dedicated to *operations*, *agile development* and other adjacent domains)
- good reputation of author or publisher

[8] https://martinfowler.com/microservices/
[9] https://medium.com/netflix-techblog
[10] https://labs.spotify.com/

Table 3.1: Number of articles including keyword *microservices* per year

	SpringierLink	**IEEE Explore**	**ScienceDirect**	**ACM DL**
2013	5	1	0	0
2014	3	3	0	0
2015	30	24	6	7
2016	125	76	30	40
2017	218	153	53	62
2018	350	204	118	65

The final list of book, that were analyzed thoroughly and are considered the essential engineering knowledge about the domain of MSA:

- Building Microservices [5] — provide an comprehensive overview of MSA, includes currently used patterns and best practices
- Building Evolutionary Architectures [10] — focuses on how to design SW systems to better absorb changes (i.e. not limited to MSA), includes best practices
- Microservice Patterns [26] — focused on advanced architectural patterns, but still relevant for all the MSA domain
- Tao of Microservices [27] — provide an comprehensive overview of MSA, provides another point of view, author often uses logical argumentation or statistical data to support arguments, tends to be systematical

There are also two books freely available on the internet, both sponsored by industrial giants. [28], sponsored by CA[11], offers good overall introduction, but it's scope is completely covered by [5]. [29], sponsored by Microsoft to promote their Azure[12] platform, first 30 % contains very brief and understandable introduction to MSA domain, suitable for complete beginners. The remaining 70 % is dedicated to Azure.

Summary

The industry literature provides an overall vision, it is coherent, comprehensible and concise. It describes a lot of aspects of MSA, yet it claims it is not complete. The literature also lacks formal definitions, leastwise it uses seemingly uniform terminology. All of the books are focused on *getting things done* and do not see any value formal and systematic approach.

[11] https://www.ca.com/us.html
[12] https://azure.microsoft.com/

3.1.3 Academic Literature

There were two main goals for this part of the review:

- to get an insight to the state-of-the-art research of MSA
- to find some formal definitions, a terminology, or taxonomy — the topics the two previous categories of sources do not offer

Search engines used for the academic sources were: SpringierLink[13], IEEE Explore[14], ScienceDirect[15], ACM Digital Library[16] in a discipline *Computer Science*, *IT* or similar.

The retrieved articles were be divided into these categories:

- **Introductions to MSA** — general summaries of the MSA domain [24, 30, 31], some providing formalization and definitions [22]
- **Case study** — experience with some particular technology (e.g., *Jolie* programming language [32]) or domain (eCommerce [33], parking system [34])
- **Narrow focused** — focus on some particular aspect of MSA (e.g., financial rentability [35], monitoring and testing [36])
- **Literature review** — exactly one [23]

Summary

The **literature review** by Martin Garriga [23] served as the guidepost for the scientific literature until the year 2017. Garriga's review was done using the systematic literature review (SLR) method [37], therefore it's considered to be complete.

Alongside the exhaustive literature review, author proposes a MSA taxonomy extensively covering many aspects of the architecture style.

The articles in **Introduction to MSA** somehow summarize the expertise already published by the community or professional literature. A few of them attempt to provide formal definitions of commonly used terms, on the other hand, the definitions are not respected out of the scope of the single paper.

The **case study** and **narrow focused** introduce new information, unfortunately they were too specialized to fit into the scope of this thesis.

[13] https://link.springer.com/
[14] https://ieeexplore.ieee.org/
[15] https://www.sciencedirect.com/
[16] https://dl.acm.org/

3.2 Related Work

3.2.1 Academic Literature

Normalized System Institute books

The first published NS book [7] provides a brief discussion of compliance of the service-oriented architecture (SOA) with NS. Authors note that SOA may exhibit desired properties of evolvability, but do not provide any further argumentation.

The second NS book [1] contains plenty of related information:

- Chapter *12 Toward Stable Modular Software Architectures* provides a prescriptive design rules for stable software architectures.
- Chapter *15 Normalized Elements for Software Architectures* presents essential building blocks of evolvable architectures — *normalized elements* — and shows the implementation of those elements on code examples.

Content of both chapters is highly relatable to the focus of this thesis. Although the knowledge is not effortlessly applicable to the distinctive domain of microservice architecture.

Science Journals

The domain of *evolvability of microservice architecture* seems to be a virgin territory in scientific journals. Searches performed in databases of available science publishers (see 3.1.3) yielded exactly zero relevant results. The search results were not always an empty list (due to generality of some keywords, e.g., *change* or *distributed system*), but the after closer examination the returned results were all considered irrelevant.

Keywords were all possible pairs composed of members of two sets:

- microservices, microservice architecture, service oriented architecture, distributed systems
- evolvability, change, normalized systems

3.2.2 Industry Literature

The book *Building Evolutionary Architectures* [10] to some extent overlaps with focus of this thesis — as its topic is resistance of software system to change and a great portion is dedicated to microservice architecture. On the other hand, the authors of the book does not have any formal theory to back up the arguments. Every argument is based on their, undoubtedly immense, experience.

3.2.3 The Community

Bloggers and programmers often consider *evolvability* a virtue. However, it is not a frequently discussed topic and the community does not provide any relevant information on this particular problem.

Chapter 4

Analysis of the Microservice Architecture

Microservice architecture is described in process view — simply as a collection of processes communicating with each other. Although logical and development views may be organized organized in any way, MSA strongly endorses one rule: share nothing with others. This extremely radical decoupling allows swift development of independently deployable artifacts.

This architecture style is exploited by leading companies in the online business — e.g., Amazon, Ebay, Netflix, Spotify, Uber, SoundCloud and many

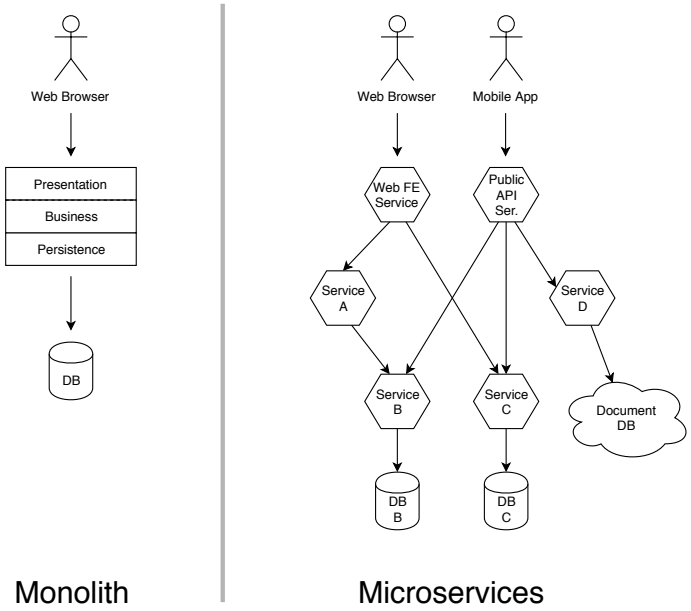

Figure 4.1: Monolith vs. Microservices Illustrated

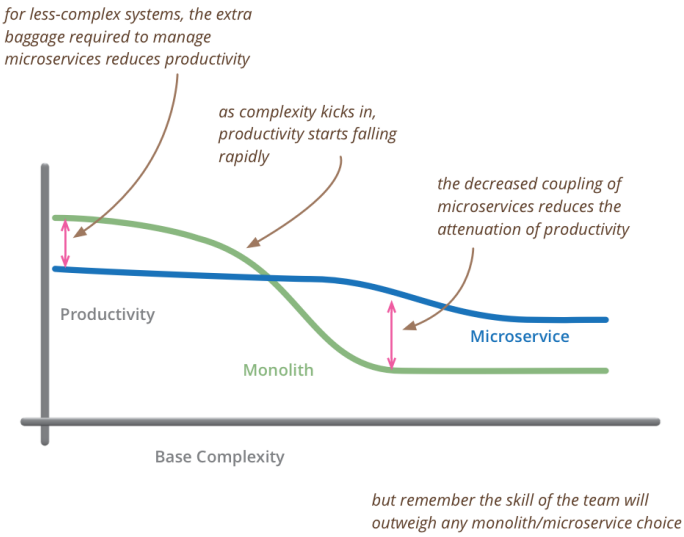

Figure 4.2: Monolith vs. Microservices — Productivity to Complexity ratio [45]

more [38]. The enterprises praise adoption of this style due to its two crucial features: easy *scalability* to their extreme dimensions [39, 40, 41, 42], and rapid development of new features — *evolvability*. Modern enterprises are able to create new functionality on an unprecedented rates - online marketplace Etsy is able to release new features 50 times a day and Amazon deploys new code to production every 11.7 seconds [43].

Loose coupling and independently deployable artifacts are the essential principles of MSA. Not only it allows teams to deploy their artifacts services without waiting for other teams, it allows them to choose any 3rd party technology they want to use. Teams handle th

In spite of all the listed advantages, microservice architecture is no silver bullet. This style adds a lot of obvious as well as hidden complexity — complicated operations, asynchronous communication, fallacies of distributed computing [44], changes in organization necessary to deal with *Conway's law* [6]. (More in subsection 4.1.5: *Trade-offs of a Distributed Architecture*).

It's not easy to decide when the pros MSA compensate the cons. Analysis of the proper decision making is too complex to fit in scope of this thesis, but the general rule of thumb is:

> Do not even consider microservices unless you have a system that's too complex to manage as a monolith. [45]

Microservices is a buzzword now and *since the successful large companies adopted this architecture, it must be the key to their success* — this is a portrayal of a dangerous phenomenon called *Microservice Envy*. This assumption

may lead a lot of R&D departments to unnecessary struggle. The Thought-Works company made a website dedicated to monitoring the state-of-the-art of MSA knowledge and tooling to help other teams to avoid this pitfall.[17]

4.1 (De)composition of Microservice Application

This section looks closer on the eminent implications and concerns creating a microservice application.

