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

The microservices architecture provides various advantages such as agility, independent scalability etc., as compared to the monolithic architecture and thus has gained a lot of attention. However, implementing microservices architecture is still a challenge as many concepts within the microservices architecture including granularity and modeling process are not yet clearly defined and documented. This research attempts to provide a clear understanding of these concepts and finally create a comprehensive guidelines for implementing microservices.
Various keywords from definitions provided by different authors are taken and cateogorized into various conceptual areas. These concepts along with the keywords are researched thoroughly to understand the microservices architecture. Additionally, the three important drivers: quality attributes, constraints and principles, are also focussed for creating the guidelines.
The findings of this research indicate that eventhough microservices emphasize on the concept of 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. Furthermore, in order to identify microservices, either the domain driven design approach or the use case refactoring approach can be used. Both these approaches can be effective in identifying microservices, but the concept of bounded context in domain driven design approach identifies autonomous services with single responsibility. Apart from literature, a detail study of the architectural approach used in industry named SAP Hybris is conducted. The interviews conducted with their key personnels has given important insight into the process of modeling as well as operating microservices. Moreover, the challenges for implementing microservices as well as the approach to tackle them are done based on the literature review and the interviews done at SAP Hybris.
Finally, the findings are used to create a detail guidelines for implementing microservices. The guidelines captures how to approach modeling microservices architecture and how to tackle its operational complexities.

Contents

1 Introduction

The software architecture is the set of principles assisting software architect and developers for system or application design [**Dashofy:2009aa**]. It defines a process to decompose a system into modules, components and specifies their interactions [**Brown:2015aa**]. In this chapter, two different architectural approaches namely monolithic architectural approach and microservices achitectural approach are discussed. Firstly, a conceptual understanding of the monolithic architecture approach is presented, which is followed by its various advantages and disadvantages. Secondly, an overview of the microservices architecture approach is explained. In Section 1.3, the motivation for the current research is discussed which is then followed by a list of research questions. Finally, in Section 1.4 and Section 1.5, the approach that is used to conduct the current research is presented. The purpose of this chapter is to provide a basic background context for the following chapters.

1.1 Monolithic Architectural Approach

A mononlithic architectural approach is one in which even a modular application is deployed as a single artifact. Figure 1.1 shows the architecture of an Online-Store application that has a clear separation of components such as Catalog, Order etc., as well as respective models such as Product, Order etc. Despite the modular decomposition of the application, all the components are deployed as a single application artifact on the server.[**Richardson:2014aa**][**Richardson:2014ab**]

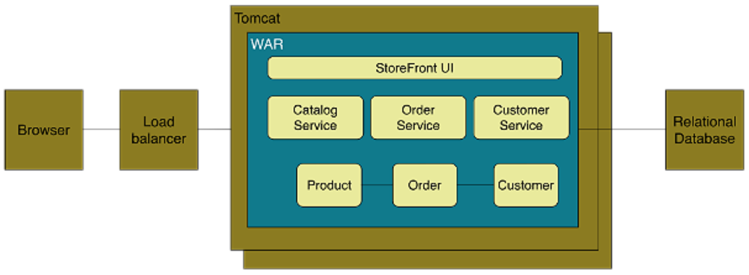


Figure 1.1: Monolith Example from [**Richardson:2014aa**]

1.1.1 Types of the Monolithic Architectural Approach

According to [**Annett:2014aa**], 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 called Module Monolithic Architecture. An example of this architecture is shown in Figure 1.2. The application on the left side of the figure, has all the code in the same codebase in the form of packages and classes without clear definition of modules and gets compiled to a single artifact. However, the application on the right side is developed as a number of modular codebase, each has separate codebase and can be compiled to different artifact. In the application shown in the left side of the figure, the various parts of the codebase can reference each other directly.

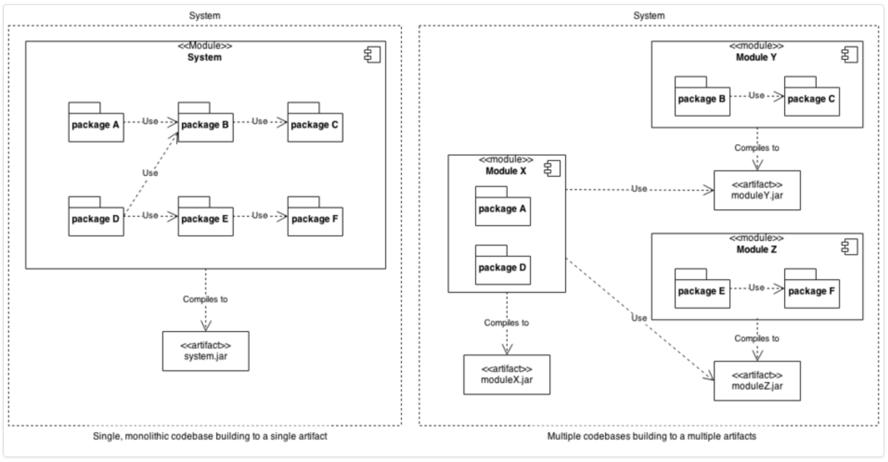


Figure 1.2: Module Monolith-Example from [**Annett:2014aa**]

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

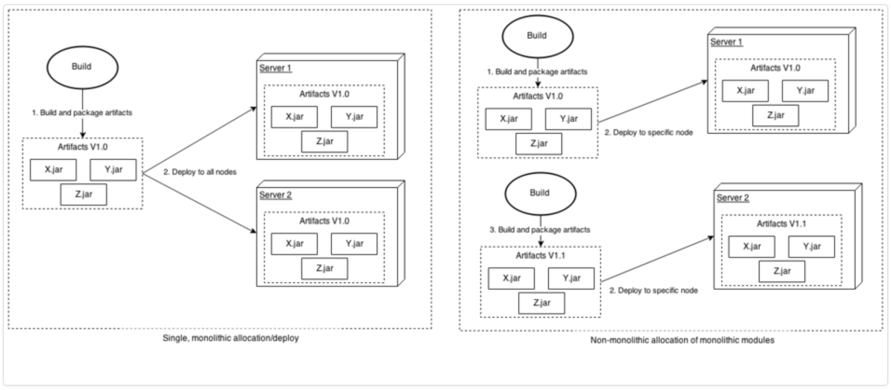

Build
1. Build and package artifacts
Artifacts V1.0
X.jar
Y.jar
Z.jar
2. Deploy to all nodes
Server 1
Server 2
Single, monolithic allocation/deploy
2. Deploy to specific node
Artifacts V1.1
3. Build and package artifacts
Non-monolithic allocation of monolithic modules

Figure 1.3: Allocation Monolith-Example from [**Annett:2014aa**]

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

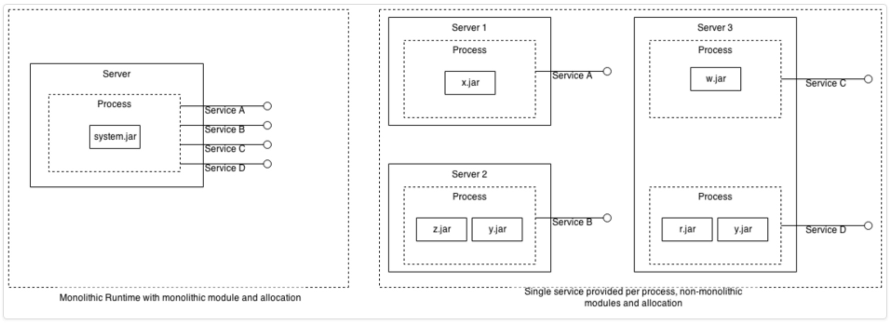

Server
Process
system.jar
Service A
Service B
Service C
Service D
Monolithic Runtime with monolithic module and allocation
Server 1
x.jar
Server 2
z.jar
y.jar
Server 3
w.jar
r.jar
Single service provided per process, non-monolithic modules and allocation

Figure 1.4: Runtime Monolith-Example from [**Annett:2014aa**]

1.1.2 Advantages of the Monolithic Architectural Approach

The monolithic architectural approach is appropriate for small applications and has the following benefits [**Richardson:2014ab**][**Fowler:2014aa**][**Gupta:2015aa**][**Abram:2014aa**] :

- 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 the Monolithic Architectural Approach

As the requirements of the system grows over time, the corresponding system's codebase becomes large and the size of team increases. Under such circumstances, the monolithic architecture faces many challenges as explained below [**Namiot:2014aa**] [**Newman:2015aa**][**Abram:2014aa**] [**Richardson:2014aa**] [**Richardson:2014ab**] [**Gupta:2015aa**] :

- **Limited Agility**: As the whole application has a single codebase, even changing a small feature to release it in the production takes time. Firstly, a small change can trigger changes to other dependent code. In huge monolithic applications, 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 the production, the whole application has to be deployed. Thus continuous delivery gets slower. This is 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. Overall, it will slow down the speed of understandability, execution and testing.

- **Difficult Team Structure**: The division of a team as well as assigning tasks to the team members can be challenging. Most common ways to partition teams in monolithic architectureal approach are by technology and by geography. However, partitioning either by technology or by geography may not be appropriate in all cases. 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 question who should find the problem, is it either the operations team or the last person to commited the code. The approprate team structure and the ownership of code 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 project need to follow the same techonology stack throughout the lifecycle of an application. However, if the requirement changes then there can be situations when specific features can be best solved by different set of technologies. Additionally, not all the features in the application are the same and hence cannot be solved using same technology. 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 monolithic applications, it is very difficult to migrate to a newer technology stack and it can rather be a painfull process.

- **Limited Scalability**: The scaling of monolith application can be performed 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 resiliency.**[MacVittie:2014aa][Namiot:2014aa]**

1.2 Microservices Architectural Approach

With monolith, it is easy to start development. But as the system gets bigger and complicated over 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 and agility need to be maintained for the whole lifetime of the application. It becomes complicated due to the fact that the system needs to be updated continuously as the as the requirements keep changing and evolving over time. In order to tackle these disadvantages, microservices architecture style is followed.
The microservices architecture uses the approach of decomposing an application into smaller autonomous components. The following Section 1.2.1 provides details of the 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 captured various disadvantages related to the monolithic architectural approach. In **[Fisher:2015aa]**, authors propose an approach called scale cube to address the challenges related to agility, scalability and productivity. The scale cube 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 of the system.

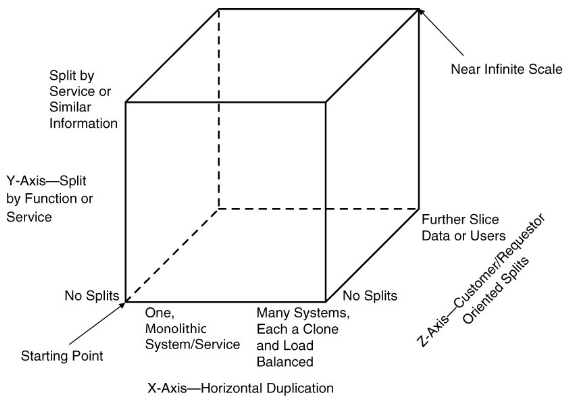

Split by Service or Similar Information
Near Infinite Scale
Y-Axis—Split by Function or Service
Further Slice Data or Users
Z-Axis—Customer/Requestor Oriented Splits
No Splits
No Splits
One, Monolithic System/Service
Many Systems, Each a Clone and Load Balanced
Starting Point
X-Axis—Horizontal Duplication

Figure 1.5: Scale Cube from **[Fisher:2015aa]**

The scaling along each dimensions are described below. **[Fisher:2015aa][MacVittie:2014aa][Richardson:**

1. X-axis Scaling: It is achieved by cloning the application and data along multiple servers. A pool of requests are sent to a load balancer and the requests are deligated to the servers. Each of the server has the latest version of the application and access to all the data required. 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 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 obtained 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 the 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 provided added functionality or a new functionality can be tested to a small group and thus minimizes the risk.

3. Y-axis Scaling: The scaling along this dimension means the spliting of the application's responsibility. The separation can be done either by data, by the actions performed on the data or by the 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 are 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 in 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 are a standard way of sharing functionalities among various services and teams. This capability is provided as a feature of the programming languages. The shared libraries however do not provide technology heterogeneity. Furthermore, a) independent scaling b) independent deployment and c) independent maintainance cannot be achieved unless the libraries are dynamically linked. In such cases, any small change in the library leads to redeployment of the whole system. Moreover, sharing the code is a form of coupling which should be avoided.

Decomposing an application into individual features gives various advantages such that each feature can be a) scaled independently b) deployed independently c) agile and d) assigned easily to teams. Microservices uses the decomposition technique proposed by the scale cube. The detail advantages of microservices architectural approach are discussed further in Section 6.1.

1.2.2 Definitions

There are several definitions given by several pioneers and early adapters of the microservices architectural approach.

Definition 1: [**Richardson:2014ac**]

"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: [**Wootton:2014aa**]

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

Definition 3: [**Cockcroft:2015aa**]

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

Definition 4: **[Fowler:2014aa][Radchenko:2015aa]**

"Microservices are Service-Oriented Architecture done right."

Definition 5: **[Fowler:2014aa]**

"Microservices 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."
Similar to other architectural approaches, microservices presents its approach by increasing cohesion and decreasing coupling. Besides that, it breaks down the system along business domains (following the single responsibility principle) into granular and autonomous services running on separate processes. Additionally, the architecture focuses on the collaboration of these services using light weight mechanisms.

1.3 Motivation

The various definitions presented in the previous 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 understand and realize the microservices architecture correctly and effectively. The first two terms in list relate to the runtime operational qualities of microservices whereas the later two address modeling qualities. With that consideration, it indicates that the definitions highlights on both modeling as well as operational aspects of any software applications.
However, if an attempt is made to have a clear indepth understanding of each of the key terms then various questions can be raised without suitable answers. The different

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

#	Questions	Type
1	How small should be the size of microservices?	Modeling
2	How do services interact and coordinate with eachother?	Operation
3	How to deploy and maintain services independently when there are dependencies among services?	Operation
4	How to derive and map microservices from business capabilities?	Modeling
5	What are the challenges those need to be tackled when implementing microservices and how to tackle the challenges?	Operation

Table 1.1: Various Questions related to Microservices

Without clear answers to these questions, it is difficult to say that the definitions presented in Section 1.2.2 are adequate to follow the microservices architecture. To a large extent, the process of creating microservices is not clearly documented. The research in this area of formalizing the process of modeling and operating microservices is still in its infancy. Additionally, eventhough a lot of industries such as amazon, netflix etc. are following this architecture, the process of modeling and implementing microservices is not clear. The purpose of this research is to provide a clear understanding about the process of designing microservices by focusing on the following questions.

Research Questions

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

1.4 Research Approach

A research is an iterative procedure with the goal of collecting as many relevant documents as possible so that the research questions could be answered rationally. Therefore, it becomes very important to follow a consistent research procedure. For the current research, the approach is chosen by refering to [**np:2007aa**]. It consists of two major phases. a. Data Collection Phase and b. Data Synthesis Phase
The following sections explain each of the phases in detail.

1.4.1 Data Collection Phase

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

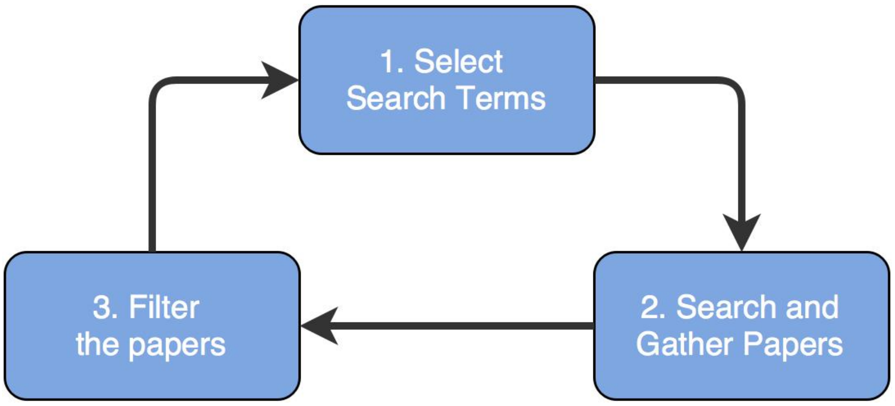


Figure 1.6: Data Collection Phase

1. **Select Search Terms**
   At first, various search terms which define the research topics and questions, are selected using the 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 listed in Table 1.2.

2. **Search and Gather Papers**
   The search terms selected in the previous step are utilized to discover various research 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 research papers collected from various resources are filtered first to check if they have profound base to back up their result. Using only authentic papers is important to provide rational base to the 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 a good base in terms of sources and provide references of the past studies?
   c) Are there any case studies or examples provided to verify the result of the reseach?

1.4.2 Data Synthesis Phase

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

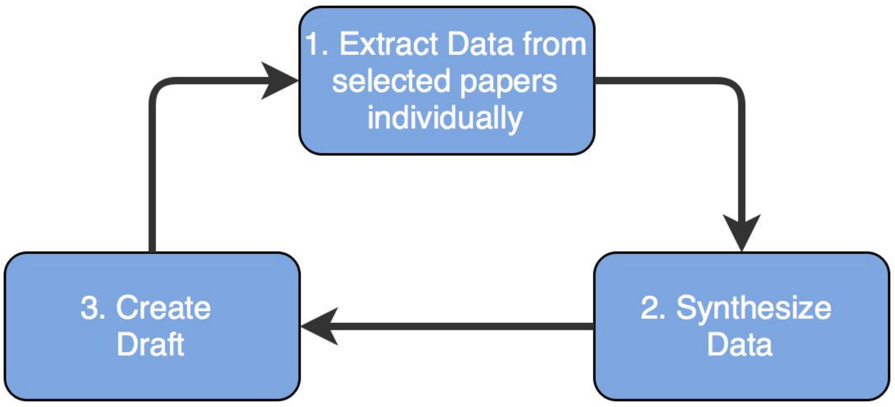


Figure 1.7: Data Synthesis Phase

1. **Extract Data from selected papers individually**
   Firstly, each research paper is 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 content from all research papers are then synthesized.

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

1.5 Research Strategy

The Section 1.4 emphasizes the importance of having a consistent research procedure. It focusses on how to conduct a research in a consistent manner. In addition, it is equally important to understand what to research. 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 is 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 relevance 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, the first group of areas to focus are various aspects shown in columns such as size, quality of microservices etc.
The next group of areas which need to be understood, are various drivers for defining microservices architecture. According to [**Brown:2015aa**], the important drivers for software 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 architectural approach. The better knowledge of the constraint and moreover the solution to these constraints is useful in the process of explaining the architecture.

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** is studied thoroughly.

Considering both the Table 1.2 and List 1.5, the first area to understand are various quality attributes influencing microservices architecture. Additionally, the process of identifying and modeling individual microservices from problem domain is defined. These areas are first researched based on various academic research papers. Then, the approach used by the industry for adopting microservices architecture is studied in order to answer the research questions. The industrial case study is performed in SAP Hybris. Finally, the various constraints and challenges which affect microservices are identified. The results of the previous research is used to create various principles for implementing microservices architecture. These principles provide sufficient ground to create guidelines for modeling and operating microservices.

1.6 Problem Statement

Considering the case that the size of microservice is an important concept and is discussed a lot, but there is no concrete answer about what defines the size of a microservice. 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 is to understand the concept of granularity in the context of microservices.

2 Granularity

In this chapter, a detailed concept regarding the size of the microservices is discussed. A major focus is to research various qualitative aspects which define the granularity of the microservices. Section 2.2 lists various principles which guide identifying correct size by considering various aspects. Finally, in Section 2.3, various interpretations defining the dimensions of the microservices are presented.

2.1 Introduction

The granularity of a service is often ambiguous and has different interpretations. In simple term, it refers to the size of the service. However, the size itself can be vague. It cannot be defined as a single quantitative value because the concepts defining granularity are subjective in nature. For example, if we choose activities supported by the service to determine its granularity then we cannot have one single value instead a hierarchical list of activities, where each activity can either refer to a) a simple state change, b) any action performed by an actor or c) a complete business process. **[Linthicum:2015aa]**, **[Raf-Haesen:2015aa]**
Although, the interest upon the granularity of a component or service for the business users highly depends upon their business value, there is no doubt that the granularity affects the architecture of a system. The granularity of a service should reflect upon the business perspective and should also consider the impact upon the overall architecture. If we consider other variations of the software architecture, we have come from object oriented to component based architecture 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 the real world objects, each unit representing fine grained abstraction with some attributes and functionalities.
Such abstraction is a good approach towards simplicity in development and easier understanding of the application. However, it is not always sufficient when high order business goals have to be implemented. It indicates the necessity of coarser-grained units than the units of object oriented paradigm. Moreover, the component based development introduces the concept of business components which target the business

problems and are coarser grained. Similarly, in service oriented architecture, the services provide access to the application and each application is composed of various component services. [**Linthicum:2015aa**]

2.2 Basic Principles of the Service Granularity

In addition to the quantitative perspective of granularity, it can be helpful to understand the qualitative aspects of granularity. The list below specifies some basic principles derived from various research papers and defines the qualitative properties of granularity. These principles are helpful to determine the correct granularity of the microservices.

1. The correct granularity of a service is dependent upon the time. The various supporting technologies that evolve over time can 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 (PaaS) technologies, it is fast and easy for deployment automation. These technologies makes it easy to support large number of services. Finally, the creation and operation of multiple fine-grained services is possible.[**Peter-Herzum:2000aa**]

2. A good candidate for a service should be independent of the implementation but should depend upon the understandability of domain experts [**Raf-Haesen:2015aa**; **Peter-Herzum:2000aa**].

3. A service should be an autonomous reusable component and should support various cohesions such as functional cohesion (group similar functions), temporal cohesion (change in the service should not affect other services), run-time cohesion (allocate similar runtime environment for similar jobs; eg. provide same address space for jobs of similar computing intensity) and actor cohesion (a component should provide service to similar users). [**Raf-Haesen:2015aa**], [**Peter-Herzum:2000aa**]

4. A service should not support huge number of operations. In such case, any change in the service will affect high number of customers and there will be no unified and simplified view on the functionality. But, if the interface of the service is small, it will be easy to maintain and understand.[**Raf-Haesen:2015aa**], [**Pierre-Reldin:2007aa**]

5. A service should provide transaction integrity and compensation. The functionalities supported by the 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. **[Raf-Haesen:2015aa]**, **[Foody:2005aa]** **[Bianco:2007aa]**

6. The notion of right granularity is more important than that of fine or coarse. The size depends upon the usage condition and moreover depends on balancing various qualities such as reusability, network usage, completeness of context etc. **[Raf-Haesen:2015aa]**, **[Lawrence-Wilkes:2004aa]**

7. The level of abstraction of the services should reflect the real world business activities. The situation will help to map business requirements and technical capabilities easily. **[Pierre-Reldin:2007aa]**

8. If there are ordering dependencies between operations, it can be easy to implement and test if the dependent operations are all combined into a single service **[Bianco:2007aa]**.

9. There can be two better approaches for breaking down an abstraction. One way is to separate redundant data or data with different semantic 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 code can be divided into two independent fine-grained services: Customer Registration and Code Generation, each service will have limited scope and separate context of Customer.**[Pierre-Reldin:2007aa]**

10. If there are functionalities 'A, B, C' provided by a service which are more likely to change than other functionalities 'D, E' in the service. It is better to extract the functionalities 'A, B, C' into a fine-grained service so that any further change on the functionalities will affect only limited number of consumers. **[Bianco:2007aa]**

2.3 Dimensions of the Service Granularity

As already mentioned in Section 2.1, it is not easy to define granularity of a service quantitatively. However, understanding granularity can be easier if we can project it along various dimensions, where each dimension is a qualifying attribute responsible for determining size of a service. Eventhough the dimensions discussed in this section do not give the precise quantity to identify granularity, it definitely gives the hint to locate the service in granularity space. Moreover, it can be interesting and beneficial to know how these dimensions come together for defining the size of the microservices.

2.3.1 Dimensions given by Interface Perception of Consumers

One way to define the granularity is by understanding the perspective of the consumer towards the service interface. The various properties of the service interface responsible to define its size are listed below.

