FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master´s Thesis in Informatics

Designing a business platform using microservices.

Rajendra Kharbuja

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Designing a business platform using microservices.

Entwerfen einer Business-Plattform mit microservices.

Author:	Rajendra Kharbuja
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Manoj Mahabaleshwar
Submission Date:	

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Rajendra Kharbuja

Acknowledgments

Abstract

Microservice architecture provides various advantages compared to Monolith architecture and thus has gained a lot of attention. However, there are still many aspects of microservices architecture which are not clearly documented and communicated. This research attempts to provide a clear picture regarding various concepts of microservices architecture such as granularity, modeling process and finally create a comprehensive guidelines for implementing microservices.
Various keywords from definitions provided by different authors were taken and cateogorized into various conceptual areas. Theses concepts along with the keywords were researched throroughly to understand the concepts of microservices architecture. Additionally, the three important drivers: quality attributes, constraints and principles, are also focussed for creating guidelines.
A lot of interesting findings are made. Although, the concept of microservices emphasizes on creating small services, the notion of appropriate granularity is more important and depends upon four basic concepts which are : single responsibility, autonomy, infrastructure capability and business value. Additionally, the quality attributes such as coupling, cohesion etc should also be considered for identification of microservices. A list of basic metrices are created, which can make it easy to qualify microservices.
In order to identify microservices, either domain driven design or use case refactoring can be used. Both these approach can be effective but the concept of bounded context in domain driven design identifies autonomous services with single responsibility. Apart from literature, a detail study of the architectural approach used in industry called SAP Hybris is done. A number of interviews conducted with their key personnels give important insight into the process of modeling as well as operating microservices. Again, challenges for implementing microservices as well as the approach to tackle them are done based on the literature and interviews done at SAP Hybris.
Finally, all the findings were used to create a detail guidelines for implementing microservices. This is one of the major outcome of the research which dictates how to approach modeling microservices architecture as well as its operational complexities.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
- 1.1 Monolith Architecture Style . . . 1
  - 1.1.1 Types of Monolith Architecture Style . . . 1
  - 1.1.2 Advantages of Monolith Architecture Style . . . 2
  - 1.1.3 Disadvantages of Monolith Architecture Style . . . 3
- 1.2 Microservice Architecture Style . . . 5
  - 1.2.1 Decomposition of an Application . . . 5
  - 1.2.2 Definitions . . . 7
- 1.3 Motivation . . . 8
- 1.4 Research Approach . . . 9
  - 1.4.1 Data Collection Phase . . . 9
  - 1.4.2 Data Synthesis Phase . . . 10
- 1.5 Research Strategy . . . 11
- 1.6 Problem Statement . . . 13

2 Granularity **14**
- 2.1 Introduction . . . 14
- 2.2 Basic Principles to Service Granularity . . . 15
- 2.3 Dimensions of Granularity . . . 16
  - 2.3.1 Dimension by Interface . . . 16
  - 2.3.2 Dimension by Interface Realization . . . 17
  - 2.3.3 R^3 Dimension . . . 19
  - 2.3.4 Retrospective . . . 20
- 2.4 Problem Statement . . . 22

Acronyms **23**

List of Figures **25**

List of Tables **26**

Bibliography **27**

1 Introduction

Architecture is the set of principles assisting software architect and developers for system or application design. [DMT09] It defines process to decompose a system into modules, components and specifies their interactions. [Bro15] In this chapter, two different architectural approaches will be taken. Firstly, a conceptual understanding of monolithic architecture style will be presented, which will then be followed by its various advantages and disadvantages. Then, an overview of microservices architecture will be given. In Section 1.3, the motivation for the current research is explained which is then followed by list of various reseach questions 1.3 being focused. Finally, in Section 1.4 and Section 1.5, the approach which will be used to conduct current research is presented. The purpose of this chapter is to provide a basic background context for the following chapters.

1.1 Monolith Architecture Style

A Mononlith Architecture Style is one in which an application is deployed as a single artifact. The architecture inside the application can be modular and clean. In order to clarify, the figure 1.1 shows architecture of an Online-Store application. The application has clear separation of components such as Catalog, Order and Service as well as respective models such as Product, Order etc. Despite of that, all the units of the application are deployed in tomcat as a single war file.[Ric14a][Ric14c]

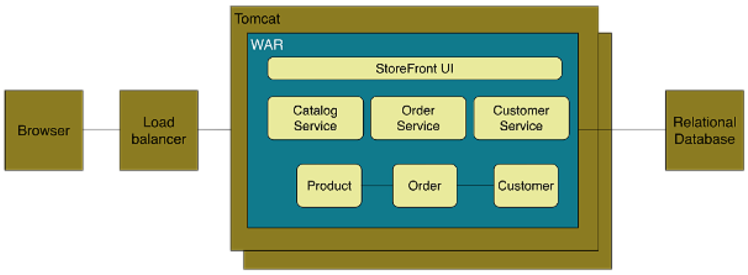

Figure 1.1: Monolith Example from [Ric14a]

1.1.1 Types of Monolith Architecture Style

According to [Ann14], a monolith can be of several types depending upon the viewpoint, as shown below:

1. Module Monolith: If all the code to realize an application share the same codebase and need to be compiled together to create a single artifact for the whole application then the architecture is Module Monolith Architecture. An example is show in Figure 1.2. The application on the left of the figure, has all the code in the same codebase in the form of packages and classes without clear definition of modules and get compiled to a single artifact. However, the application on the right is developed as a number of modular codebase, each has separate codebase and can be compiled to different artifact. The modules uses the produced artifacts which is different than the earlier case where the code referenced each other directly.

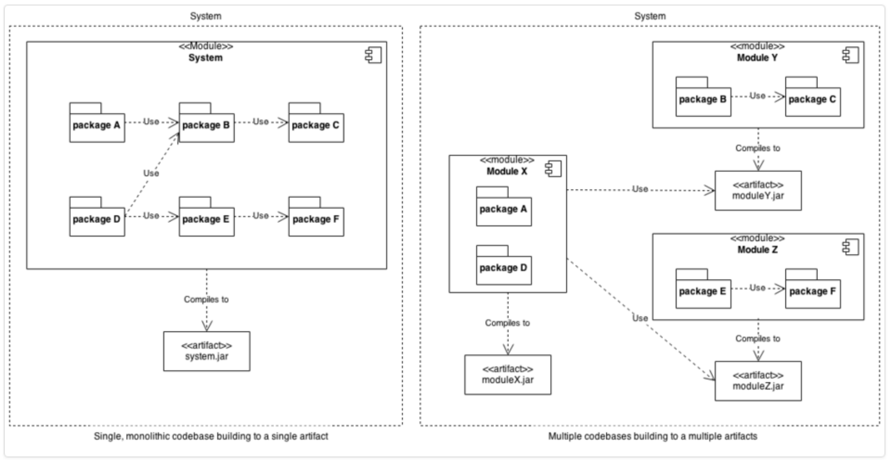

Figure 1.2: Module Monolith Example from [Ann14]

2. Allocation Monolith: An Allocation Monolith is created when all code is deployed to all the servers as a single version. This means that all the components running on the servers have the same versions at any time. The Figure 1.3 gives an example of allocation monolith. The system on the left have same version of artifact for all the components on all the servers. It does not make any differenct whether or not the system has single codebase and artifact. However, the system on the right as shown in the figure is realized with multiple version of the artifacts in different servers at any time.

