ADOPTION OF THE MICROSERVICE ARCHITECTURE

Maryanne Ndungu

Master of Science Thesis
Study Programme in Software Engineering
Faculty of Science and Engineering
ÄBO AKADEMI
March, 2019
Abstract

The ever changing world of technology and increased consumer demands have challenged system and software architects to design architectures that are resilient to change so as to remain relevant and also meet requirements. This has led to the evolution of software design and development over the decades. Unfortunately, the challenges brought about by the exponential growth of a system's code base over time are persistent and continue to demand for innovative solutions. Microservices have therefore emerged as an attempt at providing innovative guidelines to designing architecture.

The microservice architecture, through its characteristics, affords some solutions to managing complexities related to the growth of a system. The quality attributes attained by the implementation of the microservice architecture such as such as loose coupling, high cohesion, resilience and scalability are undeniably appealing. While microservices offer numerous business advantages such as faster-time-to-market, the adoption of the microservice architecture is more challenging compared to other architectures.

This thesis aims at providing a detailed understanding of microservices, its concepts and its benefits. It focuses on presenting implementation strategies and the best possible approaches to adopting the microservice architecture. A goal-oriented approach to adopting microservices is highly encouraged for companies and organizations that want to derive maximum business value from microservices. In its attempt to provide guidance to adopting microservices, this thesis discusses different ways of adopting the architecture and how to identify microservices.

As an outcome, this thesis concludes that there is no correct way of implementing microservices. The microservice architecture only provides guidelines to designing a system. An organization’s microservice design can only be correct if the organization achieves its goals and attains business value from the implementation of the architecture.

**Key words:** Microservices, architecture, services, microservice adoption
Concept Definitions

OOP - Object-Oriented Programming
MVC - Model-View-Controller
SOA - Service Oriented Architecture
SOAP - Simple Object Access Protocol
ESB - Enterprise Service Bus
DLLs - Dynamic-link libraries
HTTP - Hypertext Transfer Protocol
REST - Representational State Transfer
API - Application Programming Interface
URL - Uniform Resource Locator
CRM - Customer Relationship Management
UI - User Interface
DevOps - Development and Operations
SSH - Secure Shell
VM - Virtual Machine
DDD - Domain Driven Design
XML - Extensible Markup Language
BPEL - Business Process Execution Language
WSDL - Web Services Definition Language
# TABLE OF CONTENTS

ABSTRACT .................................................................................................................................

CONCEPT DEFINITIONS .............................................................................................................

1 INTRODUCTION ......................................................................................................................1

1.1 PURPOSE ..............................................................................................................................1

1.2 EVOLUTION OF APPLICATION DEVELOPMENT ..............................................................2

  1.2.1 Monolithic architecture .................................................................................................2

  1.2.2 Object Oriented Programming ......................................................................................4

  1.2.3 Service Oriented Architecture ......................................................................................5

  1.2.4 New Market Demand .....................................................................................................5

  1.2.5 Microservices to the rescue ..........................................................................................6

2 MICROSERVICES ....................................................................................................................7

2.1 INTRODUCTION ....................................................................................................................7

2.2 DEFINITIONS ........................................................................................................................8

  2.2.1 Definition 1: ...................................................................................................................8

  2.2.2 Definition 2: ...................................................................................................................8

  2.2.3 Definition 3: ...................................................................................................................8

2.3 CHARACTERISTICS OF MICROSERVICES .........................................................................8

  2.3.1 Single purpose ...............................................................................................................8

  2.3.2 Encapsulation ...............................................................................................................9

  2.3.3 Ownership ....................................................................................................................9

  2.3.4 Autonomy .....................................................................................................................10

  2.3.5 Multiple versions .........................................................................................................10

2.4 MONOLITHS VS MICROSERVICES ....................................................................................11

2.5 SOA VS. MICROSERVICES ................................................................................................12

  2.5.1 Service Taxonomy .........................................................................................................12

  2.5.2 Service Granularity .......................................................................................................14

  2.5.3 Service Ownership and Team Coordination .................................................................14

  2.5.4 Component Sharing ......................................................................................................16

2.6 WHY USE THE MICROSERVICE ARCHITECTURE ..........................................................17

  2.6.1 Faster Time-to-Market .................................................................................................17

  2.6.2 Better and Less Complex Code ....................................................................................18

  2.6.3 Accountability .............................................................................................................18

  2.6.4 Enhanced Domain Expertise .......................................................................................18

  2.6.5 Security .........................................................................................................................19

2.7 DISADVANTAGES OF MICROSERVICES ..........................................................................19

  2.7.1 Outer complexity is More Difficult ...............................................................................19

  2.7.2 Organizational Maturity ...............................................................................................20

  2.7.3 Duplication ....................................................................................................................21

  2.7.4 Eventual Consistency ...................................................................................................21

3 QUALITY ATTRIBUTES OF MICROSERVICES .................................................................23

  3.1 AUTONOMY AND INDEPENDENCE ..................................................................................23

  3.2 COUPLING AND COHESION..............................................................................................23

  3.3 RESILIENCE AND AVAILABILITY .....................................................................................24

  3.4 MODIFIABILITY AND MAINTAINABILITY .........................................................................24
4 FACTORS TO CONSIDER WHEN ADOPTING THE MICROSERVICE ARCHITECTURE

4.1 STRATEGIC GOALS ................................................................. 27
4.2 TOOLS AND PROCESSES .......................................................... 28
4.3 ORGANIZATION STRUCTURE AND CULTURE .............................. 30
4.4 BUSINESS DOMAINS AND BOUNDED CONTEXTS ......................... 31
4.5 THE SIZE OF A SERVICE (GRANULARITY) .................................... 32

5 HOW TO ADOPT MICROSERVICES ............................................ 34

5.1 BUILD NEW ...................................................................... 34
  5.1.1 Start Big ..................................................................... 34
  5.1.2 Start Small ................................................................... 35
5.2 EXTEND THE MONOLITH ......................................................... 36
5.3 DECOMPOSE THE MONOLITH ............................................... 37
  5.3.1 Stop digging ................................................................ 38
  5.3.2 Split frontend and backend ................................................. 39
  5.3.3 Extract services .............................................................. 39

6 STUDY CASE: MODERNIZATION OF VIVO’S SYSTEM ................... 41

6.1 CASE STUDY APPROACH .......................................................... 41
6.2 SCENARIO DESCRIPTION .......................................................... 42
6.3 WHY ADOPT MICROSERVICES .................................................. 42
6.4 ADOPTION APPROACH AND PROCEDURE ................................ 44
  6.4.1 Setting Strategic Goals ....................................................... 44
  6.4.2 Extracting Services ............................................................ 45
  6.4.3 Creating teams ................................................................. 46
  6.4.4 Inner architecture ............................................................. 47
  6.4.5 Outer architecture ............................................................. 48
  6.4.6 Current status ................................................................. 49
6.5 CHALLENGES ENCOUNTERED ................................................... 49
6.6 ACHIEVEMENTS .................................................................. 50
6.7 QUALITY ATTRIBUTES ATTAINED .......................................... 51

7 CONCLUSION ........................................................................ 53

REFERENCES ............................................................................. 56

APPENDIX .................................................................................. 59
1 Introduction

The world is ever changing in terms of how we do businesses, perform operations and carry out processes. The internet has become the driving and dominant force behind present-day businesses and organizations. Consequently, changing how we design and build systems is necessary in order to support these processes. At the inception of a software project, software engineers have every intention to build applications that will be relevant and functional for the foreseeable future. Unfortunately, applications have the tendency of increasingly becoming complex as they evolve over time. Instinctively, software engineers become aware of the rigidity and fragility of the application. For instance, making a small change to the structure of an application can introduce undesirable effects that ripple through the entire application. Complexity and the need to adapt to the ever-changing world introduce development challenges that demand for innovative solutions and prudent decisions during the architectural design stage.

1.1 Purpose

The microservice architecture has emerged as a paradigm that focuses on dividing a system into several services that are small, independent and related to a particular business function. The focus of this thesis is to discover how large-scale software development can be supported by the microservice architecture for minimal complexity and easy maintainability. In addition, this thesis aims at looking into the best approaches of adopting the Microservice architecture, especially where a monolithic application already exists. It achieves this by performing an analysis of research studies that focus on the challenges of monolithic applications and identify solutions provided by appropriate adoption of the microservice architecture.

The microservice architecture has received a substantial amount of attention from practitioners in the software development industry. The architecture is being highly recommended by product design companies and consultancy firms and has been successfully implemented by big companies such Amazon and Netflix. Although this approach of architecture has a great deal of benefits, the challenges of adopting the
architecture are considerably complex. The motivation of this thesis is derived from a limitation of research studies performed on the microservice architecture in order to provide guidance to stakeholders who may be willing to adopt this approach despite its challenges.