1. Functionality: It defines 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 by the service. Another type of the functionality can be parameterized and it can be optionally provided depending upon some values. Based on the scope of the functionality, the service can be either fine-grained or course-grained than the other services. Considering functionality criterion, a service offering basic CRUD functionality is fine-grained than a service which is offers some data accumulation using orchestration. [**Raf-Haesen:2015aa**]

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 the input data granularity, which is the amount of data consumed or needed by the service in order to accomplish its tasks. The other type 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 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. [**Raf-Haesen:2015aa**]

3. Business Value: According to [**Rolland:2015aa**], 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. The level of aggregation is directly influenced by the target business goal of the services. An atomic service can 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. [**Raf-Haesen:2015aa**]

2.3.2 Dimensions given by Interface Realization

The Section 2.3.1 provides the aspect of granularity determined by the perception of customer on service interface. However, there can be different opinions on the same dimenstions listed in Section 2.3.1, when it it comes to the imlementation. In ths section, the same dimensions of granularity listed in the Section 2.3.1 are analyzed again considering various aspects at implementation level.

1. Functionality: In Section 2.3.1, it is mentioned that an orchestration service has higher granularity than its constituent services with regards to the default functionality. If the realization effort is focused, it may only include compositional and/or compensation logic because the individual tasks are already acomplshed by the constituent services. Thus, in terms of the effort in realizing the orchestration service it is fine-grained than from the view point of interface by consumer. **[Raf-Haesen:2015aa]**

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 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. **[Raf-Haesen:2015aa]**

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 service 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 critical to analyze the complexity associated with the implementation, reliablity and upgrading of the data management service. From the realization point of view, the data service is coarse-grained than its view by consumer. **[Raf-Haesen:2015aa]**

Principle: A high Functionality granularity does not necessarily mean high business value granularity
It may seem that the dimensions: functionality and business value have direct proportional relationship. The business value of a service reflects business goals

however the functionality refers to the amount of work performed 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 may be of a very low business value to the enterprise. [**Raf-Haesen:2015aa**]

2.3.3 R^3 Dimension

Keen [**Keen:2015aa**] and later Weill and Broadbent [**Weill:1998aa**] introduced a separate group of criteria to measure the granularity of a service. The granularity is evaluated in terms of two dimensions as shown in the Figure 2.1

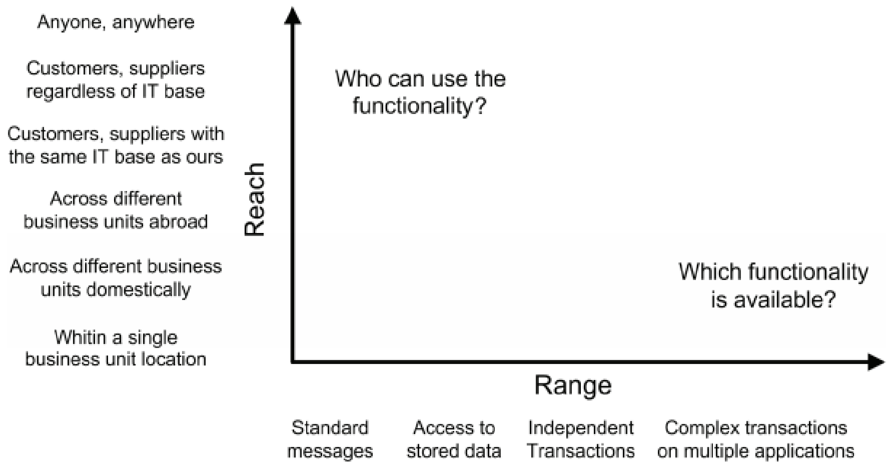

Anyone, anywhere
Customers, suppliers regardless of IT base
Customers, suppliers with the same IT base as ours
Across different business units abroad
Across different business units domestically
Whitin a single business unit location
Reach
Who can use the functionality?
Which functionality is available?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

Figure 2.1: Reach and Range model from [**Keen:2015aa**; **Weill:1998aa**]

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

2. Range: It gives answer to "Which functionality is available? ". It defines the extent to which the information can be accessed from the service or shared with the service. The various levels of functionalities can be simple data access, 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.

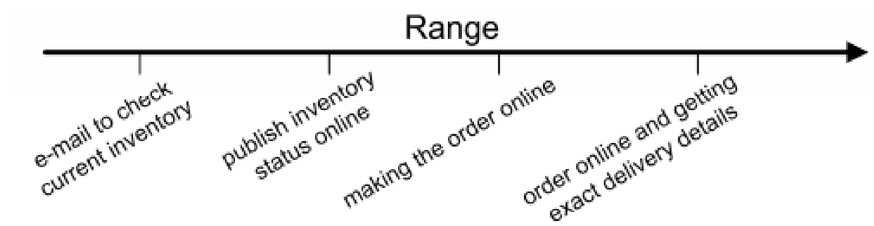

Range
e-mail to check current inventory
publish inventory status online
making the order online
order online and getting exact delivery details

Figure 2.2: Example to show varying 'Range' from [**Keen:2015aa**; **Weill:1998aa**]

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 level of data access. As the service grows, 'Reach' and 'Range' also peaks up, which means the extent of consumers as well as the kind of functionality increase. This adds complexity to the service. The solution proposed by [**Cockburn:2001aa**] is to divide the architecture into services. However, only 'Reach' and 'Range' can not be enough to define the service. It is equally important to determine scope of the individual services. The functionality of the service is defined in two distinct dimensions 'which kind of functionality' and 'how much functionality'. So, this leads to another dimension of the service as described below. [**Keen:2015aa**; **Weill:1998aa**; **Pierre-Reldin:2007aa**]

3. Realm: It tries to create a boundary around the scope of the functionality provided by the service. If we take the same example as shown by Figure 2.2, only 'range' 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 an important role here. So, we can have two different services each with same 'Reach' and 'Range' however different 'Realm' for customer order and Sales order. [**Keen:2015aa**; **Weill:1998aa**; **Pierre-Reldin:2007aa**]

The consideration of all the 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 than a fine-grained service. The Figure 2.3 and Figure 2.4 show such volume-granularity analogy given by R^3 model. [**Keen:2015aa**; **Weill:1998aa**; **Pierre-Reldin:2007aa**]

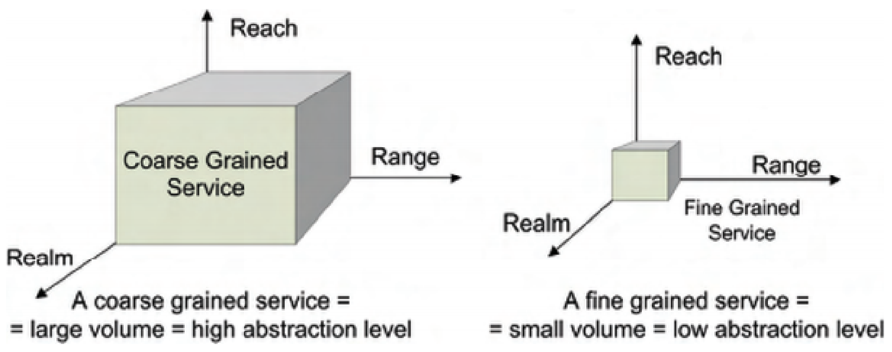

Reach
Coarse Grained Service
Range
Realm
A coarse grained service =
= large volume = high abstraction level
Reach
Range
Realm
Fine Grained Service
A fine grained service =
= small volume = low abstraction level

Figure 2.3: R^3 Volume-Granularity Analogy to show direct dependence of granularity and volume [**Pierre-Reldin:2007aa**]

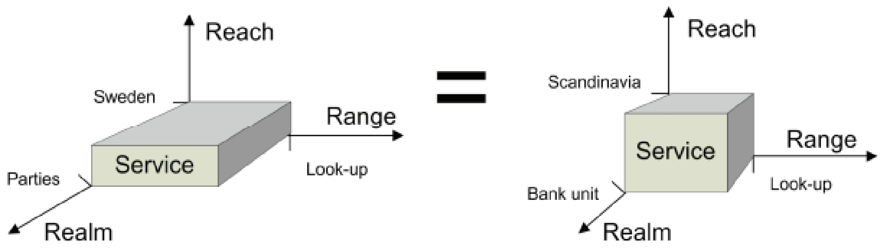


Figure 2.4: R^3 Volume-Granularity Analogy to show same granularity with different dimension along axes [**Pierre-Reldin:2007aa**]

2.3.4 Retrospective

The Section 2.3.1 and Section 2.3.2 divides granularity of a service along three different directions: data, functionality and business value. Moreover, the interpretations from the perspective of consumer and then the viewpoint of the producer for realization are made. Again, the Section 2.3.3 divides the aspect of granularity along a) Reach, b) Range, and c) Realm space, the granularity given by the volume in the R^3 space. Despite the good explanation regarding each dimenstions such as data, functionality and business value, the classification does 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 used to determine if a service is fine-grained than other service. The R^3 model in Section 2.3.3 also provides various dimensions of granularity. This makes it easy to visualize the granularity and provides flexibility to compare the granularity between various services.
A case study **[Pierre-Reldin:2007aa]** using data, functionality and business criteria to evaluate the impact of granularity on the complexity in the architecture was conducted. The study was performed at KBC Bank Insurance Group of central Europe. Along the study, the impact of each kind of granularity was 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 follows 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 functionalities 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 many services are additionally needed to accomplish high level functionalities. Furthermore, it increases 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. **[Pierre-Reldin:2007aa]**

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 operation, infrastructure etc if they choose low granularity. **[Pierre-Reldin:2007aa]**

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 dimension 'reach' of a service has to be increased from domestic to global in an organization then the dimension '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. **[Pierre-Reldin:2007aa]**

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 a) Reach, b) Range, and c) Realm. It will help to decrease development and maintenance overload. **[Pierre-Reldin:2007aa]**

2.4 Problem Statement

In this chapter, various qualitative aspects related to the granularity of the microservices are analyzed. The interesting thing to notice is various dimensions of the size across data, functionality and business value. But it is also necessary to look into them quantitatively. Additionally, the various principles listed in Section 2.2 points towards other quality attributes of microservices including coupling, cohestion etc. The granularity is not the only attribute which is important while modeling the microservices but there are other quality attributes which affect the granularity and also influence on the overall quality of the microservices. The next task is to study the various quality attributes affecting the microservices and derive their quantitative metrics.

3 Quality Attributes of Microservices

This chapter defines various quality attributes of the microservices. Also, it provides different quantitative approaches to evaluate the differnt quality attributes of the microservices. The Section 3.2 presents a list of quality attributes compiled from various research papers. In Section 3.3, various metrics to evaluate these quality attributes are discussed. Furthermore, in Section 3.4, a list of basic metrices (derived from the complex metrics discussed in section 3.3) shows the simplified way to determine the quality of the microservices. Again, Section 3.5 lists various principles to show the impact of various attributes on the quality of microservices. Finally, relationship amongst various quality attributes is discussed in Section 3.6.

3.1 Introduction

One of the various goals of software engineering is to develop application with high quality. In case of the applications following service oriented architecture, the complexity increases over time. So, for these applications, quality assessment becomes very crucial. [**Zhang:2009aa**; **Goeb:2011aa**; **Nematzadeh:2014aa**] Quality models have been invented over time to evaluate quality of a software. A quality model is defined by quality attributes and quality metrics. Quality attributes are the expected properties of software artifacts defined by the chosen quality model. And, quality metrics give the techniques to measure the quality attributes. [**Mancioppi:2015aa**] The software quality attributes can again be categorized into two types: a) internal attributes and b) external attributes. [**Mancioppi:2015aa**; **Briand:1996aa**] The internal quality attributes are the design time attributes which needs to be fulfilled during the design of the software. Some of the internal quality attributes are loose coupling, cohesion, granularity, autonomy etc. [**Rostampour:2011aa**; **Sindhgatta:2015aa**; **Elhag:2014aa**] On the other hand, external quality attributes are the traits of the software artifacts produced at the end of SDLC (Software Development Life Cycle). Some of them are reusability, maintainability etc. [**Elhag:2014aa**; **Mancioppi:2015aa**; **Feuerlicht:2007aa**; **Feuerlicht:2013aa**] The external quality attributes can only be measured after the end of development. However, it has been evident that internal quality attributes have huge impact upon the value of external quality attributes and thus can be used to predict them [**Henry:1990aa**; **Briand:2015aa**; **Alshayeb:2003aa**; **Bingu-Shim:2008aa**]. The evaluation of both internal and external quality attributes are valuable in order to produce high quality software. **[Mancioppi:2015aa; Perepletchikov:2007aa; Mikhail-Perepletchikov:2015aa]**

3.2 Quality Attributes

As already mentioned in Section 3.1, the internal and external quality attributes determine the overall quality of the service getting composed. There are different researches and studies which have been performed to find the factors affecting the service qualities. Based on the published research papers, a comprehensive Table 3.1 has been created. The table provides a minimum list of quality attributes which have been considerd in various research papers for determining the quality of the services.

#	Attribute	**[Sindhgatta:2015aa]**	**[Xiao-jun:2015aa]**	**[Saad-Alahmari:2011aa]**	**[Bingu-Shim:2008aa]**	**[Ma:2009aa]**	**[Feuerlicht:2007aa]**
1	Coupling	✓	✓	✓	✓	✓	✓
2	Cohesion	✓	X	✓	✓	✓	✓
3	Autonomy	✓	X	X	X	X	X
4	Granularity	✓	✓	✓	✓	✓	✓
5	Reusability	✓	X	X	✓	X	✓
6	Abstraction	✓	X	X	X	X	X
7	Complexity	X	X	X	✓	X	X

Table 3.1: Quality Attributes

The Table 3.1 gives an overview of the research papers around the quality attributes of the services. From the table, it can be deduced that most of the papers focus on coupling and granularity of the service and only few of them focus on other attributes such as complexity and autonomy.

Coupling refers to the dependency and interaction among services. The interaction becomes inevitable when a service requires a functionality provided by another service for accomplishing its own goals. Similarly, Cohesion of any system is the extent of appropriate arrangement of elements to perform the functionalities. It affects the degree of understandability and sensibility of its interface for its consumers. Whereas, the complexity attribute provides the way to evaluate the difficulty in understanding and reasoning the context of the service.**[Elhag:2014aa]**
The reusability attribute for a service measures the degree to which it can be used by multiple consumer services and can undergo multiple composition to accoplish various high order functionalities. **[Feuerlicht:2007aa]**
Autonomy is a broad term which refers to the ability of a service for a) self-containment, b) self-controlling, and c) self-governance. Autonomy defines the boundary and context of the service. **[Ma:2007aa]** Finally, another important quality attribute is granularity and is described in Chapter 2. The basic signifance of granularity is the diversity and size of the functionalities offered by the service. **[Elhag:2014aa]**

3.3 Quality Metrics

There have been many researches conducted to define quality metrics for components. A major portion of the researches have been made for object oriented and component based architectural approaches. These metrics are refined to be used in service oriented systems [**Xiao-jun:2015aa**; **Sindhgatta:2015aa**]. Firstly, various research papers defining the quality metrics of service oriented systems are collected. The research papers which have conducted their findings or proposition based on decent amount of scientific researches and have produced convincing result are only selected.
It is interesting to notice that the evaluation presented in the research papers are only performed using the case studies of their own but not real life scenarios. Moreover, they have used diversity of equations to define the quality metrics. Additionally, all the research papers do not focus on all the quality attributes. So, a fair comparision and selection of single dominant way to evaluate the quality metrics is difficult. Nevertheless, all of them do not discuss about the relationship between all the quality metrics [**Elhag:2014aa**]. This creates difficulty to map all the quality metrics into a common cohesive family.
Around such situation, this section facilitates the understanding of the quality metrics. Despite the case that there is no consensus on threshold value of quality metrics which defines the optimal quality level, the method discussed in this section can still be used to evaluate and compare the quality of microservices. This definitely helps to choose the identify good microservices based on various quality attributes. Furthermore, the quality metrics defined in research papers are quite complex to understand and apply. The complex metrics can be broken down further to the simple understandable terms called the basic metrics. The basic metrics are the driving factors for the quality attributes and are also utilized by most of the the research papers to define the quality metrics.
The remaining part of this section discusses the metrics definitions proposed in various research papers and analyzes them to identify their conceptual simplest form.

3.3.1 Context and Notations

Before looking into the definitions of metrics, it can be helpful to know related terms, assumptions and their respective notations.

- the business domain is realized with various processes defined as $P = \{p_1, p_2 ... p_s\}$
- a set of services realizing the application is defined as $S = \{s_1, s_2 ... s_s\}$; s_s is the total number of services in the application

- for any service $s \epsilon S$, the set of operations provided by s is given by $O(s) = \{o_1, o_2, ...o_o\}$ and $|O(s)|$ = O
- if an operation $o \epsilon O(s)$ has a set of input and output messages given by M(o). Then the set of messages of all operations of the service is given by $M(s) = \cup_{o \epsilon O(s)} M(o)$
- the consumers of the service $s \epsilon S$ is given by $S_{consumer}(s) = \{S_{c1}, S_2, ...Sn_c\}$; n_c gives the number of consumer services
- the producers for the service or the number of services to which the service $s \epsilon S$ is dependent upon is given by $S_{producer}(s) = \{S_{c1}, S_2, ...Sn_p\}$; n_p gives the number of producer services

3.3.2 Coupling Metrics

[**Sindhgatta:2015aa**] defines following metrics to determine coupling

$$SOCI(s) = number\_of\_operations\_invoked$$

$$= |\{o_i \epsilon s_i : \exists_{o \epsilon s} calls(o, o_i) \land s \neq s_i\}|$$

$$ISCI(s) = number\_of\_services\_invoked$$

$$= |\{s_i : \exists_{o \epsilon s}, \exists_{o_i \epsilon s_i} calls(o, o_i) \land s \neq s_i\}|$$

$$SMCI(s) = size\_of\_message\_required\_from\_other\_services$$

$$= | \cup M(o_i) : (o_i \epsilon s_i) \lor (\exists_{o \epsilon s}, \exists_{o_i \epsilon s_i} calls(o, o_i) \land s \neq s_i)|$$

where,

- SOCI is Service Operation Coupling Index
- ISCI is Inter Service Coupling Index
- SMCI is Service Message Coupling Index
- $call(o, o_i)$ represents the call made from service 'o' to service $'o_i'$

[**Xiao-jun:2015aa**] defines

$$Coupling(S_i) = p \sum_{j=1}^{n_s} (-log(P_L(j)))$$

$$= \frac{1}{n} \sum_{j=1}^{n_s} (-log(P_L(j)))$$

where,

- n_s is the total number of services connected
- 'n' is the total number of services in the entire application
- $p = \frac{1}{n}$ gives the probability of a service participating in any connection

[**Kazemi:2011aa**] defines

$$Coupling = \frac{dependency\_on\_business\_entities\_of\_other\_services}{number\_of\_operations\_and\_dependencies}$$

$$= \frac{\sum_{i \epsilon D_s} \sum_{k \epsilon O(s)} CCO(i,k)}{|O(s)|.K}$$

where,

- $D_s = \{D_1, D_2, ...D_k\}$ is set of dependencies the service has on other services
- $K = |D_s|$
- CCO is Conceptual Coupling betweeen service operations and obtained from BE X EBP (CRUD) matrix table constructed with business entities and business operations, where BE is business entity and EBP any logical process defined in an operation.

[**Bingu-Shim:2008aa**] defines

$$coupling(s) = \frac{number\_of\_services\_connected}{number\_of\_services}$$

$$= \frac{n_c + n_p}{n_s}$$

where,

- n_c is number of consumer services
- n_p number of dependent services
- n_s total number of services

[**Saad-Alahmari:2011aa**] defines

$$coupling = \frac{number\_of\_invocation}{number\_of\_services}$$

$$= \frac{\sum_{i=1}^{|O(s)|}(S_{i,sync} + S_{i,async})}{|S|}$$

where,

- $S_{i,sync}$ is the synchronous invocation in the operation O_i
- $S_{i,async}$ is the asynchronous invocation in the operation O_i

The Table 3.2 shows simpler form of metrics proposed for coupling. Although, the table shows different way of evaluating coupling from different research papers, they somehow agree to the basic metrics to calculate coupling. It can be deduced that the metrics to evaluate coupling uses basic metrics such as a) number of operations, b) number of provider services, and c) number of messages.

3.3.3 Cohesion Metrics

[**Sindhgatta:2015aa**] defines following metric for cohesion.

$$SFCI(s) = \frac{number\_of\_operations\_using\_same\_message}{number\_of\_operations}$$

$$= \frac{max(\mu(m))}{|O(s)|}$$

where,

- SFCI is Service Functional Cohesion Index

#	Papers	Metrics Definition
1	**[Sindhgatta:2015aa]**	SOCI : number of operation of other services invoked by the service ISCI : number of services invoked by a service SMCI : total number of messages from information model required by the operations
2	**[Xiao-jun:2015aa]**	Coupling is evaluated as information entropy of a complex service component where entropy is calculated using various data such as total number of atomic components connected, total number of links to atomic components and total number of atomic components
3	**[Kazemi:2011aa]**	The coupling is measured by using the dependency of a service operations with operations of other services. Additionally, the dependency is considered to have different weight value for each kind of operation such as Create, Read, Update, Delete (CRUD) and based on the type of business entity involved. The weight is referred from the CRUD matrix constructed in the process.
4	**[Bingu-Shim:2008aa]**	evaluated by the average number of directly connected services
5	**[Saad-Alahmari:2011aa]**	defines coupling as the average number of synchronous and asynchronous invocations in all the operations of the service

Table 3.2: Coupling Metrics

- the number of operations using a message 'm' is $\mu(m)$ such that $m \epsilon M(s)$ and |O(s)| > 0
  The service is considered highly cohesive if all the operations use one common message. This implies that a higly cohesive service operates on a small set of key business entities as guided by the messages and are relavant to the service and all the operations operate on the same set of key business entities. **[Sindhgatta:2015aa]**

[Bingu-Shim:2008aa] defines

$$cohesion = \frac{n_s}{|M(s)|}$$

[Perepletchikov:2007aa] defines

$$SIDC = \frac{number\_of\_operations\_sharing\_same\_parameter}{total\_number\_of\_parameters\_in\_service}$$

$$= \frac{n_{op}}{n_{pt}}$$

$$SIUC = \frac{sum\_of\_number\_of\_operations\_used\_by\_each\_consumer}{product\_of\_total\_no\_of\_consumers\_and\_operations}$$
$$= \frac{n_{oc}}{n_c * |O(s)|}$$

$$SSUC = \frac{sum\_of\_number\_of\_sequentials\_operations\_accessed\_by\_each\_consumer}{product\_of\_total\_no\_of\_consumers\_and\_operations}$$

$$= \frac{n_{so}}{n_c * |O(s)|}$$

where,

- SIDC is Service Interface Data Cohesion
- SIUC is Service Interface Usage Cohesion
- SSUC is Service Sequential Usage Cohesion
- n_{op} is the number of operations in the service which share the same parameter types
- n_{pt} is the total number of distinct parameter types in the service
- n_{oc} is the total number of operations in the service used by consumers
- n_{so} is the total number of sequentially accessed operations by the clients of the service

[Saad-Alahmari:2011aa] defines

$$cohesion = \frac{max(\mu(OFG, ODG))}{|O(s)|}$$

where,

- OFG and ODG are Operation Gunctionality Granularity and Operation Data Granularity respectively as calculated in Section 3.3.4
- $\mu(OFG, ODG)$ gives the number of operations with specific value of OFG and ODG

#	Papers	Metrics Definition
1	[**Sindhgatta:2015aa**]	defines SFCI which measures the fraction of operations using similar messages out of total number of operations in the service operations of the service
2	[**Perepletchikov:2007aa**]	SIDC : defines cohesiveness as the fraction of operations based on commonality of the messages they operate on SIUC : defines the degree of consumption pattern of the service operations which is based on the similarity of consumers of the operations SIUC : defines the cohesion based on the sequential consumption behavior of more than one operations of a service by other services
3	[**Bingu-Shim:2008aa**]	cohesion is evaluated by the inverse of average number of consumed messages by a service
4	[**Saad-Alahmari:2011aa**]	cohesion is defined as the consensus among the operations of the service regarding the functionality and data granularity which represents the type of parameters and the operations importance as calculated in the Section 3.3.4

Table 3.3: Cohesion Metrics

The Table 3.3 gives simplified view for the cohesion metrics. The papers presented agree on some basic metrics such as a) similarity of the operation usage behavior, b) commanility in the messages consumed by operations and c) similarity in the size of operations.

