Cloud Computing Architecture and Deployment

Master Thesis

By

Ms. Mayuri Mansubrao Deshmukh

March 2019

SRH Hochschule Heidelberg

School of Informatics

Dr.Peter Misch

Prof.Dr.Gerd Moeckel

Cloud Computing Architecture and Deployment

Matriculation ID:11009241

March 2019

A thesis submitted as apart of requirement for the qualifying examination for the degree of Master of Science

Thesis Advisory Committee

………………………………………………………… (Internal Supervisor)

Dr. Peter Misch

SRH University of Applied Computer Science

………………………………………………………… (Internal Supervisor)

Prof. Dr. Gerd Moeckel

SRH University of Applied Computer Science

AFFIDAVIT

Herewith I declare:

- that I have independently composed the chapter of the Master Thesis for which I am named as the author.
- that I did not use any other sources and additives but the ones specified.
- that I did not submit this work for any other examination procedure.

Heidelberg, ________________________

ACKNOWLEDGEMENT

Firstly, I would like to sincerely thank my Dr. Peter Misch for giving me the opportunity to study and research on this topic. It would not have been possible without his support, motivation, enthusiasm and immense knowledge. His huge input and guidance were immensely high for the time of research and problem-solving content. Additionally, I would like to thank Prof. Dr. Gerd Moeckel. My study and research work would not have been possible without him. His guidance, suggestion was priceless, and he was very supportive during the entire course of my thesis.

A big thanks to my family and friends for enduring everlasting assistance, continuous support and for always standing by me during the course of my thesis. The support was indeed priceless!

It was because of such wonderful people around me that the study could see the day of light!

ABSTRACT

One of the most important migration is happening towards cloud in the business world. Monolithic architecture is considered as a standard methodology for creating a cloud-based application. This architecture is feasible as long as the size of the application remains small. If large monolithic application with a huge codebase needs to be scaled it becomes troublesome and also causes problem on the production level if any error or failure is detected. In case of hardware failure, the complete system is migrated to the healthy servers to overcome these problems most of the business firms are moving over to cloud. Cloud providers provide services like monitoring and logging and eventually inform the developers about any error occurrences.

Although this does not solve the problem of scaling a large codebase monolithic application. To solve this problem, this thesis provides a methodology for implementing microservice architecture over cloud. Microservices are smaller, independent components that are loosely linked with each other. This research also gives a brief description of concepts a linked with microservices. The proposed methodology makes use of docker and kubernetes for containerization and orchestration. Kubernetes helps both developers and the operation team in monitoring and logging of the containerized application during events of failures. Making use of docker provides the development environment faced during execution which are dependent on the hardware and software attributes of the system.

Keywords: Cloud Computing, Microservices, Docker, Kubernetes

Contents

I Introduction 1

II Cloud Computing 3

2.1 Definition 3

2.2 According to NIST Cloud Computing 4

2.2.1 Cloud Service Models 5

2.2.2 Cloud Deployment Model 7

2.3 Advantages 8

2.4 Disadvantages 8

2.5 Summary 9

III Microservice 10

3.1 Definitions of Microservices 10

3.2 Monolithic Architecture 12

3.3 Advantages and Disadvantages of Monolithic 13

3.4 Microservice Architecture 15

3.5 Communication Mechanism 18

3.5.1 Representational State Transfer (REST) 18

3.5.2 Messaging 19

3.5.3 Remote Procedure Calls (RPC) 19

3.5.4 API gateways 20

3.5.5 The Service Registry 21

3.5.6 Dependencies and Data Sharing 24

3.6 Advantages of microservices 26

3.6.1 Single responsibility 26

3.6.2 Autonomy 26

3.6.3 Heterogeneity 26

3.6.4 Resilience 26

3.6.5 Scaling 27

3.6.6 Easy deployment 27

3.6.7 Composability 27

3.6.8 Replaceability 27

3.6.9 Small teams 27

3.7 Disadvantages of Microservices 28

3.8 Comparison between microservice and monolithic architecture..28
3.9 Fault tolerance in Microservices 31
3.9.2 Stability Patterns 34
3.9.3 Fault Tolerance Mechanism providers..35
3.10 Monitoring of microservices 37
3.10.1 Audit logging 37
3.10.2 Application metrics 37
3.10.3 Distributed tracing 37
3.10.4 Health check API..38
3.10.5 Exception tracking..38
3.10.6 Log aggregation 38
3.10.7 Monitoring solutions...39
3.11 Related Work...42
3.11.1 CAP Theorem...42
3.11.2 CAP Theorem and Microservices 42
3.11.3 Conway's Law 42
3.11.4 Conway's Law and Microservices...43
3.11.5 Domain Driven Design (DDD) 43
3.11.6 Bounded Context..43
3.11.7 Domain Driven Design and Microservices 43
3.11.8 Continuous Delivery 44
3.11.9 Continuous Integration 45
3.11.10 Deployment Pipeline...45
3.11.11 Continuous Delivery and Microservices..47
3.11.12 DevOps 47
3.11.13 DevOps and Microservices..48
3.11.14 Scalability 48
3.11.15 History 48
3.12 Summary 49
IV Docker 51
4.1 Real-World Use Case for Docker 52
4.2 Virtualization vs Containerization 52
4.2.1 Virtual Machines (VM) 52

4.2.2 Containers 53
4.3 Why we need docker? 54
4.3.1 Workflow before Docker 54
4.3.2 Workflow after Docker 55
4.4 The Docker Engine 56
4.5 Docker Architecture 56
4.5.1 Docker client 57
4.5.2 Dockerhost 57
4.5.3 Docker registries 59
4.6 Dockerfile 60
4.6.1 BUILD phase 60
4.6.2 RUN phase 61
4.7 Docker-compose 61
4.8 Docker terminology 61
4.9 The Container in Cloud Computing 62
4.9.1 Benefit of Containers in Cloud Computing 62
4.10 Summary 63
V Kubernetes 64
5.1 Understanding the Core of What Kubernetes Does 65
5.2 Understanding the Parameters architecture of a Kubernetes cluster 66
5.2.1 Components 67
5.2.2 Node (worker) parameters 67
5.3 Studying an application in Kubernetes 68
5.4 Kubernetes Concepts 70
5.5 What is Pod? 71
5.6 Features of Kubernetes 72
5.7 Summary 72
VI Implementation of proposed architecture 73
6.1 Working 74
6.2 Features of the proposed architecture 76
Conclusion 77
References 78