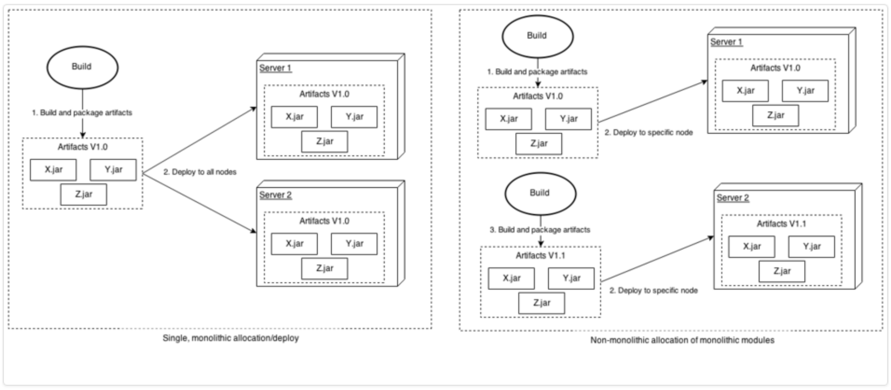
Build
1. Build and package artifacts
Artifacts V1.0
X.jar
Y.jar
Z.jar
2. Deploy to all nodes
Server 1
Server 2
Single, monolithic allocation/deploy
2. Deploy to specific node
3. Build and package artifacts
Artifacts V1.1
Non-monolithic allocation of monolithic modules

Figure 1.3: Allocation Monolith Example from [Ann14]

3. Runtime Monolith: In Runtime Monolith, the whole application is run under a single process. The left system in the Figure 1.4 shows an example of runtime monolith where a single server process is responsible for whole application. Whereas the system on the right has allocated multiple server process to run distinct set of component artifacts of the application.

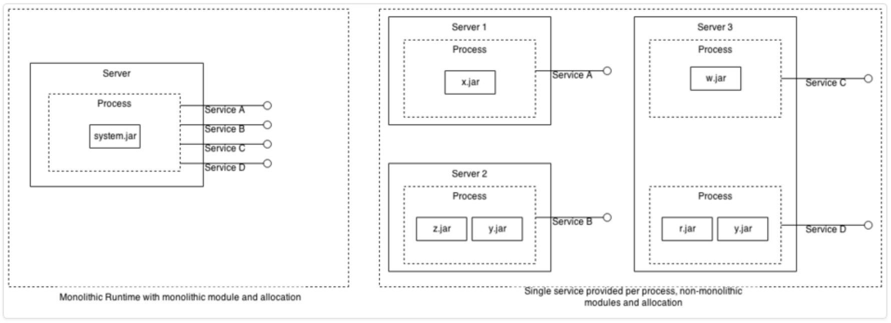
Server
Process
system.jar
Service A
Service B
Service C
Service D
Monolithic Runtime with monolithic module and allocation
Server 1
x.jar
Server 2
z.jar
y.jar
Server 3
w.jar
r.jar
Single service provided per process, non-monolithic modules and allocation

Figure 1.4: Runtime Monolith Example from [Ann14]

1.1.2 Advantages of Monolith Architecture Style

The Monolith architecture is appropriate for small application and has following benifits:[Ric14c][FL14][Gup15][Abr14]

- It is easy to develop a monolith application since various development tools including IDEs are created around the single application concept. Furthermore, it is also easy to test the application by creating appropriate environment on the developer's machine.

- The deployment can be simply achieved by moving the single artifact for the application to an appropriate directory in the server.
- The scaling can be clearly and easily done by replicating the application horizontally across multiple servers behind a load balancer as shown in Figure 1.1
- The different teams are working on the same codebase so sharing the functionality can be easier.

1.1.3 Disadvantages of Monolith Architecture Style

As the requirement grows with time, alongside application becomes huge and the size of team increases, then the monolith architecture faces many problems. The challenges of monolith architecture are as given below:[NS14][New15][Abr14][Ric14a][Ric14c][Gup15]

- Limited Agility: As the whole application has single codebase, even changing a small feature to release it in production takes time. Firstly, the small change can also trigger changes to other dependent code. In huge monolith application it is very difficult to manage modularity especially when all the team members are working on the same codebase. Secondly, to deploy a small change in production, the whole application has to be deployed. Thus continuous delivery gets slower. This will be more problematic when multiple changes have to be released on a daily basis. The slow pace and low frequency of release will highly affect agility.

- Decrease in Productivity: It is difficult to understand the application especially for a new developer because of the size of codebase. Although it also depends upon the structure of the codebase, it will still be difficult to grasp the significance of the code when there is no hard modular boundary. Additionally, a developer can be intimidated due to need to see the whole application at once from outwards to inwards direction. Secondly, the development environment can be slow to load the whole application and at the same time the deployment will also be slow. In overall, it will slow down the speed of understandability, execution and testing.

- Difficult Team Structure: The division of team as well as assigning tasks to the team can be tricky. Most common ways to partition teams in monolith are by technology and by geography. However, each one cannot be used in all the situations. In any case, the communication among the teams can be difficult and slow. Additionally, it is not easy to assign complete vertical ownership to a team for a particular feature from development to release. If something goes wrong in the deployment, there is always a confusion who should find the problem, either operations team or the last person to commit etc. The approprate team structure and ownership are very important for agility.

- Longterm Commitment to Technology stack: The technology to use is chosen before the development phase by analysing the requirements and the maturity of current technology at that time. All the teams in the architecture need to follow the same techonology stack throughout the lifecycle of application. However, if the requirement changes then there can be situation when the features can be best solved by different sets of technology. Additionally, not all the features in the application are same so cannot be treated accordingly and cannot be solved by same technology as well. Nevertheless, the technology advances rapidly. So, the solution thought right at the time of planning can be outdated and there can be a better solution available at present. In monolith application, it is very difficult to migrate to new technology stack and it can rather be a painfull process.