3.3.4 Granularity Metrics

[**Sindhgatta:2015aa**] defines

$$SCG = number\_of\_operations = |O(s)|$$

$$SDG = size\_of\_messages = |M(s)|$$

where,

- SCG is Service Capability Granularity
- SDG is Service Data Granularity

[**Bingu-Shim:2008aa**] defines

$$granularity = \frac{number\_of\_operations}{size\_of\_messages} = \frac{|O(s)|^2}{|M(s)|^2}$$

[Saad-Alahmari:2011aa] defines

$$ODG = fraction\_of\_total\_weight\_of\_input\_and\_output\_parameters$$
$$= \left(\frac{\sum_{i=1}^{n_i o} W_{pi}}{\sum_{i=1}^{n_i s} W_{pi}} + \frac{\sum_{j=1}^{n_j o} W_{pj}}{\sum_{j=1}^{n_j s} W_{pj}} \right)$$

$$OFG = complexity\_weightage\_of\_operation = \frac{W_i(o)}{\sum_{i=1}^{|O(S)|} W_i(o)}$$

$$granularity = sum\_of\_product\_of\_data\_and\_functionality\_granularity$$
$$= \sum_{i=1}^{|O(S)|} ODG(i) * OFG(i)$$

where,

- ODG is Operation Data Granularity
- OFG is Operation Functionality Granularity
- W_{pi} is the weight of input parameter
- W_{pj} is the weight of output parameter
- $n_i o$ is the number of input parameters in an operation
- $n_j o$ is the number of output parameters in an operation
- $n_i s$ is the total number of input parameters in the service
- $n_j s$ is the total number of output parameters in the service
- $W_i(o)$ is the weight of an operation of the service
- $|O(S)|$ is the number operations in the service

The Table 3.4 presents various view of granularity metrics based on different research papers. Based on the Table 3.4, the granularity is evaluated using some basic metrics such as a) the number and type of the parameters, b) number of operations and c) number of messages consumed.

#	Papers	Metrics Definition
1	[**Sindhgatta:2015aa**]	the service capability granularity is calculated by the number of operations in a service and the data granularity by the number of messages consumed by the operations of the service
2	[**Saad-Alahmari:2011aa**]	ODG : evaluated in terms of number of input and output parameters and their type such as simple, user-defined and complex OFG : defined as the level of logic provided by the operations of the services SOG : defined by the sum of product of data granularity and functionality granularity for all operations in the service
3	[**Bingu-Shim:2008aa**]	evaluated as the ratio of squared number of operations of the service to the squared number of messages consumed by the service

Table 3.4: Granularity Metrics

3.3.5 Complexity Metrics

[**Zhang:2009aa**] defines

$$RIS = \frac{IS(s_i)}{|S|}$$

$$RCS = \frac{coupling\_of\_service}{number\_of\_services} = \frac{CS(s_i)}{|S|}$$

where,

- $CS(s_i)$ gives coupling value of a service
- |S| is the number of services to realize the application
- $IS(s_i)$ gives importance weight of a service in an application

The complexity is rather obtained by relative coupling than by coupling on its own. A low value of RCS indicates that the coupling is lower than the count of services where as RCS with value 1 indicates that the coupling is equal to the number of services. This represents high amount of complexity.
Similarly, a high value of RIS indicates that a lot of services are dependent upon the service and the service is of critical value for other services. This increases complexity as any changes or problem in the service affects a large number of services to a high

extent.
[**Saad-Alahmari:2011aa**] defines

$$Complexity(s) = \frac{Service\_Granularity}{number\_of\_services}$$

$$= \frac{\sum_{i=1}^{|O(s)|}(SG(i))^2}{|S|}$$

where,

- | S | is the number of services to realize the application
- SG(i) gives the granularity of ith service
  calculated as described in Section 3.3.4

#	Papers	Metrics Definition
1	[**Zhang:2009aa**]	RCS : complexity evaluated by the degree of coupling for a service and evaluated as the fraction of its coupling to the total number of services RIS : measured as the fraction of total dependency weight of consumers upon the service to the total number of services
2	[**Saad-Alahmari:2011aa**]	the complexity is calculated using the granunarity of service operations

Table 3.5: Complexity Metrics

The Table 3.5 provides the way to interprete complexity metrics. The complexity is highly dependent upong a) coupling and b) functionality granularity of the service.

3.3.6 Autonomy Metrics

[**Rostampour:2011aa**] defines

$$Self-Containment(SLC) = sum\_of\_CRUD\_coefficients\_for\_each\_Business\_entity$$

$$= \frac{1}{h_2 - l_2 + 1}\sum_{i=l_2}^{h_2}\sum_{sr \epsilon SR}(BE_{i,sr}XV_{sr})$$

$$Dependency(DEP) = dependency\_of\_service\_on\_other\_service\_business\_entities$$
$$= \frac{\sum_{j=L1_i}^{h1_i} \sum_{k=1}^{BE} V_{sr_{jk}} - \sum_{j=L1_i}^{h1_i} \sum_{k=L2_i}^{j2_i} V_{sr_{jk}}}{nc}$$

$$\text{autonomy} = \begin{cases} SLC - DEP & \text{if SLC > DEP} \\ 0 & \text{otherwise} \end{cases}$$

where,

- nc is the number of relations with other services
- $BE_{i,sr} = 1$ if the service performs action sr on the ith BE and sr represents any CRUD operation and V_{sr} is the coefficient depending upon the type of action
- $V_{sr_{jk}}$ is the corresponding value of the action in jkth element of CRUD matrix, it gives the weight of corresponding business capability affecting a business entity
- $l1_i, h1_i, l2_i, h2_i$ are bounding indices in CRUD matrix of ith service

#	Papers	Metrics Definition
1	**[Rostampour:2011aa]**	SLC : defined as the degree of control of a service upon its operations to act on its Business entities only DEP : given by the degree of coupling of the service with other services autonomy is given by the difference of SLC and DEP if SLC > DEP else it is taken as 0

Table 3.6: Autonomy Metrics

The Table 3.6 shows a way to interprete autonomy. It is calculated by the difference SLC-DEP when SLC is greater than DEP. In other cases it is taken as zero. So, autonomy increases as the operations of the services have full control upon its business entities but decreases if the service is dependent upon other services.

3.3.7 Reusability Metrics

[Sindhgatta:2015aa] defines

$$\begin{aligned} Reusability &= number\_of\_existing\_consumers \\ &= |S_{consumers}| \end{aligned}$$

[Bingu-Shim:2008aa] defines

$$Reusability = \frac{Cohesion - granularity + Consumability - coupling}{2}$$

#	Papers	Metrics Definition
1	**[Sindhgatta:2015aa]**	SRI defines reusability as the number of existing consumers of the service
2	**[Bingu-Shim:2008aa]**	evaluated from coupling, cohesion, granularity and consumability of a service where consumability is the chance of the service being discovered and depends upon the fraction of operations in the service

Table 3.7: Reusability Metrics

The table 3.7 shows the reusability metrics evaluation. Reusability depends upon a) coupling, b) cohesion and c) granularity. It decreases as coupling and granularity increases.

3.4 Basic Quality Metrics

The Section 3.3 presents various quality metrics to evaluate quality attributes based upon different research papers. The metrics are analyzed the metrics in order to derive them in simplest form possible. Eventually, the tables demonstrating the simplest definitions of the metrics, shows that the quality attributes are measured on the basis of some basic metrics. Based on the Section 3.3 and based on the research papers, this section lists basic metrics.

Considering the Table 3.8, it can be deduced that cohesion of a service increases with the number of operations using similar messages, using similar parameters and serving same consumer. The same process can be followed for other quality attributes.

#	Metrics	Coupling	Cohesion	Granularity	Complexity	Autonomy	Reusability
1	number of service operations invoked by the service	+			+	-	-
2	number of operation using similar messages		+				+
3	number of operation used by same consumer		+				+
4	number of operation using similar parameters		+				+
5	number of operation with similar scope or capability		+				+
6	scope of operation			+	+		-
7	number of operations			+	+		-
8	number of parameters in operation			+	+		-
9	type of parameters in operation			+	+		-
10	number and size of messages used by operations	+		+	+	-	-
11	type of messages used by operations		+				+
12	number of consumer services	+			+	-	+
13	number of producer services	+			+	-	-
14	type of operation and business entity invoked by the service	+			+	-	-
15	number of consumers accessing same operation		+				+
16	number of consumer with similar operation usage sequence		+				+
17	dependency degree or imporance of service operation to other service				+		
18	degree of control of operation to its business entities					+	

Table 3.8: Basic Quality Metrics

Additionally, the effect of the basic metrics upon various quality attributes are also evaluated and stated under each column such as 'coupling', 'cohesion' etc. Here, the meaning of the various symbols to demonstrate the affect are as listed below:

- + the basic metric affect the quality attribute proportionlly
- - the basic metric inversely affect the quality attribute
- there is no evidence found regarding the relationship from the papers

3.5 Principles based on Quality Attributes

The Section 3.1 has already mentioned that there are two distinct kind of quality attributes: external and internal. The external quality attributes cannot be evaluated during design however can be predicted using the internal quality attributes. This kind of relationship can be used to design the services with good quality. Moreover, it is important to identify how each internal attributes affect the external attributes. The understanding of such relationship will be helpful to determine the combined effect of the quality attributes and to identify the service with appropriate level of quality based on the requirement. The remaining part of the section lists some principles to show existing relationship among various quality attributes and the desired value of the quality attributes for the services.

1. When a service has large number of operations, it means it has large number of consumers. It will highly affect maintainability because a small change in the operations of the service will be propagated to large number of consumers. But again, if the service is too fine granular, there will be high network coupling. [**Feuerlicht:2007aa**; **Xiao-jun:2015aa**; **Bianco:2007aa**]

2. Low coupling improves understandability, reusability and scalability where as high coupling decreases maintainability [**Kazemi:2011aa**; **Erl:2005aa**; **Josuttis:2007aa**].

3. A good strategy for grouping operations to realize a service is to combine the ones those are used together. This indicates that most of the operations are used by same client. It highly improves maintainability by limiting the number of affected consumer in the event of change in the service. [**Xiao-jun:2015aa**]

4. A high cohesive quality attribute defines a good service. The service is easy to understand, test, change and is more stable. These properties highly support maintainability. enumerate[**np:2001aa**]

5. A service with operations those are cohesive and have low coupling, make the service reusable [**Washizakia:2003aa**; **Feuerlicht:2007aa**; **Ma:2009aa**].

6. Services must be selected in a way so that they focus on a single business functionality. This highly follows the concept of low coupling. [**Perepletchikov:2007aa**; **Sindhgatta:2015aa**]

7. Maintainability is divided into four distinct concepts: analyzability, changeability, stability and testability [**np:2001aa**]. A highly cohesive and low-coupled design should be easy to analyze and test. Moreover, such a system will be stable and easy to change. [**Perepletchikov:2007aa**]

8. The complexity of a service is determined by granularity. A coarse-grained service has higher complexity. However, as the size of the service decreases, the complexity of the over system governance also increases. [**Saad-Alahmari:2011aa**]

9. The complexity depends upon coupling. The complexity of a service is defined in terms of the number of dependencies of a service, number of operations as well

as number and size of messages used by the service operations. [**Saad-Alahmari:2011aa**; **Sindhgatta:2015aa**; **Lee:2001aa**]

10. The selection of an appropriate service has to deal with multi-objective optimization problem. The quality attributes are not independent in all aspects. Depending upon goals of the architecture, tradeoffs have to be made for mutually exclusive attributes. For example, when choosing between coarse-grained and fine-grained service, various factors such as governance, high network roundtrip etc. also should be considered. [**Jamshidi:2008aa**]

11. Business entity convergence of a service, which is the degree of control over specific business entities, is an important quality for the selection of service. For example: It is be better to create a single service to create and update an order. In that way change and control over the business entity is localized to a single service.[**Ma:2009aa**]

12. Increasing the granularity decreases cohesion but also decreases the amount of message flow across the network. This is because, as the boundary of the service increases, more communication is handled within the service boundary. However, this can be true again only if highly related operations are merged to build the service. This suggests for the optimum granularity by handling cohesion and coupling in an appropriate level.[**Ma:2009aa**; **Bianco:2007aa**]

13. As the scope of the functionality provided by the service increases, the reusability decreases [**Feuerlicht:2007aa**].

14. If the service has high interface granularity, the operations operates on coarse-grained messages. This can affect the service performance. On the other hand, if the service has too fine-grained interface, then there will be a lot of messages required for the same task, which then can again affect the overall performance. [**Bianco:2007aa**]

3.6 Relationship among Quality Attributes

In order to determine the appropriate level of quality for a service, it is also important to know the relationship between various quality attributes. This knowledge will be helpful to decide tradeoffs among the quality attributes in the situation when it is not possible to achieve best of all. Based on the Section 3.2 and Section 3.5, the identified relationship among the quality attributes are shown in the Table 3.9.

#	Quality Attributes	Coupling	Cohesion	Granularity
1	Coupling		-	+
2	Cohesion	-		
3	Granularity	+		
4	Complexity	+		+
5	Reusability	-	+	-
6	Autonomy	-		
7	Maintainability	-	+	-

Table 3.9: Relationship among quality attributes

Here, the meaning of the various symbols showing nature of relationship are as listed below:

- + the quality attribute on the column affects the quality attribute on the corresponding row positively
- - the quality attribute on the column affects the quality attribute on the corresponding row inversely
- there is no enough evidence found regarding the relationship from the papers or these are same attributes

Based on the Table 3.9, if granularity of a microservice increases then coupling of other microservices on it, will also increase. Similarly, relationship among other quality attributes can also be derived using the table. The idea regarding impact of various quality attributes amongst each other can be helpful to make decision regarding trade offs.

3.7 Conclusion

There is no doubt that granularity is an important aspect of a microservice however there are other different factors such as coupling, cohesion etc. which affect granularity as well as the overall quality of the service. Again, knowing the way to evaluate these qualities in terms of a quantitative figure can be helpful for easy decision regarding quality. However, most of the metrices are quite complex so the Section 3.4 compiled these complex mectrics in terms of simple metrices. The factors used to define these basic metrices are the ones which are easily accessible to developers and architects during desigin and development phases. So, these basic metrices can be an efficient

and easy way to determine quality of microservices.
Moreover, the external quality attributes such as reusability, scalability etc can be controlled by fixing internal quality attributes such as coupling, cohesion, autonomy etc. So, finding an easy way to evaluate and fix internal quality attributes can eventually make it easier to achieve microservices with satisfactory value of external quality attributes.
Nevertheless, the quality attributes are not mutually exclusive but are dependent on each other. The Table 3.9 shows basic relationship among them. The table can be helpful when needed to perform trade-offs among various quality attributes depending upon goals.

3.8 Problem Statement

Having collected important concepts regarding various quality attributes, which is one of the major drivers for defining architecture as mentioned in Section 1.4, the next important concept is to find the process of modeling microservices as stated in the Table 1.2. The quality attributes can be a major input for deciding the process of identifying microservices in a problem domain.

4 Modeling Microservices

In this chapter, various approaches to model microservices are identified from literature. Precisely, two distinct approaches are described in sections 4.2 and 4.3 respectively. For each process, a detailed set of steps are formulated. Finally, in order to make it clear, an example problem domain is taken, which is then broken down into microservices using each of these two distinct models.

4.1 Introduction

In the previous chapters, granularity and other quality attributes of a microservice are focussed. The qualitative as well as quantitative aspects of granularity are discussed. Additionally, various quality metrics were discussed to measure different quality attributes of a microservce. Moreover, a set of principle guidelines and basic metrics to determine the quality of the microservices, are also listed.
In addition to that, it is equally important to agree on the process to identify the microservices. In this chapter, various ways to model microservices are discussed. Basically, there are three high level approaches to identify services: Top-down, bottom-up and meet-in-the-middle.
The top-down approach defines the process on the basis of the business model. At first, various models are designed to capture the overall system. Using the overall design, services are identified upfront in the analysis phase.
The next approach is bottom-up, which relies on the existing architecture and existing application. The existing application and architecture are studied to identify the cohesion and consistency of the features provided by various components of the system. This information is used to aggregate the features and identify services in order to overcome the problems in the existing system.
Finally, the meet-in-the-middle approach is a hybrid approach of both top-down and bottom-up approaches. In this approach, the complete analysis of system is not performed upfront as the case of top-down approach. Where as, a set of priority areas are identified and used to analyse their business model resulting in a set of services. The services thus achieved are further analysed critically using bottom-up approach to identify problems. The problems are handled in next iterations where the same process as before is followed. The approach is continued for other areas of the system in the

order of their priority. **[Pierre-Reldin:2007aa][Arsanjani:2004aa]**
Considering the motive of the thesis, which is mainly concerned with green field development, top-down approaches is taken into consideration. In this chapter, two different strategies discussed in literature which belong to the category of top-down approache, are explained in detail.

4.2 Modeling Using Use cases

A use case is an efficient as well as a fancy way of capturing requirements. Another technique of eliciting requirement is by features specification. However, the feature specification technique limits itself to answer only "what the system is intended to do". On the other hand, use case goes further and specifies "what the system does for any specific kind of user". In this way, it gives a way to specify and futhermore validate the expectations as well as concerns of stakeholders at the very early phase of software development. Undoubtedly for the same reason, it is not just a tool for requirement specification but an important software engineering technique which guides the software engineering cycle. Using use cases, the software can be developed to focus on the concerns that are valueable to the stakeholders and test accordingly.**[Ng:2004aa]**
It can be valueable to see the ways it has been defined.

Definition 1: **[Jacobson:1987aa]**

"A use case is a sequence of actions performed by the system to yield an observable result of value to a particular user."

Definition 2: **[Rumbaugh:1999aa]**

"A use case is a description of a set of sequences of actions, including variants, that a system performs that yield an observable result of value to an actor."

4.2.1 Use cases Refactoring

According to **[Jacobson:1987aa]** it is not always intuitive to modularize use cases directly. As per the definitions provided in Section 4.2, a use case consists of a number of ordered functions together to accomplish a certain goal. **[Jacobson:1987aa]** defines these cohesive functionalites distributed across use cases as clusters. And the process of identifying the clusters scattered around the use cases, is the process of identifying services. According to research papers [**Ng:2004aa**] and [**Jacobson:2003aa**], use cases possibly consist of various cross-cutting concerns. The papers use the term 'use case module' to define the cohesive set of tasks. Figure 4.1 demonstrates the cross cutting functionalities needed by a use case in order to accomplish its goal. For example, the use case 'does A' has to perform separate functionalities on different domain entities X and Y. Similarly, the use case 'does B' needs to perform distinct functions on entities X, Y and Z.

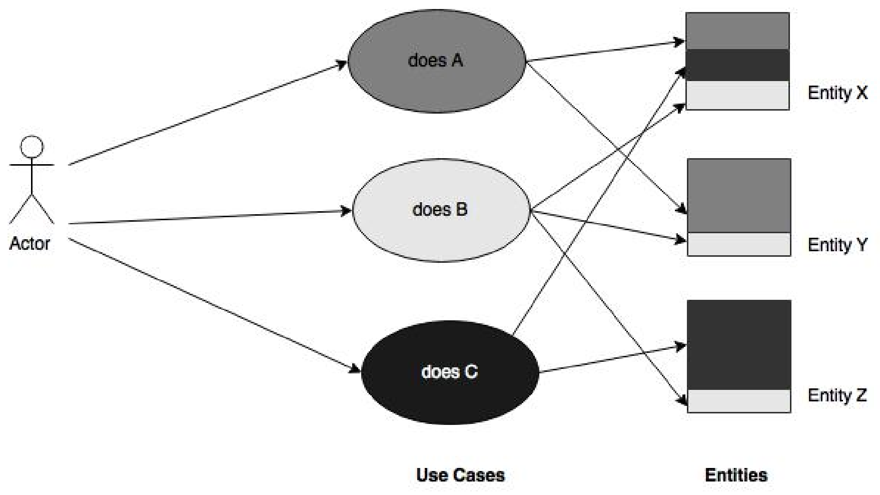


Figure 4.1: Use cases with cross-cutting concerns [**Ng:2004aa**]

This situation prompts for the analysis of the use cases for cross cutting tasks and refactoring of the use cases in order to segregate the cohesive functionalities. Refactoring helps to achieve the right level of abstraction and granularity by improving functionality cohesion as well as elimination of redundancy and finally promoting the reusablity. [**Doh:2007aa**]
Furthermore, the refactoring is assisted by various relationships in use case model to represent dependencies between use cases, which are a) include, b) generalization and c) extend. [**Ng:2004aa**]

4.2.2 Process for Use Cases Refactoring

In order to refactor the use cases and finally map use cases to the services, the papers [**Kim:2006aa**], [**Yun:2006aa**] and [**Doh:2007aa**] provide a comprehensive method. In this approach, use case models are first created and then refactored to create new set of use cases to accomplish high cohesion in functionality and loose coupling. This will ultimately create use cases supporting modularity. **[Fareghzadeh:2008aa]** There are three distinct steps for service identification using service refactoring.

1. **TaskTree Generation**
   In this step, the initial use case model created during problem domain analysis is used to create task trees for each distinct use case. The task trees provide sequence of individual tasks required to accomplish the goal of the respective use cases.

2. **Use Cases Refactoring**
   In this step, the task tree generated in the previous step is analysed. As already mentioned in the Section 4.2.1, the initial use cases consist of various cross cutting functionalities which acts on different business entities. In order to minimize that, refactoring is performed. The detail rules are provided in Section 4.2.3.

3. **Service Identification**
   The use cases achieved after refactoring have correct level of abstractions and granularity, representing cohesive business functionality. The resulting individual use cases promote reuse of common functionality [**Doh:2007aa**]. Furthermore, the approach considers the concerns of stakeholders in terms of cohesive business functionalities, to be represented by use cases [**Fareghzadeh:2008aa**]. Additionally, the process also clarifies the dependencies between various use cases. Thus, the final use cases obtained can be directly maped to individual services.

4.2.3 Rules for Use Cases Refactoring

The context is first defined before distinct rules are presented.

Context 1

If U represents use case model of the application, t_i represents any task of U and T being the set of tasks of U. Then, $\forall t_i \in T$, t_i exists in the post-refactoring model U'. The refactoring of a use case model preserves the set of tasks.

Context 2

A refactoring rule R is defined as a 3-tuple (a. Parameters, b. Preconditions and c. Postconditions). Parameters are the entities involved in the refactoring, precondition defines the condition which must be satisfied by the use case u in order for R to be applied in u. Postcondition defines the state of U after R is applied.
The various refactoring rules are as listed below.

4.2.3.1 Decomposition Refactoring

When the use case is complex and composed of various functionally independent tasks, the tasks can be taken out of the task tree for the use case and represented as a new use case. The Table 4.1 shows the decomposition rule in detail.

Parameters	u: a use case to be decomposed t: represents task tree of u t′: a subtask tree of t
Preconditions	t′ is functionally independent of u
Postconditions	1. new use case u′ containing task tree t′ is generated 2. a dependency is created between u and u′

Table 4.1: Decomposition Rule

4.2.3.2 Equivalence Refactoring

If the two use cases share their tasks in the task tree, we can conclude that they are equivalent and redundant in the use case model. The Table 4.2 shows the rules for the refactoring.

Parameters	u_1 , u_2: two distinct use cases
Preconditions	the task trees of u_1 and u_2 have same behavior
Postconditions	1. u_2 is replaced by u_1 2. all the relationship of u_2 are fulfilled by u_1 3. u_2 has no relationship with any other use cases

Table 4.2: Equivalence Rule

4.2.3.3 Composition Refactoring

When there are two or more small-grained use cases such that they have related tasks, the use cases can be represented by a composite single use case.

Parameters	u_1, u_2: the fine grained use cases t_1, t_2: the task trees of u_1 and u_2 respectively
Precondition	t_1 and t_2 are functionally related.
Postconditions	1. u_1 and u_2 are merged to a new unit use case u 2. u has new task tree given by $t_1 \cup t_2$ 3. the dependencies of u_1 and u_2 are handled by u 4. u_1 and u_2 are deleted along with their task trees t_1 and t_2

Table 4.3: Composition Rule

4.2.3.4 Generalization Refactoring

When multiple use cases share some volume of dependent set of tasks in their task trees, it can be implied that the common set of tasks can be represented by a new use case. The Table 4.4 provides the specific of the rule.

Parameters	u_1 , u_2: two distinct use cases t_1 , t_2: task trees of u_1 and u_2 respectively
Precondition	t_1 and t_2 share a common set of task $t = \{x_1, x_2...x_n\}$
Postconditions	1. a new use case u is created using task tree $t = \{x_1, x_2...x_n\}$ 2. relationship between u with u_1 and between u with u_2 is created 3. the task tree $t = \{x_1, x_2...x_n\}$ is removed from task tree of both u_1 and u_2 4. the common relationship of both u_1 and u_2 are handled by u and removed from them

Table 4.4: Generalization Rule

4.2.3.5 Merge Refactoring

When first use case is just specific for second use case and the second use case is only the consumer for it, the two use cases can be merged into one.