List of Figures

Figure 2.1 Cloud computing architecture (Strijkers, Makkes, & Demchenko, 2012).....4
Figure 2. 2 Proposed architecture appears in PaaS (Author).....6
Figure 3. 1 Monolithic Architecture (Author).....12
Figure 3. 2 Microservice architecture (Author).....15
Figure 3. 3 Self-registration (Peyrott, The service registry, 2015). 22
Figure 3. 4 Client-side discovery (Peyrott, The service registry, 2015).....23
Figure 3. 5 Service-side discovery (Peyrott, The service registry, 2015).....24
Figure 3. 6 Geographic Comparison between microservices and Monolithic (Google, 2019). 29
Figure 3. 7 Comparison between monolithic and microservices over the past 5 years (Google, 2019). 29
Figure 3. 8 Comparison between characteristics of monolithic and microservice architecture (Author)...30
Figure 3. 9 Fault path (Madani & Jamali, 2018). 31
Figure 3. 10 Type of Fault Tolerance (Madani & Jamali, 2018).....32
Figure 3. 11 Continuous delivery process diagram (Continuous delivery, n.d.) 46
Figure 4. 1 Virtual Machine (Wagner, Torre, & Rousos, 2018) 52
Figure 4. 2 Docker Container (Wagner, Torre, & Rousos, 2018).....53
Figure 4. 3 Traditional deployment workflow (without Docker) (Matthias & Kane, 2015).....55
Figure 4. 4 Docker deployment workflow (Matthias & Kane, 2015). 55
Figure 4. 5 Docker engine (Nickoloff, 2016).....56
Figure 4. 6 Docker Architecture (Nickoloff, 2016). 57
Figure 5. 1 Core of what Kubernetes does (Lukša, 2017). 65
Figure 5. 2 Kubernetes Parameters (Author).....66
Figure 5. 3 Architecture of Kubernetes cluster (Lukša, 2017).....66
Figure 5. 4 Functioning inside a container (Lukša, 2017).....69
Figure 6. 1 Proposed architecture on cloud (PaaS) (Author) 73

List of Tables

Table 3. 1 Aspects of REST (Fielding & Reschke, 2014) (Feng, Shen, & Fan, 2009).....18
Table 4. 1 Difference in VM vs Containers (Bauer, 2018) 54
Table 4. 2 Important Docker Commands 60

I Introduction

Some years ago, most software applications were built in a monolithic style. They were running as single processes across a handful of servers. These systems are still working today and they have continuous release cycles and are updated frequently. At one end of this cycle, there are application developers who bring together the entire application as one unit and pass it onto production unit, who are responsible for deploying and monitoring the efficient functioning. Whenever a hardware failure occurs irrespective of the cause, the production unit migrates the entire application and the related frameworks to the healthy functioning servers. Scaling which referred to increasing or decreasing the size of an application was easy when the application and codebase was small, although in case of a huge application with a large code base, scaling took a toll on the developers. Scaling would require creating multiple instances of the same code base which made testing and further deployment difficult.

Nowadays the applications that were created using monolithic architecture have found a new direction and are migrating towards the most trending topic of this and the past decade called as microservices. These monolithic applications are cut down depending on the different functionalities within it and made to run as an independent service. These independent services are coined as microservices which are attached to the other services in a loose manner, wherein the interaction is as minimal as possible. Microservices are basically, small components having their independent code, their personal database that can be sharable as well, when brought together can be referred to as an entire huge application. The most important reason for the development of microservices was scaling. Microservices can tackle scaling in every possible way be it deployment, testing, automation and monitoring. Each microservice can be scaled as a single service. Scaling a microservices does not refer to scaling the entire application. This makes deployment easy and monitoring easier. This makes changing and updating a quick process from business point of view as well.

But passing and distributing a microservice based application should have a systematic approach. This brings to the concept of Docker and its containers. Containerizing of an application refers to bringing together the entire application with all its packages, libraries, database, the platform and its specification together. Container gets its name from ship containers wherein everything is brought and put inside the container so as to get it shipped.

But it is becoming increasingly difficult to configure, administrate the system as a whole with significant amounts of transportable components and an ever-growing mass of data centers. It is difficult to determine where each of these components can reach high resource utilization usage and thus reduce hardware costs. This eventually increases the manual work even in the time where everything is getting automated. Therefore, automation is of the utmost importance which can handle failures efficiently, automate the configuration process, and monitor it. To have these difficulties sorted Kubernetes is brought into play.

Kubernetes reduces the burden of the shoulders of not only the developers but provides an extensible assist to the production unit of an organization wherein the main task of the production unit is monitoring Kubernetes while it monitors the application for you. The most important concern that was discussed above was automation which is being relieved by the implementation of Kubernetes. Kubernetes provides features like automation and rescheduling of an application whenever there is occurrence of a hardware failure. Kubernetes also carries a high fault tolerant mechanism itself.

Having kubernetes will not only benefit the microservices architecture-based applications run on local infrastructure but also on cloud. Which brings us to the final concern of this project which is cloud. Nowadays, it’s getting clumsy and difficult to manage huge server base for each organization and having their backups maintained as well. Most organizations these are preferring to move onto cloud for their data storage issue. Having everything on cloud reduces the concern of handling it by ourselves, this concern is taken care of by the cloud vendors. This project showcases the modification in the PaaS layer of the cloud computing model, wherein it portrays implementation of microservices based architecture that is containerized using Docker and deployed using Kubernetes over Cloud.

What are the Research Questions?

1. Why is there a need for microservice architecture?
2. How to make applications portable?
3. How to deploy the application on cloud?

II Cloud Computing

2.1 Definition

Cloud computing is a paradigm for enabling global, accessible, on-demand network access to a mutual supply of scalable system framework resources (e.g., storage, networks, servers, applications, and services) that can be delivered quickly and made quickly accessible with as least as possible management support and service providers interaction. Following are the cloud models important features (Mell & Grance, 2011).

- Three service models (IaaS, PaaS, SaaS).
- Four deployment models (Public Cloud, Private Cloud, Hybrid Cloud, Community Cloud).

From the above, there are 5 essential factors to be covered namely,

1 On-demand self-service

Customers get computing resources on-demand and self-service. Cloud-computing customers use an automated interface and get the processing power, storage, and network they need, with no need for human intervention.

2 Broad network access

They can access these resources over the network

3 Resource pooling

The provider of those assets has a big supply of them and allocate them to customers out of the pool. That allows the provider to get markets of scale by buying in bulk. Customers don't have to know or care about the exact physical location of those assets.

4 Rapid elasticity

The resources are elastic. Clients who need more resources can get more rapidly. When they need less, they can scale back.

5 Measured Service

The consumers pay for what they use, as they go. If they end using resources, they stop paying.

The biggest part of Cloud computing is that one can begin using services and unlike traditional computing where one has to go through the capital expenditure model. Capital expenditure model is one where needed computing resources are rented from own data centre or available service providers through related or new processes.