1.2 Evolution of Application Development

The architecture of a system is what enables it to evolve, adapt and be flexible to change over its lifetime and therefore provide standard and efficient service. A well-designed architecture accommodates frequent changes all through the development and maintenance phases of a system. For several decades, software engineers have been working on developing architecture designs that allow systems to meet both functionality requirements and quality attributes requirements. Over the years, software engineers have been able to device architectures, methods and patterns that support the development of flexible systems. (Dragoni, et al., 2016)

1.2.1 Monolithic architecture

Monolithic architecture is a traditional approach of software design where all of a system’s modules or services are contained in one application which is deployed as a single artifact. Figure 1-1 below shows an example of a monolithic application. It contains several business logic services in one application which is packaged and deployed as a single war file.
The monolithic architectural approach is best suited for certain goals and particularly convenient for small applications. It is not only easy to develop, but also easy to test. The deployment of a monolith involves creating a single artifact and moving it to the server, a process that is relatively simple. Scaling is easily achieved by replication of the application on various servers and the use of a load balancer for traffic distribution. (Karbuja, 2016)

Over time, as the application grows, the code base becomes larger and development challenges arise among them being:

Software complexity: Due to changing user demands and requirements, software is constantly changing and its functionality is expanding. As the monolith’s code base becomes larger, the application becomes increasingly difficult to evolve and maintain due to its complexity. (Dragoni, et al., 2016)

Minimal Agility: The development of a monolith involves all teams working on the same code base. The introduction of a new feature requires that the task be distributed to different teams and the features be integrated into the system. Furthermore, deploying
a small change in the application requires the whole application to be deployed. The entire process of adding even the smallest feature and delivering it into production can take a long time. Slow continuous delivery can make a company miss a market opportunity. (Karbuja, 2016)

**Fragility:** In a monolithic application, modules tend to be tightly coupled and over time inter-dependencies build up. Making simple changes to one module in the application can trigger changes in another dependent module causing the entire system to break. (Dragoni, et al., 2016)

**Limited Scalability:** A Monolith typically offers many services within a single application. The challenge arises when some services are on high demand and they need to be scaled. The conventional strategy of managing increased traffic in a monolith involves replication of the application across servers and the use of a load balancer to distribute traffic. Since all services are in a single application, the entire application has to be scaled and therefore a large amount of server resources is utilized even though not entirely needed. (Karbuja, 2016)

**Technology lock-in:** Decisions on technologies to be used for development of a monolithic application are made in early stages during requirements analysis. Teams find themselves in a technology lock-in where they are compelled to use the same language and framework for the entire life-cycle of the project. With ever evolving requirements and new features, the technology stack may no longer offer the best solutions. (Karbuja, 2016)

### 1.2.2 Object Oriented Programming

The challenges associated with large scale software development are not a new phenomenon. Software Engineers have experienced these challenges since the 1960s.
The field of software design underwent intense research in a bid to find techniques of managing large scale software development. An abundance of research in the practical adoption of software architecture and its concepts in software development led to the advent of Object-Oriented Programming (OOP) in the 1980s. The OOP design paradigm provides several approaches on translating software design into code by use of a group of persistent solutions, a concept known as patterns. A considerable example of an OOP architectural design pattern that is frequently used by developers to manage large scale software is the Model-View-Controller (MVC) pattern. (Dragoni, et al., 2016)

1.2.3 Service Oriented Architecture

The concept of separation of concerns is a design principle that is highly emphasized in OOP and has gained a lot of consideration as a profoundly recommended concept for designing large scale systems. Component-based development has emerged as a consequence, providing greater management capability and easy maintenance for complex systems. The Service Oriented Architecture (SOA) emerged from a combination of the concepts of OOP and componentization of applications. In SOA, an application is divided into several parts known as services. A service provides functionality that is accessible to other services through various protocols such as Simple Object Access Protocol (SOAP). A communication system such as the Enterprise Service Bus (ESB) is used. (Dragoni, et al., 2016)

1.2.4 New Market Demand

Increased consumer demands and expectations have compelled companies to reorient themselves in order to thrive and survive in today’s consumer driven market. Consequently, the adoption of cloud computing by companies has become imminent. Cloud computing facilitates the deployment of applications in a manner that can scale computing resources, enable hot deployments and allow continuous delivery. The need for companies to have their applications running on the cloud has been accelerated by the growing demand for efficiency in operations and high availability. For the same
reason, companies need to innovate as fast as possible and therefore there is an increased need for continuous deployment. While adopting cloud computing, most enterprises’ attempt at deploying monolithic applications has resulted to major challenges and inability to take full advantage of cloud computing power. (Goetsch, 2017)

1.2.5 Microservices to the rescue

The monolithic approach of developing software was favorable in the era of physical storage and websites. Nowadays, systems have become extremely large so that deploying them as a single artifact is impractical. A good example is to imagine amazon.com being deployed as a single war file. Such an exercise would be inconceivable. In order to overcome the challenges associated with deploying monoliths to the cloud and to take full advantage of cloud computing capacity, the microservice architecture has emerged as a further iteration of the SOA with the aim of reducing the size of applications and thus lessening their complexity. The first iteration was focused on the idea of encapsulation and componentization while the latest iteration is focused on independent development and deployment. The microservice architecture provides a guideline for the design and implementation of an application as a set of small distributed services. (Goetsch, 2017)
2 Microservices

2.1 Introduction

Even though the microservice architecture has only recently emerged as a popular architectural style, the concepts behind the architecture have been around for decades. For instance, these concepts have been implemented in the Unix operating system:

1. A program should do one thing and do it exceptionally well. The program should handle only one responsibility. When a new job arises, create a new program to avoid complicating the old program with an additional responsibility.
2. Anticipate that the output of a program will be the input of another, yet unknown program. Let the output be in its simplest form and avoid adding irrelevant information to it. Input formats should be kept simple.
3. Design software with the intention of developing and releasing within a short time, preferably a few weeks, in order for the software to be tried and tested. Be flexible to throwing away inept parts or rebuilding programs.

—Doug McIlroy, one of the founders of Unix and inventor of the Unix pipe, 1978

Prominent server-side development languages such as Java, Python and C++ are equipped with abstraction capabilities that minimize complexity of applications. However, the frameworks for these languages are intended to build monoliths which produce single executable artefacts. With time, as monolithic applications begin to grow, they do not follow the design concepts mentioned above and therefore encounter major development challenges associated with complexity. For decades, software engineers and designers have been actively investigating ways of breaking down the complexity of monoliths. Their attempts led to the invention of Unix pipes then to dynamic-link libraries (DLLs) followed by Object Oriented Programming and finally to Service Oriented Architecture. It is because of these research efforts and advances in software design and development methodologies that the microservice architecture has emerged and gained popularity as a reliable choice of architectural style. (Goetsch, 2017) (Alshuqayran, et al., 2016)
2.2 Definitions

The following are some of the definitions of microservice architecture that have been given by several early adapters of the architecture.

2.2.1 Definition 1:

"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." (Karbuja, 2016)

2.2.2 Definition 2:

"Microservices are individual pieces of business functionality that are independently developed, deployed, and managed by a small team of people from different disciplines." (Goetsch, 2017)

2.2.3 Definition 3:

"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." (Lewis & Fowler, 2014)

2.3 Characteristics of Microservices

2.3.1 Single purpose

This characteristic is based on the principle that a program ought to do one thing and it should do it well. Monolithic applications provide multiple services within a single code base that could contain millions of lines of code. A commerce application, for example,
would handle carts, orders, inventory, products and prices in the same application. With the microservice architecture, each service handles a single responsibility. It is generally expected that the code base of a service will instinctively grow over time to accommodate more functionality as the business evolves. However, bloating and increased complexity can be avoided by delegating the responsibility of development of each service to small teams of 2 – 15 developers. If this number needs to be exceeded, then that is a sign that the microservice is doing too many things and should probably be split into two. A smaller team of developers is also more likely to maintain focus and ensure that the microservice stays within its goal of handling one responsibility. (Goetsch, 2017)

2.3.2 Encapsulation

Microservices should own their data and hide their implementation. Each microservice should have its own data storage volume and should keep it private. A microservice's persistent data can only be accessed via its clearly defined API. The drawbacks of sharing data volumes and access to them is tight coupling, in that, your microservice's availability becomes directly dependent on a database that is managed by another microservice. Encapsulation ensures loose coupling between services and maintains explicit dependencies therefore, avoiding unwarranted complexities. Another major benefit of encapsulation is the fact that any changes made to a particular service's database does not interfere with any other service. (Goetsch, 2017)

2.3.3 Ownership

A large monolithic application often has a large team of developers working on it. For example, an application consisting of tens of millions of lines of code may have a team of around 100 developers. An individual developer will contribute only one percent of the application. As such, the application is generally viewed as a black box that no one fully understands and there is seldom any incentive to take responsibility for it. In other words, there is largely no feeling of ownership among team members. The application
runs the risk of becoming more complex due to developers acting in their own interest and not for the betterment of the application. (Goetsch, 2017)

The microservice architecture thrives mainly due to ownership. Organizing developers around microservices leads to small autonomous teams of 2-15 members who are responsible for the microservice from its development stage all the way to its deployment. The team is independent and makes its own decisions, therefore increasing motivation and efficiency. The team members also develop an empathetic relationship with the users of their service and a commitment to their success. (Goetsch, 2017)