- Limited Scalability: The scalability of monolith application can be done in either of two ways. The first way is to replicate the application along many servers and dividing the incoming request using a load balancer in front of the servers. Another approach is using identical copies of the application in multiple servers as in previous case but partitioning the database access instead of user request. Both of these scaling approaches improves the capacity and availability of the application. However, the individual requirement regarding scaling for each component can be different but cannot be fulfilled with this approach. Also, the complexity of the monolith application remains the same because we are replicating the whole application. Additionally, if there is a problem in a component the same problem can affect all the servers running the copies of the application, this does not improve resilency.[Mac14][NS14]

1.2 Microservice Architecture Style

With monolith, it is easy to start development. But as the system gets bigger and complicated along time, it becomes very difficult to be agile and productive. The disadvantages listed in Section 1.1.3 outweighs its advantages as the system gets old. The various qualities such as scalability, agility need to be maintained for the whole lifetime of the application and it becomes complicated by the fact that the system needs to be updated side by side because the requirements always keeps coming. In order to tackle these disadvantages, microservices architecture style is followed.
Microservices architecture uses the approach of decomposing an application into smaller autonomous components. The following Section 1.2.1 provides details of various decomposition techniques.

1.2.1 Decomposition of an Application

There are various ways to decompose an application. This section discuss two different ways of breaking down an application.

1.2.1.1 Scale Cube

The Section 1.1.3 specified various disadvantages related to monolith architecture style. The book [FA15] provides a way to solve most of the discussed problems related to agility, scalability, productivity etc. It provides three dimensions of scalability as shown in figure 1.5 which can be applied alone or simultaneously depending upon the situation and desired goals.

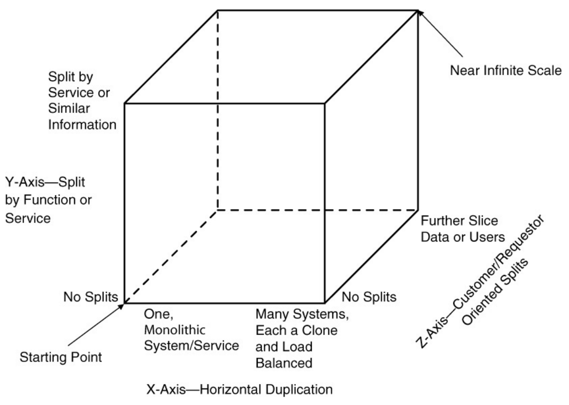
Split by Service or Similar Information
Near Infinite Scale
Y-Axis—Split by Function or Service
Further Slice Data or Users
Z-Axis—Customer/Requestor Oriented Splits
No Splits
No Splits
One, Monolithic System/Service
Many Systems, Each a Clone and Load Balanced
Starting Point
X-Axis—Horizontal Duplication

Figure 1.5: Scale Cube from [FA15]

The scaling along each dimensions are described below. [FA15][Mac14][Ric14a]

1. X-axis Scaling: It is done by cloning the application and data along multiple servers. A pool of requests are applied into a load balancer and the requests are deligated to any of the servers. Each of the server has the full capability of the application and full access to all the data required so in this respect it does not make any difference which server fulfills the request. Rather, it is about how many requests are fulfilled at any time. It is easy to scale along X-axis as the number of requests increases. The solution is as simple as to add additional clones. However, with this type of scaling, it does not scale with the increase in data. Moreover, it also does not scale when there are large variation in the frequency of any type of requests or there are dominant requests types because all the requests are handled in an unbiased way and allocated to servers in the same way.

2. Z-axis Scaling: The scaling is done by spliting the request based on certain criteria or information regarding the requestor or customer affected by the request. It is different than X-axis scaling in the way that the servers are responsible for different kinds of requests. Normally, the servers have same copy of the application but some can have additional functionalies depending upon the requests expected. The Z-axis scaling helps in fault isolation and transaction scalability. Using this scaling, certain group of customers can be given added functionality or a new functionality can be tested to a small group and thus minimizing the risk.

3. Y-axis Scaling: The scaling along this dimension means the spliting of the application responsibility. The separation can be done either by data, by the actions performed on the data or by combination of both. The respective ways can be referred to as resoure-oriented or service-oriented splits. While the x-axis or z-axis split were rather duplication of work along servers, y-axis is more about specialization of work along servers. The major advantage of this scaling is that each request is scaled and handled differently according to its necessity. As the logic along with the data to be worked on are separated, developers can focus and work on small section at a time. This will increase productivity as well as agility. Additionally, a fault on a component is isolated and can be handled gracefully without affecting rest of the application. However, scaling along Y-axis can be costly compared to scaling along other dimensions.

1.2.1.2 Shared Libraries

Libraries is a standard way of sharing functionalities among various services and teams. The capability is provided by the feature of programming language. However there are various downsides to this approach. Firstly, it does not provide technology heterogeneity. Next, unless the library is dynamically linked, independent scaling, deployment and maintainance cannot be achieved. So, in other cases, any small change in the library leads to redeployment of whole system. Sharing code is a form of coupling which should be avoided.

Decomposing an application in terms of composition of individual features where each feature can be scaled and deployed independently, gives various advantages along agility, performance and team organization as well. Thus, microservices uses the decomposition technique given by scale cube. A detail advantages of microservices are discussed further in Section **??**.

1.2.2 Definitions

There are several definitions given by several pioneers and early adapters of the style.

Definition 1: [Ric14b]

"It is the way to functionally decompose an application into a set of collaborating services, each with a set of narrow, related functions, developed and deployed independently, with its own database."

Definition 2: [Woo14]

"It is a style of software architecture that involves delivering systems as a set of very small, granular, independent collaborating services."

Definition 3: [Coc15]

"Microservice is a loosely coupled Service-Oriented Architecture with bounded contexts."

Definition 4: [FL14][RTS15]

"Microservices are Service-Oriented Architecture done right."

Definition 5: [FL14]

"Microservice architecture style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."
As other architectural styles, microservices presents its approach by increasing cohesion and decreasing coupling. Besides that, it breaks down system along business domains following single responsibility principle into granular and autonomous services running on separate processes. Additionally, the architecture focus on the collaboration of these services using light weight mechanisms.

1.3 Motivation

The various definitions presented in section highlights different key terms such as:

1. collaborating services
2. developed and deployed independently
3. build around business capabilities
4. small, granular services

These concepts are very important to be understood in order to approach microservices correctly and effectively. The first two terms relate to runtime operational qualities of microservices whereas the next two address modeling qualities. With that consideration, it indicates that the definitions given are complete which focus on both aspects of any software applications.
However, if attempt is made to have clear indepth understanding of each of the key terms then various questions can be raised without appropriate answers. The various

questions (given in column 'Questions') related to modeling and operations (given in column 'Type') are listed in the Table 1.1.