Parameters	u, u′: the use cases r defines the dependency of u with u′ t, t′: the task trees of u and u′ respectively
Precondition	there is no dependency of other use cases with u′ except u
Postconditions	1. u′ is merged to u 2. u has new task tree given by $t \cup t'$ 3. r is removed

Table 4.5: Merge Rule

4.2.3.6 Deletion Refactoring

When a use case is defined but no relationship can be agreed with other use cases or actors then it can be referred that the use case represent redundant set of tasks which has already been defined by other use cases.

Parameters	u: a distinct use case
Precondition	the use case u has no relationship with any other use cases and actors
Postconditions	the use case u is deleted

Table 4.6: Deletion Rule

4.2.4 Example Scenario

In this section, a case study is taken as an example, which is modeled using use case. Finally, the use case model is refactored using the rules listed by 4.2.3.

Case Study

The case study is about an international hotel named 'XYZ' which has branches in various locations. In each location, it offers rooms with varying facilities and prices. In order to make it easy for customers to find rooms irrespective of the location of the customers, it is planning to offer an online room booking application. Using this application, any registered customer can search for a room according to his/her requirement in any location. When customer is satisfied, the room can be booked online. The customer is sent notification by email after booking. During the starting days of the application, the payment is accepted in person only, after the customer gets into the respective branch. Finally, if the customer wants to cancel the room, he/she can do online anytime. The customer will also be notified by e-mail to confirm the cancelation of booking. It is an initial case study so only important features and conditions are considered here.

The remaining part of this section presents the steps to indentify the services using use cases refactoring following the process defined in the Section 4.2.2 and rules presented in the Section 4.2.3.

Step 1:

The initial analysis of the case study produces use case model as shown by figure 4.2.

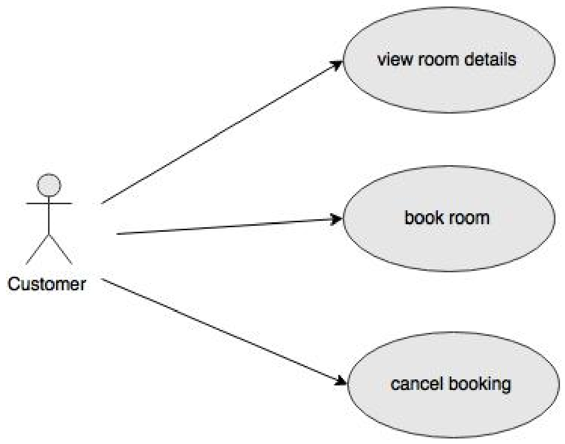


Figure 4.2: Initial Use Case Model for Online Room Booking Application

Step 2:

For each use case, task trees are generated listing the functionalities needed to accomplish the desired goal of the respective use cases.

Use Case	Task Tree
book room	customer enters login credentials system validates the credentials customer enters location and time details for booking system fetch the details of empty rooms according to the data provided system displays the details in muliple pages customer choose the room and submits for booking system generates a booking number System updates the room system sends notification to the customer regarding booking
cancel booking	customer enters login credentials system validates the credentials customer enters the booking number system validates the booking number customer cancels the booking system updates the room system sends notification to the customer regarding cancelation
view room details	customer enters login credentials system validate the credentials customer enter location and time system fetch the details of empty rooms according to the data provided system display the details in multiple pages

Table 4.7: Task Trees for Initial Use Cases

Step 3:
The initial task trees created for each use case at Step 2 is analysed. There are quite a few set of tasks which are functionally independent of the goal for respective use case. Similarly, there are few tasks which are common in multiple use cases. The following Table 4.8 lists those tasks from the tasks trees 4.7 which are either functionally independent from their corresponding use cases or common in multiple use cases.

Tasks Type	Tasks
Independent Tasks	system validates the credentials system fetch the details of empty rooms according to the data provided system generates a booking number system validates the booking number system updates the room system sends notification to the customer
Common Tasks	system validates the credentials system fetch the details of empty rooms according to the data provided system updates the room system sends notification to the customer

Table 4.8: Common and Independent Tasks

Now, for independent tasks the docomposition rule explained by 4.1 can be applied and for common tasks, the generalization rule given by 4.4 can be applied. The use cases obtained after applying these rules are shown by the Figure 4.3

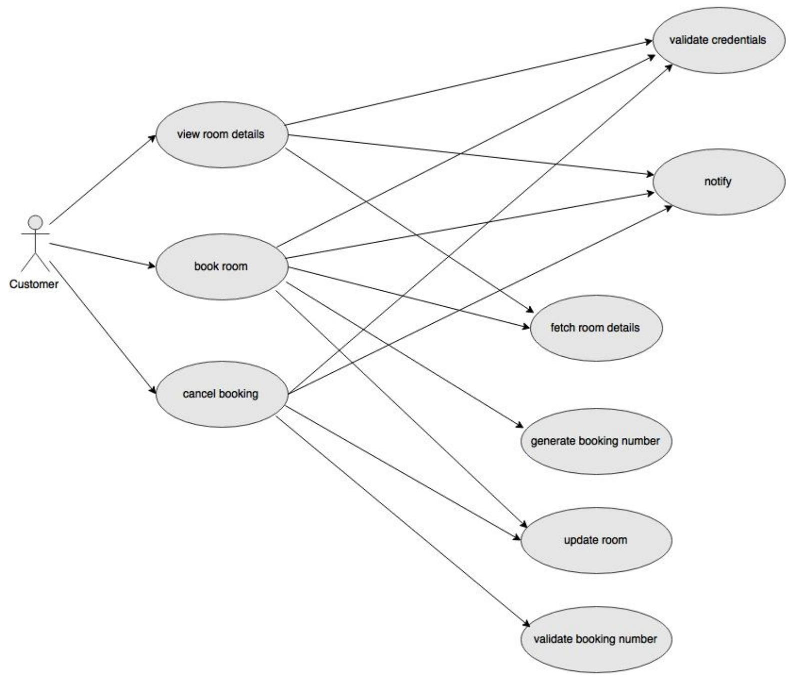


Figure 4.3: Use Case Model after applying Decomposition and Generalization rules

Step 4:

The use case model 4.3 obtained in Step 3 is analysed again for further refactoring. It can be seen that the use cases ’fetch room details’ and ’update room’ are fine grained and related to same business model ’Room’. Similarly, the use cases ’generate booking number’ and ’validate booking number’ are related to the business entity ’Booking Number’. The composition rule listed by 4.3 can be applied to these use cases, which will then result in use case model shown by the Figure 4.4

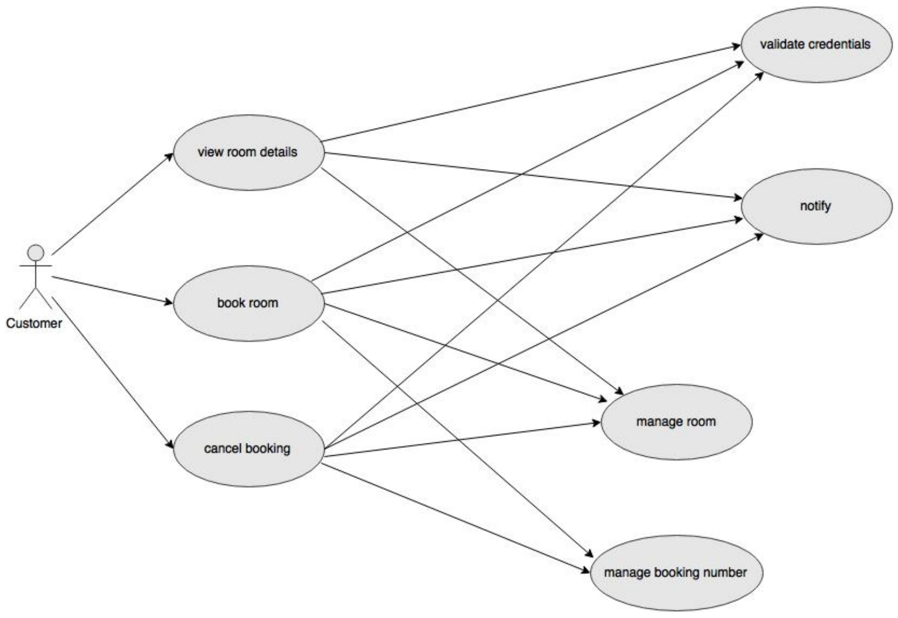


Figure 4.4: Use Case Model after applying Composition rule

Step 5:
Finally, the use cases obtained in step 4 is used to identify the service candidates. The final use cases obtained in Step 4 have appropriate level of granularity and cohesive functionalities. Most importantly, the refactoring has now separated the cross cutting concerns in terms of various reusable fine grained use cases as shown by the Figure 4.4. If we compare the the final use case model shown by the Figure 4.5 with respect to the Figure 4.1, then it can be implied that the functionalities operating on business entities as well as the business logic serving separate concerns are separated and represented by individual use cases. So, the each use case can be mapped to individual services. The services are as listed in the bottom of the Figure 4.5.

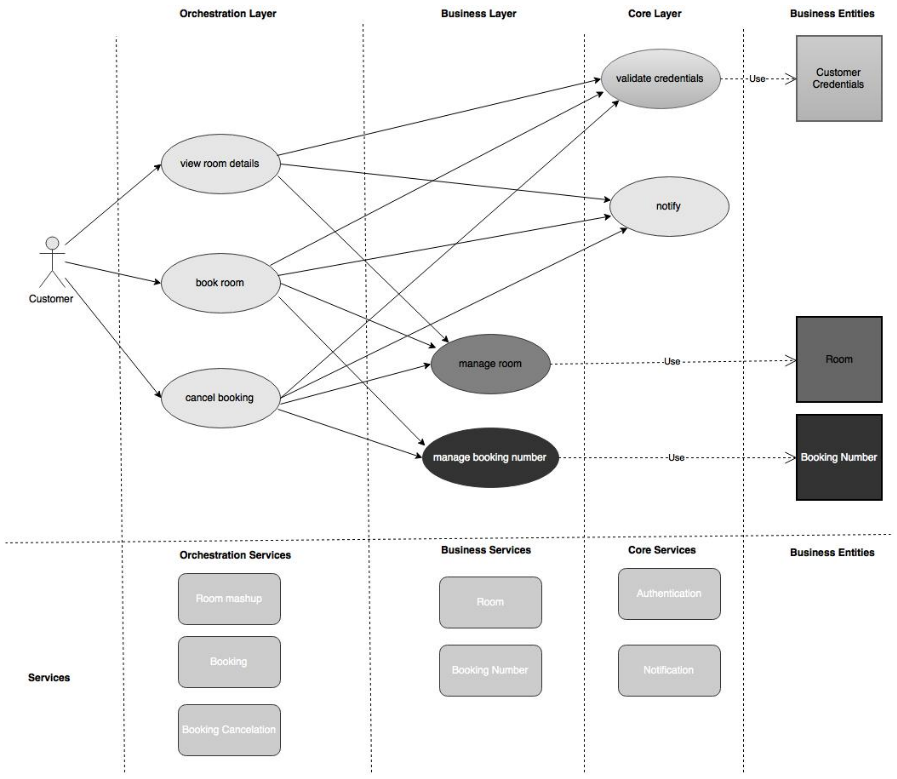


Figure 4.5: Use Case Model for identification of service candidates

Microservices Layers
The services obtained as shown in the Figure 4.5 from the refactoring process, creates various levels of abstractions. The levels of abstractions represents different level of functionality. The services at the bottom layer are core services which serve many higher level services. These services are not dependent on any other services. ′Notification′ and ′Authentication′ are core services obtained from refactoring. The layer above the core layer is business layer. The business services can either have entity services, which control some related business entities or can have task services which provides some specific business logic to higher level services. ′Room′ and ′Booking Number′ are the entity services obtained from refactoring. Finally, the top layer is mashup layer which contains high level business logic and collaborates with business services as well as core services. ′Room mashup′ , ′Booking′ and ′Booking Cancelation′ are orchestration layer services. The individual services as well as their respective layers are shown in the Figure 4.5. **[Fareghzadeh:2008aa][Emig:2015aa][Zimmermann:2005aa]**

4.3 Modeling using Domain Driven Design

Understanding the problem domain can be a very useful way to design software. The common way to understand the problem domain and communicate them to the team is by using various design models. The design models are the abstraction as well as representation of the problem space. However, when the abstractions are created, there are chances that important concepts or data are ignored or misread. This can highly impact the quality of software produced. The software thus developed may not reflect the real world situation or problem entirely.
Domain Driven Design provides a way to represent the real world problem space so that all the important concepts and data from real world remains intact in the model. The domain model thus captured respects the differences as well as the agreement in the concepts across various parts of the problem space. Moreover, it provides a way to divide the problem space into manageable independent partitions and makes it easy for developers as well as all stakeholders to focus on the area of concern and be agile. Finally, the domain models act as the understandable common view of the business for both the domain experts as well as the developers. The domain driven design makes sure that the software developed complies with business needs.**[Evans:2003aa][Vernon:2013aa]**

4.3.1 Process to implement Domain Driven Design

There are three basic parts to implement domain driven design:

1. Ubiquitous Language
2. Strategical Design
3. Tactical Design

As the focus is to describe the process of modeling microservices, only the first two parts are relevant and are focused in the later part of this chapter.

4.3.1.1 Ubiquitous Language

Ubiquitous Language is a common language agreed among domain experts and developers in a team. It is important to have a common understanding about the concepts of a problem doomain and ubiquitous language is the way to assure that. Domain Experts understand the domain in terms of their own jargon and concept. It is difficult for a developer to understand them. Usually, during desing and development, developers translates those jargons into the terms that they understand easily. However

along the way of translation, major domain concepts can get lost and the immediate value of the resulting solution might decrease tremendously. In order to prevent creation of such low valued solution, there should be an approach to define common vocabularies and concepts understood by all domain experts as well as the developers. These common vocabularies and concepts make the core of the ubiquitous language.[**Evans:2003aa**][**Vernon:2013aa**]
In addition to the common vocabularies and concepts, domain models provide backbone to create ubiquitous language. The domain models represent not only artifacts but also define functionalities, rules and strategies. It is a way to express the common understanding in a visual form, providing an easy tool to comprehend.[**Evans:2003aa**][**Fowler:2006aa**]
According to [**Fowler:2003aa**], domain model should not be confused with data model. The data model represents the business in datacentric view however each domain object contains data as well as logic closely related to the data contained. Domain models are conceptual models rather than software artifacts but can be effectively visualized using UML. [**Scott:2001aa**]

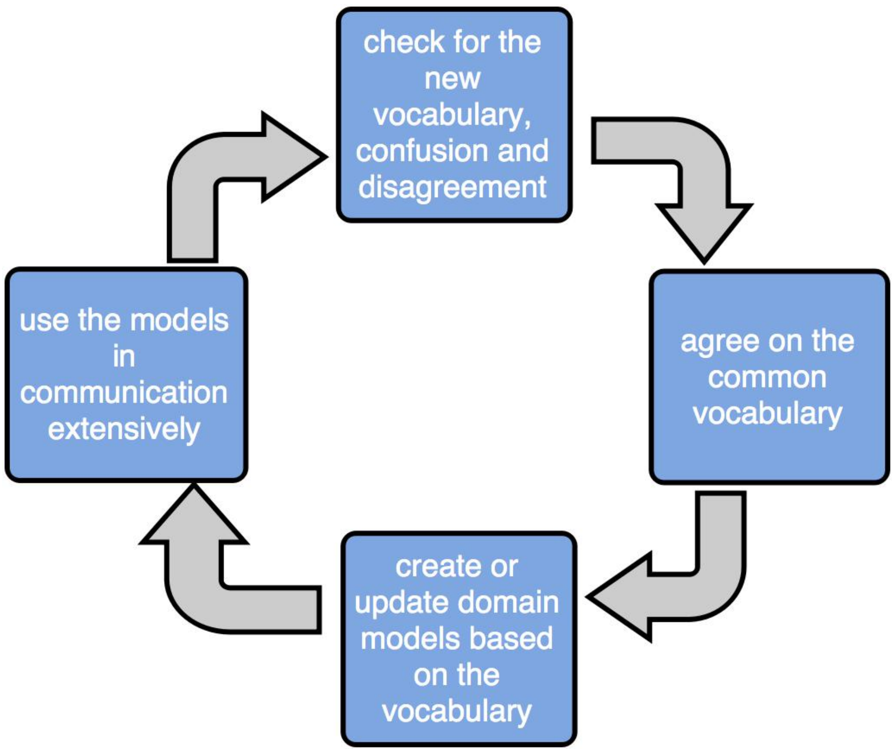

check for the new vocabulary, confusion and disagreement
agree on the common vocabulary
create or update domain models based on the vocabulary
use the models in communication extensively

Figure 4.6: Process to define Ubiquitous Language [**Evans:2003aa**]

The Figure 4.6 visualizes the process of discovering the ubiquitous language in any problem domain. It is not a time discrete phenomenon but a continous procedure with various phases occuring in a cycle throughout the development.
Firstly, on arrival of any new term, concept or any confusion, the contextual meaning of those terms are made clear before adding to the domain vocabulary. Secondly, the vocabularies and concepts are used to create various domain models following UML. The domain models and various vocabularies form the common laguage among domain experts and developers within the team. It is very important to use only domain vocabularies for communication and understanding which helps to reflect the hidden domain concepts in the implementation. Additionally, it also creates opportunities to find new vocabularies and refine existing ones in the event of confusion or disagreement. **[Evans:2003aa]**
Thus, the common vocabulary and domain models form the core of the ubiquitous language and the only way of coming up with the better ubiquitous language is by applying it in communication extensively. Ubiquitous language is not just a collection or documentation of terms but is the approach of communication within a team.

4.3.1.2 Strategical Design

When applying model-driven approach to an entire enterprise, the domain models get too large and complicated. It becomes difficult to analyze and understand all at once. Furthermore, it gets worse as the system gets bigger and complex. The strategical design provides a way to divide the entire domain models into small, manageable and inter-operable parts which can work together with low inter-dependency in order to provide the functionalities of the entire domain. The goal is to divide the system into modular parts which can be easily integrated. Additionally, the all-cohesive unified domain models of the entire enterprise cannot reflect differences in contextual vocabularies and concepts. **[Fowler:2014ab][Evans:2003aa][Vernon:2013aa]**
The strategical design specifies two major steps which are crucial in identifying microservices.

Step 1: Divide problem domain into subdomains

The domain represents the problem space solved by the software. The domain can be divided into various sub-domains based on the organizational structure of the enterprise and each sub-domain being responsible for certain area of the entire problem space. The Research Paper [**Engels:2015aa**] provides a comprehensive set of steps to identify sub-domains.

1. **Identify core business functionalities and map them into domain:**
   A core business functionality represents the direct business capability which holds high importance and should be provided by the enterprise in order for it to succeed. For example, for any general e-commerce enterprise, the core focus is to receive the high amount orders from the customers and to maintain the inventory to fulfil the orders. So, for those kind of e-commerce enterprises, 'Order Management' and 'Inventory Management' are some of the core domains.

2. **Identify generic and supporting subdomains:**
   The business capabilities which are viable for the success of business but do not represent the specialization of the enterprise falls into the supporting subdomains. The supporting subdomain does not require the enterprise to excel in these areas. Additionally, if the functionalities are not specific to the business but in a way support the business functionalities, then these are covered by generic subdomains. For example: for an e-commerce enterprise, 'Payment Management' is a supporting subdomain whereas 'Reporting' and 'Authentication' are generic subdomains.[**Vernon:2013aa**]

3. **Divide existing subdomains with multiple independent strategies:**
   The subdomains found from earlier steps are analyzed to check if there are mutually independent strategies to handle the same functionality. The subdomain can then be further divided into multiple subdomains along each dimension of strategies. For example: the 'Payment Managment' subdomain discovered in earlier step can have different way of handling online payment depending upon the provider such as bank or paypal etc. In that case 'Payment Management' can be further divided into 'Bank Payment Management' and 'Paypal Payment Management'.

Step 2: Indentify bounded contexts

As stated earlier of this section, the attempt to use a single set of domain models for the entire enterprise adds complexity for understanding and analysis. There is another important reason for not using the unified domain models, which is called unification of the domain models. Unification represents internal consistency of the

terms and concepts described by the domain models. It is not always possible to have consistent meaning of terms without any contradictory rules throughout the enterprise. The identification of the logically consistent and inconsistent domain model creates a boundary around domain model. This boundary will bind all the terms which share consistent meaning from those which are different so that there is clear understanding of the concept inside the boundary without any confusion. The shared context inside the boundary is defined as the bounded context.
Definition 1:
" A Bounded Context delimits the applicability of a particular model so that team members hava a clear and shared understanding of what has to be consistent and how it relates to other contexts. A bounded context has specific functionality and explicitly defines boundary around team organization, code bases and database schemas."**[Evans:2003aa]**
Definition 2:
" A Bounded Context is an explicit boundary within which a domain model exists. Inside the boundary all terms and phrases of the Ubiquitous Language have specific meaning, and the model reflects the Language with exactness." **[Vernon:2013aa]**
Definition 3: " A Bounded Context is a specific responsibility enforced by explicit boundaries." **[Mike:2012aa]**
Thus, the idea of bounded context provide the applicability of the ubiquitous language inside its boundary, performing a specific responsibility. Outside the bounded contexts, teams have different ubiquitous language with different terms, concepts, meanings and functionalities. In addition to identify independent and specific responsibilities inside a subdomain in order to discover bounded contexts, there are some general rules which can be considered as well.

1. **Assign individual bounded context to the subdomains identified in Step 1**
   Each subdomain has distinct functionalities and complete set of independent domain models to accomplish the functionalities. Thus, one to one mapping of subdomain and bounded context is possible ideally. However, in real cases, it may not be always possible which can be tackled using the rules discussed in further steps. For example: Order Management and Inventory Management are two distinct sub-domains and also two distinct bounded context each with own consistent ubiquitous language. **[Fowler:2014ab][Gorodinski:2013aa]**

2. **Identify polysemes**
   Polysemes are the terms which have different meaning in separate contexts. If the example of the generic subdomain 'Authentication' is taken, it has a model 'Account'. The same model 'Account' can refer to multiple concept such as

social account identified by e-mail or registration account identified by unique username. The handling of these two separate concepts is also different which clearly suggests two separate context for each within 'Authentication' subdomain. Additionally, there can be polysemes in two separate domains as well. For example: there is also 'Account' model to identify bank account in the subdomain 'Bank Payment Management'. The identification of polysemes is vital to create clear boundary around which the concept and handling of such polysemes are consistent. [**Fowler:2014ab**]

3. **Identify independent generic functionality supporting the subdomain**
   There can be a necessity of independent technical functionality required to accomplish the business functionality of the subdomain only. Since it is only required by the particular subdomain, the functionality cannot be nominated as a generic subdomain. In such a case, the independent technical functionality can be realized as a separate bounded context. For example, in the subdomain 'Inventory Management', the functionality of full-text search can be assigned a separate bounded context as it is independent.[**Gorodinski:2013aa**]

4.3.2 Microservices and Bounded Context

Analyzing various definitions of microservices provided in Section 1.2, a list of important features with respect to specific category such as granularity or quality was compiled in the table 1.2. In addition to the definitions of bounded context provided in Section 4.3.1.2, the following definition of Domain Driven Design can also be helpful to find analogy between a microservice and a bounded context.

Definition: Domain Driven Design

"Domain Driven Design is about explaining what your company does in isolated parts, in performing these specific parts and you can have a dedicated team to do this stuff, and you can have different tools for different teams." [**Riggins:2015aa**]

Using Table 1.2 and concepts extracted from various definitions of bounded context as well as Domain Driven Design, the Table 4.9 is generated.

The Table 4.9 lists various features expected by a microservices. Additionally, it also shows that the tabulated features are fulfilled by a bounded context. Thus, it can be implied that a microservice is conceptually analogous to a bounded context. The

#	Expected features of a microservice	Fulfilled by Bounded Context
1	Loosely coupled, related functions	✓
2	Developed and deployed independently	✓
3	Own database	✓
4	Different database technologies	✓
5	Build around Business Capabilities	✓
6	Different Programming Languages	✓

Table 4.9: Analogy of Microservice and Bounded Context

Section 4.3.1 provided the detailed process to discover bounded contexts within a large domain, which ultimately also provides guidelines to determine microservices using domain driven design approach.
Furthermore, there can be found enough evidence regarding the analogy as well as practical implementation of the concept. The table lists various articles which agree on the analogy and in most cases, have utilized the concepts to create microservices in real world.