2.3.4 Autonomy

Ownership is dependent on autonomy. As mentioned above, a team responsible for a microservice is independent and should be able to design, develop, deploy and maintain a microservice in complete isolation without the need for coordination with another team. The team is fully responsible for the outcome of the microservice and therefore has the freedom to make decisions on all matters tools, technologies and methodologies. Different microservices solve different business problems. Each team should be able to select a technology stack best suited to meet their particular need. Understandably, organizations have the need to standardize some outer details such as API protocols, alerting, messaging and logging but not necessarily the product that support the implementation. Each microservice is required to expose only its API; hence the internal implementation technicalities are of no major consequence. (Goetsch, 2017)

2.3.5 Multiple versions

The ability to make more than one version of a microservice deployment at the same time in the same environment is yet another distinguishing characteristic of the microservice architecture. For example, it is possible to have version 1.9, 2.0, 2.1 and 2.2 of the products microservice all running in the production environment at the same time. This is made possible by the use of URL, for example (/products/v1.9 or products?version=1.9). Clients can therefore make http requests to a particular version of the microservice. This is particularly beneficial for companies that are keen to
promptly release minimum viable products (MVPs) to their clients. The microservice architecture's capability to support multiple versions is an exceptional strength, one that other architectures such as the monolithic architecture do not offer. (Goetsch, 2017)

### 2.4 Monoliths vs Microservices

The monolithic architecture has been extensively explained in the previous chapter, detailing its benefits and caveats. The monolithic architecture is structured totally differently from the microservice architecture in terms of both technological and organizational aspects. The table below summarizes the most significant differences between the two architectures.

<table>
<thead>
<tr>
<th>Software monolith</th>
<th>Microservices</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single application</td>
<td>Array of many small services with limited functionality</td>
</tr>
<tr>
<td>Entire application needs to be deployed</td>
<td>Microservices can be deployed separately</td>
</tr>
<tr>
<td>One datastore for entire application</td>
<td>Each Microservice has its own datastore.</td>
</tr>
<tr>
<td>Communication within application</td>
<td>Remote calls, usually REST calls via HTTP</td>
</tr>
<tr>
<td>Separation between developers and ops</td>
<td>Cooperation of developers and ops to maintain stable operations</td>
</tr>
<tr>
<td>State lies in external application at runtime</td>
<td>States are stored centrally, individual instances are stateless</td>
</tr>
</tbody>
</table>

Table 2-1: Major differences between monolithic and microservice architectures


2.5 SOA vs. Microservices

From a general definition point of view, microservices and SOA (Service Oriented Architecture) appear similar. They both put emphasis on dividing large systems into distributed services. The services become the primary components of the architectures and for the implementation of system and business functionalities. From this perspective, it is difficult to distinguish between them. We shall demonstrate the differences between microservices and SOA by comparing the characteristics of their services and their capabilities.

2.5.1 Service Taxonomy

Service taxonomy relates to the classification of services in an architecture pattern. There are several ways of classifying services. Services can be classified according to the tasks they perform in the architecture, for example, some services implement business functionalities while others implement non-business functionalities such as security or logging. (Richards, 2016)

![Service Taxonomy Diagram](image)

Figure 2-1: Service taxonomy in microservice architecture
The service taxonomy of the microservice architecture is limited to only two service classifications namely functional services and infrastructure services. Functional services consist of services that implement and support particular business functions or operations. On the other hand, infrastructure services perform supporting tasks such as authorization, authentication, monitoring and logging. Important for this comparison is the fact that infrastructure services in microservices are not accessible outside the architecture and are only shared internally amongst services. Functional services are accessible to the outside world via an API implementation. (Richards, 2016)

Service taxonomy in the SOA architecture is formal and distinctively classifies its services into four types as shown in Figure 2-2 above. Business Services are course-grained and high-level services that define the main business operations. Their representation is generally through XML, Business Process Execution Language (BPEL) or Web Services Definition Language (WSDL). Enterprise services are course grained but concrete services that implement the business functionality defined by business services and are usually shared amongst other services in the architecture. Enterprise services are dependent on infrastructure and application services in order to complete requests. Application services are fine grained services that implement business functions that are very specific in the application context. Enterprise services
can be accessed through a user interface or via an enterprise service. Infrastructure service, just like in the microservice architecture, are used to implement non-functional supporting tasks such as auditing and monitoring. (Richards, 2016)

2.5.2 Service Granularity

The service granularity between the two architectures is significantly different. As the prefix ‘micro’ in microservices suggests, the services in the architecture are fine-grained and small. They are single purpose services and aim to do only one thing and do it well. Their scope is limited to the implementation of a single business function. On the other hand, the size of SOA services ranges from fine-grained application services to large, course-grained enterprise services. It is not unusual to come across an SOA service that represents a large business product or even an entire subsystem. SOA depends on several services to execute an individual business request. (Richards, 2016)

2.5.3 Service Ownership and Team Coordination

Service ownership refers to a group in the organization that is responsible for designing, developing and maintaining a service. Due to the fact that microservices have a smaller service taxonomy (infrastructure services and functional services), development teams in this architecture may have ownership of both the infrastructure and functional services. Teams do not specialize in the development of one type of service but rather consist of experts across different fields and are therefore self-sufficient. (Richards, 2016)
On the other hand, services across the SOA architecture have different owners. Business services tend to be owned by the business users in the organization. Architects and shared services teams own the enterprise services. Application services are the responsibility of application development teams while infrastructure services could be owned by either the application development team or the infrastructure development team. Figure 2-4 below shows service ownership in SOA architecture.
The relevance of service ownership becomes apparent in the overall coordination of services. In SOA, the development of a complete business request requires coordination among several teams. The constant need for consultations across service level owners impedes fast development. In microservices, there is minimal need for coordination of teams or services in order to fulfill a single business function. Coordination across teams, if required, is achieved by the creation of small teams that can quickly develop, test and deploy services. (Richards, 2016)

2.5.4 Component Sharing

SOA is fundamentally built to maximize component sharing by using the enterprise level shared services. While component sharing in SOA helps in reducing duplication of functionalities, it leads to tight coupling among components, thus increasing the risks associated with making changes. In distributed architectures, sharing of some services will always be inevitable, an example being infrastructure services. However, while SOA tries to share as much as possible, microservices try to minimize component sharing by using the concept of ‘bounded context’. Services are therefore self-reliant and contain minimal dependencies. (Richards, 2016)
2.6 Why use the Microservice architecture

2.6.1 Faster Time-to-Market

The Microservice architecture offers the capability to make independent deployment of smaller services. This is the most significant advantage of microservices since it enables rapid release of new and innovative features to users and customers. The ability to promptly get an innovative feature into the market can be of substantial benefit to your company. With monoliths, a new feature comes to completion after several iterations, usually four iterations. Consequently, a release to production often occurs quarterly, a
considerably long time for both the consumers and the business. The consequences of late delivery to the market can be detrimental to a business. (Goetsch, 2017)

2.6.2 Better and Less Complex Code

By nature, microservices are small in size and the scope of the service is often limited to a particular business function. The quality of code is usually better since it is owned by a small team that is completely responsible for the microservice. As a result, the code tends to be less complex and of better quality. While monoliths might have up to tens of millions of lines of code, it follows that the development, maintenance and updating of a microservice is easier compared to that of monolithic applications. (Goetsch, 2017)

2.6.3 Accountability

The adoption of the microservice architecture requires that teams be organized around microservices. As such, small teams of about 2-15 developers are created for each microservice. The team has freedom to make choices concerning architecture, technology and tools for the microservice, therefore creating a sense of ownership on the part of team members. Due to the microservice's limited scope of functionality and the size of the team, every developer is involved in the project from its inception to its production stage. This develops a sense of accountability and responsibility among members of the team. (Goetsch, 2017)

2.6.4 Enhanced Domain Expertise

The microservice architecture organizes services around business capabilities. For example, an online shopping system would have different microservices for products, inventory, pricing, orders, promotions, carts and catalog. With small teams created around the business functionalities, each team will include one or two domain experts. These experts have extensive domain-specific expertise and experience and are responsible for the features and functionality. They collaborate with developers in order
to develop very deep domain-specific functionalities and algorithms that support the business function. (Goetsch, 2017)

2.6.5 Security

With large monoliths, security is oftentimes an afterthought (Goetsch, 2017). The application is usually developed first and then security is implemented later, generally only in the UI. Usually, developers do not implement security features due to complexity of the code. Microservices on the other hand are quite different. They use APIs to expose only small parts of their functionality. All requests pass through the API gateway or the load balancer where they are evaluated to check if they can be passed on to the API endpoint. The APIs implement permissions and roles that allow only authorized users to access information. For example, an admin user in an organization is allowed to send HTTP POST, GET, UPDATE and DELETE requests to the customer’s microservice API. A normal user in the same organization is only allowed to make HTTP GET requests only. It is also possible to track who or what is sending requests to each API. (Goetsch, 2017)

2.7 Disadvantages of Microservices

While the advantages of the microservice-based approach are intriguing, it is worthwhile to note that it might not be suitable for every business context. There are no distinct pros and cons between microservices and monoliths that support opting for either of the architectures; adopting either of them involves a trade-off.