#	Questions	Type
1	How small should be size of microservices?	Modeling
2	How does the collaboration among services happen?	Operation
3	How to deploy and maintain independently when there are dependencies among services?	Operation
4	How to map microservices from business capabilities?	Modeling
5	What are the challenges need to be tackled and how to?	Operation

Table 1.1: Various Questions related to Microservices

Without clear answer to these questions, it is difficult to say that the definitions presented in Section 1.2.2 are complete and enough to follow the microservices architecture. The process of creating microservices is not clearly documented. There is no enough research papers defining a thorough process of modeling and operating microservices. Additionally, a lot of industries such as amazon, netflix etc are following this architecture but it is not clear about the process they are using to define and implement microservices. The purpose of the research is to have a clear understanding about the process of designing microservices by focusing on following questions.

Research Questions

1. How are boundary and size of microservices defined?
2. How business capabilities are mapped to define microservices?
3. What are the best practices to tackle challenges introduced by microservices?
    a) How does the collaboration among services happen?
    b) How to deploy and maintain independently when there are dependencies among services?
    c) How to monitor microservices?

1.4 Research Approach

A research is an iterative procedure with a goal of collecting as many relevant documents as possible so that the research questions could be answered rationally. So, it becomes very important to follow a consistent research procedure. For the current research, the approach is chosen by refering to [np07]. It consists of two major phases.

1. Data Collection Phase
2. Data Synthesis Phase

The following sections explains each phase in detail.

1.4.1 Data Collection Phase

In this phase, papers related to the reseach questions are collected. The Figure 1.6 shows basic steps in this phase.

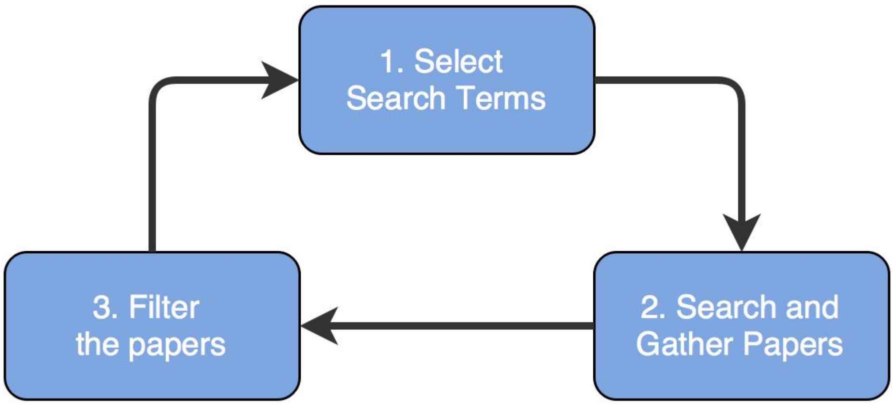

Figure 1.6: Data Collection Phase

1. **Select Search Terms**
   At first, various search terms which defines the research topics and questions well, are selected using following strategies.
   a) keywords from research questions and various definitions
   b) synonyms of keywords
   c) accepted and popular terms from academics and industries
   d) references discovered from selected papers

   A consise list of initial keywords which are selected from various definitions of microservices are given in Table 1.2.

2. **Search and Gather Papers**
   The search terms selected in previous step are utilized to discover various papers. In order to achieve that, the search terms are used against various resources listed below.

   a) google scholar
   b) IEEExplore
   c) ACM Digital Library
   d) Researchgate
   e) Books
   f) Technical Articles

3. **Filter the Papers** Finally, the various papers collected from various resources are filtered first to check if the papers have profound base to back up their result. Using only authentic papers is important to provide rational base to current research. The various criteria used to filter the papers are listed below.
   a) Is the paper relevant to answer the research question?
   b) Does the paper have good base in terms of sources and provide references of the past studies?
   c) Are there any case studies or examples provided to verify result of the reseach?

1.4.2 Data Synthesis Phase

The next phase is to gather data from the selected papers to create meaningful output and provide direction to the research. The Figure 1.7 shows various steps within this phase.

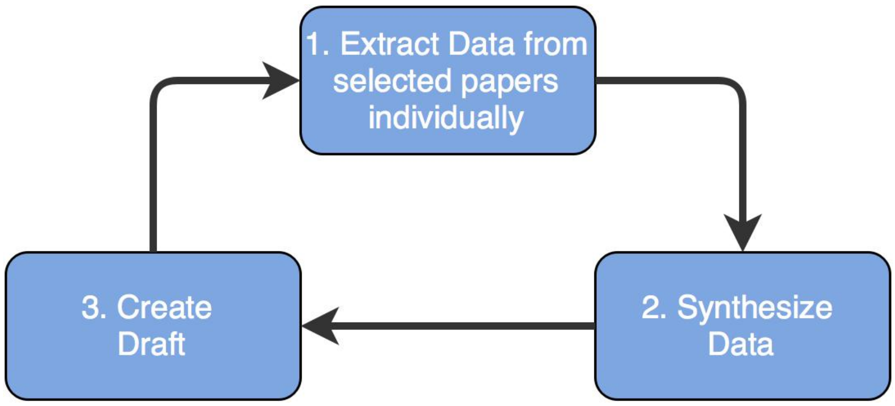

Figure 1.7: Data Synthesis Phase

1. **Extract Data from selected papers individually**
   Firstly, each papers are scanned and important data relevant to research are collected.

2. **Synthesize Data**
   The data collected from each paper are revised and compared for their similarities in concepts as well as differences in opinion. These individual contents from all papers are then synthesized.

3. **Create Draft**
   Finally, draft for the observations are created for future reference which will be used later for creating final report.

1.5 Research Strategy

The Section 1.4 emphasize the importance of having consistent research procedure. Additionally, it is important to define a good strategy for achieving goals of the research. A good strategy is to narrow down the areas for conducting research. In this section, a list of areas will be identified.
The various definitions in Section 1.2.2 show that the authors have their own interpretation of microservices but at the same time agree upon some basic concepts. Nevertheless, each definition can be used to understand different aspect of the microservices. A distinct set of keywords can be identified from these definitions which represents different aspects. The Table 1.2 shows various keywords to focus. Additionally, there are other columns in the table which represents various aspects of microservices architecture. These columns are checked or unchecked to represent the relevancy of each keyword.

#	keywords	size	Quality of good microservice	communication	process to model microservices
1	Collaborating Services			✓	
2	Communicating with lightweight mechanism like http			✓	
3	Loosely coupled, related functions		✓	✓	
4	Developed and deployed independently				✓
5	Own database		✓		✓
6	Different database technologies				✓
7	Service Oriented Architecture		✓		✓
8	Bounded Context	✓	✓		✓
9	Build around Business Capabilities	✓	✓		✓
10	Different Programming Languages				✓

Table 1.2: Keywords extracted from various definitions of Microservice

Finally, answering various questions around these keywords can be the **first approach** to understand the topics shown in columns such as size, quality of microservices etc.