#	Articles	Agreement on the Concept	Utilized Bounded Context to create Microservices
1	**[Mauro:2015aa]**	✓	✓
2	**[Hughson:2014aa]**	✓	
3	**[Fowler:2014aa]**	✓	
4	**[Sokhan:2015aa]**	✓	
5	**[Daya:2015aa]**	✓	✓
6	**[Riggins:2015aa]**	✓	
7	**[Beard:2015aa]**	✓	✓
8	**[Krylovskiy:2015aa]**	✓	✓
9	**[Viennot:2015aa]**	✓	✓
10	**[Balalaie:2015aa]**	✓	✓

Table 4.10: Application of Bounded Context to create Microservices

4.3.3 Example Scenario

In this section, a case study of a system is used to identify microservices following the rules listed in the Section 4.3.1.

Case Study

The case study is for the same hotel introduced in the Section 4.2.4. The hotel 'XYZ' is thinking of upgrading its current system and adding new business use cases in order to tune its competitive edge in the market.
Previously, the hotel only supported booking of the rooms, payment by cash, cancelation of booking and finally viewing of the room details.
The hotel has planned to provide various package offerings to a) general customers and b) specific group of customers satisfying certain profile. Firstly, a hotel staff creates a package with name, description, validity time period, applicable type of room, location of room and discount associated with the package. Furthermore, the package also contains certain constraints such as a) maximum number of offerings and b) maximum number of offerings per customer. The offering is represented by a coupon and has unique indentification code. Once a package is created, the package has to be activated by the hotel staff or the activation is triggered after certain preset time. Only the activated packages are visible to the customers. When the customers apply for the active packages, the customer profile is validated against the expected profile for the package and then a new coupon is sent to the customer after a successful validation.
The room booking workflow has no changes with respect to the old system where a customer can view list of rooms with various information and send room booking request to the system. The system then sends confirmation with unique booking code to the customer.
Finally, the payment can be performed by debit card or by paypal. During the payment, the customer can specify a valid coupon. In such case, the payment system redeems the coupon so that the respective discount represented by the coupon is deducted from the overall price.
Additionally, a detailed report is made available for the customers showing their various transactions with the hotel. Similarly, hotel staffs have other various types of reports such as profit statement and room booking status.
The steps listed in 4.3.1 can be used to design the system defined by the Case Study 4.3.3. **Step 1:**
With reference to the steps shown by the Figure 4.6, the first step is to find out new domain vocabularies, understand their meaning, agree on them and use them as the common vocabularies. The Table 4.11 lists some important domain terms taken from the case study 4.3.3 and clarifies the meaning for each. It is interesting to notice that the term 'Profile' has two different meaning for 'Package' context and 'Customer' context, so it is a polyseme.

Domain Terms	Definition
Package	a discount offering to certain group of customers
Profile	the expected characteristics of customers to be elligible to apply a package
Discount	the reduction value offered in the total price for hotel service
Constraint	defines the limitation of creating coupons for any package
Coupon	represents the authorized assignment of a valid package to a valid customer, which can be used later
Customer Profile	the current characteristics of a customer
Room Booking	assignment of a room to a customer for certain period
Booking Code	a unique code representing the valid room booking
Redeem Coupon	request to use the coupon whereby the associated discount value is deducted from the total price
Price	the total amount for the hotel service

Table 4.11: Domain Keywords

Step 2:
Next, the subdomains are identified following the step 1 4.3.1.2 of strategical design. The core domains, supporting domains and generic domains are identified and listed in the table.

Core Domains	1 Package	
	2 Booking	
	3 Checkout	
Supporting Domains	1 Payment	1.1 Paypal
		1.2 CardPayment
	2 User Management	2.1 Customer Management
		2.2 Staff Management
	3 Room	
Generic Domains	1 Reporting	
	2 Authentication	

Table 4.12: Subdomains

The supporting domain "User Management" has different stragegy for Customers and Staffs so can be further divided into "Customer Management" and "Staff Management" subdomains. Similarly, using the same rule, the "Payment" subdomain can also be further divided into "Paypal" and "CardPayment".
Step 3:
The next action is to identify bounded context following the process described in step 4.3.1.2. The process is implemented in each subdomains. In order to accomplish this, domain models can be constructed for each subdomain. Along the process, the various ambguities in the domain models and clear understanding of the concepts are made

visible. This helps to achieve consistent ubiquitious language with the unification of domain models as well as shared vocabulary inside each subdomain. The boundary around the uniform and consistent domain models represents the bounded contexts. Additionally, the rules listed in the Section 4.3.1.2 can be applied wherever appropriate to identify bounded contexts. The following Section will provide the analysis of major subdomains for identifying the bounded contexts within.

Customer Management

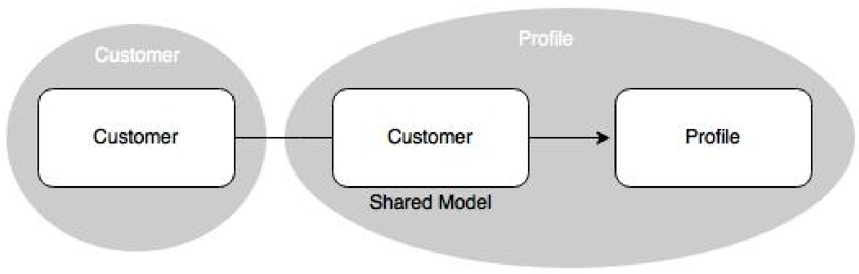


Figure 4.7: Domain Model for Customer Management

The management of customer includes two distinct tasks which are managing the customer itself and management of attributes of each customer representing profile of the customer at any point of time. Although, these two functionalities are related to the customer, they are actually loosely coupled and have different performance requirement. For eg: the change in the decision of adding attribute to a customer profile does not change the way customer is managed and also the rate at which customer profile is updated is definitely higher than the rate at which customer is managed. Additionally, not all the attributes from customer is relevant to customer profile which makes 'Customer' a polyseme and a shared model between the two bounded contexts 'Customer' and 'Profile'.

Booking

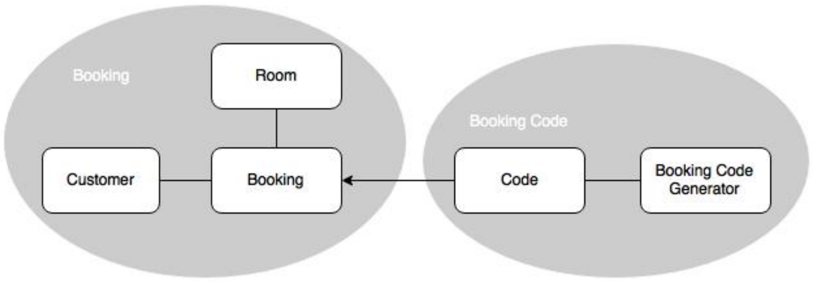


Figure 4.8: Domain Model for Booking

The Figure 4.8 shows the domain models of "Booking" subdomain. All the domain models and terms inside the subdomain are consistent and clear. However, the models ’Room’, ’Customer’, ’Booking’ and ’Code’ are closely related to booking functionality whereas the functionality of generating the booking code can be considered independent of booking. This gives two loosely coupled bounded contexts: "Booking" and "Booking Code". It is also interesting to notice that the models ’Customer’ and ’Room’ are polysemes for the bounded contexts a) ’Customer’ and b) ’Room’.

Checkout

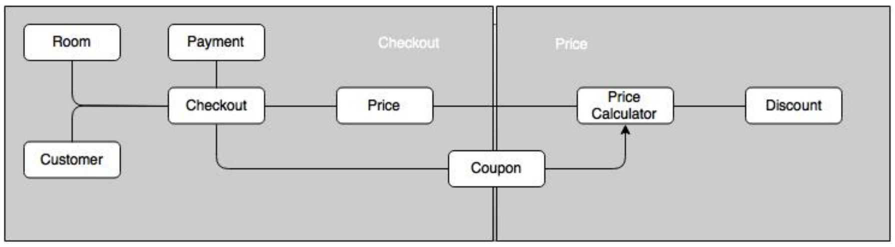


Figure 4.9: Domain Model for Checkout

The overall billing price calculation during checkout, which includes a) validation of coupon and b) redeem of the coupon over the total price, is independent of the orchestration logic during checkout. The subdomain can be realized into two different bounded contexts 'Checkout' and 'Price'. Additionally, the domain models 'Customer, 'Discount', 'Room', 'coupon' and 'price' are polysemes for these bounded context with respect to other bounded contexts.

Package Management

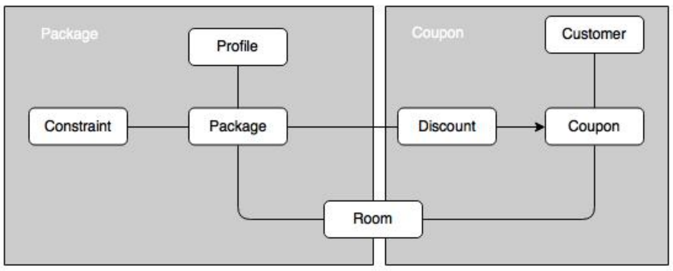


Figure 4.10: Domain Model for Package Management

The logic of package management can be considered independent of the way coupon is generated and validated. The packages are created by staff memeber whereas a coupon is created upon receiving a request from customer. Additionally, these are very specific and independent responsibilities, eligible of having own bounded contexts. The only data shared by these bounded contexts are about the information regarding eligible rooms and discount values. It should be made clear at this point that the 'Profile' domain model here is very different than 'Profile' domain model from 'Profile' boundend context of 'Customer Management' subdomain. In here, 'Profile' model defines the attributes of generic customer eligible for a specific package, however the 'Profile' domain model in 'Profile' bounded context provides the updated characteristics of any individual customer at the current moment. Similarly, the models such as 'Customer' and 'Room' are polysemes.
The same process can be applied to identify bounded contexts for rest of the subdomains.

4.4 Conclusion

The use cases refactoring process discussed in Section 4.2 highlights an important concept regarding use cases fulfilling multiple cross-cutting concerns and provides an approach of breaking down these concerns into individual use case modules, each focused on cohesive tasks. This kind of decomposition truely relates to the characteristics of microservices. On the other hand, domain driven design technique presented in Section 4.3 provides a different approach and focus on understanding the problem domain well using ubiquitous language, dividing the big problem domain into multiple manageable sub-domains and then finally using the concept of bounded context to determine individual microservices. The microservices thus obtained are autonomous and focus on single responsibility.
The use cases refactoring process utilizes use cases as its modeling tool, which is not only easy but also the familiar tool for architects and developers. This makes use cases refactoring a faster approach to decompose a problem domain. Whereas, domain driven design has ubiquitous language and bounded context at its core. Both of these concepts are complex and new to most architects as well as developers. At the same time, understanding problem domain correctly at the first attempt is difficult and the understanding process can take quite a few iterations to get the decomposition right. Thus, domain driven design is not the faster approach of modeling microservices.
Visualizing a problem domain in terms of diferent use cases seems like a natural way but dividing a big problem domain into smaller independent manageable sub-domains as stated by domain driven desing, can be a smarter way to decompose. So, in case of big and complex problem domain, it can be very difficult to visualize in terms of large number of big use cases trees and task trees. This may lead to microservices without appropriate level of granularity and quality attributes.
A good thing about use cases refactoring is that it provides very easy way to break down use cases along various level of functionalites or abstractions. However, it undermines another crucial concept which is data ownership and data decompostion. In use cases refactoring, an explicit assumption is made about an entity model being a single source of truth for whole domain, but it is rarely true. Using this approach does not necessarily produce autonomous services. However, the concept of ubiquitous lanugage and bounded context used by domain driven design, highly focus on different and unique local representation of same entity. This highly favors the autonomy of the resulting microservices. At the same time, the bounded contexts are responsible for small sets of cohesive functionality. This adheres to single responsibility principle. Thus, bounded context represents the appropriate size for a microservice.

4.5 Problem Statement

It is interesting to understand the various strategies defined in the literature to decompose a problem domain into microservices. The concept of microservices architecture is getting very popular recently among a lot of industries. So, the understanding of the process used in industries can also add bigger value to the thesis. For this reason, the next step is to study architecture at SAP Hybris as well as the process it followed for modeling microservices to build its commerce platform.

5 Architecture at SAP Hybris

In this chapter, a detail research of the architecture at SAP Hybris is conducted. Firstly, the Section 5.2 clarifies vision of YaaS. Secondly, the basic principles for modeling and developing microservices are listed in Section 5.3. In order to perceive a clear understanding about modeling and deployment process at SAP Hybris, Section 5.4 presents detailed interview conducted with various key personnels. The modeling process deduced from interviews is further clarified with an example in Section 5.6. Finally, the process of continuous deployment followed at SAP Hybris, which is one of the major processes when using microservices, is discussed in Section 5.7.

5.1 Overview

YaaS provides a variety of business services related to different domains such as Commerce, Marketing, Billing etc. Using these offered services, the developers can create their own business services focusing more on their business requirements.
The Figure 5.1 provides the overview of YaaS. YaaS provides various business processes as a service (bPaaS) essential to develop other applications and services, thus filling up the gap between SaaS and PaaS.

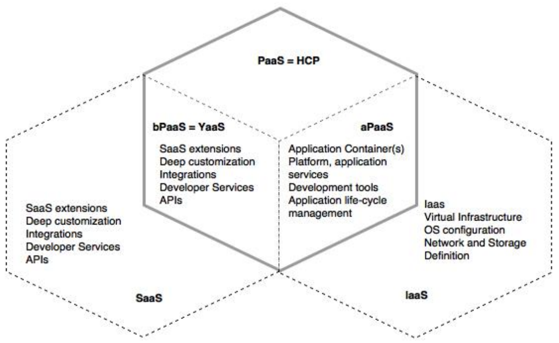


Figure 5.1: YaaS and HCP [**Hirsch:2015aa**]

5.2 Vision

The vision of YaaS can be made clear with the following statement.

"A cloud platform that allows everyone to easily develop, extend and sell services and applications."
[Stubbe:2015aa]

Additionally, the vision at SAP Hybris can be broadly categorized into following objectives.

1. **Cloud First**
   The different parts of the application need to be scaled independently.
2. **Autonomy**
   The development teams should be able to develop their modules independent of other teams and able to freely choose the technology that fits the job.
3. **Retain Speed**
   The new features can only be of high value to customers if could be able to be released as fast as possible.
4. **Community**
   It should be possible for the components to be shared across internal and external developers.

The keywords from various definitions of microservices discussed in Section 1.2 as well as the characteristics of microservices signifies clearly that microservices architecture can be a good fit for YaaS architecture.

5.3 SAP Hybris Architecture Principles

The Agile Manifesto [**Beck:2011aa**] provides various principles to develop a software in a better way by minimizing the time for delivering new feature. It focuses on fast response to the requirement changes by continuous delivery of software artifacts with close collaboration between customer and self-organizing teams.
The Reactive Manifesto [**Boner:2014aa**] lists various qualities of a reactive system which includes responsiveness (acceptable consistent response time, quick detection and solution of problem), resilience (responsive during the event of failure) , elasticity

(responsive during varying amount of workload) and asynchronous message passing. Additionally, loose coupling is also highly focused.
Furthermore, the twelve factors from Heroku [**Wiggins:2012aa**] provides a methodology for minimizing time and cost to develop software applications as services. It emphasizes on scalability of applications, explicit declaration as well as isolation of dependencies among the components, multiple continuous deployments from a single version controlled codebase with separate pipelines for build, release and run.
Finally, the microservices architectural approach provides techiques of developing an application as a collection of autonomous small sized services focused on single responsibility. [Section 1.2] It focuses on independent deployment capability of individual microservice and suggests to use lightweight mechanisms such as http for communication among services. The architecture offers various advantages, few of them being a) independent scalability of each microservice, b) resilience achieved by isolating failure in a component and c) technology heterogeneity among various development teams. [**Newman:2015aa**]
Following the principles mentioned above, a list of principles are compiled to be used as guidelines for creating microservices at SAP Hybris. [**Stubbe:2015aa**]

The y-Factors

1. **Self-Sufficient Teams**
   The teams have freedom for any decision related to the design and development of their components. This freedom is balanced by the responsibility for the team to handle the complete lifecycle of their components including deploying, running, enhancing performance in production and troubleshooting in case of any problems.

2. **Open Technology landscape**
   The teams are independent to choose any technology that they believe fits the requirement. They are completely responsible for the quality of their product. This gives the teams a feeling of ownership and satisfaction for their products.

3. **Release early, release often**
   The agile manifesto and twelve factors from Heroku also focus on continuous delivery of product to the clients. This decreases time of feedback and also results in high customer satisfaction. The teams are responsible to create build and delivery pipelines for all available environments.

4. **Responsibility**
The teams are the only responsible groups to work directly with the customers on the behalf of their products. They need to work on the feedbacks provided by the customers. It increases the quality of the products and relationship with customers. All the responsibilities including scaling, maintaining, supporting and improving products are handled by the respective teams.

5. **APIs first**
API is a contract between the service and the consumers. The decision regarding design and development of API is very crucial. API can be one of the greatest assets if good or else can be a huge liability. [**Bloch:2016aa**] The articles [**Bloch:2016aa**] and [**Blanchette:2008aa**] list various characteristics of good API including a) simplicity, b) extensibility, c) maintainability, d) completeness, e) small and f) focussing on single functionality. Furthermore, a good approach to develop API is to first design iteratively before implementation in order to understand the requirement clearly. Another important aspect of a good API is a complete and updated documentation.

6. **Predictable and easy-to-use UI**
The user interfaces should be simple, consistent across the system and also comply with various user friendly patterns.[**Sollenberger:2012aa**] The articles [**Martin:2013aa**] and [**Porter:2016aa**] specify additional principles to be considered when designing user interfaces. A few of them are a) providing clarity with regard to purpose, b) smart organization and c) respecting the expectation as well as requirements of customers.

7. **Small and Simple Services** A service should be small and focused on cohesive functionalities. The concept closely relates to the single responsibility principle. [**Martin:2016aa**] A good approach is to explicitly create boundaries around business capabilities.[**Newman:2015aa**]

8. **Scalability of technology**
The choice of technologies should be cloud friendly such that the products can scale cost-efficiently and without any delay. It is influenced by the elasticity principle provided by the reactive manifesto.[**Boner:2014aa**]

9. **Design for failure** The service should be responsive at the time of failure. It can be possible by the containment and the isolation of failure within each component. Similarly, the recovery should be handled gracefully without affecting the overall availability of the entire system. [**Boner:2014aa**]

10. **Independent Services**
    The services should be autonomous. Each service should be able to be deployed independently. The services should be loosely coupled, they could be changed independently of eachother. The concept is highly enforced by exposing functionalities via APIs and using lightweight network calls as only way of communication among services. [**Newman:2015aa**]

11. **Understand Your System**
    It is crucial to have a good understanding of problem domain in order to create a good design. The concept of domain driven design strongly supports this approach and motivates to indentify individual autonomous components, their boundaries and the communication patterns among them. [**Newman:2015aa**] Furthermore, it is also important to understand the expectations of the consumers regarding performance and then to realize them accordingly. However, this can only be possible by installing necessary operational capabilities such as a) continuous delivery, b) monitoring, c) scaling and d) resilience.

5.4 Modeling Microservices at SAP Hybris

With the intension to anticipate the overall belief, culture and practice followed by SAP Hybris for developing YaaS, a number of interviews are conducted with a subset of key personnels who are directly involved in YaaS at different roles. The list of interviewees with their corresponsing roles is shown in the Table 5.1. In order to preserve identity, the real names are replaced with forged ones.

Names	Roles
John Doe	Product Manager
Ivan Horvat	Senior Developer
Jane Doe	Product Manager
Mario Rossi	Product Manager
Nanashi No Gombe	Product Manager
Hans Meier	Architect
Otto Normalverbraucher	Product Manager
Jan Kowalski	Senior Developer

Table 5.1: Interviewee List

5.4.1 Hypothesis

During the process of solving the concerned research questions, various topics of high value have been discovered. The topics are:

1. Granularity
2. Quality Attributes
3. Process to design microservices

Similarly, based on research findings, a list of hypothesis has been built for each topic listed in the list above.

Hypothesis 1:

The correct size of microservices is determined by:

1. **Single Responsibility Principle**
2. **Autonomy, and**
3. **Infrastructure Capability**

According S.Newman, a good microservice should be small and should focus on accomplishing one thing well. Additionally, it could be independently deployed and updated. This strongly suggests the requirement of SRP and autonomy respectively. [**Newman:2015aa**] Futhermore, M. Stine also agrees that the size of a microservice should be determined by single responsibility principle [**Stine:2014aa**].
Also, the principle mentioned in 2.3.4 indicates that the advancement in the current technology and culture of the organization influences the size of the microservices. [**Pierre-Reldin:2007aa**]

Hypothesis 2:

Domain driven design is the optimum approach to design microservices.

The concept of domain driven design to design the microservices is suggested by S. Newman and M. Fowler [**Newman:2015aa**] [**Fowler:2014aa**]. Similary, there can be found evidence of domain driven design being used in various projects to design microservices. The list of projects is shown in the Table 4.10.

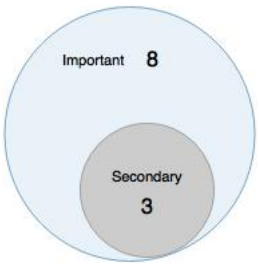
5.4.2 Interview Compilation

For each of the topics listed in Section 5.4.1, a list of questionaires is prepared. The questions are asked to the interviewees. In the remaining part of this section, the responses from the interviews on the questionaires are compiled.

1. Granularity

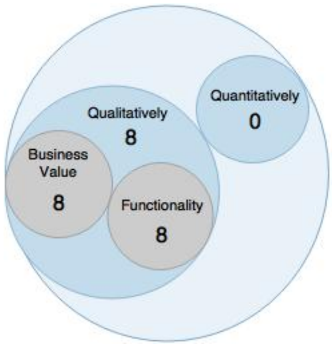
Question 1.1

"Do you consider the size of microservices when desigining microservices architecture?"



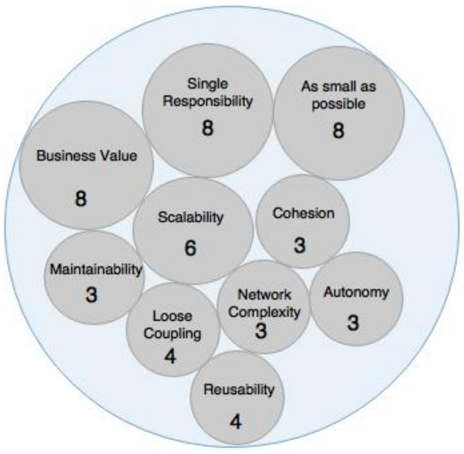
Question 1.2

"How do you measure size of a microservice?"



Question 1.3

"What factors do you consider when you decide the size of a microservice?"



Question 1.4

"Suppose that you do not have the agile culture like continuous integration and delivery automation and also cloud infrastructure. How will it affect your decision regarding microservices and their size?"

Response 1.4	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
No microservices at all, start with Monolith, extract one microservice at a time as automation matures	1	1	0	1	1	1	1	1	**7**
Start with bigger sized microservices with high business value, low coupling and single Responsibility	0	0	1	0	1	0	1	0	**3**

Question 1.5

"Are you facing any complexities because of microservices architecture and the size you have chosen?"

Response 1.5	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
communication overhead in network	0	1	0	0	0	0	1	0	**2**
complexity in failure handling	0	1	0	0	0	0	0	0	**1**
Monitoring is difficult	0	1	0	0	0	0	1	0	**1**
Operational support is costly	0	1	1	0	0	0	0	0	**2**
difficult to trace a request	0	1	1	0	0	0	0	0	**2**
updates can be difficult due to dependencies among microservices	0	0	1	0	0	0	0	0	**1**

2. Quality Attributes

Question 2.1

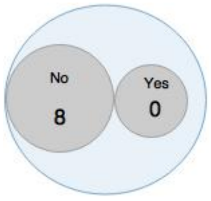
"Do you consider any quality attributes when selecting microservices?"