4.1.1 Microservice Scope

Microservice applications are composed of *microservices*. The preposition *micro* suggests existence of some *milliservices* or *nanoservices*. Or at least that the *microservice* itself should be very small. This might be very misleading interpretation. If we consider the common rule, that the microservice code base should be owned by one cross-functional team [5, 10, 3] with top limit of 8-10 people, the deployed artifact might not be small at all.

MSA could be used for project starting from scratch (a *greenfield* project), or for splitting a monolithical code base (a *brownfield* project). Regardless of the origin, the recommended first step is always deep analysis of the application domain. The well-known methodology for this purpose is *domain driven design* (DDD) exquisitely described in [46].

Based on a particular domain model, the development team can identify real-world components. Each component is responsible for a *business capability* — in DDD it is called a *bounded context*. Seams between the bounded contexts are the first candidates for microservice boundaries.

This approach is the essence mentioned in the industry-oriented books [5, 10, 26, 28] as well as science journals [47, 48, 49, 50, 51] and many blog posts (e.g., [3, 4]). Yet, this is the only advice that is given by the many authors of the professional literature. The rest is upon the particular domain, experience of the enterprise architecture and the team delivering the service.

4.1.2 Inner vs. Outer Architecture

Until now, the MSA was discussed as a cluster of independently deployed services and communication between them. On the other hand, the resultant artifact was always addressed as one *application* — that means the services construct an integral system. In order to keep balance between the freedom of the inner workings of a MS and complexity of operations, another border between *inner* and *outer* architecture must be defined (see fig. The Figure 4.3: *Inner vs. Outer Architecture*).

[17] https://www.thoughtworks.com/radar/techniques/microservice-envy

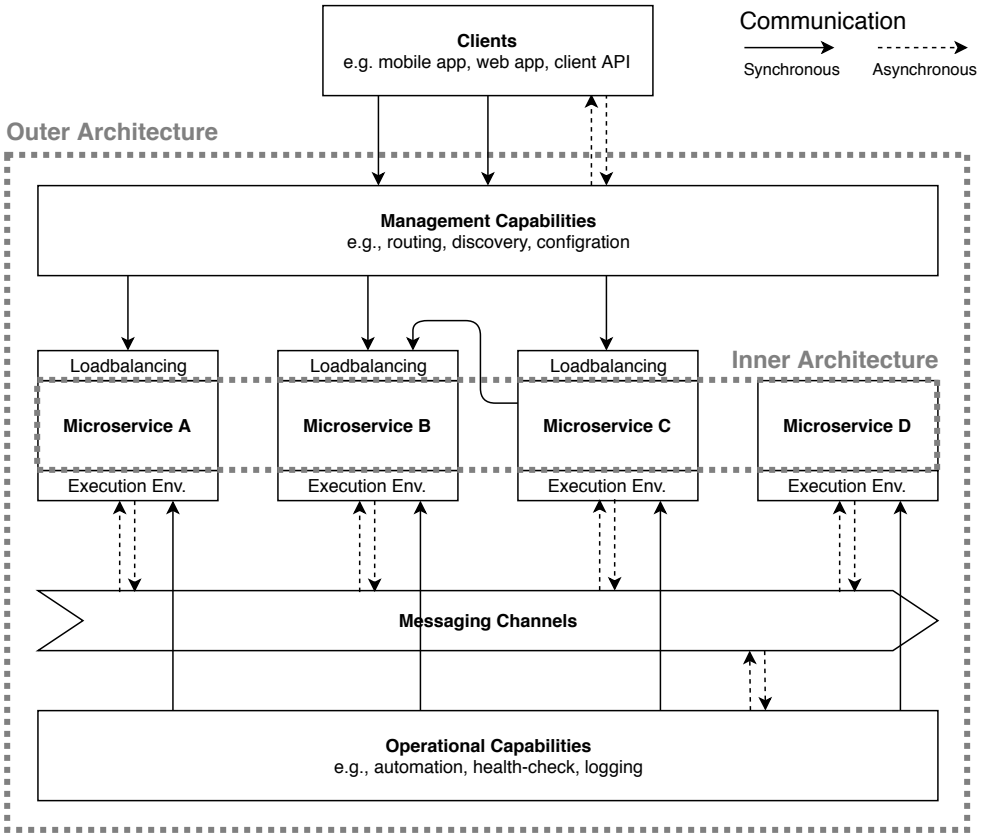

Figure 4.3: Inner vs. Outer Architecture

The **outer architecture** includes direct communication between services and other supporting systems (e.g., monitoring and logging), the execution environments (e.g., docker containers or VMs), networking (including load-balancing, routing etc.) and messaging channels (i.e. message brokers). It does not describe just the inter-connections, but more importantly the *contracts* between them (i.e. the API definitions).

The **inner architecture** is everything else — the internal business logic, the choice of programming language, 3rd party libraries and frameworks (if it's possible to run them in the supported execution environment), the data persistence (everything from ORM through schema to the choice of DBMS is hidden from the outer world — as long as it is able to be executed in the unified environment).

4.1.3 Technological diversity

The freedom of the inner architecture is often considered an advantage of MSA. It allows developers to *select the best weapons* — the best programming

language, the best 3rd party technology, the most suitable DB schema. It allows experimentation with all the aspects and that keeps the developers motivated [5, 52], fosters the culture of innovation [53], reduces the chance of lock-ins for outdated technology [5] and helps keep the whole application loosely coupled (see 4.1.4).

4.1.4 Strong Module Boundaries

A common recommendation in SW development is to have clearly defined code module boundaries. It's the manifestation of loose coupling and Douglas McIlroy's vision (see subsection 2.1.2: *Essential Principles*). MSA is no different.

Respecting the strong module boundaries does not depend on the architecture style — it is possible to create a perfectly modular monolithic app. It is only the matter of developer's discipline.

The microservice architecture magnifies the need for modularization by its intrinsic features, e.g., separate build artifacts for independent deployment, freedom of choice in technologies or persistence encapsulation.

Therefore, a developer feels much stronger guilt for including code from another service, when he knows the code he *borrows* might be rewritten to completely different programming language without prior notice.

Sometimes the discipline means breaking some established but blindly-followed guidelines and best practices. For example the undisputed do not repeat yourself (DRY) principle must be in context of MSA be applied only on the microservice level, not across the whole application.

Authors of the up-to-date industry literature [5, 10, 28] also recommend an extreme caution when creating an internal shared library of common functions and/or data structures (e.g., DTOs) since it leads to unrestrained coupling and the time invested into creation of such artifacts limits the technological diversity.

4.1.5 Trade-offs of a Distributed Architecture

This section analyses the costs of adoption of the microservice architecture. Every benefit listed in the previous section has a cost of its own. And almost all of them increase the overall complexity of the system.

4.1.5.1 Distribution

MSA utilizes a distributed system to provide fine-grained modularity. The fact it is distributed bring a lot of complexity and possible unintended consequences.

L. Peter Deutsch compiled a list called *Fallacies of distributed computing* [54] in 1994. The list consists of eight false assumptions which must be kept in mind while designing and operating a distributed system:

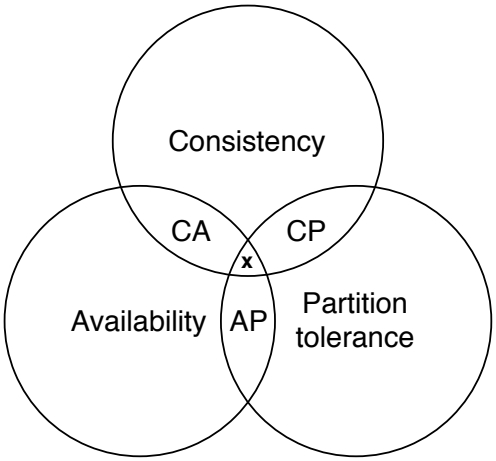

Figure 4.4: CAP Theorem

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology does not change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

All of the listed fallacies are still valid [44], although effects of some are were alleviated due to the advance of related technology.

- **The network is reliable** has a whole subsubsection 4.4.1.2: *Availability and Resilience* dedicated to dealing with this phenomenon.
- **Latency is zero**, **transport cost is zero**, and **bandwidth is infinite** have a great impact on applications performance. The ways how to mitigate this are discussed in section 4.2: *Inter-microservice Communication*.
- **Topology does not change**, and **the network is homogeneous** is mitigated using e.g. service discovery, discussed in **??**: **??**
- **The network is secure**, and **there is one administrator** belong to the application operations domain, therefore are out of scope of this analysis.

4.1.5.2 Eventual Consistency

The MSA is a distributed architecture therefore it exhibits implications of the CAP (also known as Brewer's) theorem [55]. The theorem states that

a distributed data store can't provide more than two out of the following properties:

- **Consistency** — each read receives the most recent write or an error.
- **Availability** — each request receives a non-error response, although without a guarantee to contain the most recent write.
- **Partition tolerance** — the data store responds with an non-error reply despite an erratic number of messages being lost or delayed in communication between nodes.

The implications of this phenomenon are known to all users of some of the present-day web applications — missing updated data, inconsistent notifications counts, etc. That's the nature of distributed architecture: the request to update data is handled by green node, the request for listing the data is handled by blue node. Until the blue and green nodes exchange information, the user is stuck in an *inconsistency window*.