In Cloud computing, resources such as servers, storage, network, and many other business applications can quickly be provisioned without any human interaction. The billing of these services is estimated either per minute or per hours depending upon various services given by the separate Cloud computing platforms. Firms offering these computing services are known as cloud service providers, and they charge for Cloud computing services based on usage.

2.2 According to NIST Cloud Computing

The terms Cloud and Cloud computing are both used interchangeably in further sections to ease out the understanding. The cloud architecture consists of five different parts which work together to provide on-demand services. Figure 2.1 is taken from National Institute of Standards and Technology (NIST) Cloud computing reference architecture (Strijkers, Makkes, & Demchenko, 2012). It shows cloud architecture and its five components i.e. cloud provider, cloud consumer, cloud carrier, cloud auditor and cloud broker. These are briefly described as further.

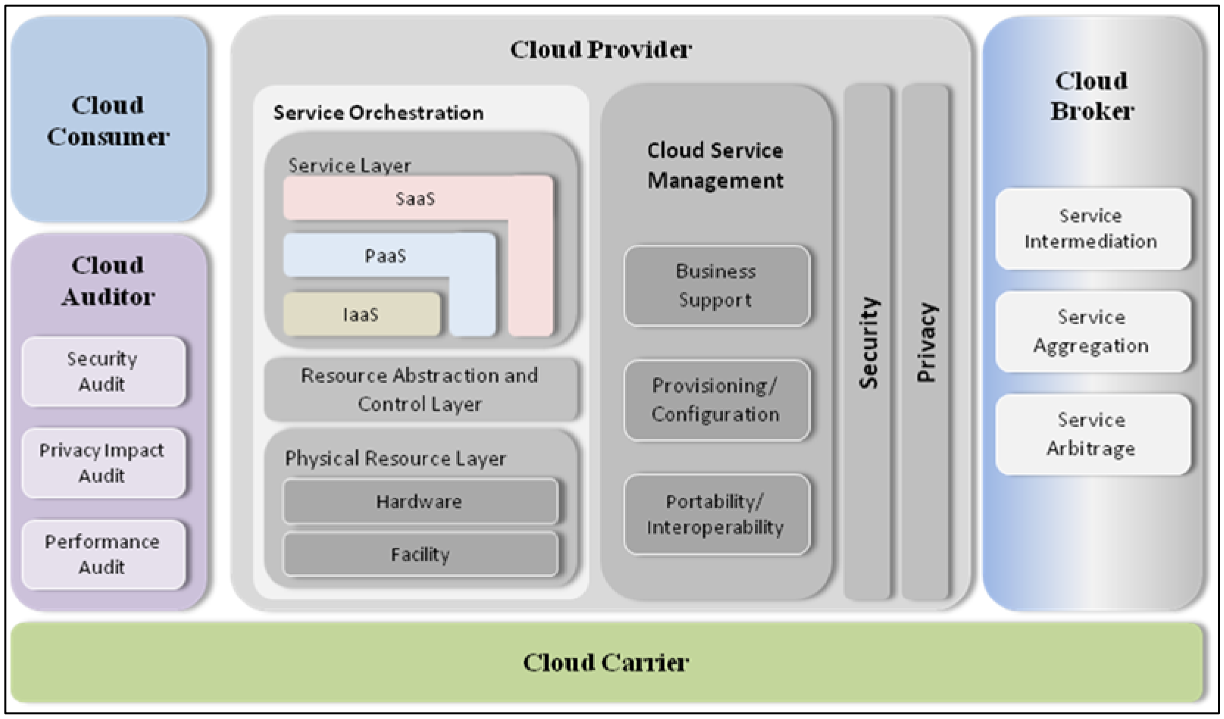

Figure 2.1 Cloud computing architecture (Strijkers, Makkes, & Demchenko, 2012).

Cloud Provider

The cloud providers are firms that gives cloud services. The cloud provider has control over the IT infrastructure and manages the technical outages if planned or unplanned. Cloud provider also assures that agreed Service Level Agreements (SLA's) are achieved (Joomla, 2018).

Cloud Consumer

A cloud consumer is a person(s) or a firms using cloud service(s) and having an agreement with a cloud provider or cloud broker (Joomla, Cloud Consumer, 2018).

Cloud Carrier

The cloud carriers are networks and telecommunication industries, which assure that services from cloud provider are available to cloud consumers. Cloud carrier works closely with the cloud provider to meet agreed SLA's (Joomla, Cloud Carrier, 2018).

Cloud Broker

The cloud brokers are third-party companies, which work closely with both cloud providers and cloud consumers. Generally, these are consulting companies, and so they can quickly sell various cloud solutions to their existing clients as well as to new clients (Joomla, Cloud Broker, 2018).

Cloud Auditor

The cloud auditors are third parties who are specialized in an independent assessment of cloud services offered by cloud providers. A cloud auditor can audit various areas such as security, privacy, performance, licensing, operations and the other regions to highlight the gaps against multiple services and data privacy standards (Joomla, Cloud Auditor, 2018).

2.2.1 Cloud Service Models

There are multiple cloud services offered by various cloud providers. These are divided mainly into three categories of services, i.e. Infrastructure as a service, Platform as a Service and Software as a Service also known as SPI (SaaS, PaaS, IaaS) (Bokhari, Shallal, & Tamandan, 2016).

Infrastructure as a Service (IaaS)

This layer is the most wide-scale service used amongst the three service models. IaaS provides virtual infrastructure as well as raw hardware for building, maintaining and eliminating storage, virtual machines, and virtual network over the Internet in the cloud provider's infrastructure. Most of IaaS providers have capabilities to produce virtual servers with different forms for example; more CPUs for computing with standard RAM as well as can add more RAM later as per demand. Like other service models, infrastructure assets size up or down and are billed as per real usage only. Leading cloud providers can connect a virtual network in a cloud to a company's network through the VPN, and it appears as a scalable IT infrastructure to existing IT infrastructure.

The big advantage of an IaaS solution is a fast rise or drop of infrastructure on demand, lower risk in Return of Investment (ROI), decreasing the human resource and hardware costs, programmed scaling of computing power, etc.

Example of this type of IaaS services is VM, storage, servers, network, etc.

Platform as a Service (PaaS)

PaaS is a layer on the head of the IaaS solution. The PaaS service gives a platform for building new applications and software by developers or customers across the Internet. Customers can rent scalable virtualized servers, attached facilities, and can quickly size the demands as per requirement. Customers do not have power over the network, storage, servers, and operating system but they can control deployed applications and configurations. Costing model for PaaS could depend on various parts, for example, some Input/output requests, storage usage in Gigabytes, data transfer per Gigabytes, etc.