Response 2.1	John	Ivan	Jane	Mario	Nanashi	Hans	Otto	Jan	**Score**
Loose Coupling	1	1	1	1	1	1	1	1	**8**
Cohesion	1	1	1	1	1	1	0	1	**7**
Autonomy	1	1	0	0	0	1	1	0	**4**
Scalability	1	0	0	1	1	0	1	1	**5**
Complexity	1	0	0	0	1	0	1	0	**3**
Reusability	0	1	0	0	0	0	1	0	**2**

Table 5.2: Response 2.1

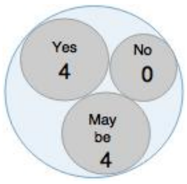
Question 2.2

"Do you consider any metrics for evaluating the quality attributes?"



Question 2.3

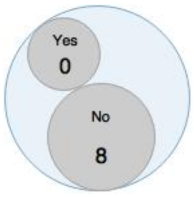
"Do you think the table of basic metrics can be helpful?"



3. Process to design microservices

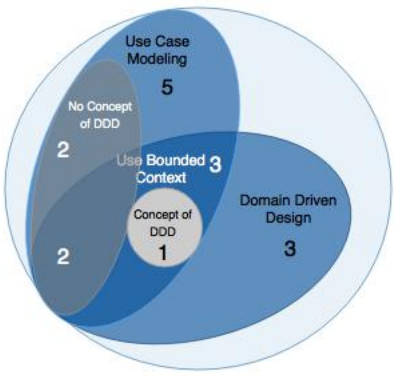
Question 3.1

"Do you follow specific set of consistent procedures across the teams to discover microservices?"



Question 3.2

"Can you define the process you follow to come up with microservices?"



5.4.3 Interview Reflection on Hypothesis

The data gathered from the interviews which are compiled in the Section 5.4.2 can be a good source to analyze the hypothesis listed on the Section 5.4.1.

Hypothesis 1

The correct size of microservices is determined by:

1. **Single Responsibility Principle**
2. **Autonomy, and**
3. **Infrastructure Capability**

The response from Question 1.3 [5.4.2] has helped to point out some important constraints which play significant role in industries while choosing appropriate size of the microservices. It can be deduced that the answer closely relate to the Hypothesis 5.4.1. The Figure 5.2 attempts to clarify the relationship among the terms with the hypothesis.

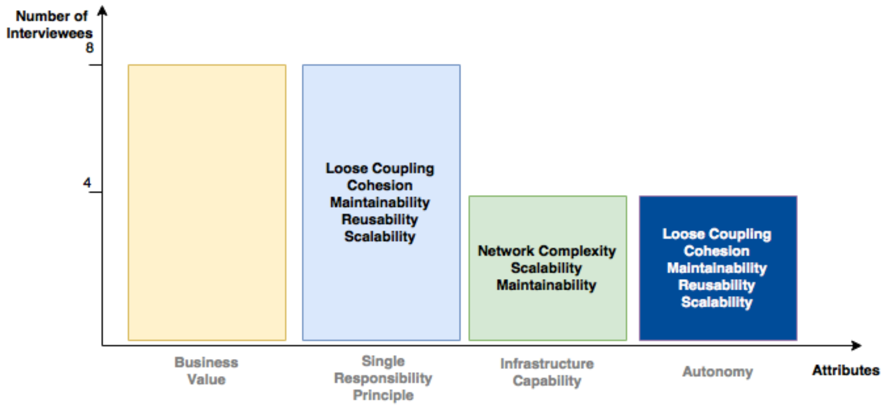


Figure 5.2: Attributes grouping from interview response

The interview unfolds various quality attributes affecting the correct size of a microservice. Moreover, these quality attributes can be arranged into three major groups, which are:

1. Single Responsibility Principle
2. Automony
3. Infrastructure Capability

The response from the Question 1.4 [5.4.2] highly suggests the significance of proper infrastructure to determine the size of the microservices and also regarding the choice of using microservices architecture. This highly moves in the direction to support the hypothesis.
Finally, there appears one additional constraint to affect the size of the microservices, which is not covered by hypothesis but mentioned by all interviewees, the **business value** provided by the microservices.

Hypothesis 2

Domain driven design is the optimum approach to design microservices.
From the response to Question 3.1 [5.4.2], it can be implied that the organization has no consistent set of specific guidelines which are agreed and pratised across all teams. However, the various reactions to the Question 3.2 [5.4.2] suggest that the process fall under two classes. The representation of response from 5.4.2 is repeated here again for convenience 5.3.

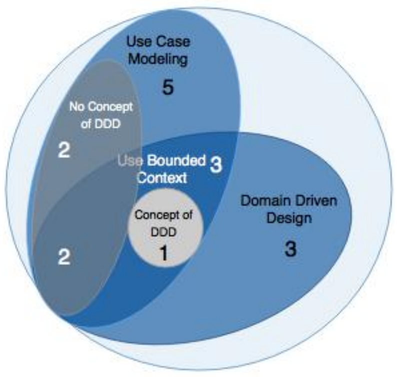


Figure 5.3: Process variations to design microservices

According to one category of reaction, some teams functionally divide a usecase into small functions based upon various factors such as single responsiblity and the requirements of performance. This strongly resembles to the use cases modeling technique as described in the Chapter 4.2.
Moreover, within the same group, a partition of teams are also using the concept of bounded context to ensure autonomy, which also suggests towards the direction of domain driven design approach as mentioned in the Chapter 4.3. It is interesting to find out that some of the teams are using the concept of bounded context implicitly even without its knowledge. The rest of the teams who are not using bounded context, have no idea about the concept. It can be implied that they may have different view once they get familiar with the concept.
Finally, the rest of the interviewees agree on domain driven design approach to be the process of designing microservices. So, the response is of mixed nature, both use cases modeling approach and domain driven design approach are used in desigining microservices. However, it should also be stated that domain driven design are used on its own or along with usecase modeling to design the autonomous microservices.

5.5 Modeling Approach at SAP Hybris derived by interview compilation

The responses from the interviews as compiled in Section 5.4.2 are analyzed for each topic listed in 5.4.1 separately.

Granularity

Granularity of a microservice is considered as an important aspect when implementing microservices architecture. The first impression in major cases is to make the size of microservice as small as possible. However there are also various attributes and concepts which affects the decision regarding the size of the microservices. The major aspects are single responsibility principle, autonomy and infrastructure capability.
The single responsibility influences to make the size of a microservice as small as possible so that the microservice has minimum cohesive functionality and less number of consumers. On the other hand, in order to make the microservice autonomous, the microservice should have full control over its resources and have less dependencies upon other microservices for accomplishing its core logic. This suggests that the size of the microservice should be big enough to cover the transactional boundary required for its core logic and have full control on its business resources.
Moreover, the small the size of the service is, the number of required microservices in-

creases for accomplishing the functionalities of the application. This increases network complexity. Additionally, the logical complexity as well as maintainability also increase. Undoubtedly, the independent scalability of individual microservice also increases. However, this increases the operational complexity for maintaining and deploying the microservices due to the increased number of microservices.
As shown in Figure 5.2, various factors affect the size of microservices. The research findings as well as interview response leads to the idea that the question regarding "size" of a microservices has no straight answer. The answer itself is a multiple objective optimization problem and depends upon a) the level of abstraction of the problem domain, b) self-governance requirement, c) self-containment requirement and d) the honest capability of the organization to handle the operational complexity.
Finally, there is an additional deciding factor called "Business Value". The chosen size of the microservice should provide business value and competitive advantage to the organization and closely lean towards the organizational goals. A decision for a group of cohesive functionalities to be assigned as a single microservice means that a dedicated amount of resources in terms of development, scaling, maintainance, deployment and cloud infrastructure have to be assigned. A rational decision can be to evaluate the expected business value of the microservice against the expected cost value required by it.

Quality Attributes

The teams do not follow any quantitative metrics for measuring various quality attributes. However, they agree on a common list of quality attributes to be considered as shown in the Table 5.2 . According to the reactions from interviews, the basic metrics to evaluate the quality attributes visualized in the Table 3.8 can be helpful to model microservices architecture in an efficient way.

Process to design microservices

There is no single consistent process followed across all the teams at SAP Hybris. Each team has its own steps to come up with microservices. However, the steps and decision are highly influenced by a set of YaaS standard principles 5.3 and vision of SAP Hybris 5.2. The reactions from the interviewees are already visualized in 5.3.
The process followed by a portion of teams closely relate to usecase modeling approach, where a usecase is divided into several smaller abstractions guided by a) cohesion, b) reusability and c) single responsibility principle. Within this group, a major number of teams also use the concept of bounded context in order to realize autonomous boundary around the microservices. Finally, the remaining portion of the teams closely follow

domain driven design approach, where a problem domain is divided into sub-domains. In each sub-domain, various bounded contexts are discovered, which is then mapped into microservices.
The domain driven design approach can be considerd as the most suitable technique to design microservices. Following this approach, not only the problem domain is functionally divided into cohesive group of functionalities but also leads to a natural boundary around the cohesive functionalities closely relating to the real world scenario of the organization. Moreover, following the domain driven design approach, similarities as well as differences in concepts are acknowledged. Finally, the ownership of business resources and core logic are well preserved.

5.6 Case Study

YaaS provides a cloud platform where the customers can buy and sell applications and services related to the commerce domain. In order to accomplish this, YaaS offers various basic business and core functionalities as microservices. The customers can either use or extend these services and create new services and applications focusing on their individual requirements and goals.
The following part of this section discuss the steps used to identify microservices under commerce domain in YaaS.
Step 1:
The problem domain is studied thoroughly to identify core domains, supporting domains and generic domains.

#	Sub-domain	Type	Description
1	Checkout	Core	handles overall process from cart creation to selling of cart items.
2	Product	Supporting	deals with product inventory
3	Customer	Supporting	deals with customer inventory
4	Coupon	Supporting	deals with coupon management
5	Order	Supporting	handles orders management
6	Site	Supporting	handles site configuration
7	Tax	Supporting	handles tax configuration and tax calculation
8	Payment	Supporting	handles payment for orders
9	Email	Generic	provides REST api for sending emails
10	Pubsub	Generic	provides asynchronous event based notification service
11	Account	Generic	handles users and roles management
12	OAuth2	Generic	provides authentication for clients to access resource of resource-owner
13	Schema	Generic	storing schema of documents
14	Document	Generic	provides storage apis for storing and accessing documents

Table 5.3: Sub-domains in YaaS

Step 2:
The major functionality of the commerce platform is to sell products, which makes 'Checkout' the core sub-domain. The sub-domain is also responsible for a bunch of other independent functionalities such as cart management and cart calculation. These individual independent functionalities can be rightfully represented by bounded contexts as shown in the Table 5.4. The bounded contexts are realized using microservices. The other major reason for them to be made microservices, in addition to single independent responsibility, is different scalability requirements for each functionality.

#	Bounded Context	Description
1	Checkout	handles the orchestration logic for checkout
2	Cart	handles cart inventory
3	Cart Calculation	calculates net value of cart considering various constraints such as tax, discount etc.

Table 5.4: Bounded Contexts in Checkout

Step 3:
The supporting sub-domain "Product" deals with various functionalities related to "Product".

#	Bounded Context	Description
1	Category	manage category of products
2	Product	manage product inventory
3	Price	manage price for products

Table 5.5: Functional Bounded Contexts in Product

The bounded contexts listed in Table 5.5 are based upon the independent responsibility and different performance requirement. Although, each functionality deals with business entity "Product", the conceptual meaning and scope of "Product" is different in each bounded context. "Product" is a polyseme.
Again, there are two additional functionalities identified. The first one is a technical functionality to provide indexing of products so that searching of products can be can be efficient. The functionality is generic, technical and independent. It has a different bounded context and can be realized using different microservice. Furthermore, there is a requirement for mashing up information from microservices such as a) product, b) price and c) category in order to limit the network calls from client to each individual service and to improve performance. This is realized using a separate mashup service. Also, these functionalities have different performance requirement. The identified bounded contexts are shown in Table 5.6.

#	Bounded Context	Description
1	Algolia Search	product indexing to improve search performance
2	Product Detail	provide mashup of product, price and category

Table 5.6: Supporting and Generic Bounded Contexts in Product

It is important to notice that the decision to realize the functionalities as microservices considered various technical aspects as performance, autonomy, etc. and also business value. The individual microservices such as "Product", "Category", "Checkout" have high business value to YaaS because they are good candidate services required by the partners and customers of YaaS.
The various microservices for the sub-domains "Checkout" and "Product", identified following the steps listed above, forms layers of the microservices based on their level of abstraction and reusability. The layers are shown in Figure 5.4. The core microservices are technical generic microservices used by business microservices and form the bottom layer. On the top of generic microservices, are the domain specific business microservices focused on domain specific business capabilities. The individual business functionalites provided by business microservices are then used by Mashup layer to create high level microservices by orchestrating them.

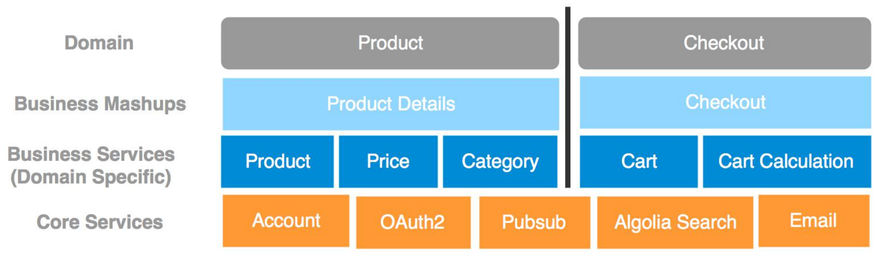


Figure 5.4: Microservices Layers

Step 4:
The same process from steps 1-3 are applied to remaining sub-domains listed in Table 5.3. The Table 5.7 shows a list of services discovered in each sub-domains within the commerce domain. For example, 'Tax' and 'Tax Avalara' are microservices under 'Tax' sub-domain. Similarly, 'Order' and 'Order Details' are microservices related to 'Order' subdomain.

Sub-domain	Microservice	Description
Checkout	checkout cart Cart Calculation	handles the orchestration logic for checkout handles cart inventory calculates net value of cart considering various constraints such as tax, discount etc.
Product	Product Product Details Category Price	manages product inventory provides mashup of product, price and category manages category of products manage price for products
Order	Order Order Details	handles order management provide mashup of orders and products
Site	Site Shipping	handles site configuration handles shipping configuration for site
Tax	Tax Tax Avalara	handles tax configuration handles tax calculation using Avalara
Customer	Customer	deals with customer inventory
Coupon	Coupon	deals with coupon management
Payment	Payment Stripe	handles payment for oders using Stripe

Table 5.7: Services in YaaS Commerce domain

Including all the microservices in core, supporting and generic sub-domains for only commerce domain, the resulting architecture is as shown in Figure 5.5. At the bottom are various generic microservices such as Pubsub, Document etc. They are utilized by various business microservices such as Customer, Coupon etc. On the top are various mashup microservices such as Checkout, Order-Details and they utilize various business microservices and core microservices underneath.

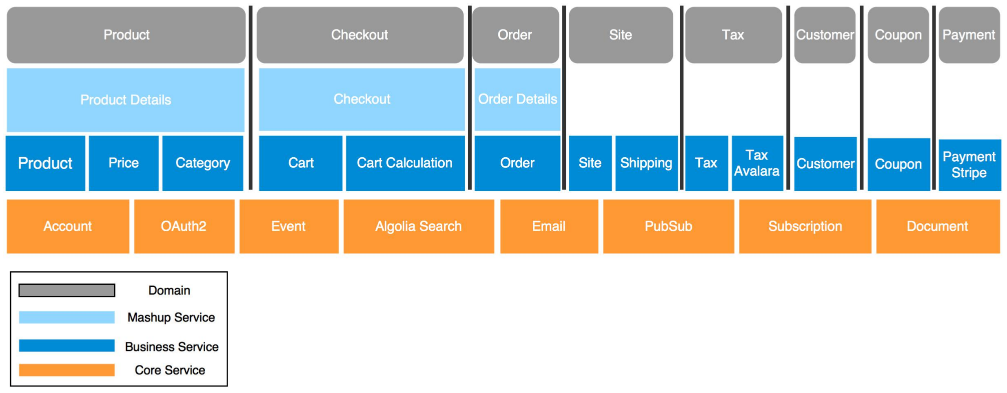


Figure 5.5: Microservices Layers For Commerce Domain

5.7 Deployment Workflow

Another major aspect apart from modeling microservices is continuous deployment. One of the major goals of implementing microservices architecture is agility. Agility can only be maintained when any small change in each microservice can be deployed into production smoothly without affecting other microservices.

5.7.1 SourceCode Management

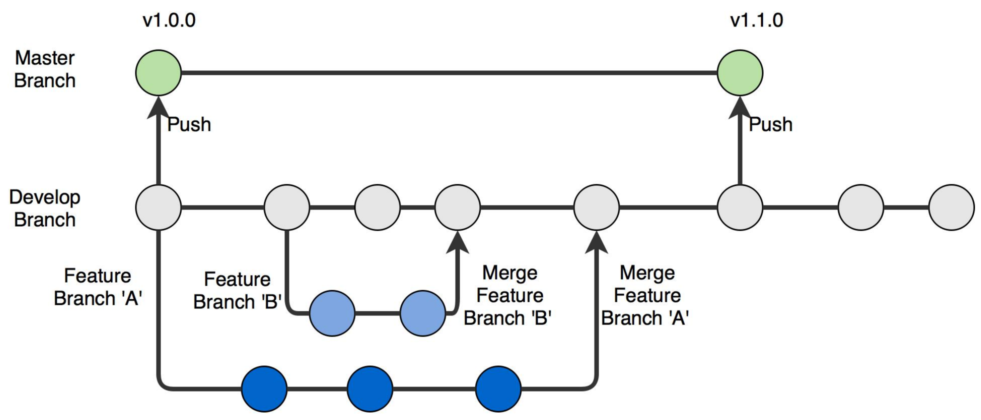


Figure 5.6: Sourcecode Management at SAP Hybris

The sourcecode files for each microservices are maintained in BitBucket version control system. The files are managed using Git. In order to support continuous deployment and efficient collaboration among the developers working in the same microservice, various branches are maintained within each repository. The major branch is called "Master" branch, which is used only for continuous deployment in production. Another branch is "Develop" branch and is used by developers to add changes continuously. It is used to create development version of the microservice and undergoes various level of tests. When sufficient features are ready and verified in "Develop" branch, they are merged into "Master" branch and realeased from "Master" branch. In this way, "Master" branch has always unbroken production ready sourcecode.
For adding any major changes or features in the microservices, a new "Feature" branch is created from "Develop" branch and undergoes various levels of testing before it is merged with "Develop" branch. In this way, development and verification of any change can happen independently.

5.7.2 Continuous Deployment

In Section 5.7.1, the concept of maintaining feature branches are discussed. In this section, the process of deploying the changes of feature branch to the production is discussed.
The Figure 5.7 shows the complete deployment flow for a microservice. Firstly, a developer "A" pushes his/her local changes to the remote "Feature" branch at BitBucket. The changes triggers Teamcity "Build" phase where the changes are first compiled and then a series of unit tests run against the newly built application which is again followed by a series of integration tests.
If the tests are successful, the developer will create a pull request in BitBucket, representing a request to merge his/her changes to "Develop" branch. Developer "B" from the same team will then review his/her changes and provides comments on BitBucket interface. After the changes are made and developer "B" is satisfied, he/she approves the pull request.
The developer "A" or "B" can merge the feature branch into develop branch. This again triggers "Build" phase in Teamcity. In "Build" phase, the develop branch gets compiled, followed by unit tests and integration tests. If the tests are successful, then "Deployment" phase is triggered when code is deployed to "Stage" environment and a group of smoke tests run to verify the deployment. Now, project owner or quality assurance team can access the microservice from stage and perform user acceptance tests. If there are any feedbacks or changes to be made, it will follow the same process from the beginning.
Now, when there are sufficient features and it is time for release, the "Develop" branch is merged to "Master" branch. The merge will again trigger "Build" phase in Teamcity. After successful tests, the "Master" branch is first deployed into "Stage" environment, where a series of smoke tests runs. After the deployment is successful without any error, the "Master" branch is deployed into "Production" environment, where various smoke tests run again to re-assure the deployment.

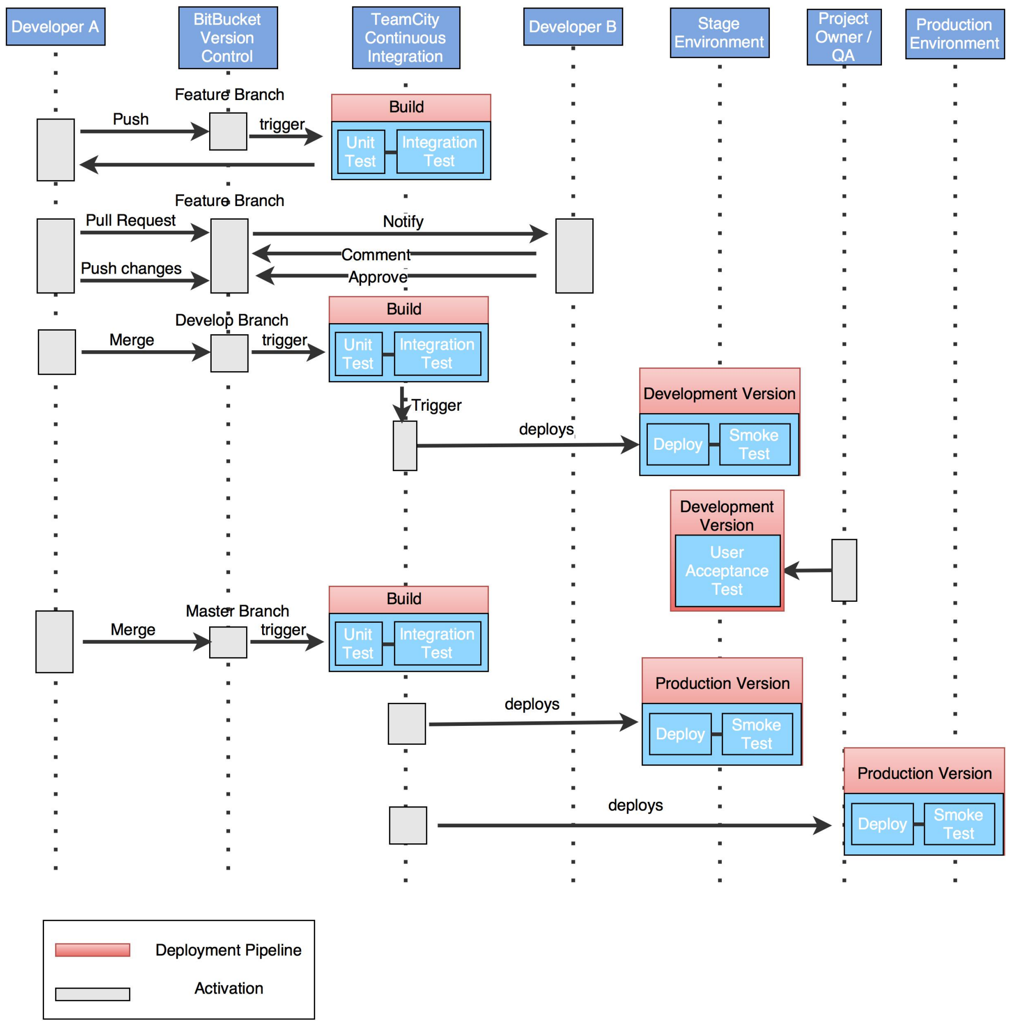


Figure 5.7: Continuous Deployment at SAP Hybris

The deployment is performed using cloud foundry which consists of AWS cloud underneath. The Figure 5.5 uncovers only microservices at the backend related to commerce domain. The complete architecture covering other domains such as Marketing etc and all layers including User-Interface, PaaS is shown by Figure 5.8. Apart from Commerce domain, there are various other domains such as Marketing, Sales etc. On each domain, same process as discussed in the Case Study 5.6 is followed to identify various microservices. For example, in 'Marketing' domain, there are microservices such as 'Loyalty', Loyalty Mashup' services.