2.7.1 Outer complexity is More Difficult

Microservices provide coherent advantages when it comes to large scale software development: Division of business functionality into smaller services leads to reduced complexity hence supporting easy development of new features. However, the adoption
of microservices does not result in magical vanishing of a system's development and maintenance complexity but is often viewed as a tradeoff between inner and outer complexity. Microservices make complexity more visible and unambiguous hence facilitating its proper handling and management. For example, in microservices, division of business functionality into smaller services often means packaging and deployment of 100 services instead of 5 modules. In the context of deployment and operations, a monolith is advantageous in that it can be deployed as a single artefact. With microservices, a container orchestration system, service discovery and load balancer are required to ensure correct deployment, communication and management of the services. This represents a total shift in complexity from inside the services to the connections between them and their management. (Goetsch, 2017)

2.7.2 Organizational Maturity

Organizations that are looking forward to adopting the microservices architecture must have a mature organizational structure and culture. In what has come to be known as the Conway's Law, Melvin Conway wrote:

Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure. (Bailey, Godbole, Knutson, & Krein, 2013)

The structure of an organization determines the kind of software they create. Centralized organizations structured along layers often focus on the technology layer and establish teams based on technical competencies. Eventually, the organization is comprised of a UI team, a database team, application logic team and the operations team. Simple changes to the system therefore require cross-layer coordination and approval which takes a lot of time and is costly. (Goetsch, 2017)

The microservice approach to organizational structure necessitates the creation of small teams structured along business capability. The small teams comprise of members with different technical competencies necessary for the implementation of a specific business function throughout its life cycle; from its design to
deployment stages. The adoption of the DevOps culture has emerged as a way of breaking the boundaries that exist between development and operations teams. In practice, every team is responsible for both the development and operations tasks for their microservice. The organizational structure and culture should be driven by the importance of speed in development and the value of being reactive to the ever-changing market. (Goetsch, 2017)

2.7.3 Duplication

Considering that each microservice is independently deployable by nature, the microservice architecture allows for autonomous teams that have freedom to make decisions on the technology stack that is best suited for their business functionality. Eventually, an organization could have too much technology diversity for each microservice and could become overwhelmed. Some organizations end up providing a limited technology set to choose from.

The other form of duplication appears in the instances of the products and services. With monoliths, only one instance of the database is shared among modules. The monolith is also deployed as a single artefact. On the other hand, the microservice architecture requires that each service have its own instance of the database. One of the biggest advantages of the microservice architecture involves the ability to scale services based on their demand. This allows for the deployment of more than one instance of each microservice. This might not be technically efficient but the eventual advantage of the architecture is speed. (Goetsch, 2017)

2.7.4 Eventual Consistency

The microservice architecture insists on the use of distributed data management which involves each service having its own database. This introduces the issue of eventual consistency where several copies of a record from different timestamps exist in the databases of different services. In monolithic applications, the existence of only one centralized database ensures data consistency such that when data is written to the database, it immediately becomes visible to the whole application. However, as the
application grows, the need for caching increases in order to improve performance. Cache validation eventually becomes a problem and data consistency can no longer be guaranteed. Monoliths are therefore not completely exempted from eventual consistency problem although they suffer less of it due to centralized data management. A similar data model is not efficient for microservices since it introduces strong coupling and interferes with development speed. The adoption of the microservice architecture means embracing eventual consistency. (Goetsch, 2017)
3 Quality Attributes of Microservices

3.1 Autonomy and Independence

Autonomy refers to the ability of a service to be self-contained, self-controlling and self-governing with an explicitly defined boundary (Rostampour; Kazemi; Shams; Jamshidi; & Nasirzadeh, 2011). The division of a microservice application into smaller services enables the services to be designed, developed and deployed independently. Independent deployment in turn means that each service in the application is operated independently from other services. Communication with other services occurs through their published interfaces. If a service becomes obsolete or deprecated, it is generally easy to remove the service from the application without a hustle and without breaking other services. Autonomy is one of the fundamental considerations applied when determining how a microservice architecture will be broken down into services. The autonomy of a service has a significant effect on other attributes of the service such as reusability, maintainability and discoverability. Additionally, it greatly influences the effectiveness of a service as part of a greater microservice application. (Rostampour; Kazemi; Shams; Jamshidi; & Nasirzadeh, 2011)

3.2 Coupling and Cohesion

The ultimate objective of most microservice applications is to provide for independent deployment of services which in turn increases the speed at which the system evolves and hence faster time to market. To achieve this, the application has to implement bounded service contexts enforced by clearly defined boundaries. Coupling and cohesion are the most fundamental attributes for establishing service boundaries. The two attributes also determine the correlation and interdependence between the services.

Quynh & Thang define coupling as the degree to which each service relies on each one of other services. According to their research, since coupling is the degree of interdependence between services, the degree can be high or low depending on the
strength of association or bonding. They explain that the software complexity of a system will be dependent on the strength of the interconnection and interdependency between components. Low coupling is therefore the desired quality attribute of a service. They define cohesion as the degree to which the components of a service belong together and therefore, it is a measure of how strongly related all elements of a component are to each other. The advantages of high cohesion include reduced complexity and increased maintainability. These attributes are enhanced by the fact that logical changes to a component will require minimal changes to other components. Increased reusability of components caused by high cohesion enables system maintainability. (Quynh & Thang, 2009) (Newman, 2015)

3.3 Resilience and Availability

Resilience refers to the ability of a system to withstand failure while availability is the ability of a system to be accessible and operational when needed for use. The microservice architecture is designed and built for failure. The architecture achieves this by isolating and containing failure in each service. When an application is composed of many different services, there is no single point of failure. If one service goes down, it does not break the entire application and other services are not affected. During the time of failure, the system is responsive and the recovery is managed independently and with care so as not to affect other services. Ensuring that parts of a system can fail and recover without breaking the whole system guarantees high availability for the system. (Karbuja, 2016)

3.4 Modifiability and Maintainability

While the relatively huge and complex monolithic applications may contain hundreds of thousands up to millions of lines of code, the microservice architecture contains very small services that can be comprised of about a thousand lines of code. A smaller code-base greatly reduces complexity and makes it easier to maintain the code. Services are
also autonomous, loosely coupled and accessed via API endpoints only. These characteristics promote explicit dependencies between services and in turn reduced complexity. The reduced costs of modifying a micro service-based system increases its maintainability. However, if API endpoints need to be changed, it might require more work since it can be challenging to find all the services that use this particular service endpoints and the effects these changes might have on them. (O'Brien;Bass;& Merson, 2005)

3.5 Scalability

Scalability refers to the ability of a Microservice-based system to be enlarged in order to accommodate increasing user demand without degradation of the system’s performance or other quality attributes. Microservices are small, independent and offer different services. Due to user behavior and preferences, certain services receive more user traffic over time, thus the need for additional resources. When a particular service becomes more popular than others, a microservice-based system is able to create multiple instances of this individual service due to the fact that microservices are small and independently deployable. The existence of a load balancer in the system’s architecture also makes sure that traffic is evenly distributed to the service instances. Ultimately, an increase in user demands does not require scaling the whole system. The microservice architecture effortlessly allows expanding of the system by creating additional services for popular services only, thus avoiding wasting of resources. (O'Brien;Bass;& Merson, 2005)

3.6 Performance

Performance as a quality attribute can be regarded from different perspectives. Generally, it is associated with timeliness and the ability of the software to meet timing demands in terms of response time, throughput and timeliness. Response time refers to the time a software takes to process a request while throughput refers to the number of
requests that can be processed per unit time. Timeliness refers to a software’s capability to meet deadlines in a tolerable amount of time. Performance is fundamentally one of the most important quality attributes of a software. Unfortunately, it is negatively affected in relation to the microservice architecture.

There are two major factors that affect performance in microservices. Firstly, the microservice architecture requires that services be deployed independently in order to achieve autonomy and maintain resilience. This means that services are deployed in different containers and at times in different machines. The need for services to communicate over the network arises, consequently leading to network latency issues and response time uncertainties. Communication over the network will tend to degrade the performance of a micro service-based software compared to other applications that use in-memory call technique. As a result, the microservice architecture is not suitable for real-time systems since timeliness is a safety critical requirement.

Secondly, the location of service end-points might not be known beforehand by consumer services. This means that for a consumer service to interact with a producer service, the consumer service has to make a pre-call to a service directory in order to get the location of the producer service. This additional call to the directory service increases the amount of time required to process the request. A solution to reducing the response time in this case could involve elimination of the call to the service directory by hard coding the location of the provider service. Unfortunately, this approach reduces scalability and availability since it prevents the deployment of replicas to multiple locations and hinders moving of deployed services from one location to another without affecting the users. (O'Brien; Bass; & Merson, 2005)
4 Factors to consider when adopting the Microservice architecture

Much has been mentioned about the advantages and disadvantages of microservices in the previous chapters. The benefits and quality attributes achieved by the implementation of the microservice architecture are unquestionably appealing. While the case for the microservice architecture might be convincing in theory, it is rarely as simple in practice. The microservice architecture eventually ends up being more complicated than other architectures. This is because it is a distributed system and distributed programming is challenging. As mentioned in Chapter 2, deployment, communication and management of hundreds of services, as required in the microservice architecture, shifts complexity outwards.