There are benefits of a PaaS model for example versatility in load increase and decrease through a development process. There is also no need to maintain infrastructure procedures in this case, but to lay down appropriate rules for instantaneous load balance. Since PaaS has several design principles, different clients from different locations can configure an application. Security is a shared duty between a cloud provider and a cloud client in PaaS.

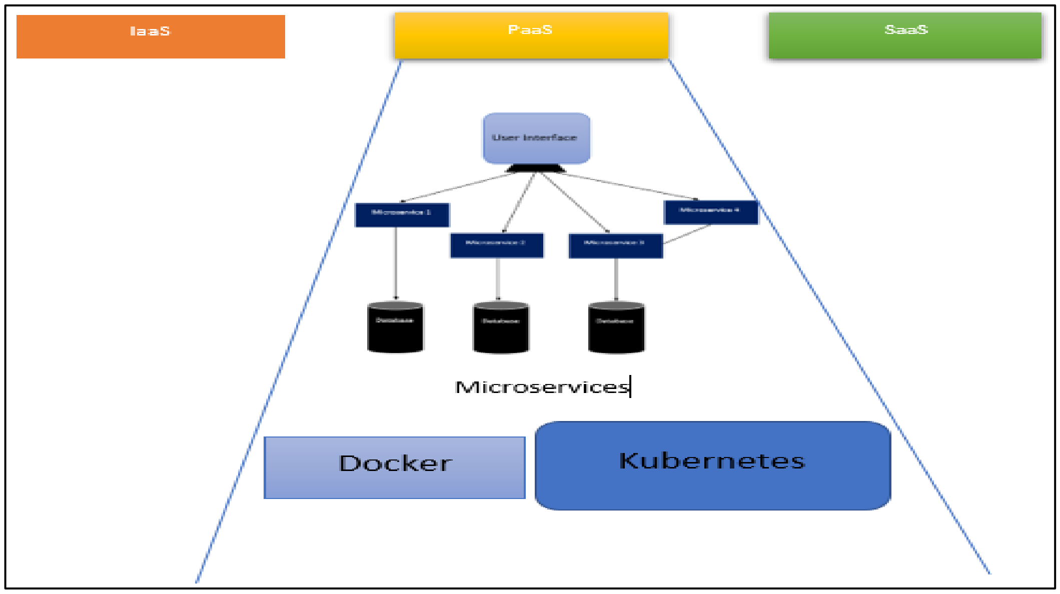

Figure 2. 2 Proposed architecture appears in PaaS (Author).

Most of the development and migration is happening on PaaS (Barde, 2013).

Software as a Service (SaaS)

SaaS is a layer on top of the PaaS solution. In SaaS, users can manage applications via thin clients without installing those applications on local private computers. SaaS provider oversees the management of client's infrastructure and licensing of software applications. Some of the examples of the SaaS model are Microsoft OneDrive, Microsoft Office 365, etc. The network is a crucial part for excellent consumer experience, and thus SaaS offering is suitable for lightweight applications related to heavyweight applications such as 3D games. A cost model of SaaS-based systems changes for applications as some are charged as per usage and some have a set charge for a certain period. A benefit of a SaaS model is a cheaper licensing cost since its design principle is one too many, that is the same applications are used by multiple clients in likeness yet managing the isolation of each client. Other advantages are such as cheaper operations and maintenance cost which are also taken care of by a SaaS provider as its infrastructure is established and operated by the SaaS provider.

Example of this type of SaaS services is Salesforce, Google Apps, Microsoft Office 365, etc.

2.2.2 Cloud Deployment Model

A cloud deployment structure shows a particular type of a cloud environment, primarily characterized by possession or control, size, and control of access. Cloud deployment models are divided into Private Cloud, Public Cloud, Hybrid Cloud, and Community Cloud.

Private Cloud

In Private Cloud, the infrastructure is provisioned exclusively for a private organization having many customers for example business units, third parties, or vendors accessing the same resources. Private clouds are generally tested, operated and controlled by the private organization. The infrastructure can be either on premise or off premise. In the last decade the majority of large organizations, having their own data centres, have transformed into private cloud-based solutions through virtualization technology.

Public Cloud

Public cloud depends on standard Cloud computing model where a provider assures the availability of Virtual machines, storage or applications to the public via the Internet. As per NIST, the definition of public cloud is described as:

„The cloud infrastructure is allowed for open use by the general public. It may be maintained, managed, and operated by a business, academic, or government organization, or some combination of them.„ (Cloud Strategy Partners)

These cloud infrastructures are hosted on cloud provider's huge scale data centres spread across globe solving disaster recovery problems. In public cloud, enterprises have no control over data and processing environments Also, because customers are connected to Public Clouds through Internet connections, Service Level Agreements (SLAs) may be complicated to meet if the Internet connections are not working correctly.

Hybrid Cloud

A hybrid cloud consists of different cloud providers for the same consumer. As an example, a banking customer may want to keep its business-related financial data in a private cloud, and it can operate marketing related tasks in a different cloud. Hybrid cloud is quite popular among enterprise customers. This is due to their sizeable existing infrastructure, which is not so straightforward to move onto the public cloud. Thus, the enterprise either move a part of their existing resources into a cloud or start new projects directly in the cloud.

Community Cloud

Community Cloud is provisioned for a specific group of users or organizations. A community cloud is almost like a public cloud, but its use is only for a particular community of cloud consumers. Community members not only share the responsibility of laying the foundation of the infrastructure and its evolution but also ensures that only allowed parties to have access to it unless otherwise agreed.

2.3 Advantages

Lower-cost computers for users: As the application runs on the cloud, the cloud is responsible for its processing which thereby, reduces the cost on system enhancement.

Better performance: As the program files are on the cloud and not on the user's personal computer, the internal network will perform better as no network traffic will appear.

Fewer IT infrastructure costs: Implementing cloud computing technology focuses on reducing unnecessary cost and focusing on investing on the specific servers required by the IT department.

Less maintenance costs: Cloud services are provided by cloud vendor, so, the cost of maintaining huge servers gets eliminated resulting in less maintenance cost.

Reduced software costs: Employees working on cloud-based application need to have access to the cloud and its dependent software, doing so, decreases the need to have the software’s installed on all the systems of that department.

Software updates eliminated: Software’s are required to be updated on regular basis but working on cloud computing-based technology the organizations need not worry for any updates.

Increase in computing power: Implementing cloud computing, the user is not confined to the system's power and can benefit from the cloud's power as well.

Storage capacity limitless: Storage can be scaled based on the requirement of a project. Cloud vendor's offer services for upgrading and degrading a subscribed pack.

Data safety secured: Having implemented cloud erases the worries of disk failures or any calamities at the company.

Documents can be accessed worldwide: Only requirement is a working internet and one can retrieve the documents from over cloud.