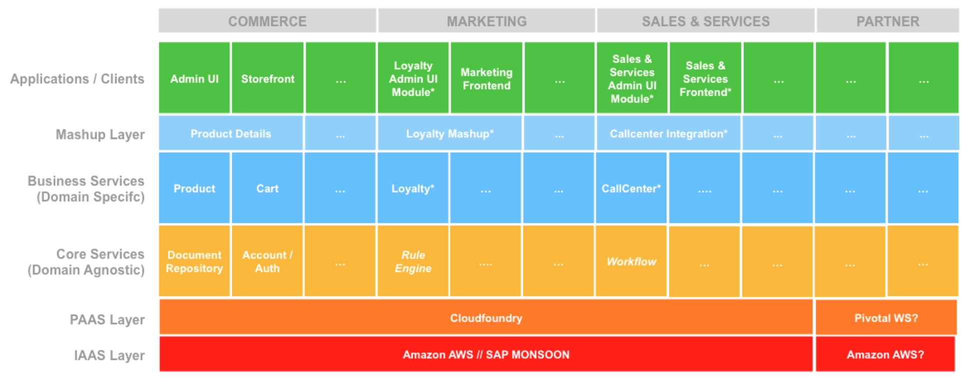


Figure 5.8: SAP Hybris Architecture

5.8 Problem Statement

In this chapter, the modeling approach used in SAP Hybris is discussed. Again, the continuous deployment process for each microservice is presented. This adds value to the findings made regarding modeling process from literature review. Now, as stated in Section 1.4, the constraints or challenges are one of the drivers for defining guidelines. So, the next step is to research about various challenges when implementing microservices and find the techniques to tackle them. While looking into that, it can also be interesting to find out how these challenges are being handled at SAP Hybris.

6 Challanges of Microservices Architecture

This chapter focuses on various challenges faced while implementing microservices. In Section 6.1, various advantages of the microservices architecural approach along with the challenges are listed. Then each challenge is broken down into several sub-challenges and discussed further after Section 6.2. For each challenge, various techniques which can be used to handle them are also presented. Additionally, in order to give the insight about approach in industries, the techniques used in SAP Hybris are also presented.

6.1 Introduction

The Section 1.1.3 lists some drawbacks of monolithic architecture. These disadvantages have become the motivation for adopting the microservices architectural approach. The microservices architecture offers opportunities in various aspects however it can also be challenging to utilize them properly. The responses from Interview Question 5.4.2 also highlight some prominent challanges. In this chapter, the challanges and advantages of microservices architectural approach are discussed. **[Fowler:2015aa]**

Advantages	Challanges
Strong Modular Boundaries It is not completely true that the monoliths have weaker modular structure than microservices. But, as the system gets bigger, it is very easy for a monolith to turn into a big ball of mud. However, it is very difficult to do the same with the microservices. Each microservice is a cohesive unit with full control upon its business entities. The only way to access its data and functionalities is through its API.	**Distributed System Complexity** The infrastructure of the microservices architecture is distributed, which introduces many complications, as listed by 8 fallacies **[Factor:2014aa]**. The calls are remote and are imminent to accomplish business goals. The remote calls are slower than local calls and affect the performance to a great deal. Additionally, network is not completely reliable which makes it necessary to handle failures.

Independent Deployment
Due to the automony characteristic of the microservices, each microservice can be deployed independently. Deployment of microservices is thus easy compared to monolith application where a small change needs the whole system to be deployed.[**Newman:2015aa**]

Integration
It is challanging to prevent breaking other microservices when deploying a microservice. Similarly, as each microservice has its own data, the collaboration among the microservices and sharing of data can be complex.

Agile
Each microservice is focused to single responsibility, changes are easy to implement. At the same time, as the microservices are autonomous, they can be deployed independently, decreasing the release cycle time.

Operational Complexity
As the number of microservices increases, it becomes difficult to deploy in an acceptable speed. Additionally, realising new version of the microservices becomes more complicated as the frequency of changes increase. Similary, as the granularity of microservices decreases, the number of microservices increases which shifts the complexity towards the interconnections. Ultimately, it becomes difficult to monitor and debug microservices.

In the following sections of this chapter, various ways to tackle the challenges listed in Section 6.1 are discussed in detail.

6.2 Integration

The collaboration among various microservices whilst maintaining autonomous deployment is challenging. In this section, various challanges associated with the integration and their potential remedies are discussed.

6.2.1 Sharing Data

An easiest way for collaboration among the microservices is to allow the microservices to access and update a common datasource. However, using this kind of integration creates various problems.[**Newman:2015aa**]
Problems

1. The shared database acts as a point of coupling among the collaborating microservices. If a microservice make any changes to its data schema, there is high

probability that other services need to be changed as well. Loose Coupling is compromised.

2. The business logic related to the shared data may be spread across multiple services. Changing the business logic is difficult. Cohesion is compromised.

3. Multiple services are tied to a single database technology. Migrating to a different technology at any point is hard.

Alternatives
There are three distinct alternatives to tackle the problems listed above.[**Richardson:2015aa**]

1. **private tables per service** - multiple microservices share the same database underneath but each microservice owns a set of tables.

2. **schema per service** - multiple microservices share same database however each microservice owns its own database schema.

3. **database server per service** - each microservice has a dedicated database server underneath.

Techniques

The logical separation of data among services discussed in the Section 6.2.1 increases autonomy but also makes it difficult to access and create consistent view of data. However, the problems can be compensated using following approaches.[**Richardson:2016aa**] [**Richardson:2015aa**]

1. The implementation of business transaction which spans multiple microservices is difficult and also not recommended because of Theorem **??**. The solution is to apply eventual consistency **??** focusing more on availability.

2. The implementation of queries to join data from multiple databases can be complicated. There are two alternatives to achieve this.
   a) A separate mashup microservice can be used to handle the logic to join data from multiple microservices by accessing respective APIs.
   b) CQRS pattern **??** can be used by maintaining separate a) model, and b) logic for updating as well as querying data.

3. The need for sharing data among various autonomous services cannot be avoided completely. It can be achieved by one of the following approaches.

a) The data can be directly accessed by using the resource owner's API.

b) The data can be duplicated into another microservice which accesses the data. The duplicate data can be made consistent with the owner's data using event driven approach.

SAP Hybris uses the technique of maintaining private schema per microservice as listed in the Section 6.2.1 in order to share data. For this purpose, there is a generic microservice called "Document" which handles the task of managing data for all microservices per each tenant and application. Furthermore, SAP Hybris follows eventual consistency and creates various mashup services for collective view of data from multiple services. Additionally, data are duplicated to decrease the network overload and when fresh data are required, resource owner's API is accessed directly.

6.2.2 Inter-Service Communication

A microservice is an autonomous component but the necessity of interaction among microservices cannot be denied. The various possible interactions among microservices is shown in Figure 6.1.[**Richardson:2015ab**]

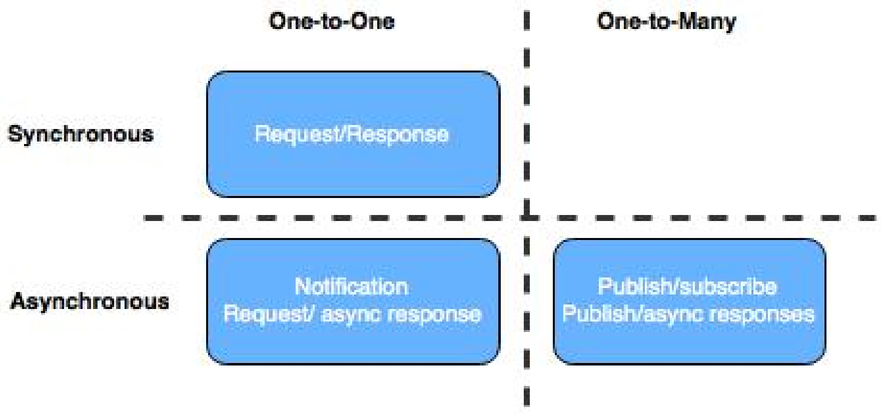


Figure 6.1: Inteaction Styles among microservices [[**Richardson:2015ab**]]

One to One interactions

1. **Request/Response**: It is synchronous interaction where client sends request and expects for response from the server.

2. **Notification**: It is asynchronous one way interaction where client sends request but no reply is sent from the server.

3. **Request/async Response**: It is asynchronous interaction where client requests server but does not get blocked waiting for the response. The server send the response asynchronously.

One to Many interactions

1. **Publish/subscribe**: A microservice publishes notification which is consumed by other interested microservices.

2. **Publish/async responses**: A microservice publishes request which is consumed by interested microservices. The microservices then send asynchronous responses.

Techniques

6.2.2.1 Synchronous and Asynchronous

Each of the interactions listed in Section 6.2.2 has its own place in an application. An application can contain a mixture of these interactions. However, a clear idea about the requirement as well as thorough knowledge regarding drawbacks associated with each interaction is necessary before choosing any type of interaction for a specific case.**[Newman:2015aa][Richardson:2014aa][Morris:2015aa]**

Synchronous Interaction

In synchronous interaction, it is simple to understand the flow, The synchronous approach is a natural way of communication. However, as the volume of interactions get higher and distributes to various levels or branches, it increases the overall blocking period of client and thus increases latency. The synchronous nature of interaction increases temporal coupling between microservices because it demands both consumer and producer microservices to be active at the same time. Since, the synchronous client needs to know the address and port of the server to communicate, it also inducess location coupling. With the cloud deployment and auto-scaling, this is not simple. Finally, to tackle the location coupling, service discovery mechanishm is recommended to be used.

Asynchronous Interaction

The asynchronous interaction decouples the microservices. It also improves latency as the client is not blocked waiting for the response. However, there is one additional component called message broker to be managed. Additionally, the conceptual understanding of the flow is not natural and not easy to reason about.

SAP Hybris uses both asynchronous as well as synchronous communication where appropriate. For, synchronous, it uses REST calls where as for asynchronous messaging it uses a generic microservice called "PubSub", which again uses Apache Kafka for managing publication and subscription of messages. An instance of the various interaction is shown in the Example 6.2.2.2.

6.2.2.2 Example

The Figure 6.2 shows various interactions during creation of an order. At first, the client, ’checkout microservice’ in this case, sends Restful Post order request to the ’Order microservice’. Next, the ’order microservice’ publishes ’Order Created’ event and then sends response to the client. The ’OrderDetails microservice’, which is subscribed to the event ’Order Created’ acts on the event by first fetching necessary product details from the ’Product microservice’ using Restful Get request. Finally, the ’OrderDetails service’ sends request to the ’Email microservice’ for sending email to the customer regarding order details but does not wait for response. The ’Email service’ sends response to ’OrderDetails service’ when email is sent successfully.

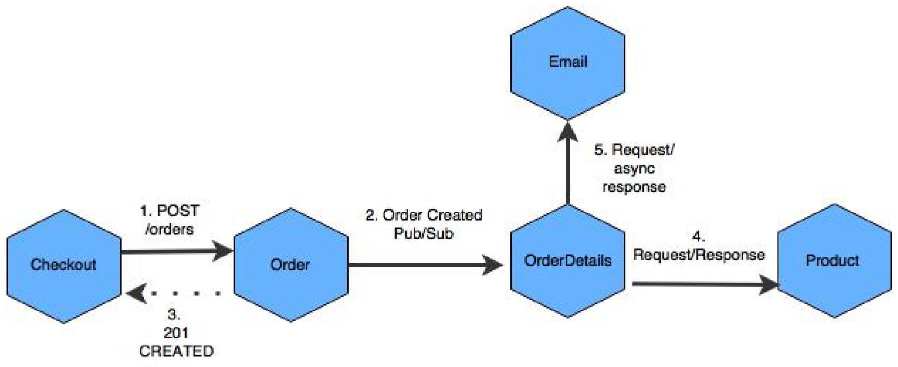


Figure 6.2: Interaction during Order creation

6.3 Distributed System Complexity

6.3.1 Breaking Change

Although microservices are autonomous components, they still need to communicate. Moreover, they also undergo changes frequently. However, some changes can be backward incompatible and break their consumers. The breaking changes are inevitable but the impact of the breaking changes can be somehow reduced by applying various techniques. [**Newman:2015aa**]

Techniques

1. **Avoid as much as possible**
   A good approach to reduce impact of incompatible changes of producer microservices to the consumers is to defer it as long as it is possible. One way is by choosing the correct integration technology such that loose coupling is maintained. For example, using REST as an integration technology is better approach than using shared database because the former approach encapsulates the underlying implementation and consumers are only tied to the interfaces.
   Another approach is to apply TolerantReader pattern when accessing provider's API [**Fowler:2011aa**]. The consumer can get liberal while reading data from the provider. Without blindly accepting everything from the server, the consumer can filter the required data only without caring about the payload structure and the sequence of data. So, if the consumer is tolerant while accessing the provider, the consumer service can still work even if the provider adds new fields or change the structure of the response.

2. **Catch breaking changes early**
   It is also crucial to identify the breaking change as soon as it happens. It can be achieved by using consumer-driven contracts. Consumers will write the contract to define what they expect from the producer's API in the form of various integration tests. These integration tests can become a part of the build process for the producer microservice. In this way, any breaking change can be detected in the build process before the API is accessed by the real consumers.

3. **Use Semantic Versioning**
   Semantic Versioning is a way to identify the state of current artifact using compact information. It has the form MAJOR.MINOR.PATCH. MAJOR number is increased when backward incompatible changes are made. Similarly, MINOR

number is increased when backward compatible functionalities are added. Finally, PATCH number represents that bug fixes are made which are backward compatible.
With semantic versioning, the consumers can clearly know if the provider's current update may breaks their API or not. For example, if consumer is using 1.1.3 version of provider's API and the new updated version is 2.1.3, then the consumer should expect some breaking changes and react accordingly.

4. **Coexist Different Endpoints**
   Whenever a breaking change on a service is deployed, the new deployment should be carefully managed not to fail all the consumers immediately. One way is to support old version of end points as well as new ones. This gives some time for consumers to react on their side. Once all the consumers are upgraded to use new version of the provider, the old versioned end point can be removed. However, using this approach means that additional tests are required to verify both versions of the end points.

5. **Coexist concurrent service versions**
   Another approach is to support both version of the microservice at once. The requests from the old versioned consumers need to be routed to old versioned producer microservice and requests from new versioned consumers to the new versioned producer microservice. It is mostly used when the cost of updating old consumers is high. However, this also means that two different versions have to be maintained and operated smoothly.

At SAP Hybris, breaking change is avoided as much as possible using REST and also Tolerant Reader pattern. When data is read from another microservice, the data is first handled by internal mapper library which maps the input data into the internal representation to be used by microservice. Additionally, for providing detailed information regarding state of current version of the microservice, semantic versioning is used. If there is breaking change in few endpoints, then multiple version of them is also maintained for short period of time giving the consumer fair amount of time for migrattion. Similarly, if there are breaking changes in most of the endpoints then the entire microservice is maintained with different versions.

6.3.2 Handling Failures

In the microservices architectural approach, communication among the microservices is achieved along quite unreliable network, failures are inevitable. So, it becomes challenging as well as highly necessary to handle the failures. Many strategies can be applied for the purpose. **[Newman:2015aa][Richardson:2015ab][Nygard:2007aa]**

Techniques

1. **Timeouts**
   A service can get blocked indefinitely waiting for response from the provider. To prevent this, a threshold value of waiting time can be set. However, care should be taken not to choose very low or high value for waiting time.

2. **Circuit Breaker**
   If a request to a service keeps failing, there is no value to keep sending request to the same server. It can be a better approach to identify when the request fails and stop sending further request to the provider microservice assuming that there is problem with the connection. By failing fast in such a way will not only save the waiting time for the client but also reduces unnecessary network load. Circuit Breaker pattern helps to accomplish the same. A circuit breaker has three states as shown in the Figure 6.3. In normal cases when the connection to the provider is working fine, it is in 'closed' state and all the connections to the provider goes through it. Once the connection starts failing and meets the threshold (can be number of failures or frequency of failures), the circuit breaker switch to 'open' state. At this state, any more connections through the circuit breaker fail straight away. After certain time interval, the state changes to 'half open' to check the state of provider again. Any connection request at this point passes through. If the connection is succesful, the state switches back to 'closed' state, else to 'open' state.**[Fowler:2014ac]** **[Newman:2015aa]** **[Nygard:2007aa]**

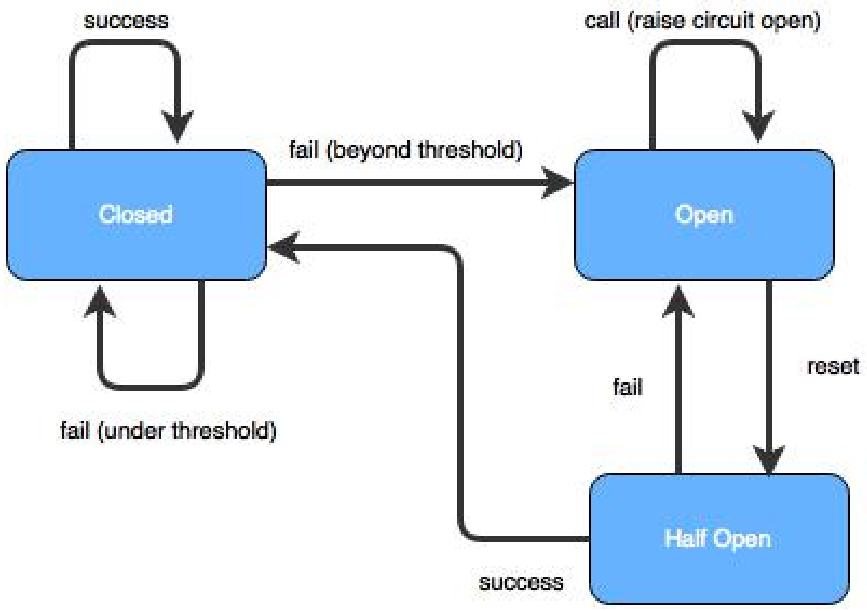


Figure 6.3: States of a circuit breaker [[**Fowler:2014ac**]]

3. **Bulk Head**
   Bulk Head is the approach of segregating various resources as per requirement and assigning them to respective purposes. Maintaining such threshold in available resources will save any resource from being constrained whenever there is problem in any part. One example of such bulkhead is the assignment of separate connection pool for each downstream resources. In this way, if there is a problem in the request from one connection pool, it will not affect another connection pool and also not consume all available resources. [**Newman:2015aa**] [**Nygard:2007aa**]

4. **Provide fallbacks**
   Various fallback logic can be applied along with timeout or circuit breaker to provide alternative mechanism to respond. For example, cached data or default value can be returned as a fallback mechanism.

At SAP Hybris, synchronous requests are made recursively for certain number of times, each attempt waits for the response only for certain fixed amount of time depending upon the use case. In order to save network resources and provide fault tolerance, circuit breaker pattern is used.

6.4 Operational Complexity

6.4.1 Monitoring

With monolithic deployment, monitoring can be achieved by sifting through few logs on the server and looking various server metrics. The complexity can increase to some extent if the application is scaled along multiple servers. In that case, monitoring can be performed by accumulating various server metrics and going through each log file on each host. The complexity goes to whole new level when monitoring microservices deployment because of large number of individual log files produced by the microservices. In addtion to the difficulty of going through each log file, it is more challenging to trace any request contextually along all the log files.

Techniques

There are various ways to work around these complexities [**Newman:2015aa**] [**Simone:2014aa**].

1. **Log Aggregation and Visualition**
   A large number of logs in different servers can be intimidating. The tracing of logs can be easier if the logs could be aggregated in a centralized location and then visualized in a better way such as graphs. One of such solutions can be achieved using ELK stack as shown in the Figure 6.4. [**Anicas:2014aa**] [**Newman:2015aa**]

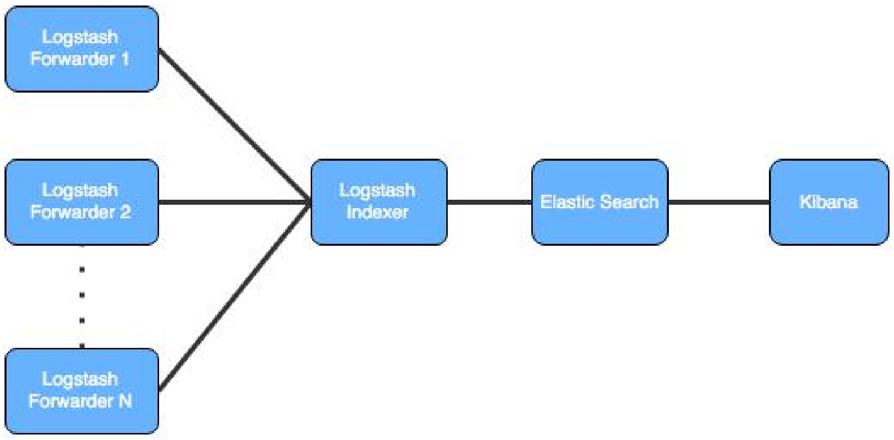


Figure 6.4: ELK stack

The ELK stack consists of following major components.

   a) Logstash Forwarder
      It is the component installed in each node which forwards the selected log files to the central Logstash server.
   b) Logstash Indexer
      It is the central logstash component which gathers all the log files and process them.
   c) Elastic Search
      The Logstash Indexer forwards the log data and stores into Elastic Search.
   d) Kibana
      It provides a web interface to search and visualize the log data.
      The stack makes it easy to trace logs and visualize graphs along various metrics.

2. **Synthetic Monitoring**
   Another useful aspect of monitoring is to collect various health status such as availability, response time, etc. There are many tools to make it easier. One such tool is ′uptime′, which is a remote monitoring application and provides email notification as well as features such as webhook to notify a url using http POST. Rather than waiting for something to get wrong, a selected set of important business logic can be tested at regular interval. The result can be fed into notification subsystem, which will trigger notification to the responsible parties. This ensures confidence that any problem can be detected as soon as possible and can be worked on sooner. **[Simone:2014aa] [Newman:2015aa]**

3. **Correlation ID**
   A request from a client is fulfilled by a number of microservices, each microservice being responsible for certain part of the whole functionality. The request passes through various microservices until it succeeds. For this reason, it is difficult to track the request as well as visualize the complete logical path. ′Correlation ID′ is an unique identification code assigned to the request by the microservice at the user end and passes the ′Correlation ID′ towards the downstream microservices. Each downstream microservice does the same and pass along the ID. In this way, a complete picture of any request is easier to visualize.

SAP Hybris uses ELK stack for monitoring and log visualization. In order to check the health status of the microservices, ′uptime′ tool is used. The webhook from the tool updates the central status page showing the current status of all the existing microservices. Any problem triggered by ′uptime′ is also reflected in this page. Similary, a set of smoke tests are triggered from continuous integration tool such as TeamCity at regular interval. In order to track the requests at each microservices, a unique code is assigned to each request. The code is assigned by API-Proxy server which is the central gateway for any incoming request into YaaS.

6.4.2 Deployment

A major advantage of the microservices is that the features can updated and delivered quickly due to the autonomity and independent deployability of each microservice. However, with the increase in the number of microservices and the rate of changes that can happen in each microservice, fast deployment is not straight forward. The culture of doing deployment for monolithic applications does not work perfectly on microservices. This section discusses how continuous deployment can be achieved in the microservices.
There are two major parts in deployment. In order to speed up deployment in microservices, approach to accomplish each part has to be changed slightly.

Techiniques

1. **Continuous Integration**
   It is a software development practice in which newly checked in code are followed by the automated build and verification phase to ensure that the integration was successful. Following continuous integration does not only automate the artifact creation process but also reduces the feedback cycle of the code providing opportunity to improve quality.
   To ensure autonomy and agility in microservices, a better approach is to assign separate sourcecode repository and separate build for each microservice as shown in the Figure 6.5. Each repository is mapped to a separate build in Continuous Integration server. Any changes in the microservice sourcecode repository triggers corresponding build in Continuous Integration server and creates a single artifact for that microservice. An additional advantage of this approach is clear assignment of the repository ownership and build responsibility to the teams who owns the respective microservices. **[Newman:2015aa] [Fowler:2006ab]**

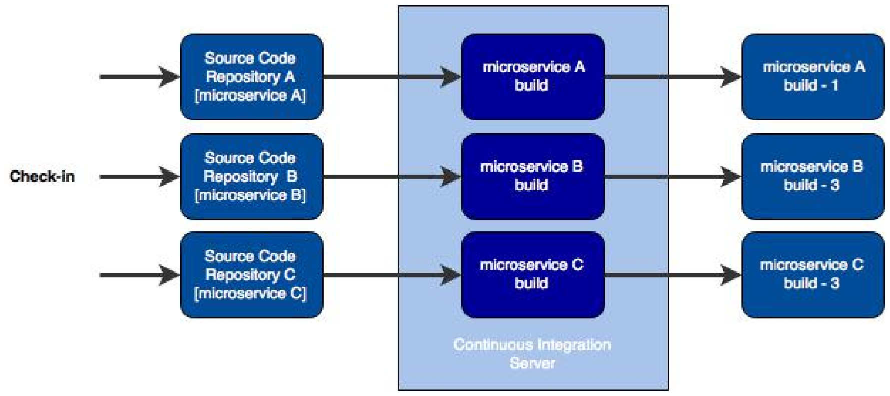

Check-in
Source Code Repository A [microservice A]
Source Code Repository B [microservice B]
Source Code Repository C [microservice C]
microservice A build
microservice B build
microservice C build
Continuous Integration Server
microservice A build - 1
microservice B build - 3
microservice C build - 3

Figure 6.5: Continuous Integration [**Newman:2015aa**]

2. Continuous Delivery
   It is a discipline in which every small change is verified immediately to check deployability of the application. There are two major concepts associated with continuous deployment. First one is continuous integration of the sourcecode and creation of artifact. Next one is continuous delivery in which the artifact is fed into a build pipeline. A build pipeline is a series of build stages, each stage responsible for the verification of certain aspects of the artifact. A successful verification at each stage ensures more readiness of the artifact to release.