The **next approach** to look around the architecture process is to understand its various drivers. According to [Bro15], the important drivers which clarify the architecture are:

1. **Quality Attributes**
   The non-functional requirements have high impact on the resulting architecture. It is important to consider various quality attributes to define the process of architecture.

2. **Constraints**
   There are always limitations or disadvantages faced by any architecture however the better knowledge is useful in the process of explaining the architecture with satisfaction.

3. **Principles**
   The various principles provides a consistent approaches to tackle various limitations. They are the keys to define guidelines.

So, in order to define the process of modeling microservices, the key aspects related to **quality attributes**, **constraints** and **principles** will be studied thoroughly.

Considering both the Table 1.2 and List 1.5, the approach to be used to define the guidelines will be to first look deep into various quality attributes influencing microservices architecture. Then, the process of identifying or modeling individual microservices from problem domain will be defined. At first, the research is performed based on various research papers. Then, the approach used by industry adopting microservices architecture will be studied in order to answer the research questions. The study will be performed on SAP Hybris. Finally, the various constraints or challenges which affects microservices will be identified. The results of previous research will be mapped into a form of various principles and these principles will ultimately provide sufficient ground to create guidelines for microservices architecture.

1.6 Problem Statement

Considering the case that the size of microservice is an important concept and is discussed a lot, but no concrete answer regarding this topic exists. Moreover, the idea of size has a lot of interpretations and no definite answer regarding how small a microservice should be. So, the first step should be to understand the concept of granularity in the context of microservices. Additionally, it would be great if some answers regarding the appropriate size of microservices could be given.

2 Granularity

In this chapter, a detailed concept regarding size of microservices will be discussed. A major focus would be to research various qualitative aspects of granularity of microservices. Section 2.2 lists various principles which can guide to model correct size considering various important aspects. Later, in Section 2.3, various interpretations defining dimensions of microservices are presented. Finally, the secion provides a complete picture of various factors which determine granularity.

2.1 Introduction

Granularity of a service is often ambiguous and has different interpretation. In simple terms, it refers to the size of the service. However, the size itself can be vague. It can neither be defined as a single quantitative value nor it can be defined in terms of single dependent criterion. It is difficult to define granularity in terms of number because the concepts defining granularity are subjective in nature. If we choose an activity supported by the service to determine its granularity then we cannot have one fixed value instead a hierarchical list of answers; where an activity can either refer to a simple state change, any action performed by an actor or a complete business process. [Linnp], [Hae+np]

Although, the interest upon the granularity of a component or service for the business users only depends upon their business value, there is no doubt that the granularity affects the architecture of a system. The honest granularity of a service should reflect upon both business perspective and should also consider the impact upon the overall architecture.

If we consider other units of software application, we come from object oriented to component based and then to service oriented development. Such a transistion has been considered with the increase in size of the individual unit. The increase in size is contributed by the interpretation or the choice of the abstraction used. For example, in case of object oriented paradigm, the abstraction is chosen to represent close impression of real world objects, each unit representing fine grained abstraction with some attributes and functionalities.

Such abstraction is a good approach towards development simplicity and understanding, however it is not sufficient when high order business goals have to be implemented.

It indicates the necessity of coarser-grained units than units of object oriented paradigm. Moreover, component based development introduced the concept of business components which target the business problems and are coarser grained. The services provide access to application where each application is composed of various component services. [Linnp]

2.2 Basic Principles to Service Granularity

In addition to quantitative perspective on granularity, it can be helpful to approach the granularity qualitatively. The points below are some basic principles derived from various scientific and research papers, which attempt to define the qualitative properties of granularity. Hopefully, these principles will be helpful to come with the service of right granularity.

1. The correct granularity of a component or a service is dependent upon the time. The various supporting technologies that evolve during time can also be an important factor to define the level of decomposition. For eg: with the improvement in virtualization, containerization as well as platform as a service technologies, it is fast and easy for deployment automation which supports multiple fine-grained services creation.[HS00]

2. A good candidate for a service should be independent of the implementation but should depend upon the understandability of domain experts.[Hae+np; HS00]

3. A service should be an autonomous reusable component and should support various cohesion such as functional (group similar functions), temporal(change in the service should not affect other services), run-time(allocate similar runtime environment for similar jobs; eg. provide same address space for jobs of similar computing intensity) and actor (a component should provide service to similar users). [Hae+np], [HS00]

4. A service should not support huge number of operations. If it happens, it will affect high number of customers on any change and there will be no unified view on the functionality. Furthermore, if the interface of the service is small, it will be easy to maintain and understand.[Hae+np], [RS07]

5. A service should provide transaction integrity and compensation. The activities supported by a service should be within the scope of one transaction. Additionally, the compensation should be provided when the transaction fails. If each operation provided by the service map to one transaction, then it will improve availability and fault-recovery. [Hae+np], [Foo05] [BKM07]

6. The notion of right granularity is more important than that of fine or coarse. It depends upon the usage condition and moreover is about balancing various qualities such as reusability, network usage, completeness of context etc. [Hae+np], [WV04]

7. The level of abstraction of the services should reflect the real world business activities. Doing so will help to map business requirements and technical capabilities. [RS07]

8. If there are ordering dependencies between operations, it will be easy to implement and test if the dependent operations are all combined into a single service. [BKM07]

9. There can be two better approaches for breaking down an abstraction. One way is to separate redundant data or data with different semanting meaning. The other approach is to divide services with limited control and scope. For example: A Customer Enrollment service which deals with registration of new customers and assignment of unique customer ID can be divided into two independent fine-grained services: Customer Registration and ID Generation, each service will have limited scope and separate context of Customer.[RS07]

10. If there are functionalities provided by a service which are more likely to change than other functionalities. It is better to separate the functionality into a fine-grained service so that any further change on the functionality will affect only limited number of consumers. [BKM07]

2.3 Dimensions of Granularity

As already mentioned in Section 2.1, it is not easy to define granularity of a service quantitatively. However, it can be made easier to visualize granularity if we can project it along various dimensions, where each dimension is a qualifying attribute responsible for illustrating size of a service. Eventhough the dimensions discussed in this section will not give the precise quantity to identify granularity, it will definitely give the hint to locate the service in granularity space. It will be possible to compare the granularity of two distinct services. Moreover, it will be interesting and beneficial to know how these dimensions relate to each other to define the size.

2.3.1 Dimension by Interface

One way to define granularity is by the perception of the service interface as made by its consumer. The various properties of a service interface responsible to define its size are listed below.