2.4 Disadvantages

Internet connection compulsory: to connect to the cloud PC internet is required or else it is impossible to work and get a connection.

Low-speed connections will be time consuming: Web-based applications, large documents require high speed internet. Even with fast connections, sometimes you might experience delays since web-based applications can sometimes be slower. The reasons for that are because of the compulsory uploading and downloading bandwidth required by the web applications.

Stored data might not be secure: As the data is stored “in the cloud” the cloud providers might have a way to access it as well. Although many applications and data are being encrypted by the developers.

No backup: As all the data is on cloud it becomes convenient as cloud providers have multiple ways to back up the data storage. Although if by any means the data is lost, the cloud user is at lost if the user does not have a local backup of the files.

2.5 Summary

This chapter covers all the essential concepts closely related to cloud computing. Cloud service models and cloud deployment models are studied in detailed. The chapter shows migration happening on Platform as a Service which is further elaborated in chapter 6.

III Microservice

3.1 Definitions of Microservices

A microservice is a separate, loose - linked development unit that operates on a single matter. Microservice follows the same pattern like traditional Unix, where Unix did one thing at one time and did it well. Matters such as how to "combine" are left to higher layers or policy. Combining means microservices tend to avoid interdependencies: if one microservice contains a laborious demand for alternative microservices, then you ought to raise yourself if it is smart to create all of them a part of a constant unit. There are various definitions to microservices.

Sam Newman describes a microservice as a self-executable software component that collaborates with other software components within an application system. Microservices are supposed to take on a role within a specialized business domain and thus a high degree of cohesion in the sense of Robert's Single Responsibility Principle1 C. Martin (Newmann, 2015).

Microservices follow the single responsibility principle (SRP) wherein it works on concentrating on one thing and tend to produce best results for that particular service. Unlike microservices, monolithic architecture involved having the entire code as one, where different tasks were incorporated in the same code and deployed as a single architecture. Monolithic based development is simple to understand and easy to deploy. In case of scaling, when an architecture is huge the codebase tends to be long which causes problems. This results in issues during deployment and scaling. However, in case of microservices multiple platform can be used for development, and as the services focus on a single task scaling becomes easy and fast (Newmann, 2015).

James Lewis uses a microservice to refer to a small application that is independent can be deployed, scaled and tested. This application and its code should be easy to understand and do exactly one job. Like Sam Newman, he too requires one Based on the single responsibility principle (Thones, 2014).

Microservice architecture has gained importance in the last couple of years. Microservices can be summarized as services focused on their individual task which are loosely coupled (Lewis & Fowler, 2014).

The motivation for this transition comes from the very fact that perpetually maintaining a monolithic design has resulted in difficulties in keeping up in pace with new development approaches like DevOps, a vocation for preparation many times every day. In alternative hand, microservices supply a lot of flexible choices, wherever individual services go with the single responsibility principle (SRP), and that they will thus be scaled and deployed severally (Lewis & Fowler, 2014).

Dragoni et al. describe a microservice as minimal independent process interacting via messages. Insignificant in this context means that the service only implemented functionalities that serve its assigned purpose. As an example of minimal, a calculator is listed, which is minimal if and just if he just needed Arithmetic operations implemented. For example, plotting functions do not belong to its assigned task and should not be performed in the calculator microservice be (Dragoni, et al., 2016).

A microservice is a stand-alone program that has its life cycle within the system. Each service should be able to operate independently, regardless of which other services are still available in the system and on which other technologies are based. So individual services can easily scale horizontally, by starting multiple instances of it. It usually requires statelessness. Every microservice satisfies a particular assignment inside an application framework, whereby the obligations are delimited from one another so that they have as meager cooperation as conceivable with changes in the undertaking determination. Often in this context, the microservice becomes the sovereignty over the scheme of its Data given. Thus, the independent evolution of service is provided, as every service can use his database with any technology. A microservice has one tolerant and at the same time, a robust interface through which its logic and data are encapsulated become. As a result, all microservice can be automated independently. A successful test is characterized by a flawless internal functionality and implementation of the Interface Specification. Service starts in a few moments for rapid development and to support elastic scaling.

3.2 Monolithic Architecture

With the term monolith was already referred to in 1982 software code, due to internal Dependencies were difficult to maintain. When fixing bugs in one place, have yourself new mistakes crept in elsewhere (Warner, 1982).

(Dragoni, et al., 2016) Gives a more general definition of monolithic software, monolith for short called: A monolithic software application is a software application composed of modules. They are not independent of the application to which they belong. We conclude that a monolith internally may well be modularized. Some High-level languages even bring concepts to modularize software, crucial but it is that the individual modules are not independent. This has the advantage of having architectures based on actually independent services are considered monolithic if they are no longer due to lousy design are independent and have the characteristics of a deployment monolith.
From the microservice perspective, a deployment monolith is a software that functions as a single large application, which is compiled and deployed as a whole. It is a matter of a single logical execution unit. Changes to a part of the execution unit As, a result, the entire system needs to be recompiled and deployed (Wolff, 2016).

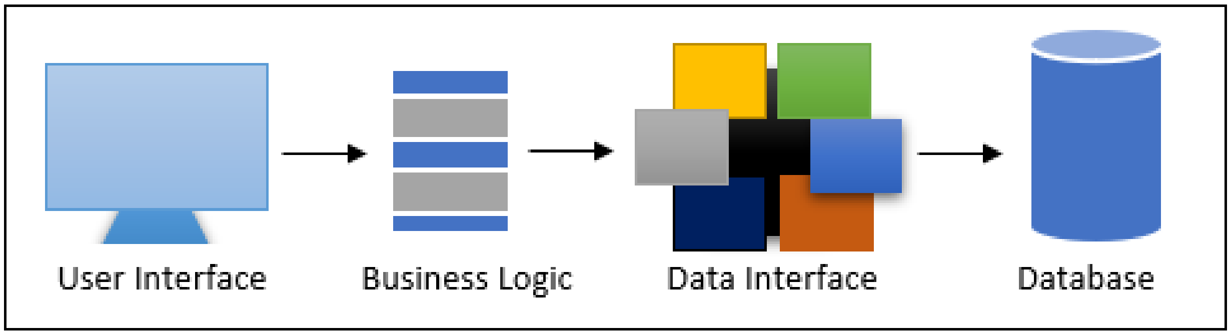

Figure 3. 1 Monolithic Architecture (Author).

Presentation tier:
The above mentioned is the appliance’s highest level. The presentation tier in case of a shopping application shows data related to services such as product browsing, purchasing and adding products to cart. It interacts with the neighboring layers eventually putting out the results within the network to the browser/client level and to all other levels. In this layer user will directly access in an undemanding way (e.g., an internet page, or operating system's GUI).

Application tier:
This tier by creative arts controls the robustness of the directly associated application.