For users, the eventual consistency is just annoying. For a programmers it is a whole lot of added complexity they have to deal with. The logic they've might end up making decisions on inconsistent information. This issue is exceptionally difficult to debug as the numbers of possible situations may rise extremely.

Design patterns guaranteeing the eventual consistency already exist (see section 4.3: *Transaction management*), but for some use cases it might not be good enough. The *eventuality* will probably never be accepted in mission-critical financial, traffic control and similar systems. Since the CAP theorem is formally proven, the MSA will be hardly ever adapted in domains with such requirements.

4.1.5.3 Operational Complexity

By splitting up the monolithic code base, the complexity of the code base itself lowers, but it leaks into the space between the microservices — into the operations of the application. The number of deployed code artifacts rise; number of DBMS and NoSQL DBs rise; load-balancers, message brokers and monitoring and management systems need to be deployed.

The swarm of tiny services is almost impossible to handle one by one. That reinforces the importance of continuous integration (CI) and continuous delivery (CD) tools and practices, as well as it enforces the implementation of the DevOps ideas and culture.

Distributed architecture also necessitate additional supporting systems, such as monitoring dashboards and centralized logging. Lack of those systems render almost impossible to trace back a bug in the application.

4.2 Inter-microservice Communication

The *inter-microservice communication* is a specialization of a general problem in IT called inter-process communication (IPC). In the professional literature the discussion of this specialization of the problem usually resorts to even greater specialization of discussing particular problems and patterns (e.g. REST vs. RPC or XML vs. JSON). An abstract taxonomy was needed, the one used here is adapted form [23] and extended.

4.2.1 Interaction Models

Interaction model defines how entities interact with each other in two perpendicular dimensions:

1. whether the requester is blocked by the reply or not
   - **synchronous** — the requester expects an immediate response from the replier, requester can't continue execution until response is received or timeout expires
   - **asynchronous** — the requester submits a request and continues execution, received response is processed independently
2. the number of receivers of a request
   - **one-to-one** — each request is processed by exactly one receiver
   - **one-to-many** — each request is processed by multiple receivers

Table 4.1: Interaction Model to Interaction Style Matrix

	one-to-one	**one-to-many**
sync.	• (sync.) request/response	—
async.	• one-way notifications • async. request/response	• publish/async. responses • publish/subscribe

4.2.2 Interaction Styles

Since both of the model dimensions must have a value, actual communication is modelled by the carthesian product of those two dimensions — an *interaction style* (see tab. 4.1).

The chosen interaction style has a major impact on the degree of final coupling between two requester and replier. In general, the synchronous request/response style causes severe coupling; the asynchronous is coupled very loosely.

- **(synchronous) request/response** — A requesting entity makes a request to a replier and waits for a response; the requester expects the response to arrive promptly, because it can't continue execution until response is received (or timeout expires).

- **asynchronous request/response** — A requesting entity makes a request to a replier, which replies occasionally. The requester is aware of the delay continues execution.

- **publish/async. responses** — A requesting entity submits a request and then waits for a certain amount of time for responses from interested repliers.

- **publish/subscribe** — An entity publishes a message to a channel; the message is consumed by zero or more subscribed consumers.

- **one-way notifications** — A requesting entity makes a request to another entity, but no reply is sent nor expected.

4.2.3 Service Contracts

Service contracts are the different means of specifying contracts (protocols) for the inter-process communication [22]. The possible values are:

- **formal** — defined through a formal, technology agnostic contract (e.g., Swagger)

- **technology-tied** — the contract is pre-defined by or tied to a specific technology (e.g., message broker client)

- **ad-hoc** — defined in a novel, ad-hoc language

4.3 Transaction management

A common practice in MSA is the *one DB per microservice* pattern (see 4.4). Consolidating data scattered around the swarm of microservices demands non-trivial architectural patterns. Transaction management is crucial aspect of the IS, as it is used to execute complex workflows.

There are two sorts of transactions in the MSA: those, which guarantee ACID [56] properties (see 4.3.1), and those which sacrifice isolation for better latency (see 4.3.2).

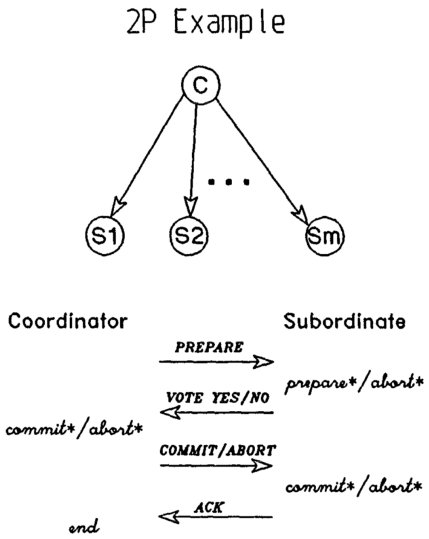

Figure 4.5: Two-phase Commit [57]. Schema from the original paper from 1986. The words in italics are commands. An * indicates that the record is forced to permanent storage.

4.3.1 N-Phase Commit

The first protocol to handle handle data transactions in distributed systems was a **2-phase commit** (see fig. 4.5). The greatest disadvantage is that is is a blocking protocol — if the coordinator stops responding, the subordinates may never resolve the started transactions and will be blocked until a commit or rollback command is received.

To resolve the issues with blocking and failures, a **3-phase commit** [58] was invented. Unsurprisingly, it incorporates one more phase to ensure all participants successfully processed the data, and it places an upper bound on the amount of time required before a transaction automatically commits or aborts. This ensures the resources held by particular transaction is always released.

4.3.2 The Saga Pattern

Although the 3-phase commit is safe and non-blocking way to manage distributed transactions, it still considerably prolongs the response time. Users are used to the comfort of *snappy* applications, therefore a different mechanism had to be adapted to maintain the low latency — the *saga*. (Surprisingly, the notion of a saga is just a few months younger than the 2-phase commit. [59])

A saga is *a sequence of local transactions executed to maintain data consistency* [59]. Every *local transaction* must exhibit the ACID properties, oth-

erwise data inconsistencies may occur.

There are two ways of coordinating sagas:

- **choreography** — has no central point of control; participants listen to other participant's events and decides if an action should be taken or not; after their action is done, they announce it with publishing a message
- **orchestration** — a central control entity issues commands to participants what operation to perform via messages

The following two sections explain the two ways on a simple example and evaluates advantages and disadvantages.

The examples depicts a simple e-shop order fullfilment flow of an order (steps 3. and 4. may be executed in parallel):

1. an order is created
2. the customer's details are verified
3. payment is verified
4. warehouse starts expediting the order
5. order is completed

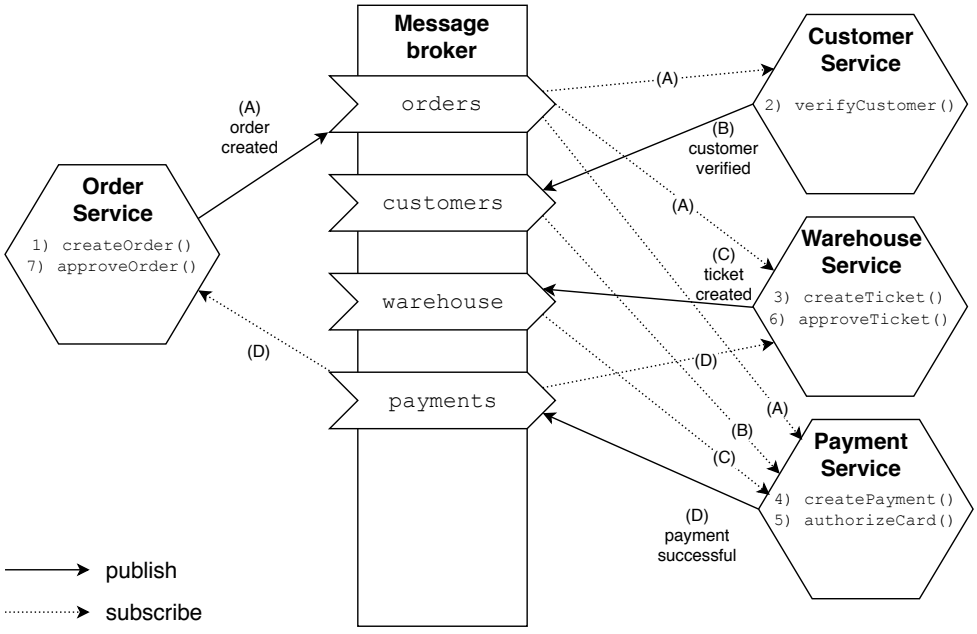

Figure 4.6: Saga Pattern — Choreography (happy flow)

4.3.2.1 Choreography

The **successful flow** is executed as follows (see fig.: 4.6):

1. **Order Service** receives an external API call to create an order, creates an instance of `Order` in state `ORDER PENDING` and publishes an `(A) order created` event.
2. **Customer Service** consumes the `(A) order created` message, verifies the customer's details and publishes an `(B) customer verified` event.
3. **Warehouse Service** consumes the `(A) order created` message, and creates an instance of `OrderTicket` in state `TICKET PENDING` and publishes an `(C) ticket created` event.
4. **Payment Service** consumes the `(A) order created`, and creates an instance of `Payment` in status `PAYMENT PENDING`.
5. **Payment Service** consumes `(B) customer verified`, `(C) ticket created` messages, and executes the card authorization, then it updates the status of `Payment` to `PAYMENT PROCESSED`, and publishes a `(D) payment successful` message.
6. **Warehouse Service** consumes the `(D) payment successful` message and updates status of `OrderTicket` to state `TICKET EXPEDITING`.
7. **Order Service** consumes the `(D) payment successful` message and updates status of `Order` to state `ORDER EXPEDITING`.