1. Functionality: It qualifies the amount of functionality offered by the service. The functionality can be either default functionality, which means some basic group of logic or operation provided in every case. Or the functionality can be parameterized and depending upon some values, it can be optionally provided. Depending upon the functionality volume, the service can be either fine-grained or course-grained than other service. Considering functionality criterion, a service offering basic CRUD functionality is fine-grained than a service which is offers some accumulated data using orchestration. [Hae+np]

2. Data: It refers to the amount of data handled or exchanged by the service. The data granularity can be of two types. The first one is input data granularity, which is the amount of data consumed or needed by the service in order to accomplish its tasks. And the other one is ouput data granularity, which is the amount of data returned by the service to its consumer. Depending upon the size and quantity of business object/objects consumed or returned by the service interface, it can be coarse or fine grained. Additionally, if the business object consumed is composed of other objects rather than primitive types, then it is coarser-grained. For example: the endpoint "PUT customers/C1234" is coarse-grained than the endpoint "PUT customers/C1234/Addresses/default" because of the size of data object expected by the service interface. [Hae+np]

3. Business Value: According to [RKKnp], each service is associated with an intention or business goal and follows some strategy to achieve that goal. The extent or magnitude of the intention can be perceived as a metric to define granularity. A service can be either atomic or aggregate of other services which depends upon the level of composition directly influenced by the extent of target business goal. An atomic service will have lower granularity than an aggregate in terms of business value. For example, sellProduct is coase-grained than acceptPayment, which is again coarse-grained than validateAccountNumber. [Hae+np]

2.3.2 Dimension by Interface Realization

The Section 2.3.1 provides the aspect of granularity with respect to the perception of customer to interface. However, there can be different opinion regarding the same properties, when it is viewed regarding the imlementation. This section takes the same aspects of granularity and try to analyze them when the services are implemented.

1. Functionality: In Section 2.3.1, it was mentioned that an orchestration service has higher granularity than its constituent services with regard to default functionality. If the realization effort is focused, it may only include compositional and/or compensation logic because the individual tasks are acomplshed by the constituent service interfaces. Thus, in terms of the effort in realizing the service it is fine-grained than from the view point of interface by consumer. [Hae+np]

2. Data: In some cases, services may utilize standard message format. For example: financial services may use SWIFT for exchanging finanicial messages. These messages are extensible and are coarse grained in itself. However, all the data accepted by the service along with the message may not be required and used in order to fulfil the business goal of the service. In that sense, the service is coarse-grained from the viewpoint of consumer but is fine-grained in realization point of view. [Hae+np]

3. Business Value: It can make a huge impact when analyzing business value of service if the realization of the service is not considered well. For example: if we consider data management interface which supports storage, retrieval and transaction of data, it can be considered as fine-grained because it will not directly impact the business goals. However, since other services are very dependent in its performance, it becomes necessary to analyse the complexity associated with the implementation, reliablity and change the infrastructure if needed. These comes with additional costs, which should well be analyzed. From the realization point of view, the data service is coarse-grained than its view by consumer. [Hae+np]

Principle: A high Functionality granularity does not necessarily mean high business value granularity
It may seem to have direct proportional relationship between functionality and business value granularity. The business value of a service reflects business goals however the functionality refers to the amount of work done by the service. A service to show customer history can be considered to have high functionality granularity because of the involved time period and database query. However, it is of very low business value to the enterprise. [Hae+np]

2.3.3 R^3 Dimension

Keen [Kee91] and later Weill and Broadbent [WB98] introduced a separate group of criteria to measure granularity of service. The granularity was evaluated in terms of two dimensions as shown in the Figure 2.1

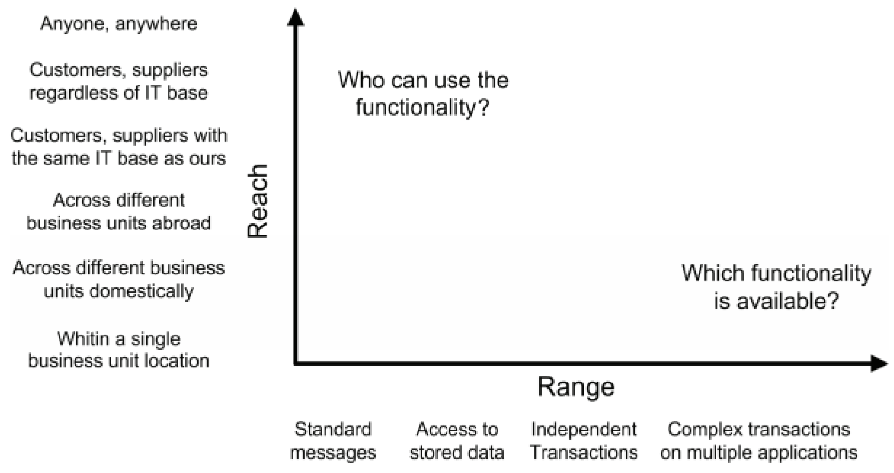
Anyone, anywhere
Customers, suppliers regardless of IT base
Customers, suppliers with the same IT base as ours
Across different business units abroad
Across different business units domestically
Whitin a single business unit location
Reach
Who can use the functionality?
Which functionality is available?
Range
Standard messages
Access to stored data
Independent Transactions
Complex transactions on multiple applications

Figure 2.1: Reach and Range model from [Kee91; WB98]

1. Reach: It provides various levels to answer the question "Who can use the functionality? ". It provides the extent of consumers, who can access the functionality provide by the service. It can be a customer, the customers within an organization, supplier etc. as given by the Figure 2.1.

2. Range: It gives answer to "Which functionality is available? ". It shows the extent to which the information can be accessed from or shared with the service. The levels of information accessed are analyzed depending upon the kind of business activities accessible from the service. It can be a simple data access, a transaction, message transfer etc as shown in the Figure 2.1 The 'Range' measures the amount of data exchanged in terms of the levels of business activities important for the organization. One example for such levels of activities with varying level of 'Range' is shown in Figure 2.2.