Data tier:
The data tier includes the pathway of information perseverance (database (DB) servers, file sharing) and thus the data access layer that embodies the pathways of perseverance and illuminates the information. This layer gives support in the form of associated API and portrays tactics to manage data keeping without exposing or making dependencies on data storage mechanism. Eliminating the dependencies on the data storage processes allows for notifying or modifying

(improvements) while customers in the application do not face any unresponsive changes. There are prices for implementation like the unification/separation of any tier and sometimes the quality cost in return improved sustainability.

3.3 Advantages and Disadvantages of Monolithic

Advantages

- It's easy to develop.
- It is straightforward to test. Only start the application and start the end-to-end testing. We will also check the chemical element for automation mistreatment with no problem.
- It is straightforward to deploy the monolithic application, copy the pre-packaged application to the server.
- Quantifiability is simple.

We tend to have a monolithic application replacement instance and raise the load balancer to further distribute the load to the new instance. However, as the size of the monolithic application increases, quantification becomes an enormous issue. The monolithic design has worked successfully for several decades. It develops and deploys as a monolith multiple of the first undefeated and largest applications. Several large-scale massive corporate applications are still being deployed as a monolith. However, there has been a paradigm shift in the work of the approach IT industries with the ever-changing market and the emergence of the latest technologies. The monolithic design that most corporations are trying to handle nowadays has some serious problems.

Disadvantages

- **Flexibility**: There is no versatile monolithic design. We tend not to be able to use entirely different technologies. At the beginning, the technology stack is set and followed throughout. Upon maturity of development, upgrading technology stack versions typically becomes troublesome, plus increasingly adopting more modern technology. Despite modularization, dependencies are growing in size as they often have a vast and complex code base. Changes often lead to undesirable effects and errors are hard to find.
- **Dependencies and reliability**: It is not reliable. If one feature goes down, the entire application might go down. Libraries used by various modules often lead to inconsistencies when new versions are required.
- **Speed of development**: Development in monolithic design is completely slow. It is troubling for new members to grasp and modify the code of a large monolithic application. The quality of code is eventually affected in time. The IDE is brimming and the scaling of the code base becomes sluggish. The bigger the device, the longer it takes to get started. These factors have a huge impact on the productivity of developers.
- **Technology Compromises**: Making a fancy application is troublesome due to technology restrictions. All modules must be based on the technology chosen, although this may be sub-optimal for individual modules or some developers are specialized in another platform.

- **Scalability**: It is difficult to rescale if a monolithic application is too large. We will produce new stone instances and raise the load balancer to distribute traffic to the new instances; however, with increasing associated load, monolithic design cannot scale. Each copy of the appliance instance is able to access all the information, making caching less efficient and increasing memory consumption and I / O traffic. Furthermore, completely different application parts have different resource needs, one might be intensive computer hardware, while another might be intensive memory. We tend to be unable to severely scale every element with monolithic design. Horizontal scaling is done by duplicating the entire application, although the load may be the responsibility of only individual modules.
- **Continuous deployment**: Extremely troublesome continuous readiness. Large monolithic applications hinder frequent deployments. We need to deploy the complete application to update one element. All modules are deployed with standardized Configured infrastructure. Different requirements have to be solved as a compromise, for example CPU and RAM. Changes in individual modules result in the redeployment of the entire application.

Because of all the higher than the disadvantages of monolithic applications, the design of the microservice is changing into a lot of and more appreciated every day. So, the question arises what microservice architecture is? Briefly, the architecture type of microservice is an associated approach to the development of one application as a collection of small services, each running in its own method and human action with lightweight mechanisms, sometimes through reposeful network services or electronic messaging. These services are engineered around business capabilities and can be deployed severely through fully machine-controlled ready machinery. There is a vacant minimum of centralized management of these services that can be developed in numerous programming languages and use various technologies for storing knowledge. Microservices are small, multiple cloud-enabled deployable units.

3.4 Microservice Architecture

Author Dragoni (Dragoni, et al., 2016) has a very simple definition: A microservice architecture is a dispersed application where all its subsystem are separate services. The type of modularization of a microservice architecture (MSA) is therefore exclusively handled by microservices at the macro level. Microservice architecture is limited to an application context and therefore is considered a software architecture. In contrast, SOA can be seen as an enterprise IT architecture that combines multiple systems.
The definition of Dragoni, et al (Dragoni, et al., 2016). and, also, call for communication to take place via a lightweight mechanism, being lightweight in the sense of Characteristic smart endpoints and dump pipes. The microservice architectural style is often used as a synonym for the microservice architecture. In doing so, aspects would enter into the context of a software architecture go out. This work, therefore, distinguishes between microservice architecture and Micro service style.

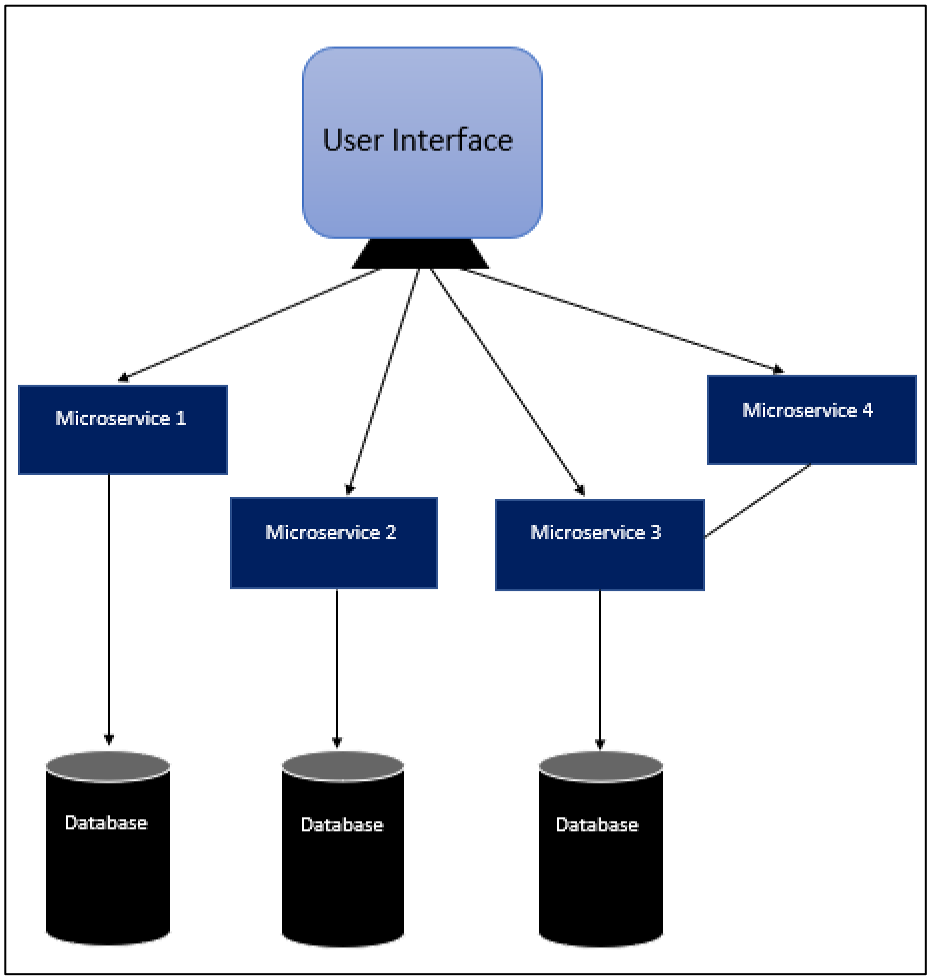

Figure 3. 2 Microservice Architecture (Author).