The flow when **a payment fails** is executed as follows (see fig.: 4.7):

1. **Order Service** receives an external API call to create an order, creates an instance of `Order` in state `ORDER PENDING` and publishes an `(A) order created` event.
2. **Customer Service** consumes the `(A) order created` message, verifies the customer's details and publishes an `(B) customer verified` event.
3. **Warehouse Service** consumes the `(A) order created` message, and creates an instance of `OrderTicket` in state `TICKET PENDING` and publishes an `(C) ticket created` event.
4. **Payment Service** consumes the `(A) order created`, and creates an instance of `Payment` in status `PAYMENT PENDING`.
5. **Payment Service** consumes `(B) customer verified`, `(C) ticket created` messages, and executes the card authorization, then updates the status of `Payment` to `PAYMENT CANCELLED`, and publishes a `(D) payment failed` message.

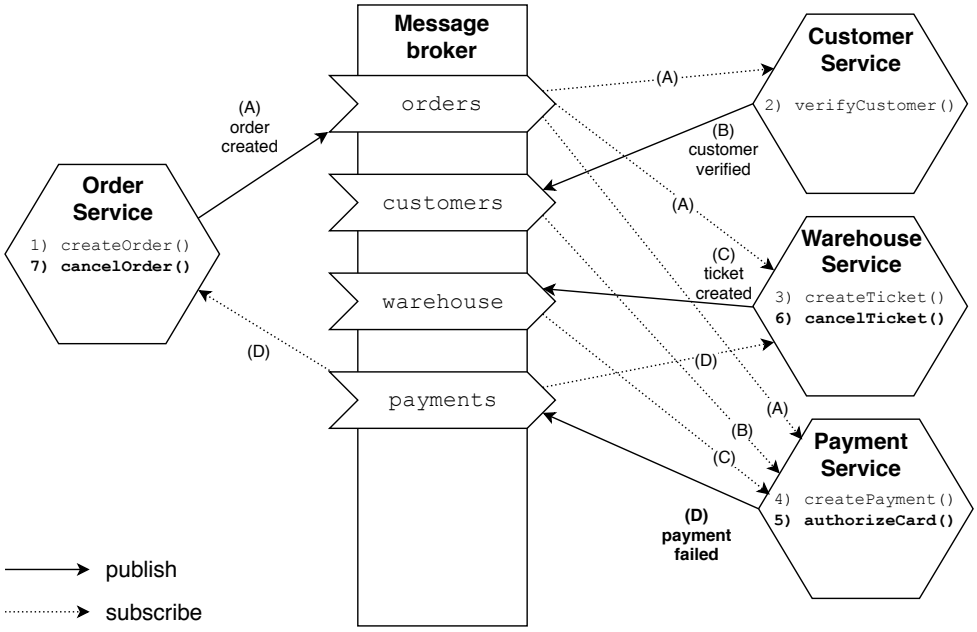

Figure 4.7: Saga Pattern — Choreography (error flow)

6. **Warehouse Service** consumes the `(D) payment failed` message and updates status of `OrderTicket` to state `TICKET CANCELLED`.

7. **Order Service** consumes the `(D) payment failed` message and updates status of `Order` to state `ORDER CANCELLED`.

To allow the choreography to function properly, messages must contain a *correlation identificator*, so the decoupled messages relate to the same instances of the objects. (For example the consumed messages `(A)` and `(D)` could not be associated with the same `OrderTicket` without an correlation id.)

Advantages

- **loose coupling** — participants listen to events and do not need to be aware of each other

- **simplicity** — services react to events, and publish events when they manipulate business objects

Disadvantages

- **difficult to understand** — there's is no central place of definition of the behaviour, it is dispersed among all participants

- **cyclic dependencies** — participants subscribe to each other's events, which might create a cyclic dependency

- **risk of tight coupling** — new functionality may cause unexpected coupling, e.g., the Warehouse Service should newly cancel the order in case some items on the order are not in stock, in that case the Order Service must subscribe to Warehouse Service messages — which tightens the coupling between the saga participants (more in [60])

4.3.2.2 Orchestration

The orchestration pattern uses an controlling entity — an *orchestrator* — to issue commands to other participants using the *publish / async. response* interaction style. The orchestrator is the only participant responsible for executing the workflow (including the failure path) and keeping the status of the workflow instance.

An orchestrator could be a dedicated service or a class inside a service containing other business logic (as is in the following example).

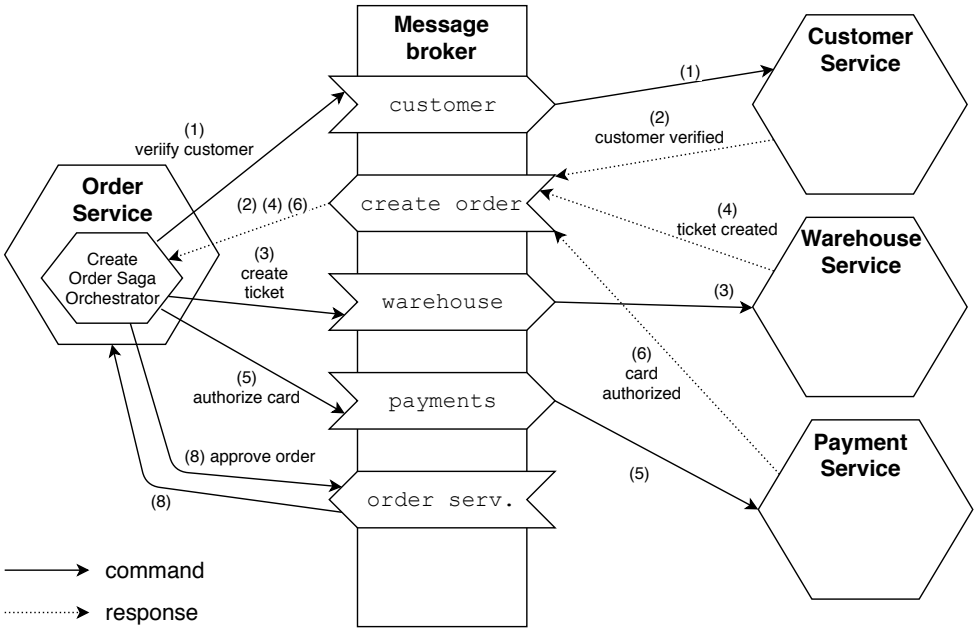

Figure 4.8: Saga Pattern — Orchestration (happy flow)

The **successful flow** is executed as follows (see fig.: 4.8):

1. **Order Service** receives an external API call to create an order, creates an instance of `CreateOrderSagaOrchestrator`.

2. `CreateOrderSagaOrchestrator` sends the `(1) verify customer` command to **Customer Service**.
3. **Customer Service** responds with `(2) customer verified` message.
4. `CreateOrderSagaOrchestrator` sends the `(3) create ticket` command to **Warehouse Service**.
5. **Warehouse Service** responds with `(4) ticket created` message.
6. `CreateOrderSagaOrchestrator` sends the `(5) authorize card` command to **Payment Service**.
7. **Payment Service** responds with `(6) card authorized` message.
8. `CreateOrderSagaOrchestrator` sends the `(7) authorize card` command to **Payment Service** which then continues with its own wofkflow.

Advantages

- **no cyclic dependencies** — orchestrator as the central entity depends on the subordinate participants, but they do not depend on the orchestrator
- **less coupling** — participants react only to the messages from the orchestrator and are not coupled to the other participants
- **better separation of concerns** — participants have less reasons to change, because they do not bear the burden of executing workflow; the orchestrator does not execute any bussiness logic but keeping the state of the workflow
- **workflow as a state machine** — because the workflow is concetrated in one place, it could be modelled and implemeted as a state machine, simplifying the design and implementation

Disadvantages

- **risk of God object** — if the orchestrator is a class inside a service (as in the example), the service will tend to contain all the related workflows, easily becoming a God object [61]
- **risk of power disbalance** — the orchestrator must only issue commands and do not execute any business logic and use the services as CRUD proxies

Table 4.2: Saga to Local Transaction Breakdown

cat.	#	Service	Transaction	Compensation Transaction
C	1	Order	createOrder()	cancelOrder()
C	2	Customer	verifyCustomer()	—
C	3	Warehouse	createTicket()	cancelTicket()
C	4	Payment	createPayment()	cancelPayment()
P	5	Payment	authorizeCard()	—
R	6	Warehouse	approveTicket()	—
R	7	Order	approveOrder()	—

4.3.2.3 Anomalies Caused by Lack of Isolation

The anomalies caused by lack of isolation encountered while executing saga are similar to anomalies of relational DBs.

List of possible anomalies:

- **lost updates** — one transaction overwrites unfinished writes of other transaction
- **dirty reads** — one transaction reads unfinished writes of other transaction
- **fuzzy/non-repeatable reads** — two different reads of one transaction make the same query and get different results because another transaction has updated the data

Local transactions may be categorized into the three following categories:

- **compensatable transactions** — transactions, that may be rolled back (using a *compensation* transaction)
- **pivot transaction** — the point of no return, only this transaction may fail irreversibly,
- **retriable transactions** — transactions guaranteed to eventually succeed (compensated by retrial)

In order to be able to compensate the the whole transaction, the local transactions must be executed in the three phases: *compensatable* — *pivot* — *retriable*. *Compensatable* and *retriable* phases may execute the local transactions in random order, even in parallel. The pivot transaction must be executed alone. (For an example of categorization of the local transaction of saga is in tab. 4.2)

4.4 Persistence

To achieve loose coupling between services, datastore is a private matter. The recommended pattern for greenfield projects is to have separated datastore per service kind (not instance). The microservice container itself is stateless, the state is kept in the database. The only allowed DB integration is with other instances of the same kind.