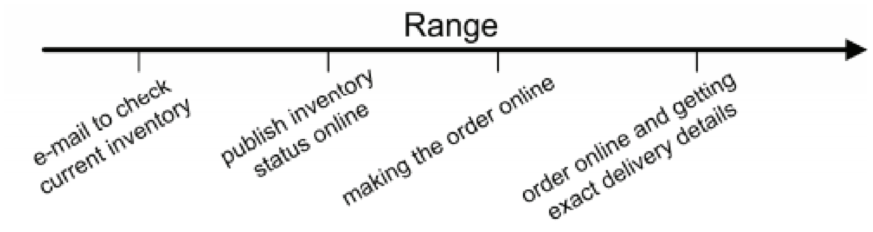
Range
e-mail to check current inventory
publish inventory status online
making the order online
order online and getting exact delivery details

Figure 2.2: Example to show varying Range from [Kee91; WB98]

In the Figure 2.2, the level of granularity increases as the functionality moves from 'accessing e-mail message' to 'publishing status online' and then to 'creating order'. It is due to the change in the amount of data access involved in each kind of functionality. Thus, the 'Range' directly depends upon the range of data access. As the service grows, alongside 'Reach' and 'Range' also peaks up, which means the extent of consumers as well as the kind of functionality increase. This will add complexity in the service. The solution proposed by [Coc01] is to divide the architecture into services. However, only 'Reach' and 'Range' will not be enough to define the service. It will be equally important to determine scope of the individual services. The functionality of the service then should be define in two distinct dimensions 'which kind of functionality' and 'how much functionality'. This leads to another dimension of service as described below. [Kee91; WB98; RS07]

3. Realm: It tries to create a boundary around the scope of the functionality provided by the service and thus clarifies the ambiguity created by 'Range'. If we take the same example as given by Figure 2.2, the range alone defining the kind of functionaly such as creating online order does not explicitly clarifies about what kind of order is under consideration. The order can be customer order or sales order. The specification of 'Realm' defining what kind of order plays role here. So, we can have two different services each with same 'Reach' and 'Range' however different 'Realm' for customer order and Sales order. [Kee91; WB98; RS07]

The consideration of all aspects of a service including 'Reach', 'Range' and 'Realm' give us a model to define granularity of a service and is called R^3 model. The volume in the R^3 space for a service gives its granularity. A coarse-grained service has higher R^3 volume then fine-grained service. The Figure 2.3 and Figure 2.4 show such volume-granularity analogy given by R^3 model. [Kee91; WB98; RS07]

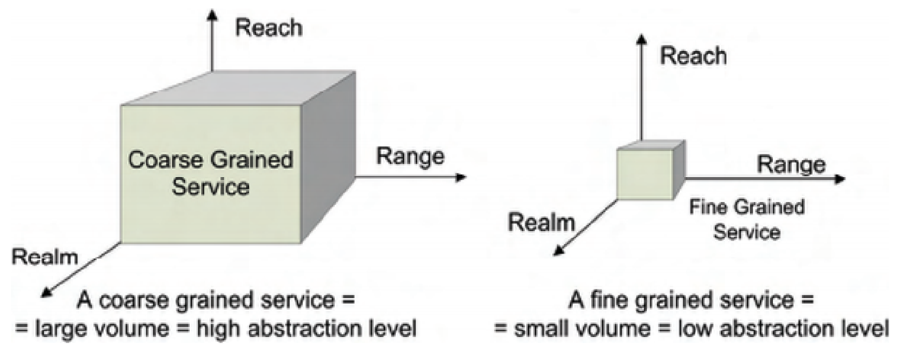
Reach
Range
Realm
Coarse Grained Service
A coarse grained service =
= large volume = high abstraction level
Reach
Range
Realm
Fine Grained Service
A fine grained service =
= small volume = low abstraction level

Figure 2.3: R^3 Volume-Granularity Analogy to show direct dependence of granularity and volume [RS07]

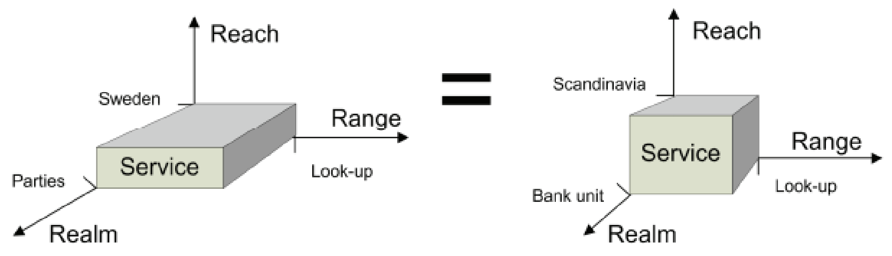

Figure 2.4: R^3 Volume-Granularity Analogy to show same granularity with different dimension along axes [RS07]

2.3.4 Retrospective

The Section 2.3.1 and Section 2.3.2 classified granularity of a service along three different directions: data, functionality and business value. Moreover, the interpretation from the viewpoint of interface by consumer and the viewpoint of producer for realization were also made. Again, the Section 2.3.3 divides the aspect of granularity along Reach, Range and Realm space, the granularity given by the volume in the R^3 space. Despite of good explanation regarding each aspect, the classification along data, functionality and business value do not provide discrete metrics to define granularity in terms of quantitatively. However, the given explanations and criteria are sufficient to compare two different services regarding granularity. These can be effectively used to classify if a service is fine-grained than other service. The R^3 model in Section 2.3.3 additionally

provides various levels of granularity along each axis. This makes it easy to at least measure the granularity and provides flexibility to compare the granularity between various services.
A case study [RS07] using data, functionality and business criteria to evaluate the impact of granularity on the complexity in the architecture is held. The study was done at KBC Bank Insurance Group of central Europe. Along the study, the impact of each kind of granularity is verified. The important results from the study are listed below.

1. A coarse-grained service in terms of 'Input Data' provides better transactional support, little communication overhead and also helps in scalability. However, there is a chance of data getting out-of-date quickly.

2. A coarse-grained service in terms of 'Output Data' also provides good communication efficiency and supports reusability.

3. A fine-grained service along 'Default Functionality' has high reusability and stability but low reuse efficiency.

4. A coarse-grained service due to more optional functionalities has high reuse efficiency, high reusability and also high stability but the implementation or realization is complicated.

5. A coarse-grained service along 'Business Value' has high consumer satisfaction value and emphasize the architecturally crucial points fulfilling business goals.

Similarly, a study was carried out in Handelsbanken in sweden, which has been working with Service-Oriented architecture. The purpose of the study was to analyze the impact of R^3 model on the architecture. It is observed that there are different categories of functionalities essential for an organization. Some functionality can have high Reach and Range with single functionality and other may have complex functionality with low Reach and Range. The categorization and realization of such functionalities should be accessed individually. It is not necessary to have same level of granularity across all services or there is no one right volume for all services. However, if there are many services with low volume, then we will need many services to accomplish a high level functionality. Additionally, it will increase interdependency among services and network complexity. Finally, the choice of granularity is completely dependent upon the IT-infrastructure of the company. The IT infrastructure decides which level of granularity it can support and how many of them. [RS07]

Principle: IT-infrastructure of an organization affects granularity.
The right value of granularity for an organization is highly influenced by its IT infrastructure. The organization should be capable of handling the complexities such as communication, runtime, infrastructure etc if they choose low granularity. [RS07]

Additionally, it is observed that a single service can be divided into number of granular services by dividing across either Reach, Range or Realm. The basic idea is to make the volume as low as possible as long as it can be supported by IT-infrastructure. Similarly, each dimension is not completely independent from other. For example: If the reach of a service has to be increased from domestic to global in an organization then the realm of the functionality has to be decreased in order to make the volume as low as possible, keep the development and runtime complexities in check. [RS07]

Principle: Keep the volume low if possible.
If supported by IT-infrastructure, it is recommended to keep the volume of service as low as possible, which can be achieved by managing the values of Reach, Range and Realm. It will help to decrease development and maintenance overload. [RS07]

2.4 Problem Statement

This chapter analysed various qualitative aspects related to granularity of microservices. The interesting thing to notice is various dimensions of size across data, functionality and business value. But it is also necessary to look into it in quantitative approach. Additionally, the various principles listed in Section 2.2 points to other qualities of microservices except granularity. The granularity is not only attribute which is important while modeling microservices but there are other quality attributes which affect granularity and influence on the overall quality of microservices. The next task will be to study various quality attributes affecting microservices and derive their quantitative metrices.