Size of a microservice

An exact definition of the size of such a service does not exist. Depending on the size and number of Microservices in the system, different aspects have to be considered. The guideline is that service ought to be as substantial as conceivable that each colleague understands its working and the code completely.

Additionally, the responsible team should be able to completely redevelop it within a few weeks, for example, to take advantage of other languages and new ones to exploit technologies. If one chooses services too big and packs too much function into a single microservice, it can no longer be serviced by a team. This should be avoided since Responsibilities between teams are then no longer clearly defined. In the meantime, it makes it laborious for team members to understand its full functionality. Another disadvantage is the more extended training of new team members, for it with increasing size of services becomes more challenging to be productive quickly. If, on the other hand, the services are too small and too many services are operated; this results in overhead to the consistency between the data of each service to obtain. Besides, the communication overhead over the network becomes higher, the more distributed the functionality of the overall system is. Ultimately, the existing infrastructure also limits so from a certain number of microservices given a high degree of automation. (Wolff, 2016).

Microservice style

When experts talk about microservices, they usually mean less the concrete service or the microservice architecture. Microservices or microservice architectural style is instead Representing a broad concept, the organization that applied Techniques, a culture of development, and above all, the nature of the modularization of systems describes components in the form of microservices. From this style also have the Definition for a microservice, and the MSA derived. Many features and ideas of the microservice style are not new and come from the UNIX world, where many independent programs in the operating system can perform particular tasks and can be combined as desired. Richard Raymond describes that UNIX Approach to The Art of Unix Programming (Raymond, 2003).

The only way to write to complex software that won't tumble on its face is to keep its worldwide sophistication down to construct it from simple and easy parts connected by well - defined interfaces, so one can hope to upgrade a part without breaking the entire software (Raymond, 2003, p. 14).

Raymond also describes the core of the microservice style. The modularization of software should take place over small parts with robust interfaces. The definition for the microservice style of Fowler and Lewis specifies the style closer. It is at the same time the basis for most of the definitions of microservice itself (Lewis & Fowler, 2014).

In short, the architectural style of microservice is a strategy to creating a single service as a suite of tiny services, each running in its own process and interacting to HTTP resource API with compact processes. These services are developed around business skills and can be utilized independently through completely automated mechanism for deployment (Lewis & Fowler, 2014).

The concept does not prescribe a programming paradigm, but places demands on the organization and infrastructure of development. The demand for automatic deployments aims to apply continuous Delivery and DevOps consistently. Only through much automation can the additional Justify and manage the expense of microservices over monoliths. Among other things, new container technologies and the ever-simpler use of Cloud offerings have significantly shaped the microservice style. Also influential for the organizational aspects of the Microservice style are insights from Conway's Law. Only by structural changes of the development departments which enables the development of independent microservices. If you forget the correct technical separation of services and teams when implementing microservices, you will become one closely coupled microservice architecture that is similar to a deployment monolith because the services are not independent. Thus, the approach to modularization and the boundary of business areas is probably the most important furthermore, in the meantime most difficult perspective for a clean Micro Services style. How this professional classification corrected is determined by definition and is different from case to case. Specialist business areas have no hard limits and can sometimes be bigger or smaller. It is recommended to start with a slightly rough subdivision. As a guide for the modularization Domain Driven Design can serve. The resulting Bounded Contexts are considered delimitation for respective teams and their services with their data schemas.

3.5 Communication Mechanism

3.5.1 Representational State Transfer (REST)

REST is a comprehensive way to design interfaces for microservices. This concept uses generally recognized techniques of the World Wide Web such as HTTP or Hypermedia and builds on the request methods and response codes contained in HTTP. The three main aspects of this concept are resources, representations, and states.

Resources	Any information that can be given a name is potentially a resource. physical objects such as a person or abstract information like today or yesterday's weather can be resources. Even a transaction between two accounts should not be represented as an action or a verb, but rather as a stand-alone resource transaction that describes a deal between two accounts. After Roy Fielding's definition is a resource not directly an entity, but rather a mapping to a set of entities. Where at a particular time exactly one concrete Entity is referenced from this set. It is even possible to define resources that pointing to an empty set of entities (Fielding & Reschke, 2014).
Representations	All resources are represented differently. For example, it can be presented with classic text without awards; it can be in a compelling format like XML or represented in a lightweight format like JSON. Another resource may refer to a multimedia entity and will, therefore, be described in one Multimedia compliant file format. Thanks to the use of HTTP can get over Content Negotiation Mechanisms An agreement of matching representation between Client and server instead of and so an even greater decoupling of these can be achieved (Fielding & Reschke, 2014).
States	In general, the interaction between two participants via a REST interface should be stateless. The server should not have to remember an application state of the client, so any request a client can be considered the first contact between client and server. The state the resource is managed on the server (Feng, Shen, & Fan, 2009). There is no standard in which REST is defined, but rather the principles of the HTTPS standards and best practices must be stuck to. For example, a REST interface should have determined behavior for the corresponding HTTP operations.

Table 3. 1 Aspects of REST (Fielding & Reschke, 2014) (Feng, Shen, & Fan, 2009).