In a legacy systems, data storage could be used to integrate multiple services. When transitioning to microservices architectures, it is mandatory to find seams in the databases and use the right technologies to split them out cleanly [5].

Due to the one datastore per service pattern, there is no limit to selection of storage technology, therefore in one

4.4.1 Operation Aspects

4.4.1.1 Scalability / Elasticity

Scalability and elasticity is the the capability to quickly react to the sudden need of increase of the overall capacity of the platform. It is realized by adding or removing service instances. This whole process is also minimize the need for human intervention, using various tools or vendor-provided mechanisms.

4.4.1.2 Availability and Resilience

Resistance to failures is a crucial feature of microservice applications. Successful companies that adapted this architecture always design their systems for failure. Netflix has developed a tool called *Chaos Monkey* [62] that is able to randomly terminate services. They are running this tool in production environment, to be sure the whole system is robust and resilient enough to withstand the erroneous termination of deployed instances.

4.4.2 Observability

Observability is the term for measuring, collecting, and analyzing various diagnostics data from an application. These signals may include metrics, logs, profiles and more.

In large-scale microservices systems, some practices from the single process systems do not scale. To achieve a usable observability, some practices must be applied:

- External requests must have assigned a unique id to track across the services (the id must be passed with each inner call)
- Include the external id in logs

- Record all the information in centralized service

Chapter 5

Towards Stable Microservice Architecture

In this chapter, key aspects of MSA are extracted from chapter 4: *Analysis of the Microservice Architecture* and the Normalized Systems theory (NS) is applied. Section 5.1 contains list of identified subdomains and discussion whether they will be subject to further analysis. The section 5.2.2 attempts to map the patterns for normalized elements to the notions of MSA. Sections following the sec. 5.2.2 closely examine the respective subdomains.

5.1 Selected Aspects

Aspects were extracted as the outline of the chapter 4 and modified. The topics selected for further discussion are marked with check mark (✔), topics rejected are marked with a cross (✖) and the decision is briefly justified.

- ✔ **Microservice scope** — a crucial question which is now answered only with a very vague guidelines. Will be discussed separately.
- ✔ **Inner vs. outer architecture** — often neglected, yet important topic, closely related to *separation of concerns*. Will be discussed separately.
- ✔ **Polyglotism, technological diversity** — one of the most hyped benefits of MSA and potential source of complexity. Will be discussed separately.
- ✖ **Trade-offs of distributed system** — although this is a set of important topics, it is actually only a list of essential problems with distributed architecture. Solutions to those problems are discussed in subsubsection 4.1.5.1: *Distribution*. Therefore this topic won't be discussed separately.

- ✖ **Interaction models and styles** — only a taxonomy, does not represent real patterns or problems. The real communication complexity is discussed in section 5.5: *Transactional Management*.
- ✔ **Service contracts** — will be discussed in section 5.3: *Inner vs. Outer Architecture* and section 5.4: *Cross-Cutting Concerns*.
- ✔ **N-phase commits** — will be discussed in section 5.5: *Transactional Management*.
- ✔ **Saga pattern** — will be discussed in section 5.5: *Transactional Management*.
- ✔ **Persistence** — will be discussed separately.
- ✔ **External APIs** — will be discussed separately.
- ✖ **Deployment and Operations** — operations fall out of scope of this thesis; the interesting topics related to this subdomain are listed in section 5.4: *Cross-Cutting Concerns*.
- ✖ **Scalability / Elasticity** — scalability is a matter of operations, therefore does not fall into the scope of this thesis. The only requirements to the microservice is to be transactional and stateless (both are discussed in in section 5.5: *Transactional Management*).
- ✖ **Availability and Resilience** — both properties are essential for MSA, therefore are taken into account at every decision or proposal. Although improving both qualities is a matter of operations, therefore out of scope of this thesis.
- ✔ **Cross-cutting concerns** — will be discussed separately.

5.2 Microservice Scope

NS states that *a software element should have only one change driver*. A software element in this case is a microservice. Applying this rule would lead to extremely granular system — a MS for each data element or workflow would need its own service. If this rule is not respected, the number of change drivers rise and combinatorial effects emerge.

5.2.1 Microservice as a Normalized System

Let us assume that the microservice itself is a Normalized System. This way it is possible to avoid the extreme granularity, but it does not help with the original question.

There is also a definition of microservice by Sam Newman: *A microservice should be rewritten by a team of developers in two weeks.* [5] Implication of

this statement is that the complexity of one MS is bounded (to the two weeks of work of one team). Considering that, microservice itself as a Normalized System does not bring any value.

5.2.2 Mapping NS Elements to Microservice Architecture

This section explores the idea of building a microservices application as a Normalised System.

A NS element is a high-level design pattern, designed to provide a basic functionality of an information system [1]. The NS elements are aptly defined in [63]:

- **data elements**, to represent data variables and structures, and including support for cross-cutting concerns such as remote access and persistence support;
- **task elements**, to represent processing instructions, and including support for cross-cutting concerns such as remote access, logging and access control;
- **flow elements**, to handle control flow and orchestrations (i.e., the execution of a number of task elements on a specific target data element in a stateful way);
- **connector elements**, to allow the interaction with external systems (via a user interface or another application);
- **trigger elements**, to offer periodic clock-like control and checking whether a task element needs to be triggered.

These elements are theoretically proven to have only one change driver (reason to change), therefore they are free from combinatorial effects [1]. Also, it is experimentally proven that it is possible to create a viable information system (IS) [64] using only these elements.

The idea is to create a microservice *building block*, similar to a *NS building element*, which would allow imitation of the NS elements using services[18].

5.3 Inner vs. Outer Architecture

The notion of inner and outer architecture resembles the NS element approach — the cross-cutting concerns are separated from the business logic.

Normalized Systems consider persistence a cross-cutting concern. Due to the MSA pattern *one DB per service*, it is the private matter of the single

[18]This is probably the point where a veteran software engineer can't contain his indignation anymore, and stops reading while yelling: *This is ridiculous, spinning up a Docker container just to have a configured cron running somewhere! What a waste!* If you are one of those, please, hold on. This suboptimal use of system resources will hopefully result in infinite evolvability.

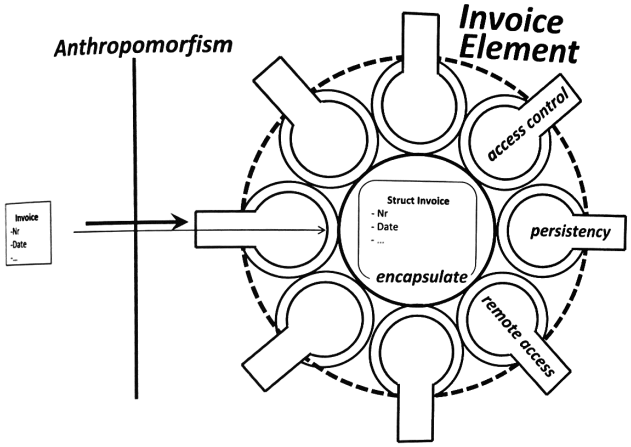

Figure 5.1: An example of NS building element [1]

service. None of the principles of the NS seem to contradict that assumption. Therefore, the definition of inner and outer architecture considered compliant with NS theory.

5.4 Cross-Cutting Concerns

Each NS element consists of business logic encapsulated in a *building block*. The building block executes the business logic code, as well as it provides access to cross-cutting concerns (e.g. persistence, access control).

This very same concept is used by various *microservice ready* frameworks, e.g., Spring Boot[19]. The Spring Boot application out-of-the-box contains some cross-cutting concerns (e.g. security and logging), others can be added via configuration (e.g. Actuator plugin for monitoring) and others can be added simply by including a dependency in a build script (e.g. JPA access).

The framework uses dependency injection to shield the programmer from implementation detail and the framework itself can be easily updated via Maven or Gradle build script.

To summarize, the engineering community is dealing with the cross-cutting concerns in a very similar manner, thus it is considered compliant with NS. (However, there is nothing to learn from NS,.)

[19]http://spring.io/projects/spring-boot

5.5 Transactional Management

5.5.1 N-phase Commit

The 2-phase as well as 3-phase commit is a relationship coordinator – cohort(s). That means it is orchestrated by a single entity. All communication and coordination is done through single entity and it is the only one keeping the state of the commit (i.e., of a workflow). Therefore, the number of dependencies in the transaction is not rising faster than the number of participants and does not became unbounded.

The fact that the transaction is blocking/non-blocking does not seem to break any of the rules of evolvability.

Therefore both transaction modes are considered compliant with a NS.

5.5.2 Saga

Saga has two possible modes of execution:

Orchestration

The orchestration is also a coordinator — cohort relationship. It exhibits the same properties as the 3-phase commit (it is non-blocking and eventually consistent). As a result, it is considered compliant with the NS theory.

It is also the preferred way to execute sagas in various literature, e.g., [26].

Coordination

Saga using the *choreography* principle exhibits a risk of tight coupling, since there's no fixed number dependencies on which of the particular participant depends, therefore, the number may eventually become unbounded and complexity rises unnecessarily. Therefore, this way of execution is not compliant with the NS theory.

This concern is raised in professional literature [26].

5.6 Polyglotism and Technological Diversity

In MSA the services communicate only via APIs or via message brokers, therefore they can not depend on each other's implementation details. Therefore, the polyglot approach does influence the architectural complexity.