Acronyms

API Application Programming Interface.

AWS Amazon Web Services.

CQRS Command Query Responsibility Segregation.

CRUD create, read, update, delete.

DEP Dependency.

HCP Hana Cloud Platform.

IDE Integrated Development Environment.

IFBS International Financial and Brokerage Services.

ISCI Inter Service Coupling Index.

ODC Operation Data Granularity.

ODG Operation Data Granularity.

OFG Operation Functionality Granularity.

PAAS Platform as a Service.

RCS Relative Coupling of Services.

REST Representational State Transfer.

RIS Relative Importance of Services.

RPC Remote Procedure Call.

SCG Service Capability Granularity.

SDG Service Data Granularity.

SDLC Software Development Life Cycle.

SFCI Service Functional Cohesion Index.

SIDC Service Interface Data Cohesion.

SIUC Service Interface Usage Cohesion.

SIUC Service Sequential Usage Cohesion.

SLC Self Containment.

SMCI Service Message Coupling Index.

SOA Service Oriented Architecture.

SOAD Service Oriented Analysis and Design.

SOAF Service Oriented Architecture Framework.

SOCI Service Operational Coupling Index.

SOG Service Operations Granularity.

SOMA Service Oriented Modeling and Architecture.

SRI Service Reuse Index.

SRP Single Responsibility Principle.

SWIFT Society for Worldwide Interbank Financial Telecommunication.

UML Unified Modeling Language.

YaaS Hybris as a Service.

List of Figures

1.1 Monolith Example from [Ric14a] 1
1.2 Module Monolith Example from [Ann14] 2
1.3 Allocation Monolith Example from [Ann14] 2
1.4 Runtime Monolith Example from [Ann14] 2
1.5 Scale Cube from [FA15] 5
1.6 Data Collection Phase 9
1.7 Data Synthesis Phase 11

2.1 Reach and Range model from [Kee91; WB98] 19
2.2 Example to show varying Range from [Kee91; WB98] 19
2.3 R^3 Volume-Granularity Analogy to show direct dependence of granularity and volume [RS07] 20
2.4 R^3 Volume-Granularity Analogy to show same granularity with different dimension along axes [RS07] 20

List of Tables

1.1 Various Questions related to Microservices 8
1.2 Keywords extracted from various definitions of Microservice 12

Bibliography

[Abr14] S. Abram. *Microservices*. Oct. 2014. URL: http://www.javacodegeeks.com/2014/10/microservices.html.

[Ann14] R. Annett. *What is a Monolith?* Nov. 2014.

[BKM07] P. Bianco, R. Kotermanski, and P. F. Merson. *Evaluating a Service-Oriented Architecture*. Tech. rep. Carnegie Mellon University, 2007.

[Bro15] S. Brown. "Software Architecture for Developers." In: (Dec. 2015).

[Coc01] A. Cockburn. *WRITING EFFECTIVE USE CASES*. Tech. rep. Addison-Wesley, 2001.

[Coc15] A. Cockcroft. *State of the Art in Mircroservices*. Feb. 2015. URL: http://www.slideshare.net/adriancockcroft/microxchg-microservices.

[DMT09] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. *Software Architecture: Foundations, Theory, and Practice*. John Wiley Sons, Jan. 2009.

[FA15] M. T. Fisher and M. L. Abbott. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Second Edition*. Addison-Wesley Professional, 2015.

[FL14] M. Fowler and J. Lewis. *Microservices*. Mar. 2014. URL: http://martinfowler.com/articles/microservices.html.

[Foo05] D. Foody. *Getting web service granularity right*. 2005.

[Gup15] A. Gupta. *Microservices, Monoliths, and NoOps*. Mar. 2015.

[Hae+np] R. Haesen, M. Snoeck, W. Lemahieu, and S. Poelmans. *On the Definition of Service Granularity and Its Architectural Impact*. Tech. rep. Katholieke Universiteit Leuven, np.

[HS00] P. Herzum and O. Sims. *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley Sons, 2000.

[Kee91] P. G. Keen. *Shaping The Future of Business Design Through Information Technology*. Harvard Business School Press, 1991.

[Linnp] D. Linthicum. *Service Oriented Architecture (SOA)*. np.

[Mac14] L. MacVittie. *The Art of Scale: Microservices, The Scale Cube and Load Balancing*. Nov. 2014.

[New15] S. Newman. *Building Microservices*. O'Reilly Media, 2015.

[np07] np. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. Keele University, 2007.

[NS14] D. Namiot and M. Sneps-Sneppe. *On Micro-services Architecture*. Tech. rep. Open Information Technologies Lab, Lomonosov Moscow State University, 2014.

[Ric14a] C. Richardson. *Microservices: Decomposing Applications for Deployability and Scalability*. May 2014.

[Ric14b] C. Richardson. *Pattern: Microservices Architecture*. 2014.

[Ric14c] C. Richardson. *Pattern: Monolithic Architecture*. 2014. URL: `http://microservices.io/patterns/monolithic.html`.

[RKKnp] C. Rolland, R. S. Kaabi, and N. Kraiem. *On ISOA: Intentional Services Oriented Architecture*. Tech. rep. Université Paris, np.

[RS07] P. Reldin and P. Sundling. *Explaining SOA Service Granularity– How IT-strategy shapes services*. Tech. rep. Linköping University, 2007.

[RTS15] G. Radchenko, O. Taipale, and D. Savchenko. *Microservices validation: Mjolnirr platformcase study*. Tech. rep. Lappeenranta University of Technology, 2015.

[WB98] P. Weill and M. Broadbent. *Leveraging The New Infrastructure: How Market Leaders Capitalize on Information Technology*. Harvard Business School Press, 1998.

[Woo14] B. Wootton. *Microservices - Not A Free Lunch!* Apr. 2014. URL: `http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html`.

[WV04] L. Wilkes and R. Veryard. "Service-Oriented Architecture: Considerations for Agile Systems." In: *np* (2004).