Even though the microservice architecture is a highly recommended approach, its appropriateness for your organization should be comprehensively assessed. An analysis of the business problem this new architecture solves for your organization ought to be deliberated in order to ascertain the motivations behind its adoption and to determine whether the organization is ready for microservices. Given the complexity of microservices and their implementation, it is beneficial to consider the following factors not only while assessing your organization’s ability to adopt the microservice architecture but also while creating a roadmap for the adoption process. (Commercetools Inc.), (Drake, 2017), (Jedrzejewski, 2018)

4.1 Strategic goals

As previously mentioned, it is crucial for an organization to acquire clarity on the motivation behind their intentions to adopt microservices and what business problems will be solved by adopting the microservice architecture. Product agility, for instance, is one of the standards that most customer centric organizations aspire to achieve. Surviving and thriving in today’s era of commerce largely depends on an organization’s ability to meet consumer demands within the shortest time possible. Speed has become
the leading competitive advantage for companies. For these organizations, it is becoming significantly important to quickly release new features to the market and rapidly deploy updates or patches in order to give consumers maximum experience. (Commercetools Inc.)

The ability to scale micro services with growing traffic is another significant incentive for an exceedingly large number of organizations to adopt the microservice architecture. Organizations often run into the challenge where their monolithic systems can no longer accommodate growing traffic due a limited number of requests they can handle. Fast growing organizations need to anticipate for growth and be resilient enough in the face of massive traffic hikes. The microservice architecture has the ability to offer numerous business benefits to organizations. However, not all organizations will need all the benefits and not all microservice architecture designs will provide all of those benefits. It is therefore crucial for an organization to determine whether its business goals match with the benefits of microservice architecture in order to come up with a design that will derive maximum business value from the architecture. (Commercetools Inc.)

4.2 Tools and Processes

A microservice system is not just a composition of service components and their communication channels. It is also a byproduct of the tools and processes chosen for its development, maintenance and deployment. The selection of suitable tools and processes is therefore a paramount factor in developing a microservice system that meets the organization’s strategic goals and contains desirable quality attributes. (Goetsch, 2017), (Nadareishvili, Mitra, McLarty, & Amundsen, 2016)

One of advantages of the microservice architecture is its ability to be a polyglot architecture. Polyglot, in this context, basically means the use of many programming languages for development of different microservices. While monolithic applications are entirely built using a single set of tools, the microservice architecture technology ecosystem is not dependent on distinct technology stacks. Every team has an
opportunity to choose the best and most suitable tools and technologies for the implementation of their micro service. This is known as inner architecture since its focus is limited to the inner operations of a particular micro service. (Goetsch, 2017)

According to Goetsch K., the outer architecture is the space between individual microservices and is the most complex part of the microservice architecture. It consists of the infrastructure onto which the respective microservice containers are deployed. It is also the environment where service discovery and communication between services takes place. For the successful operation of the outer architecture, cloud platform is a necessity. This is because the cloud enables for rapid and automated deployment of Docker containers which in turn allow each microservice to utilize its platform, infrastructure and software in its entirety. Even though every microservice team has the freedom to choose its inner architecture technology stack, the outer architecture requires that all microservices be confined in one cloud provider. This enables all teams to access shared resources such as service discovery, API gateways and messaging. (Goetsch, 2017)

Optimal building of the microservice architecture requires the adoption of standardized processes such as DevOps and the adherence of agile principles. Unlike in monolithic systems where teams are divided according to skillsets, for example database team, operations team and development team, the DevOps process demands that each team contain both development experts and operations experts. DevOps refers to an approach to software project management where development (Dev) and operations (Ops) teams come together to form one microservice team (DevOps). The main purpose of the DevOps approach is to reduce the development life cycle in order to deliver new features, updates and bug fixes fast enough to market. DevOps and microservices suit each other because teams are not only responsible for development of the code but are also accountable for the operations. Teams take care of the application from its design stage to its deployment stage and continue with its maintenance using automated continuous integration and delivery. DevOps complements microservices on the grounds that it makes communication between development and operations easier. This is because members of both fields work on a single microservice project together as one team as opposed to working as different teams on the same project. Continuous
integration and deployment facilitate iterative development which is one of the core principles of the agile methodology. (Nadareishvili, Mitra, McLarty, & Amundsen, 2016), (Jedrzejewski, 2018)

4.3 Organization Structure and Culture

To find out whether your organization is ready for the microservice architecture, it is vital to take a look at the organization’s structure and its underlying culture. In his paper title “How Committees Invent”, Mel Conway, a computer scientist, stated that a software development organization will produce a software system whose design is a replica of the organization’s communication structure. An organization’s structure is related to its teams, that is, how responsibilities are divided among teams, the size of each team, whether teams are aligned to business domains or member expertise, the nature of communication among team members and between teams, the level of cross-team dependency and the power distribution across teams. (Nadareishvili, Mitra, McLarty, & Amundsen, 2016)

The optimal organization suitable for the microservice architecture has small, self-sufficient and empowered teams responsible for the implementation of individual services that align with particular business domains. Each team contains all of the skillset required for the successful implementation of a single business domain logic for example architects, developers, product owners, operational engineers and other stakeholders. Teams that do not have complete autonomy and are not empowered will come across delays as they wait for decisions from individuals at higher power positions. The scope of a team’s service should be properly aligned to a specific business domain, failure to which, will bring about cross-team dependencies leading to further delays. The size of the teams matters because too large teams produce complex code that causes difficulties during maintenance and delays any future changes. As stated in section 5.2, it is important for the organization to adopt the DevOps process since it increases the team’s motivation when they are responsible for a service’s incremental development from its conception all the way to its maintenance stage.
Finally, a team ought to have the required skills to develop API-based services using distributed computing concepts, otherwise, costs of development could escalate due to the need for training or a need to outsource. (Nadareishvili, Mitra, McLarty, & Amundsen, 2016)

The culture of an organization entails a set of ideals, values and beliefs shared among employees that form the basis of all decisions made within the system. Often, culture signifies the level of influence of other parts of your system. Culture sets the tone on how employees behave and also influences their productivity. Communication is one aspect of culture that affects productivity in an organization. Organizing workers into teams and thus communication channels, indicates that various decisions pertaining to design have already been made. The decisions made around team membership, size and even physical location of the teams and team members also affects their communication and therefore their productivity. Putting focus on communication and coordination needs of a project will enable you to form teams that will make development easier and faster. It will also guarantee easier communication among teams and between members. Fostering innovation is another important aspect of culture in organizations. Innovation seeks opportunities to improve a team, a company or a product. One of the reasons why some organizations adopt the microservices approach is the capacity to take advantage of innovative and creative ideas. Organizations can foster creativity and innovativeness by granting a higher level of autonomy to teams. (Nadareishvili, Mitra, McLarty, & Amundsen, 2016)

### 4.4 Business Domains and Bounded contexts

One of the defining characteristics of the microservice architecture is *single purpose*. While a monolith might contain millions of lines of code implementing all required business functions, the microservice architecture contains many small services each designed to model one business domain. For instance, an e-commerce monolithic application could be built to handle inventory, products, customers, suppliers, orders, prices and shopping carts. Microservices, on the contrary, requires that each of the above business domains be handled in an individual service. Dividing business
functions into separate services is done along bounded contexts. Consequently, every service does one thing only and does it exceptionally well. This makes it easier to implement new features or changes since they will only affect one microservice. Each team is assigned to work on one bounded context which serves as a basis for one microservice. This makes it possible for teams to work independently and that they maintain their focus. It also ensures that complexity stays at its minimum. (Wolff, 2017)

Sam Newman, in his book Building Microservices, states:

If our service boundaries align to the bounded contexts in our domain, and our microservices represent those bounded contexts, we are off to an excellent start in ensuring that our microservices are loosely coupled and strongly cohesive.

4.5 The size of a service (Granularity)

As mentioned earlier, microservices require that business functions be divided into distinct business capabilities in order to form small and autonomous services. One of the most challenging questions that arise from this requirement is how micro is micro. In other words, how small should a microservice be? The granularity of a service is usually vague and often viewed from different perspectives. It is difficult to suggest a single measurable value because the interpretations of granularity are subjective. For example, if we consider lines of code (LOC) as a measurement unit for granularity, we could determine that a service should have a maximum of 1000 lines of code. However, the issue with such a quantitative value is that the measure does not put into account the business context, organizational context and how the service will be used within the larger microservice architecture. (Nadareishvili, Mitra, McLarty, & Amundsen, 2016) (Karbuja, 2016)

Since it is difficult to uphold a quantitative value for the granularity of a service, it important to focus on the qualitative element of granularity. The following principles offer guidance to developers and architects as pertains to proper service granularity.
1. Autonomy and reusability should be considered when designing service boundaries. A service should support cohesion and avoid tight coupling so that changes to the service do not affect other services.