5.7 Persistence

The one DB per MS pattern prohibits any integration throught database or other code smells, it is perfectly compliant with the *separation of concerns* design theorem.

5.8 External APIs

The both external and internal APIs must exhibit the *data version transparency*. This is easily achievable by versioning the API, e.g., using the version in URL. This already used and recommended practise in MSA.

Chapter 6

The Stable Microservice Architecture

This section attempts to apply the findings of the last chapter.

6.1 Microservice Building Block

This section proposes the *microservices building block*. It is an analogy to the NS building element (see 5.1). There are several functions this block fulfills:

- provides an execution environment for the business logic
- provides adapters for inbound and outbound ports, so the business logic is reachable by and can reach other microservices
- provides adapters for inbound and outbound ports, so the microservices are able to control the cross-cutting concerns it needs to reach (e.g., persistence access, centralized log output)
- handles the infrastructure cross-cutting concerns and shields the business logic from them (e.g., health-check, access management, service discovery client)

6.1.1 Building Block Layers

The building block consists of three layers and it is heavily inspired by the hexagonal architecture (see subsection 2.5.2: *Hexagonal Architecture*). The responsibilities of the layers are as follows:

- **business logic layer** — contains the custom code and algorithms

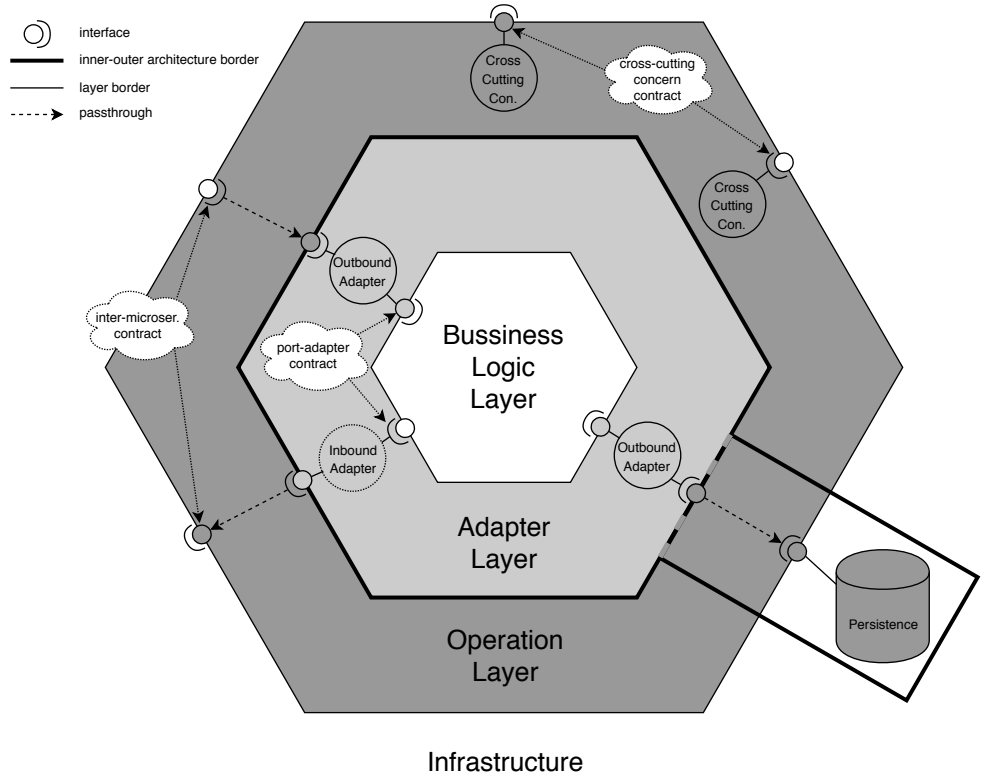

Figure 6.1: Proposed *building block* of MSA

- **adapter layer** — provides a bridge between the business logic and the runtime, separating the inner and outer architecture (see section 5.3: *Inner vs. Outer Architecture*)

- **operation layer** — the independently deployable runtime environment

The operation layer is the prefabricated MS building block. It serves the same purpose as a NS element of an application skeleton generated by the NS expander — it shields the business from the execution environment, thus making it free of combinatorial effects of the cross-cutting concerns.

The adapters fulfill the same function as in the original hexagonal architecture. It bridges the business logic and the technology surrounding it. Since the business logic does not depend on the adapters, but the adapters on the business logic. Therefore, it is necessary to define a *contract* — a language agnostic interface definition — between those two layers, so the adapters can be implement the functionality accordingly.

6.1.2 Building Block Contracts

Contracts are also the proposed way how to keep the whole system independent decoupled from the particular 3rd party technologies. It defines the interface regardless of the programming language, similar to SOAP messaging or REST calls.

Total of three categories of contracts for each building block is defined (see fig. 6.2):

- **port – adapter contracts** — decouples the business logic from the encapsulating framework; bound to programming language (or a group of compatible languages, e.g. Java, Kotlin, Scala)
- **inter-microservice contracts** — decouples the microservices themselves; it bound to the interaction styles (e.g., request/response, publish/subscribe)
- **cross-cutting concern contracts** — decouples the cross-cutting concerns from the infrastructure serving them (e.g., health-check, logging, access management)

Thanks to the contracts, the used 3rd technologies may be freely swapped. An application built using a framework A, but because of, e.g., security reasons, it needs to be swapped for a framework B. Both frameworks implement the REST standard (as the only inter-MS communication standart used). Both frameworks also implement a JSON web token (JST) authentication (as the only cross-cutting concern used). Thanks to the port – adapter contracts, the only respective adapters for framework B need to be re-implemented. Once it is done, the framework B may be deployed in the whole MS application.

6.1.3 An Example Building Block

To fullfill the functions listed in beginning of this section, the easiest way to create a building block is to assemble it from various third-party technologies. There's a plenty of tooling available that together create a complete MSA ecosystem.

Here is an example building block using a ready-made technologies: The most common way to create a independently deployable artefact in MSA is to use a application containerization tool, such as Docker[20]. The Docker is running a lightweight Linux distribution, such as AlpineLinux[21], which runs a Java Runtime Environment[22], which runs Tomcat[23], which runs a Spring-Boot[24] application. The Spring Boot application is bundled with a developer's

[20] https://www.docker.com/
[21] https://alpinelinux.org/
[22] https://openjdk.java.net/
[23] http://tomcat.apache.org/
[24] http://spring.io/projects/spring-boot

own, prepared adapters. Although this sounds extremely complicated, the Docker makes the whole process simple and automated out of the box.

Docker containers are built using *layers* stacked on top of each other. When something is changed, only the layer containing the change and layers above need to be rebuilt. Therefore rebuilding a prepared MSA building block with a custom code (the top layer) is a matter of seconds.

Even though the preparation of a single building block is relatively simple, a complete application would need a vast amount of such building blocks to cover a reasonable functionality. The viable solution seems to be the one that is utilized by Normalized Systems Expander — automatic code generation of an application skeleton based on a domain model, and then implementing the bussiness logic as a *custom code* (more about this approach in [1]).

6.2 The Normalized Elements of MSA

The empty building block from the last section may be further specialized to the normalized elements of MSA — the normalized services.

- **data services** — responsible for create, read, update, delete, search (CRUDS) operations of data
- **task services** — responsible for executing data operations on 1 or more data entities
- **flow elements** — responsible for executing workflows using saga-like transactions
- **connector elements** — responsible for external APIs
- **trigger elements** — responsible for triggering scheduled workflows and tasks

All of the elements are bundled with the vital cross-cutting concerns out of the box (logging, access control, message broker connectors, direct inbound and outbound calls etc.). The data and flow services also equipped with persistence so they persist the processed data entities as well as workflow status.

6.3 Example Usage

The following section is an example of a workflow in a system composed of Normalized elements of MSA. It represents a system excerp and shows only the communicating entities.

The usecase is very simple: *at the end of the month, calculate a salary of an employee from his timesheet.*

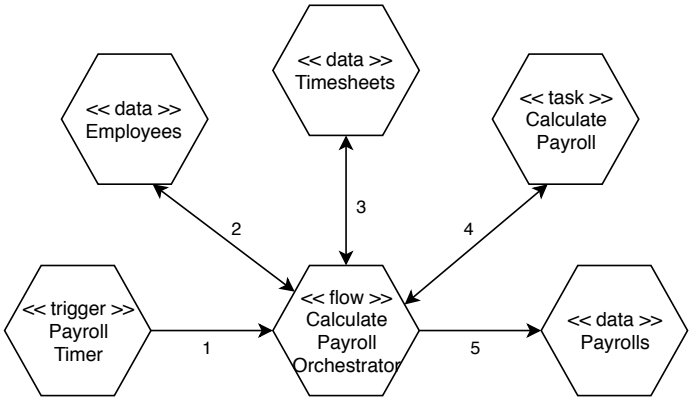

Figure 6.2: The Payroll Usecase

Scenario (for illustration see fig.: 6.2)

1. the `PayrollTimer` expires and triggers a `CalculatePayrollOrchestrator`, which created an instance of its workflow

2. the `CalculatePayrollOrchestrator` requests instance of an `Employee` data entity from `Employees` and gets a response

3. the `CalculatePayrollOrchestrator` requests instance of an `Timesheet` data entity from `Timesheets` and gets a response

4. the `CalculatePayrollOrchestrator` requests a payroll calculation from `CalculatePayroll` task service (using the `Employee` and `Timesheet` as arguments) and gets a `Payroll` entity back

5. the `CalculatePayrollOrchestrator` submits the `Payroll` entity to `Payrolls` data service.