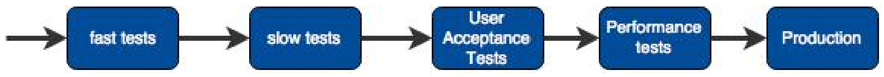


Figure 6.6: Continuous Delivery Pipeline [**Newman:2015aa**]

   A version of build pipeline is shown in the Figure 6.6. Here, faster tests are performed ahead and then a) slow tests and b) manual testing are performed down the line. This kind of successive verifications make it faster to identify problems, create clear visibility of the problems and improve the quality as well as confidence in the quality of the application.
   Each microservice has an independent build pipeline. The artifact build during continuous integration triggered by any change in the microservices is then fed into corresponding build pipeline for the microservice. [**Newman:2015aa**] [**Fowler:2013aa**]

The variation of continuous integration and deployment used at SAP Hybris is already discussed in Section 5.7.

6.5 Conclusion

In this chapter, various challenges needed to handle when implementing microservices architectural approach are listed. Each challenge is further broken down into fine constraints. Secondly, various ways of handling those constraints are discussed. Finally, the approaches used in SAP Hybris for handling the challenges are also presented. Microservices architecture provides efficient way to develop agile software. However, without a good grasp in the techniques to handle the challenges, it may not be a good idea to start with microservice in the first place.

7 Guidelines

7.1 Context

The major objective of the research is to devise some guidelines so that the task of approaching any complex problem domain using microservices architectural approach becomes easier. Additionally, a few research questions related to crucial topics on microservices have emerged. The research questions are taken as stepping stones for discovering guidelines.
At first, the concept related to the granularity of the microservices is studied in detail in Chapter 2. In this chapter, the semantic meaning of the size along with various dimensions defining granularity are discussed. Finally, various principles to evaluate the optimum size for microservices are listed.
Secondly, other quality attributes to be considered when designing good microservices are listed in Chapter 3. The quality metrics to determine each of these quality attributes are also mentioned. The various quality metrics lead to a limited list of basic metrics. Finally, various principles to determine quality of good services along with the way the quality attributes affect each other are recorded.
Furthermore, the various approaches of decomposing problem domain for identifying microservices are discussed in Chapter 4. The methods discussed are using domain driven design and using use cases refactoring. The steps involved in each approach are described along with an example.
In the previous chapters, understanding of various topics such as to a) granularity , b) quality attributes and c) process of identifying microservices are derived from literature. In the Chapter 5, an attempt is made to research further on the same topics from industry experience by studying the architecture at SAP Hybris and conducting various interviews. Finally, a series of steps are listed to visualize the process of breaking down a problem domain into the microservices.
The challenges faced while implementing microservices architecture are discussed in Chapter 6. Additionally, various ways to tackle these challenges are described.

7.2 Process to Implement Microservices Architecture

As shown in the Figure 7.1, the implementation of the microservices consists of two major parts and understanding each part is crucial in order to follow the microservices architecture.

1. **modeling microservices**
   At the core of the microservices architecture is problem domain and the importance of understanding it. Modeling microservices is the process of dividing the problem domain into various components considering various internal quality attributes such as coupling, cohesion, etc. The knowledge regarding the quality attributes is an important input when choosing the efficient process for identifying the microservices from the problem domain.

2. **operating the microservices artifacts in the production environment**
   The microservices are designed to satisfy various external quality attributes such as scalability, resilience, etc. However, the microservices also introduce critical challenges. It is not entirely possible to get the advantages from the microservices architecture unless a) these challenges are identified as well as tackled appropriately and b) certain implementation and operational practices are followed.

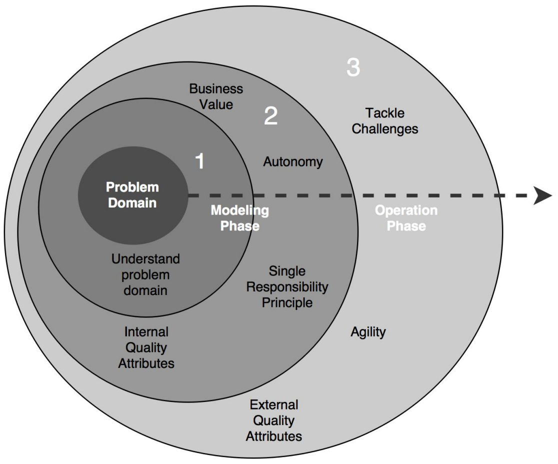

3
Business Value
2
Tackle Challenges
1
Autonomy
Problem Domain
Modeling Phase
Operation Phase
Understand problem domain
Single Responsibility Principle
Agility
Internal Quality Attributes
External Quality Attributes

Figure 7.1: Process to implement microservices architecture

The microservices architecture follows a process which guides all the way from a) understanding the problem domain, b) identifying modular components considering various attributes and c) implementing as well as operating the components to satisfy various external quality attributes. It is interesting to find how process, principles and guidelines are related. A process is based upon some basic principles and the principles helps to define guidelines based upon some specific requirements and environment. So, in order to understand the process of implementing microservices, it can be helpful to look into some basic principles.

7.3 Principles And Guidelines

Based upon the research findings as discussed in the previous chapters, various principles are listed to guide modeling microservices as well as operating them. It is highly recommended to follow these principles when considering microservices architectural approach.[**Newman:2015aa**]

1. **Correct Granularity**
   The dimension of granularity for microservice is given by a) the functionality it performs, b) data it handles and c) the business value it provides 2.3. These dimensions make it pretty obvious how to make the size of a service as low as possible. However, the notion of correct granularity is rather important and should be focused instead of trying to make the size as low as possible.2.2 There are various factors which act together and tailor the size of the microservices.
   a) One of the factors is **SRP**, which influences the functionalities and data handled by microservices to make it as small as possible. SRP encourages to focus on small set of cohesive tasks that change for the same reason. [**Stine:2014aa**] [**Newman:2015aa**] The influence of SRP on the microservices architecture is also verified by the result of interview which is compiled in the Section 5.4.2.
   b) Another important factor is **autonomy**. According to S. Newman, a microservice should be able to be deployed and updated independently [**Newman:2015aa**]. Without any surprise, autonomy is an important quality attribute of the microservices and is evaluated as its degree of control upon the operations to act on its business entities 3.3.6. As discussed in the Section 2.2, a microservice should provide transaction integrity such that it is big enough to support the activities which fall under one transaction. So, autonomy tends to limit the scope of functionalities and data such that the microservice is a) self-contained, b) self-controlling and c) can be self-governed [**Ma:2007aa**].

c) The act of realizing the size of microservices as small as possible comes with a price. As the functionality of microservice is squeezed, the number of microservices needed for any application increases. The task of deploying, provisioning and governing these microservices can be challenging. So, as stated in the principles defined in the Section 2.3.4, the decision regarding the size of service should be rational based upon the current **infrastructure and operational capability** to handle such large number of small microservices. It is further supported by the response of the interview compilation in the Section 5.4.2.

d) Finally, the size of microservices should also provide **business value** and should coincide with the business goal of the organization. It is also evident from the response of the interview compiled in the Section 5.4.2. Additionally, the business value being an important dimension of granularity of the microservices according to the Section 2.3, should be examined at the design time while decomposing the problem domain.

2. **Consider quality attributes as early as possible**
   The internal quality attributes such as coupling, cohesion, autonomy, etc. should be controlled as early as possible during the modeling and development phases. Taking care of the internal quality attributes will eventually help to control the external quality attributes of the microservices such as reusability, reliability, resilience, etc. The Table 3.8 which identifies various basic metrics can be helpful to evaluate the quality of the microservices. For one thing, the metrics such as scope of opertions, number of operations, etc. used in the basic metrics table are completely in disposal to the developers and the teams of the microservices. Secondly, it helps to identify the relationship among them as shown in the Table 3.9 and assists in managing trade offs when needed.

3. **Understand the problem domain**
   As discussed in the Chapter 4, problem domain can be analyzed using either usecases refactoring approach or domain driven design approach.
   Using usecases can break down the problem domain into various levels of abstractions, each abstraction representing lower scope of functionality. The graphical approach used in usecases modeling approach can be an effective approach for a) brainstorming, b) exploring the problem domain and c) finding new usecases.
   Another popular approach is using domain driven design to explore the problem domain. The overall process looks natural and straight forward, which focus on dividing the complex problem domain into small manageable subdomains and further into modular autonomous components. Furthermore, the approach

gives emphasis on using ubiquitous language to find autonomous modules and define their boundaries. The chapter 4 provides important guidelines for finding ubiquitous language and bounded contexts.
The graphical approach provided by usecases are more familiar to architects as well as developers and thus can be faster. However, they tend to focus only on SRP and scope of the functionalities. Following the process, may likely lose grip on very important quality attribute of the microservices called autonomy.
On the other hand, domain driven design with its approach of using ubiquitous language, focuses on autonomy. Additionally, by dividing the problem domain into smaller manageable components with limited scope, domain driven design also focuses on SRP. Although the whole proces seems natural, getting the boundary right is a complex process. A clear understanding of the problem domain is necessary in order to get the bounded context right. So, it can be an iterative process, starting with bigger boundaries at first and down to smaller modular boundaries in several iterations.
The knowledge and practice of both usecases modeling and domain driven design can be helpful. Depending upon the complexity of the problem domain and experience with it, any of them or combination of them can be used to decompose the problem domain.

4. **Culture of Automation**
With such a large number of small services, the task of managing and operating them manually can become impossible. There are three specific areas where automation can serve greatly.
    a) Automated Continuous Delivery can help building and deploying microservices frequently with consistency. Maintaining continuous integration and delivery pipeline can make sure that any new changes is consistent to old system and can be put into release in a matter of few button presses.
    b) Automated Testing is another important area to consider when there are large number of microservices and large number of changes in the backlog. Without automated testing in the delivery pipeline, it can be hard deploy changes to release environment without breaking existing functionalities.
    c) Infrastructure Automation and Provisioning should not be neglected either, especially when there can be different technology stacks in different microservices and different infrastructure environments. PaaS such as CloudFoundry can be helpful for deployment and provisioning easily whereas various configuration management tools such as Puppet, chef etc can assist to manage different technology stacks.

5. **Hide Internal Implementation**
Collaboration among the microservices is essential however high coupling by over exposing the inner-details should be avoided to save autonomy.
    a) The first on the checklist is to model the APIs in right way. Rather than breaking the application into technological boundaries, the problem domain should be clearly understood to discover clear functional boundaries and should be modeled around business domains. The concept of bounded context can be a good approach to define clear APIs, which also reduces unnecessary coupling.
    b) The APIs should be technology agnostic, which means that the technology used for collaborating between APIs should not guide the internal implementation of the APIs. Using REST and some form of RPC can be useful tackle them.
    c) In microservices, database is also an integral part of internal implementation. Already mentioned in the Section 6.2.1, sharing database tightly couples microservices as it exposes internal data structure details. In order to solve such coupling, each microservice should atleast have its own a) private tables, or b) schema, or c) at most separate database server.
    d) Additionally, in order to maintain loose coupling, the integration technology should be chosen carefully. As already discussed in the Section 6.2.2, if business requirement allows, asynchronous communication styles such as a) publish/subscribe, b) notification, and c) request/async response should be chosen over synchronous request/response.
6. **Decentralize**
Autonomy should not be limited to deployment and updates only. It should be applied in team organization as well. A team should be autonomous enough to own the microservices and control all the phases from a) developing the microservices, b) releasing them, and c) maintaining in the production environment. In order to achieve that, the teams should be trained to become domain-experts in the business areas related to their microservices. Additionaly, the architecture of microservices should follow decentralization as well. The business logic should be divided among microservices and should not be concentrated towards any specific a) god microservices and b) integration mechanism. The microservices should be smart enough to handle collaboration among themselves and the communication mechanism should be as dumb as possible such as REST and messaging. Also, choreography should be highly applied whenever possible and orchestration should only be used if the business requirement dictates so.

7. **Deploy Independently**
   Microservices should be able to be deployed independently. The Section 6.3.1 already discusses the challenges involved as well as their solutions.
   a) In order to avoid breaking changes, a) the integration technology such as REST should be used to decouple microservices and b) tollerant reader pattern should be used to implement consumers.
   b) To find the breaking changes early during development, consumer-driven contracts should be implemented as automated tests in the delivery pipeline of the microservices. Additionally, semantic versioning can be used to clearly indicate the level of new changes.
   c) When breaking changes cannot be avoided, maintaining co-existing endpoints with different versions or co-existing microservice versions can provide enough time and opportunity for consumers to get updated gracefully and without breaking APIs.
   d) Finally, the release and the deployment can be decoupled using techniques such as bluegreen deployment **??** and canary release **??**, so that new changes can be tested in production with confidence and can be released later reducing the risk.

8. **Consumer First**
   The microservices are developed in order to serve its consumers, so consumers should be the priority when designing them.
   a) APIs should be designed considering its consumers. It should be easy to a) understand, b) use and c) extend. The name as well as the link of API should be self intuitive. Additionally, documentation to follow and use API should be provided. In order help with these, there are various tools available such as swagger, RAML etc. The tools not only make it easy to design and document APIs but also provide capability to try them.[**Bloch:2016aa**] [**Blanchette:2008aa**]
   b) Another important concept is to make it easy for consumers to find the microservices itself. There are various service discovery tools such as zookeeper, etcd, consul etc that can be used.

9. **Isolate Failures**
   The microservices are build on the top of distributed system and it is not false to say the distributed system cannot be perfectly reliable [**Factor:2014aa**]. Although, with high number of the microservices and complex collaboration among

them over unreliable network, the system should not be affected until all the microservices fail. The solutions are discussed in Section 6.3.2.

a) Timeout should be implemented realizing the fact that remote calls are different than local calls and they can be slow. A realistic value of timeout should be chosen based on the use cases scenario.

b) In order to avoid leaking failures and affecting the whole system as a result, bulkhead should be used to segregate the resources and an appropriate thereshold value should be maintained for each group of resources.

c) For reducing latency, circuit breaker should be implemented which detaches a failed node after certain attempts and reattemts automatically. In this way, it not only removes unnecessary roundtrip time or waiting time but also saves network resources.

d) Finally, fallback mechanism can be used in conjunction with timeouts and circuit breaker to provide alternative mechanism such as serving from cache, etc. In this way, it not only isolates the failures but also serves the request.

10. **Highly Observable**
It can be difficult to observe the status of each microservice on each provisioned hosts considering the number of microservices. The Section 6.4.1 points out some ideas which can be used to make it easier.

    a) Use semantic monitoring to check the current behaviour of the microservices such as availability, response time etc. An example of effective tools is uptime. Additionally, trigger certain tests to verify important logic of the microservices on regular basis. One way to achieve that is to trigger smoke tests automatically.

    b) In order to get the overall picture of the whole application at one place, use logs aggregation and visualization tools such as logstash and kibana.

    c) Finally, use correlation ids to get a semantic picture of the related requests which gets branched out along many downstream microservices.

8 Conclusion

The software architecture provides a set of guidelines to decompose a system into subsystems, components and modules. Additionally, it defines the interaction amongst its individual components. Defining an architecture is an important part in software development. So, there should be precise guidelines to implement the architecture.
One of the architectural approaches is monolithic architecture in which an application is deployed as a single artifact. With this architecture, a) development, b) deployment and c) scaling is simple as long as the application is small. However, as the application gets bigger and complex, the architecture faces various disadvantages including a) limited agility, b) decrease in productivity, c) longterm commitment to technology stack, d) limited scalability, etc. This is where, a different architectural approach called 'Microservices' comes into play. In this approach, a) an application is composed using different independent, autonomous components, b) each component fulfils a single functionality and c) each component can be deployed as well as updated independently. However, there are various concepts related to the definitions of microservices such as collaboration, granularity, mapping of business capability, etc. which are not clearly documented. Moreover, there are no proper guidelines how to follow microservices architecture. This is the motivation of the current research. The objective of the research is to clarify various concepts related to the microservices and finally create a precise guidelines for implementing the microservices.
In order to achieve that, various definitions provided by different authors are taken as starting point. A list of keywords and areas are selected from them, which are further investigated during the research. Furthermore, various factors such as a) quality attributes, b) constraints and c) principles are considered as the building blocks of any architecture. So, these areas are also considered during the research to create guidelines.
Granularity of a microservice is considered as an important concept and is discussed a lot. Granularity is not a one dimensional entity but has three distinct dimensions which are a) functionality, b) data and c) business value. Increase in any of these dimension increases granularity. Also, it is important to consider the concept of 'Appropriate' granularity rather than 'Minimum' sized microservices. Appropriate granularity of any microservice is achieved by three basic concepts: **Single Responsibility Principle**, **Autonomy**, and **Infrastructure capability**.

Adding to that, other quality attributes such as coupling, cohesion, autonomy, etc. are as important as granularity. For that purpose, a list of basic metrics are created to evaluate each quality attributes easily. The quality attributes are not mutually exclusive but related to eachother. The relationship among them is clarified so that it becomes easy to determine the appropriate tradeoffs when necessary.
Another important concept which is not clearly documented is the process of identification of microservices and mapping of business capability to microservices. Two different approaches are discussed which are **Using Use Cases Refactoring** and **Using Domain Driven Design**.
Use Cases Refactoring utilizes use cases to breakdown a problem domain and refactor them based on various concepts such as similarity in functionality, similarity on entities they act, etc. A discrete set of rules for use cases refactoring are also present. On the other hand, domain driven design uses the concept of ubiquitous language and bounded context to break down a problem domain. A detailed step for each phase is also presented. Furthermore, a case study is broken down into microservices using each approach.
Use cases Refactoring is comparatively easier since architects and developers are more familiar with the concept. Domain Driven Design on the other hand, is a complex and iterative procedure. Furthermore, use cases Refactoring considers an entity as a single source of truth for all sub-domains in the entire system. Using this approach does not necessarily produce autonomous microservices but focus more on functionalities. However, the domain driven design using the concept of ubiquitous language and bounded context creates autonomous microservices with single responsibility.
After conducting research along various literature, another important part is determining the process followed in industry for which SAP Hybris is chosen. Firstly, various available documents are studied which suggested vision and principles as being the driving force for the process of implementing microservices. In order to understand indepth, interviews are conducted with various key personnels related to YaaS. A major outcome of the interview is that, in addition to factors such as autonomy, infrastructure capability and Single Responsibility Principle, **Business Value** of the microservices plays significant role when a) identifying microservices and b) defining the optimum granularity. Finally, the workflow followed during deployment of microservices in SAP Hybris is also studied in order to clarify the operational approach together with modeling process.
Understanding constraints is another important driver of architecture. The approach to handle various constraints can be helpful in defining guidelines. Microservices architecture has various advantages such as strong modularity, agility, independent deployment capability however also presents various challenges for maintaining these advantages. The challenges can be kept into three distinct groups: **Distributed System**

Complexity Integration, and **Operational Complexity**.
Different challenges along each group are discussed. Additionally, the various techniques which can be used to tackle each are listed based on the literature. Finally, the techniques used in SAP Hybris to handle each challenge are also mentioned.
With all the studies performed, the process of microservices architecture can be disected along two phases: **Modeling Phase** and **Operation Phase**.
During the modeling phase, problem domain is studied and various internal quality attributes along with business value as well as infrastructure capability are considered. Whereas during operational phase, various challenges are tackled. The effectiveness of modeling phase is visible across various external quality attributes in operation phase. Again, the principles are major driving factor for creating architectural guidelines. The principles along both modeling and operation phase are listed based on the previous findings of the research. Finally, for each principles, a set of guidelines are presented. The guidelines are one of the major outcomes of the research.

9 Related Work

Microservices is a quite new architecture and it is not surprising that there are very few research attempts conducted on the overall process of modeling them. Although there are a lot of articles which share the experience of using microservices, only few of them actually share how they achieved the resulting architecture. According to process described in [**Levcovitz:2014aa**], firstly database tables are divided into various business areas. Secondly, business functionalities and corresponding tables they act upon, are grouped together as microservices. It may only be used in special cases because an assumption is made that there exist a monolith system which has to be broken down into components as microservices and analysing the codebase is one of the necessary steps followed to relate business function with database tables. Furthermore, it does not provide any clear explanation regarding how to break the data into tables handling autonomy. Also, there is no idea about how different quality attributes will be used when breaking the monolith into various business areas. Another research paper [**Bruggemann:2013aa**] also share its process of finding microservices but does not provide enough explanation except that the microservices are mapped from product or features of the system.
The approach applied in the current research, as already defined in section 1.4, takes quality attributes and modeling process into account. It can be interesting to see the related works on these two topics.
Looking back to component oriented architecture can also be helpful to identify some concepts. The research paper [**Lee:2001aa**] describes process of using coupling and cohesion to indentify individual components. In this process, a single use case is mapped to a component such that interaction among the objects inside components is decreased. Although the whole process may not be used for services but the idea of use cases can be applied.
Again, the book by S. Newman explains about various aspects of microservices including quality attributes. According to S. Newman, the major quality attributes to be considered while modeling microservices are coupling, cohesion and autonomy [**Newman:2015aa**].
[**Ma:2009aa**] defines an iterative process evaluating portfolio of services around various quality metrices in order get the better quality services. It takes various quality attributes such as cohesion, coupling, size etc into consideration to find out their overall

metrics value. The process is iterated until the services are obtained with satisfied values. The consideration of quality attributes and evaluating them is well thought in this process but the process of coming up with the initial set of services from business domain is not well documented.
In the book by S. Newman, the process of modeling microservices is also mentioned and suggests that the concept of bounded context by domain driven design should be used to model microservices.[**Newman:2015aa**]
Additionally there are a lot of methodology standards based around SOA to model the services. Some of these methodologies are SOAF, SOMA, SOAD and others. The paper [**Ramollari:2016aa**] presents a detail list and comparision of these methodologies. Most of them are not yet implemented in industries and others not compatible with agile practices.

10 Future Directions

The current research is conducted on the basis of various academic as well as industrial researches. It would be interesting if the current research could create opportunities for new researches.
The Table 3.8 listed some basic metrics which can be used to evaluate the quality of microservices. The very first thing which can be done is to find the threshold range that can classify good and bad microservices. A certain number of microservices can be taken as sample from which microservices with high scalability, reusability or other external quality attributes can be filtered out from non performing microservices with low scalability and reusability. Then basic metrics tables can be filled along with their corresponding values for each microservice. These tables can used to define threshold values. The threshold values can be verified further by using them when creating new microservice and checking if their external quality attributes meet the expectations.
Another direction of the research can be finding the priority sequence of quality attributes for microservices. From literature, the quality attributes to focus are coupling, cohesion and autonomy. During the interview conducted with SAP Hybris, the quality attributes such as scalability and reusability were given priority when mapping functionalities to microservices. It can be valuable to research further in literature as well as in other industries regarding the priority of quality attributes they choose to identify microservices.
The basic metrics table can be leveraged to create a graphical tool which represents various microservices as nodes and connections between them as lines between nodes. It would be benefical if various basic metrics can be evaluated only with API definitions automatically. The graphical tool can be used to show the various quality attributes using the basic metrics. Depending upon the expected quality, the graphical tool can be used to change API definition. This can be a great interactive tool for optimizing quality of microservices at design time.
Nevertheless, the report can always be used as a base to conduct research on other similar industries as SAP Hybris. It can be interesting to view the result obtained from the study at various different industries. Given the situation that there are very few literature research in this area and most of the industries which are using microservices have not openly communicated the whole lifecycle process of microservices, this document can be a starting point for any company to implement microservices.

List of Figures

List of Tables