2. A potential model is an acceptable service if it is independent of the implementation but dependent on understandability of domain experts. The modelling of a service should be based on the real-world business functionalities and activities.

3. Time is an important factor that affects the granularity of a system. With time, technologies arise and others evolve to offer better support for the microservice architecture. For instance, the evolution of virtualization and container technologies has made it easier to support a vast number of services. This makes it possible to further split business functionalities and have fine-grained services.

4. A service should not support a vast number of operations since any changes would affect a large number of users. Maintaining a small number of operations in the interface helps in controlling the granularity of the service.

5. Transaction integrity and compensation should be considered and provided by the service. Every operation supported by the service should be limited within the scope of a single transaction. The service should provide compensation in the event that a transaction fails. This ensures availability and fault-recovery of the service.
5 How to Adopt Microservices

This chapter focuses on the different ways of approaching microservice adoption. The following are the three fundamental ways of adopting the microservice architecture.

5.1 Build New

In the event that an organization is creating a new and large application and has a large number of teams, then it would be appropriate to start building using the microservice architecture principles. The microservice architecture is exceptionally suitable for large projects and dynamic environments where establishing a system’s architecture in the early phases of building a system can be challenging. Microservices, however, require that the organization’s business logic be split into smaller functionalities that services will be based on. At the initial level of planning, it can be difficult to determine the criteria for separating the business domains. Below are two ways in which to approach this challenge. (Goetsch, 2017)

5.1.1 Start Big

This approach advocates for solving this problem by initially creating a few large microservices. At the beginning of a project, requirements are usually less elaborate and architects only have the bigger picture of the project. When the project consists of only a few microservices, it is easy to develop and maintain without being overwhelmed by the outer complexity that comes along with the implementation of microservices. Later, the requirements of the project become more explicit and customer demands increase. These large microservices are then gradually split into smaller microservices. Subsequently, more people can be added into the team and new teams can be gradually created to handle the new microservices. Teams can work independently and in parallel and can further subdivide their microservices into smaller ones. Figure 5-1, shows such a progressive process. (Wolff, 2017)
At the beginning of a project, it is easy to maintain the granularity of microservices since there is less complexity. However, as the code base grows, it is important that developers are perceptive of the ensuing danger of increased complexity and tight coupling. They should therefore be keen to realize when it's time to split a microservice or create a new microservice. (Wolff, 2017)

### 5.1.2 Start Small

Another way to establish microservices from business domain is to start small. This involves splitting the organization’s business logic into numerous microservices and using this as a basis for creating teams and further developing the system. The caveat to this approach is that the system architects have to be very familiar with the organization’s business logic and domains in order to create elaborate microservices. Starting off the project with many microservices could also prove to be overwhelming since each microservice has to have its own continuous delivery pipeline and all microservice deployments in the cloud have to be managed properly from the onset of the project. Figure 5-2, shows an example structure of starting small where the architects are well aware of the organization’s business domains and boundaries and
can therefore create distinct and independent microservices from the start. (Wolff, 2017)

5.2 Extend the monolith

This approach to adopting microservices requires that there is an existing monolith that has become cumbersome and difficult to support. Many organizations will find themselves in such a situation and starting a new microservice application from scratch could be unrealistic. Extending the existing monolith is a viable option in such a scenario. Extending a monolith involves creating separate small microservices which the monolith will consume.

For instance, if an organization needs to upgrade or overhaul its monolith’s inventory functionality, they could take this opportunity to create a whole new microservice for the functionality instead of making changes to the monolith. Consequently, the monolith will retrieve all inventory information from the inventory microservice. This procedure is gradually repeated for other domains and eventually all functionalities are pulled out of the monolith to remain with a complete microservice architecture. Figure 5-3, shows an example structure of adopting microservices when starting from an existing monolith. (Goetsch, 2017) (Richardson & Smith, 2016)
5.3 Decompose the Monolith

Decomposing a monolith could be the most challenging or possibly the simplest method depending on the structure of the existing monolith. A highly modularized monolith is the easiest to work with since the well-defined modules can be retrieved and independently deployed as microservices as shown in Figure 5-4 below.
However, if a monolith is tightly coupled and contains a lot of dependencies, then decomposing it can be extremely challenging. Another caveat of using this approach involves the fact that the created microservices will have to use a shared database since the monolith likely contained only one database. While decomposing a monolith can prove to be challenging, the result could be well worth it. Below are some considerable strategies to decomposing a monolith.

### 5.3.1 Stop digging

This strategy is based on the Law of Holes which states that whenever you are in a hole you should stop digging. This advice is recommended in the event that a monolith becomes too large and unmanageable and it is therefore crucial that you stop making it bigger. At this point, when new requirements arise, the functionality ought to be implemented in a separate microservice. This keeps the monolith from becoming bigger and cumbersome.
5.3.2 Split frontend and backend

The ‘stop digging’ strategy is a preventive measure but does not fix the problems related to breaking up the monolith itself. This strategy suggests shrinking the monolith by splitting the presentation layer from the data-access layer and the business logic layer. This strategy offers an opportunity for the two applications to be further developed, deployed and scaled independently. It is also beneficial to frontend developers who can now independently and rapidly develop user interface features without the need for constant coordination with backend and database teams. Rapid development of the user interface is highly valuable to an organization that needs to offer better services to its customers in order to stay ahead of competition. However, this strategy is only beneficial to an extent. It is highly likely that either the frontend or the backend application will become an unmanageable monolith and a need to further split them may arise. (Richardson & Smith, 2016)

5.3.3 Extract services

The final strategy suggests further subdividing the business logic layer to form microservices. As stated earlier, if the existing monolith is modularized, it is easiest to extract the modules and deploy them independently as microservices. However, if the monolith is not modularized, there are several ways to break it down into services. For example, an architect could decide that computational heavy modules be developed using a specific technology stack or programming language such as Java or C. Such modules will therefore be used to create independent microservices. Another way of extracting services could be on the basis of location. A subsystem could be developed by software teams in Asia while others could be developed in Africa. One of the most common reason for subdividing services according to location is that operating in a certain location might offer legal, cultural and commercial benefits.

Eric Evans in his book Domain-Driven Design, he introduces a model-centric approach to creating microservices based on an organization’s bounded contexts. Bounded contexts refer to autonomous business domains and offer a suitable basis on which to establish service boundaries. Eric Evans suggests the use of ubiquitous language used
by domain/technical experts to establish bounded contexts and thus form service boundaries. (Nadareishvili, Mitra, McLarty, & Amundsen, 2016)
6 Study Case: Modernization of Vivo’s System

One of the objectives of this thesis is to explore the practices, principles and culture that characterize the microservice architecture. This chapter introduces a scenario that seeks to answer the following questions:

1. What is the best approach to adopting microservices?
2. What are the challenges of adopting microservices?

6.1 Case Study Approach

To answer the above questions, we engage an e-commerce company in Kenya that is currently adopting the microservice architecture and, as of the date of this writing, has successfully implemented and deployed three microservices. For privacy reasons, we will not reveal the name of the company and will refer to it as ‘Vivo Group’ throughout this chapter.

The purpose of this case study is to investigate practical approaches to adopting the micro-service architecture while decomposing an existing monolithic application. The case study demonstrates the process of modernizing and e-commerce web application by modularizing it from its monolithic form into microservices.

Data collection for the case study was done by conducting interviews with Vivo employees. In addition to interviewing one of the company’s software architects, I was able to interview two software developers from development different teams. The interviews were conducted online via skype between August and September 2018. Although I had some interview questions prepared, the aim was to conduct open-ended interviews. This means that the interviews were non-structured and did not strictly follow the laid-out questions. I had the opportunity to ask elaborative questions and this led to lengthy discussions. These conversations not only provided answers to my interview questions but also helped to reveal how the participants view reality and situations. The interview questions covered the following issues: the reasons for
adopting microservices, the adoption principles and strategies, how services are extracted, structure of the architecture, team structures, deployment of services, challenges encountered and achievements.

### 6.2 Scenario Description

Vivo Group is a retail group that runs an e-commerce business whose revenue amounts to 80% of the group’s revenue. Vivo created the e-commerce platform in order to take advantage of the emergence of online shopping and the smartphone’s influence on purchases. It has an extensive web application that offers a large number of functionalities including customer registration and management, inventory, product listing and searching, ordering process and order management, pricing, shopping carts, payments and promotions. The application is based on a monolithic architecture and its deployment process involves deploying the monolith in its entirety. On the occasion that a change needs to be made or a new feature is added, the entire application is deployed afresh. The e-commerce application works side by side with other parts of the system including logistics and accounting.