6.4 Viability of the Proposed Method

As seen on the example, even such a trivial task requires a large number of services and very chatty communication between the services.

Large numbers of services mean an excess of used resources. Although, thanks to the flexible cloud environments *an excess of used resources* does not require *an excess of money* in hardware and Docker containers are lightweight compared to VMs. But still, resources are not for free and wasting them is wrong business strategy.

As is stated in the *fallacies of distributed computing* (see sec. 4.1.5.1), the network is slow and unreliable compared to local calls. The stumbling block in this case is the metric — the network latency — which is almost impossible

to scale or improve. That would render each non-trivial operation intolerably slow.

To conclude, this method is theoretically achievable, but not economically viable way to build information systems.

6.5 Other Observations

There might be some *guidelines*, *lessons learned* or proofs provided by the NS theory. For example it proves the high coupling in choreographed saga; encapsulation of bussiness logic code to a framework managing cross-cutting concerns (see 5.4).

On the other hand, this fact is also revealed by professional literature in microservices field (although, using just a simple example instead of logical proofs).

Confirmations of such recommendation or obvious facts is not credited to NS theory.

Conclusion

Author's comments on proposed solution

I, as an author, have had high expectations from this research topic. The solution I've proposed I the last chapter was wrong from the very beginning. It does not require strong imagination to realize that such system would have extremely high latency. I started this research with completely different idea.

The Normalizes Systems and microservice architecture share many characteristics: High modularity. The need for evolvability. Separation of concerns — the *do one thing but do it right* principle. Separation of states. The versioning of APIs resembles *data* and action version transparency. Both approaches appear to have the same challenges with cross cutting concerns. I believed that there must be something in Normalized Systems, that would help the craftsmanship of microservice practitioners to improve their know-how.

The development of the microservices is driven by immense experience and intuition, but it is also limited by restraint to do anything extreme — such as breaking an application to extremely small modules. And yet this has been proven to be the way to guarantee an infinite evolvability.

It was proven that applying Normalized Systems theory on microservices architecture would result in a system that is infinitely evolvable. However, it would be a waste of computational resources as well as human patience.

Evaluation of Goals

- *Perform a literature review on MS and on application of NS to MS and/or related architecture styles and patterns*

  Accomplished — the literature review was performed on vast amounts of resources. Although lacking systematical approach for the *community*, retrieved information served as an comprehensive base for the analysis.

- *Analyze the microservice architecture*

  Accomplished — the domain was analyzed thoroughly within a required scope. The key aspects and challenges of microservice architecture were identified.

- *Examine compliance of microservice architecture to Normalized Systems theory*

  Accomplished — the compliance with Normalized Systems theory was examined and discovered improvements were suggested.

- *Summarize design guidelines for MSA*

  Accomplished — a set of design rules was proposed and demonstrated on suitable example. A discussion of viability of the proposed method was done. The initially proposed case study was omitted due to the findings.

Bibliography

[1] De Bruyn, P.; Mannaert, H.; Verelst, J. *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design*. Antwerp, Belgium: Koppa, 2016, ISBN 978-90-77160-091.

[2] Amazon.com, Inc. *What Is DevOps?* [ONLINE], 2018, [accessed: 3. 6. 2018]. Available from: `https://aws.amazon.com/devops/what-is-devops/`

[3] Fowler, M.; Lewis, J. *Microservices: a definition of this new architectural term*. [ONLINE], 2014, [accessed: 1. 2. 2017]. Available from: `https://martinfowler.com/articles/microservices.html`

[4] Mahlen, P. *Modeling Microservices at Spotify with Petter Mahlen*. [ONLINE], 2106, [accessed: 5. 9. 2018]. Available from: `https://dzone.com/articles/modeling-microservices-at-spotify-with-petter-mari`

[5] Newman, S. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2015, ISBN 978-1-491-95035-7.

[6] Conway, M. *Conway's Law*. [ONLINE], 1967, [accessed: 25. 6. 2018]. Available from: `http://www.melconway.com/Home/Conways_Law.html`

[7] Mannaert, H.; Verelst, J. *Normalized Systems: Re-creating Information Technology Based on Laws for Software Evolvability*. Antwerp, Belgium: Koppa, 2009, ISBN 978-90-77160-00-8.

[8] Verelst, J.; Mannaert, H.; Huysmans, P. "IT Isn't Different After All": Implications of Normalized Systems for the Industrialization of Software Development. In *2013 IEEE 15th Conference on Business Informatics*, July 2013, ISSN 2378-1963, pp. 356–362, doi:10.1109/CBI.2013.58.

[9] Op 't Land, M.; Krouwel, M.; Dipten, E.; et al. Exploring Normalized Systems Potential for Dutch MoD's Agility. In *Practice-Driven Research on Enterprise Transformation*, *Lecture Notes in Business Information Processing*, volume 89, edited by F. Harmsen; K. Grahlmann; E. Proper, Springer Berlin Heidelberg, 2011, ISBN 978-3-642-23387-6, pp. 110–121, doi:10.1007/978-3-642-23388-3_5. Available from: `http://dx.doi.org/10.1007/978-3-642-23388-3_5`

[10] Ford, N.; Kua, P.; Parsons, R. *Building Evolutionary Architectures: Support Constant Change*. O'Reilly Media, Inc., 2017, ISBN 978-1-491-98636-3.

[11] Mannaert, H.; Verelst, J.; Ven, K. Towards evolvable software architectures based on systems theoretic stability. *Software: Practice and Experience*, volume 42, no. 1, 2012: pp. 89–116, ISSN 1097-024X, doi:10.1002/spe.1051. Available from: `http://dx.doi.org/10.1002/spe.1051`

[12] Lehman, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, volume 68, no. 9, 1980: pp. 1060–1076.

[13] McIlroy, M. D.; Buxton, J.; Naur, P.; et al. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, sn, 1968, pp. 88–98.

[14] Kolařík, V. *Applying OntoUML for structural definitions of Normalized Systems Expanders*. Bachelor's thesis, Czech Technical University in Prague, Prague, Czech Republic, May 2014.

[15] Foote, B.; Yoder, J. *Big Ball of Mud*. [ONLINE], 1999, [accessed: 4. 10. 2018]. Available from: `http://www.laputan.org/mud/mud.html`

[16] ISO/IEC 42010:2007(E). IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. Technical report, IEEE Computer Society, 2000.

[17] Clements, P.; Garlan, D.; Bass, L.; et al. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002, ISBN 0201703726.

[18] Kruchten, P. B. The 4+1 View Model of Architecture. *IEEE Software*, volume 12, no. 6, Nov 1995: pp. 42–50, ISSN 0740-7459, doi:10.1109/52.469759.

[19] Garlan, D.; Shaw, M. An Introduction to Software Architecture. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[20] Fedorov, A.; Francis, B.; Harrison, R. *Professional Active Server Pages 2.0*. Professional Series, Wrox Press, 1998, ISBN 9781861001269. Available from: `https://books.google.cz/books?id=hWZm_9NnvO8C`

[21] Wikipedia. *Monolithic application*. [ONLINE], 2018, [accessed: 20. 12. 2018]. Available from: `https://en.wikipedia.org/wiki/Monolithic_application`

[22] Dragoni, N.; Giallorenzo, S.; Lafuente, A. L.; et al. *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, ISBN 978-3-319-67425-4, pp. 195–216, doi:10.1007/978-3-319-67425-4_12. Available from: `https://doi.org/10.1007/978-3-319-67425-4_12`

[23] Garriga, M. Towards a Taxonomy of Microservices Architectures. In *Software Engineering and Formal Methods*, edited by A. Cerone; M. Roveri, Cham: Springer International Publishing, 2018, ISBN 978-3-319-74781-1, pp. 203–218.

[24] Sorgalla, J.; Rademacher, F.; Sachweh, S.; et al. On Collaborative Model-Driven Development of Microservices. In *Software Technologies: Applications and Foundations*, edited by M. Mazzara; I. Ober; G. Salaün, Cham: Springer International Publishing, 2018, ISBN 978-3-030-04771-9, pp. 596–603.

[25] Vural, H.; Koyuncu, M.; Misra, S. A Case Study on Measuring the Size of Microservices. In *Computational Science and Its Applications – ICCSA 2018*, edited by O. Gervasi; B. Murgante; S. Misra; E. Stankova; C. M. Torre; A. M. A. Rocha; D. Taniar; B. O. Apduhan; E. Tarantino; Y. Ryu, Cham: Springer International Publishing, 2018, ISBN 978-3-319-95174-4, pp. 454–463.

[26] Richardson, C. *Microservice Patterns*. Manning Publications Company, 2018, ISBN 9781617294549. Available from: `https://books.google.cz/books?id=UeK1swEACAAJ`

[27] Rodger, R. *The Tao of Microservices*. Manning Publications Company, 2017, ISBN 9781617293146. Available from: `https://books.google.cz/books?id=uosOkAEACAAJ`

[28] Nadareishvili, I.; Mitra, R.; McLarty, M.; et al. *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., first edition, 2016, ISBN 1491956259, 9781491956250.

[29] Familiar, B. *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*. Berkely, CA, USA: Apress, first edition, 2015, ISBN 1484212762, 9781484212769.