### 6.3 Why adopt Microservices

The currently running monolithic application commenced as a well thought out and designed application. It performed well and efficiently met user and system requirements. Over the years, however, the application code base has expanded and evolved in order to meet changing customer demands. This has led to increased dependencies between modules making it complex and difficult to understand. Consequently, the application has progressively become difficult to maintain or update. Furthermore, the initial architecture is no longer suitable for the company’s business requirements. For instance, the product search functionality has overwhelmingly evolved as Vivo escalates its efforts towards outperforming its competitors. Vivo has also introduced a number of self-service functionalities for their clients for the purpose
of reducing overhead and therefore decreasing costs. As a result, these modules have evolved to become very large and contain complex structures with a great deal of dependencies.

**Slow Continuous Delivery Pipeline:** In order to improve their product agility, Vivo adopted continuous delivery and created a continuous delivery pipeline. Recently, this pipeline has become slow and complicated since the whole monolith needs to be tested and run on the pipeline all at once.

**Simultaneous work is difficult:** Different teams work on different new features. Separation of individual modules is not clear and there exists a lot of dependencies. Since the whole application can only be deployed all at once, only one team is able to carry out testing and deployment at any given time. Coordination among teams is therefore necessary so as to ensure smooth release of changes into production. Unfortunately, this hinders parallel work among teams and is not only time consuming but also very costly.

**The Testing bottleneck:** Before any deployment can be done, the monolith has to go through different types of testing. Integration tests require that changes made by only one team should be included in an integration test since it is difficult to determine the source of any error if many teams run their tests concurrently. Unfortunately, a single integration test could run for roughly one hour and therefore running approximately six integration tests is attainable per day. Vivo has five teams and therefore one team can commit their changes to the pipeline every two days. In a situation where there’s an error during testing, teams have to wait for the error to be solved before they can commit their changes to production. Oftentimes, teams create feature branches that are separate from the pipeline and commit their changes while waiting for the pipeline to be clear. Integrating these changes later might cause problems. Occasionally, during merging, changes are mistakenly removed or errors emerge due to the detached development
process. Such errors can take a long time to resolve and therefore impede the work of other teams.

![Diagram](image)

Figure 6-1: Deployment of monolith slows down teams

### 6.4 Adoption Approach and Procedure

Vivo’s monolithic architecture’s shortcomings became painfully slow and cumbersome to further develop and maintain. Rapid changes in consumer demands and increased competition compelled Vivo to reconsider their application’s architecture. The team of architects at Vivo realized that a one size fits all set of principles to adopt microservices does not exist and that a microservice implementation is only ‘correct’ when it helps to reach Vivo’s goals. They therefore decided to design a microservice architecture that is most suitable for Vivo.

#### 6.4.1 Setting Strategic Goals

The first step involved setting strategic goals for adopting microservices. The e-commerce business environment has recently become highly competitive due to technology’s limitless capability which is the sole driver of online shopping. Speed has become a necessity and also a competitive advantage in the e-commerce world. For these reasons, Vivo’s main goal for adopting microservices is to achieve both product
and team agility. For Vivo, it is important that new features, updates and patches are quickly released to their consumers and thus faster time to market is a priority. Additionally, they intend to gradually adjust their organizational and team structure to a form that increases speed and supports agility. The adoption of microservices will enable them to form independent teams therefore reducing interdependence between teams and will eliminate the need for coordination.

6.4.2 Extracting Services

The existing monolithic e-commerce application was large and complex and consisted of dozens of interdependent modules. Deciding which modules to extract, how to extract them and in what order proved to be a challenge. The architects at Vivo decided that they would initially extract modules with the least interdependencies. This is because they would be the easiest to extract and they would give the team experience for further extraction. Afterwards, they would extract the modules that offer the greatest business value.

First, the Vivo team split the frontend and backend systems and deployed them independently. This led to the frontend team acquiring autonomy that was beneficial for fast development of frontend features. The architects then decided to further subdivide the backend into smaller services. They settled on a strategy that would allow them to gradually pull out modules from the monolith and create independent services that would then be consumed by the monolith. The procedure would be repeated until all business functionalities become services and the monolith eventually becomes obsolete.

Assessing which modules to extract was not an easy task since the monolith was highly coupled and locating module boundaries was ambiguous. The Vivo team decided to use the domain driven design to locate service boundaries. They used their domain’s ubiquitous language and e-commerce terms to come up with appropriate bounded contexts. Figure 6-2, shows the service boundaries that were formed.
6.4.3 Creating teams

Previously, Vivo development teams were divided according to technical expertise such as database, business logic tier, UI. Considering they were working on a single code base, every single change or function implementation had to go through all teams before it was ready for testing and production. To achieve an agile organizational structure that fosters speed, teams were formed around each microservice. The teams are now able to work independently since there is minimal need for coordination between teams. Each team has complete ownership of its microservice and hold responsibility for it from the design stage all the way to its deployment.
6.4.4 Inner architecture

While technology is an essential part of materializing a microservice, it is not the principle characteristic. The microservice architecture fundamentally concerns creating small, compact and independent teams responsible for bringing out a model of a particular business functionality. Each team is autonomous and has the freedom to select the most suitable technology stack to develop their microservice. This is what Vivo refers to as the inner architecture because it involves the internal structure of an individual microservice.

**APIs:** In the microservice architecture, every team is required to expose a single microservice endpoint to the external world. This endpoint, usually an API, is the only means by which the microservice can communicate with other microservices. Most APIs are HTTP REST APIs. This is not compulsory but has become the norm across the industry and Vivo has therefore decided to use HTTP REST.

**Circuit Breakers:** The use of circuit breakers is one of the highly recommended if not mandatory for microservices. Vivo’s microservice design policies have made it a requirement for every team. All calls to a microservice are supposed to be routed through a circuit breaker which monitors the health of the microservice. An example of its application would be; a microservice X sends a request to another microservice Y without a circuit breaker. If microservice Y breaks, it would lead to failure of microservice X as well. This is because microservice X stays hang up waiting for a response from microservice Y. This can be solved by a circuit breaker which stops forwarding requests to a microservice whenever it is having troubles or experiencing downtime.

**Containers:** Containers have recently become an essential part of deployment of microservices. Each team at Vivo is required to package its microservice in one or more containers for the purpose of creating deployment artifacts. One of the advantages of using containers is that they are very lightweight compared to virtual machines (VM).
They can be of small sizes of up to a few hundred megabytes unlike virtual machines whose sizes can go up to several gigabytes. Due to their sizes, containers make it easy for developers to spin up new containers and therefore scale up or down depending on user traffic.

6.4.5 Outer architecture

The outer architecture refers to the complex space that contains all microservices. It can also be viewed as the space between microservices that contains all infrastructure where microservices are deployed. This is the environment where individual microservices can interact and communicate to form a system. The outer architecture can be complex and at Vivo, it is assigned to an independent team comprising of the best developers. Vivo uses a cloud as a service platform for their outer architecture. While each team has the freedom to choose its own technology set, the outer architecture requires that all team use the same cloud platform to deploy and manage their microservices.

**Container orchestration:** As mentioned earlier in section 6.3.4, each team is required to use containers for deployment as a section of their inner architecture. Container orchestration forms the largest part of the outer architecture. During the era of the monolith at Vivo, deploying was easy since it entailed deploying a single artifact to the server. The management of several microservice containers is also quite straightforward. It involves a DevOps personnel to SSH into a server, set up a docker machine, run the microservice container and expose the service’s host and port. However, it is not as straightforward when an organization has hundreds of microservice containers to manage. Container Orchestration refers to a system of either physical or virtual hosts where containers run. Orchestration tools and platforms offer an interface that helps DevOps personnel deploy, manage, control and coordinate container processes. Vivo decided to use Red Hat’s commercial platform for container applications management. The platform helps developers at Vivo build releases for new versions of code, run integration tests and deploy containers to the cloud. The orchestration platform performs other complex processes such as service registry, load balancing, networking, auto scaling, storage and security.
6.4.6 Current status

The progress of the adoption process so far, has been challenging but quite interesting for the developers at Vivo. Vivo selected the first three business functionalities to be converted to microservices. The three functionalities, namely Product Search, Customer and Shopping Cart modules, led to the simultaneous creation of three teams. The teams successfully chose their technology stacks and made several iterations of development. They work independently and there is minimal need for coordination among teams. Each team is responsible for the testing of their microservice and this has made testing easier and faster. Integration testing, however, is still necessary in the last stages of deployment in which the new features of the microservice have to be tested together with the monolith. The teams have also been able to make several deployments to the cloud production environment. The new microservices are now incorporated into the system and the monolith sends requests to the microservices. The features implemented by the microservices have been pulled out of the monolith and the monolith uses interfaces to communicate with individual microservices.

6.5 Challenges encountered

While the modernization of the Vivo e-commerce is greatly beneficial for the company, the adoption of the microservice architecture is not without its challenges. Implementing microservices introduced additional complexity since every microservice requires its own infrastructure while the existing monolith needs to be supported. Many more servers are required for both microservice deployment and the support of the monolith and this presented a new level of complexity with respect to communication, monitoring and logging. Since microservice teams have the freedom to choose their technology sets, a high technology complexity arises. This means that when a team has a lot of work, it cannot receive reinforcement from other teams due to insufficient proficiency in their specific technology sets.