[30] Schwartz, A. Microservices. *Informatik-Spektrum*, volume 40, no. 6, Dec 2017: pp. 590–594, ISSN 1432-122X, doi:10.1007/s00287-017-1078-6. Available from: `https://doi.org/10.1007/s00287-017-1078-6`

[31] Gruhn, V. Die Organisation der Zukunft: Microservices. *Wirtschaftsinformatik & Management*, volume 10, no. 1, Feb 2018: pp. 52–57, ISSN 1867-5913, doi:10.1007/s35764-018-0022-0. Available from: https://doi.org/10.1007/s35764-018-0022-0

[32] Guidi, C.; Lanese, I.; Mazzara, M.; et al. *Microservices: A Language-Based Approach*. Cham: Springer International Publishing, 2017, ISBN 978-3-319-67425-4, pp. 217–225, doi:10.1007/978-3-319-67425-4_13. Available from: https://doi.org/10.1007/978-3-319-67425-4_13

[33] Hasselbring, W.; Steinacker, G. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, April 2017, pp. 243–246, doi:10.1109/ICSAW.2017.11.

[34] Yugopuspito, P.; Panduwinata, F.; Sutrisno, S. Microservices architecture: Case on the migration of reservation-based parking system. In *2017 IEEE 17th International Conference on Communication Technology (ICCT)*, Oct 2017, ISSN 2576-7828, pp. 1827–1831, doi:10.1109/ICCT.2017.8359946.

[35] Singleton, A. The Economics of Microservices. *IEEE Cloud Computing*, volume 3, no. 5, Sep. 2016: pp. 16–20, ISSN 2325-6095, doi:10.1109/MCC.2016.109.

[36] Ma, S.; Fan, C.; Chuang, Y.; et al. Using Service Dependency Graph to Analyze and Test Microservices. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, July 2018, ISSN 0730-3157, pp. 81–86, doi:10.1109/COMPSAC.2018.10207.

[37] BA, K.; Charters, S. Guidelines for performing Systematic Literature Reviews in Software Engineering. volume 2, 01 2007.

[38] Richardson, C. *Microservices architecture*. [ONLINE], 2018, [accessed: 10. 4. 2018]. Available from: https://microservices.io/articles/whoisusingmicroservices.html

[39] Shoup, R. *An Approach to Achieve Scalability and Availability of Data Stores*. [ONLINE], 2008, [accessed: 11. 11. 2017]. Available from: https://www.infoq.com/articles/ebay-scalability-best-practices

[40] Narula, M. *An Approach to Achieve Scalability and Availability of Data Stores*. [ONLINE], 2017, [accessed: 11. 11. 2017]. Available from: https://www.ebayinc.com/stories/blogs/tech/an-approach-to-achieve-scalability-and-availability-of-data-stores/

[41] Barkas, N. *Spotify: Horizontal scalability for great success.* [ONLINE], 2011, [accessed: 11. 11. 2017]. Available from: `https://www.youtube.com/watch?v=BBAfIYpDMX4`

[42] Gonzalo, P. *Microservices Architecture at Spotify.* [ONLINE], 2015, [accessed: 25. 6. 2018]. Available from: `https://medium.com/codebase/microservices-architecture-at-spotify-beac905e9622`

[43] Novak, A. *Going to Market Faster: Most Companies Are Deploying Code Weekly, Daily, or Hourly.* [ONLINE], 2016, [accessed: 9. 12. 2018]. Available from: `https://blog.newrelic.com/technology/data-culture-survey-results-faster-deployment/`

[44] Rotem-Gal-Oz, A. *Fallacies of Distributed Computing Explained.* [ONLINE], 2007, [accessed: 25. 11. 2018]. Available from: `http://www.rgoarchitects.com/Files/fallacies.pdf`

[45] Fowler, M. *Monolith First.* [ONLINE], 2015, [accessed: 7. 6. 2018]. Available from: `https://martinfowler.com/bliki/MonolithFirst.html`

[46] Evans, E. J. *Domain-Driven Design: Tacking Complexity In the Heart of Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003, ISBN 0321125215.

[47] Bogner, J.; Zimmermann, A. Towards Integrating Microservices with Adaptable Enterprise Architecture. In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, Sep. 2016, ISSN 2325-6605, pp. 1–6, doi:10.1109/EDOCW.2016.7584392.

[48] Zimmermann, A.; Sandkuhl, K.; Pretz, M.; et al. *Towards an integrated service-oriented reference enterprise architecture.* O'Reilly Media, Inc., 08 2013, ISBN 1491956259 9781491956250, 26-30 pp.

[49] Hassan, S.; Bahsoon, R. Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap. In *2016 IEEE International Conference on Services Computing (SCC)*, June 2016, pp. 813–818, doi:10.1109/SCC.2016.113.

[50] Wilde, N.; Gonen, B.; El-Sheikh, E.; et al. Approaches to the Evolution of SOA Systems. In *Emerging Trends in the Evolution of Service-Oriented and Enterprise Architectures*, edited by E. El-Sheikh; A. Zimmermann; L. C. Jain, Cham: Springer International Publishing, 2016, ISBN 978-3-319-40564-3, pp. 5–21, doi:10.1007/978-3-319-40564-3_2. Available from: `https://doi.org/10.1007/978-3-319-40564-3_2`

[51] Toffetti, G.; Brunner, S.; Blöchlinger, M.; et al. Self-managing cloud-native applications: Design, implementation, and experience. *Future*

Generation Computer Systems, volume 72, 2017: pp. 165 – 179, ISSN 0167-739X, doi:https://doi.org/10.1016/j.future.2016.09.002. Available from: http://www.sciencedirect.com/science/article/pii/S0167739X16302977

[52] Amundsen, M. *Three Pillars of Microservice Culture*. [ONLINE], 2016, [accessed: 25. 6. 2018]. Available from: https://www.oreilly.com/ideas/three-pillars-of-microservice-culture

[53] Ashkenas, R.; Spiegel, M. *Your Innovation Team Shouldn't Run Like a Well-Oiled Machine*. [ONLINE], 2015, [accessed: 20. 12. 2018]. Available from: https://hbr.org/2015/10/your-innovation-team-shouldnt-run-like-a-well-oiled-machine

[54] Wikipedia. *Fallacies of Distributed Computing*. [ONLINE], 2018, [accessed: 25. 11. 2018]. Available from: https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

[55] Gilbert, S.; Lynch, N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, volume 33, no. 2, June 2002: pp. 51–59, ISSN 0163-5700, doi:10.1145/564585.564601. Available from: http://doi.acm.org/10.1145/564585.564601

[56] Haerder, T.; Reuter, A. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, volume 15, no. 4, Dec. 1983: pp. 287–317, ISSN 0360-0300, doi:10.1145/289.291. Available from: http://doi.acm.org/10.1145/289.291

[57] Mohan, C.; Lindsay, B.; Obermarck, R. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.*, volume 11, no. 4, Dec. 1986: pp. 378–396, ISSN 0362-5915, doi:10.1145/7239.7266. Available from: http://doi.acm.org/10.1145/7239.7266

[58] Keidar, I.; Dolev, D. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences*, volume 57, no. 3, 1998: pp. 309–324.

[59] Garcia-Molina, H.; Salem, K. Sagas. *SIGMOD Rec.*, volume 16, no. 3, Dec. 1987: pp. 249–259, ISSN 0163-5808, doi:10.1145/38714.38742. Available from: http://doi.acm.org/10.1145/38714.38742

[60] Rücker, B.; Schimak, M. *Know the Flow! Microservices and Event Choreographies*. [ONLINE], 2017, [accessed: 11. 12. 2017]. Available from: https://www.infoq.com/articles/microservice-event-choreographies

[61] Wikipedia. God object. [ONLINE], 2017, [accessed: 11. 12. 2017]. Available from: https://en.wikipedia.org/w/index.php?title=God_object&oldid=812564435

[62] Netflix, I. *Chaos Monkey Documentation*. [ONLINE], 2018, [accessed: 8. 9. 2018]. Available from: https://netflix.github.io/chaosmonkey/

[63] De Bruyn, P.; Mannaert, H.; Verelst, J.; et al. Enabling Normalized Systems in Practice – Exploring a Modeling Approach. *Business & Information Systems Engineering*, volume 60, no. 1, Feb 2018: pp. 55–67, ISSN 1867-0202, doi:10.1007/s12599-017-0510-4. Available from: https://doi.org/10.1007/s12599-017-0510-4

[64] De Bruyn, P.; Mannaert, H.; Verelst, J.; et al. Enabling Normalized Systems in Practice – Exploring a Modeling Approach. *Business & Information Systems Engineering*, volume 60, no. 1, Feb 2018: pp. 55–67, ISSN 1867-0202, doi:10.1007/s12599-017-0510-4. Available from: https://doi.org/10.1007/s12599-017-0510-4

Appendix A

Acronyms

ACID atomicity, consistency, isolation, durability

API application programming interface

BE back-end

CAP consistency, availability, partition tolerance (theorem)

CD continuous delivery

CI continuous integration

CRUD create, read, update, delete

CRUDS create, read, update, delete, search

DAO data access object

DB database

DBMS database management system

DDD domain driven design

DRY do not repeat yourself

DTO data transfer object

EA enterprise architecture

FE front-end

FP functional programming

GUI graphical user interface

IPC inter-process communication

IS information system

IT information technology

JSON JavaScript object notation

JST JSON web token

MS microservice

MSA microservice architecture

MVC Model-View-Controller architecture

MVVM Model-View-ViewModel architecture

NS Normalized Systems theory

OOP object oriented programming

ORM object-relational mapping

POJO plain old Java object

R&D research and development

REST representational state transfer

RPC remote procedure call

SA software architecture

SLR systematic literature review

SOA service-oriented architecture

SOAP simple object access protocol

SW software

VM virtual machine

XML extensible markup language

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `Figures` ..... source files of figures used in the thesis
- `Text` ..... thesis text
  - `DP_Kolarik_Vincenc.pdf` ..... PDF version of the thesis
  - `src` ..... LaTeX source codes of the thesis