The complexity of supporting both the microservice and the monolith will linger for long since it takes a long time to entirely replace the monolith. The longer the migration
process takes, the more it costs to maintain the two infrastructures. Migration to the microservice architecture brought about numerous performance issues as a result of the need for microservices to communicate over the network. Communication over the network is susceptible to latency, downtime and failures. This is a risk that was anticipated and Vivo plans to solve this by improving infrastructure and adding more server resources in order to minimize failures and shorten recovery time after downtime.

6.6 Achievements

**Faster time-to-market:** As a result of adopting microservices, it is now possible for a team to deploy a new feature into production in about 30 minutes. Development, testing and deployment of a new feature now takes approximately one week. This is a lot faster compared to the one month it took for a team to deploy a totally new version of the monolith. This improvement has been enabled by the separation of concerns that occurred when business functionalities were split to individual microservices. Deployment of a new feature now solely affects one microservice. User experience by Vivo customers has vastly improved and Vivo has already begun to see increased customer interactions with the e-commerce platform. Customers now spend more time on the web application and Vivo management hopes that this will translate to increased revenue this financial year. Developers are also receiving timely feedback from users and are better able to meet their needs by having short iterations. Concisely, faster time-to-market has been achieved.

**Independent teams:** The architectural design of a monolith requires that all teams work on the same code base. Additionally, the team members at Vivo worked on a shared continuous delivery pipeline. As a result, parallel work among teams became complicated and the need for team coordination became imminent since because the entire monolith had to be deployed all at once. The implementation of microservices facilitated the creation of independent teams. Individual teams have complete ownership of their microservices and have the freedom to make decisions related to all
development stages of the microservice. The testing and deployment phases no longer have to be coordinated and the teams are now in a position to work independently.

**Introduction of Product Search functionality:** The product search functionality had become fragile and no more development was allowed due to complexity and the associated risk of breaking. After the introduction of microservices, a new team was created for the search functionality and the members were able to create a totally new microservice from scratch. The microservice architecture allowed them to throw away the preceding product search code from the monolith since it had become difficult to maintain. Building the new microservice from scratch enabled the team to create a better internal architectural design and also use a technology stack that was most suitable for the microservice. As a result, the team has managed to introduce a new search engine to the microservice. This has greatly improved the functionality’s performance and has put Vivo well ahead of competition.

### 6.7 Quality Attributes Attained

**Agility:** One of the main strategic goals behind the decision to adopt the microservice architecture at Vivo was the ability to swiftly introduce changes and new features into the e-commerce platform. Since the breaking down of business functionalities, formation of new teams and development of the microservice infrastructure, Vivo has been able to achieve agility with respect to both the product and the teams. Smaller teams working independently from each other has led to efficiency and faster deployment of innovative features. The company is now better able to release new features and offer better experiences to their customer base and therefore keep up with market demands and competition.

**Maintainability:** While Vivo’s relatively huge and complex monolithic e-commerce application contains hundreds of thousands up to millions of lines of code, the introduced microservice architecture is made up of small independent services that
contain about a thousand lines of code. The smaller code-base has greatly reduced complexity and has made it easier for the teams to develop, make changes and maintain the code. The Product Search team was even able to create a new and complete microservice within six iterations.

**Autonomy:** On of the initial steps of adopting microservices involved the splitting down of Vivo’s monolithic e-commerce application into several smaller services. This has enabled teams to design, develop and deploy the services independently. In the event that a service becomes obsolete, the microservice team can effortlessly remove the entire service from the system without breaking other services.

**Scalability:** Due to structure of microservices, Vivo now plans to improve performance and minimize downtime by expanding their infrastructure resources to facilitate scaling of services. This will enable the e-commerce application to scale a microservice upwards by allocating more resources to the service when it becomes more popular. Similarly, the application will scale a microservice downwards to avoid wasting of resources if the microservice is not receiving too much traffic. Unlike in monolithic applications where scaling affects the entire application, in microservices, scaling affects independent services and therefore there is effective distribution of resources.
7 Conclusion

The ever changing world of technology and increased consumer demands have challenged system architects to design architectures that are resilient to change so as to remain relevant and also meet requirements. Microservices have therefore emerged as an attempt at providing innovative guidelines to designing architecture as a solution to managing complexities related to the growth of a system. The microservice architecture aims at dividing into small modules. While this is not a new concept, the microservices solution is unique in that it offers a strong modularization concept that puts emphasis on complete independence. To achieve autonomy;

- The architecture requires that services handle only one task and it ought to be perform the task exceptionally well.
- Microservices are deployed independently of other services. Changes to a single service are brought to production without the involvement of any other services.
- Microservices are developed by distinct and independent teams. These teams have ownership of the service and are responsible for it from its design phase to its deployment and maintenance phases. The teams can use tools and programming languages of their own choosing.
- Each microservice contains its own data storage either as a separate database or as a different schema in a common database.
- A microservice can employ its own support services e.g. a unique database, a search engine. These support services can be brought along to the common production platform for instance as containers or virtual machines.
- Microservices communicate with each other over the network. Each microservice implements its own communication protocol such as HTTP REST APIs. Consequently, services do not need external coordination in order to communicate.

In addition to autonomy, the microservice architecture, if well-designed, helps achieve desirable quality attributes such as loose coupling and high cohesion, reusability, modifiability, scalability and resilience. It is important for architects and developers to
ensure that a microservice observes the principle of performing only one task as this leads to loose coupling and enhances reusability and modifiability while also making it easier to achieve scalability. As a result, the quality of the microservice and the system as a whole will be maintained.

While the pros for adopting the microservice architecture are indisputably appealing, the process of adopting the architecture can be lengthy, complex and filled with challenges. It is therefore necessary to thoroughly analyze whether the pros transcend the cons and whether the costs will eventually bring about substantial results. When the analysis results establish that adoption of the architecture is well worth the effort, the next step is to develop a roadmap for the adoption process. Every microservice architecture design is unique and solves different problems. In order to design an architecture that is most suitable for your organization, Chapter 4 outlines several factors that need to be considered. On top of the list is the analysis of the enterprise’s strategic goals in order to focus on implementing a design that offers the highest business value.

An organization’s structure and culture should also to be considered. According to Conway’s law, a software replicates the communication structure of an organization. Since the independence of teams is important for the successful implementation of the microservice architecture, it is important that an organization considers its levels of command and the communication structure. A challenging part of the adoption process involves establishing system boundaries and extracting services. The Domain Driven Design (DDD) approach suggests establishing domain vocabulary that can be used to describe the system and dividing the microservices along those terms. Further subdividing is then crucial in order to determine bounded contexts and build microservice that perform only one function.

There are several ways of adopting the microservice architecture. Depending on the current situation of the enterprise, architects can decide whether to build an entirely new microservice system from scratch or decompose an existing monolithic system. Chapter 5 explains the different ways one can choose to adopt microservices. It also discloses different strategies for decomposing an existing application.
This thesis presents the challenges of the traditional monolithic applications and offers solutions based on the appropriate adoption of the microservice architecture. Firstly, it provides a case for microservices in relation to its characteristics, advantages and quality attributes. It then presents a case against microservices citing its outer complexity issues and its costs. The decision to adopt the microservice architecture should be based on a careful analysis of the two cases. Secondly, this thesis demonstrates the essential factors that ought to be considered should an enterprise decide to go the microservice way. Thirdly, it illustrates the possible approaches to adopting microservices, giving easy to use strategies to decomposing an existing monolithic application. At the end of the day, there is no correct or generic way to implement microservices. Microservices can only offer guidelines and direction towards a suitable design solution. A microservice architecture design is only correct when it helps realize an enterprise’s goals and offer business value.
REFERENCES


APPENDIX

Interview Questions

1. What are the reasons behind the decision to adopt the microservice architecture for your e-commerce web application?
   a. What were the challenges faced with the previous architecture?
   b. What were the motivations behind adopting microservices

2. What are the main goals that Vivo hopes to achieve by adopting microservices?

3. How did you go about extracting services from the previous e-commerce application?
   a. How did you come up with bounded contexts?
   b. How did you identify business domains (If used)?

4. What is the organization’s communication structure and culture?
   a. How are teams formed? Are they aligned to business domains or member expertise?
   b. How are responsibilities divided among teams?
   c. What is the size of each team?
   d. What is the nature of communication among team members and between teams?
   e. What are the levels of cross-team dependency and the power distribution across teams?

5. What tools and processes did you use?
   a. Are you using agile/DevOps?
   b. What were the principles behind the processes chosen?

6. What procedures did you take while adopting the microservice architecture?
   a. What technologies did you use for the inner and outer architectures of your application?

7. What is the current status in the adoption process?

8. What goals have you successfully achieved by adopting the microservice architecture?

9. What goals have you been unable to achieve?
   a. What challenges have you encountered so far?

10. What quality attributes has the new e-commerce application attained?