3.5.2 Messaging

Messaging is based on sending messages and is typically used asynchronously. Theoretically, the request-response scheme can be implemented with messaging by sending a message and waiting for the response message. The Characteristics of messaging systems are much better suited to event-based communication. In a highly distributed system such as a microservice architecture, which with latencies and network vulnerability, messaging systems can leverage their power (Wolff, 2016). Messages survive the failure of the network because they are cached in the systems. They are delivered as soon as the recipients are reachable again. In the case of processing errors at receivers, the processing is tried yet before an error message is sent to the issuer. The asynchronous approach allows messaging systems to handle network latencies bypass much better. The systems make it possible to decouple the transmitter and receiver completely. So, it is possible, among other things, that with a Message multiple recipient are reached. Some messaging systems allow transactions in distributed systems, helping developers maintain consistency. Although messaging systems are ideally suited to the requirements of microservice architecture care must be taken to see which functions are transferred to the system. A messaging system in the form of a lightweight message broker to recommend, focusing on the delivery of messages and, where appropriate, transactions take care. If too many functions are transferred to the central system, the decentralized approach of microservices will be broken and the principle of dumb Pipes. Also, message-based systems increase the complexity of the architecture and require additional infrastructure and know-how. Developers need to be asynchronous with the peculiarities Communication will be familiarized and without a good monitoring system, it will difficult to fix a malfunction of the software system (Newmann, 2015). Last it should be aware that although message brokers are designed for robustness, such systems represent a single point of failure. A failure of such a system has, as a result, all communication between services collapses (Wolff, 2016). When used correctly, event-based messages enable a very strong decoupling between the microservices, since they only transport facts as events and not use commands to distribute instructions to other services. Every service can be on other events react as it sees fit. Is a service faulty and does not ship? Events, this does not affect other services.

3.5.3 Remote Procedure Calls (RPC)

RPC is a technique to call functions across process boundaries. In principle, Therefore, they are also suitable for having several microservices communicate with each other. An interesting approach is pursued by the Apache Thrift Project, which was initially developed by Facebook was presented (Slee, Agarwal, & Kwiatkowski, 2007). Thrift is a library designed to deliver efficient and reliable communication between backend services. It uses a language-independent initialization file specifying data types and interfaces. Using code generators are generated from the definitions of the code required for RPC communication in the supported languages (Foundation, 2018). The library supports simple data types and structs, which are also used for exception handling. Also, you can manually assign identifiers to the data fields or generate automatically, which later for the versioning of the interface be used. The big advantage of Thrift over the established structure-based protocols is the Data transmission of binary data. This eliminates a complex parsing of for example XML and enables more efficient communication (Sumaray & Makki, 2012).

3.5.4 API gateways

API gateway is accustomed compose one or 2 microservices. Microservices supply higher measurability as compared to monolithic applications. In order to scale one part of the monolithic application, all the parts of associate degree application should be scaled. In order to scale one element of the monolithic application, all the parts should be scaled. This is often the disadvantage of stone design, as a result of we've got to waste the resources and this addition will increase the quality. However, in microservices, we can scale those parts of an application solely that we would like to scale rather than all the components. Therefore, scaling in microservices is simpler than stone.

API Gateway and its need:
In all SOA, a few concerns are shared among all (or most) administrations. microservice-based architecture isn't a special case. As we said in the main post, microservice is grown nearly in confinement. Cross-cutting concerns are managed by upper layers in the product stack. The API gateway has the following task (Peyrott, 2015):

- Verification
- Transportation Safety
- Balancer
- Request Handling for fault tolerance
- Reliance goals
- Transportation models

Authentication/Verification
Gateways execute some validation called as authentication for each request. As indicated by principles that are explicit to each service, the gateway either courses the demand to the asked for microservice or returns a mistake code (or less data). All gateways when sending microservices add verification data behind the request. This enables microservices to execute explicit client rationale at whatever point required (Peyrott, 2015).

Security
Numerous gateway work as a solitary passage point for an open API. In these cases, the gateways are responsible for transport safety and after that dispatch the solicitations either by utilizing an alternate secure channel or by evacuating security imperatives that are a bit much inside the interior system. For example, for a RESTful HTTP API, a gateway may perform "SSL end": a safe SSL association is built up between the customers and the passage, and proxied demands are then sent over non-SSL associations with inner administrations (Peyrott, 2015).

Load-balancing/adjusting
Under high-load situations, gateways can disseminate demands among microservice -instances as indicated by custom rationale. Each administration may have explicit scaling constraints. Gateways are intended to adjust the heap by considering these confinements. For example, a few administrations may scale by having various instances running under various inside endpoints. Doors can dispatch the request to these endpoints to deal with the heap (Peyrott, 2015).

Request-dispatching
Indeed, even under ordinary burden situations, portals can give custom rationale to dispatching demands. In enormous models, inner endpoints are included and expelled as groups work or new microservice instances are brought forth (because of topology changes, for example). Doors may work a couple with service registration/discovery procedures or databases that portray how to dispatch each request. This gives excellent adaptability to improvement groups. Also, flawed services can be steered to reinforcement or conventional services that enable the demand to finish as opposed to fizzling (Peyrott, 2015).

Resilience goals
As the microservices manage specific concerns, some microservice-based models will in general progress toward becoming "garrulous": to perform necessary work, many requests should be sent to a wide range of services. For accommodation and execution reasons, gateways may give exteriors ("virtual" endpoints) that inside are directed to a wide range of microservices (Peyrott, 2015).

Transport transformations
Microservice are generally created in disengagement, and improvement groups have significant adaptability in picking the advancement stage. This may result in microservices that arrival information and use transports that are not advantageous for customers on the opposite side of the passage. The gateways must play out the vital changes so customers can even now speak with the microservices behind it (Peyrott, 2015).

Transport safety
Transport safety/security is handled through TLS: all public requests are received first by a reverse nginx proxy setup with sample certificates.
It can be concluded that it very well may be presumed that API gateways are a basic necessity of any microservice-based engineering. Cross-cutting concerns, for example, verification, load adjusting, reliance goals, information changes, and dynamic demand dispatching can be dealt with advantageously and conventionally. microservices would then be able to concentrate on their particular undertakings without code-duplication. This outcome is the increasingly agreeable and quicker improvement of every microservices (Peyrott, 2015).

3.5.5 The Service Registry

The service written account could be a piece of information inhabited with info on the way to dispatch requests to microservice instances. Interactions between the written statement and alternative elements are often divided into two teams, every with two subgroups (Peyrott, The service registry, 2015):

- Interactions between microservices and the registry (registration):
    a. Self.
    b. Third-party.
- Communications between clients and the registry (discovery)
    a. Client-end
    b. Server-end

Registration
Most microservice-based structures are in steady development. Administrations go here and there as advancement groups split, improve, belittle and do their work. At whatever point an administration endpoint changes, the registry has to think about the change. This is the thing that registration is about: who distributes or refreshes the data on the most proficient method to achieve each administration (Peyrott, The service registry, 2015).

Self: Self-registration powers microservices to communicate with the vault independent from anyone else. At the point when an administration goes up, it informs the registry. A similar thing happens when the administration goes down. Whatever the registry requires the administration itself must give extra information. microservices are tied in with managing a solitary concern so self-registration may appear to be an enemy of example. Be that as it may, for straightforward models, IT may be the correct decision (Peyrott, The service registry, 2015).

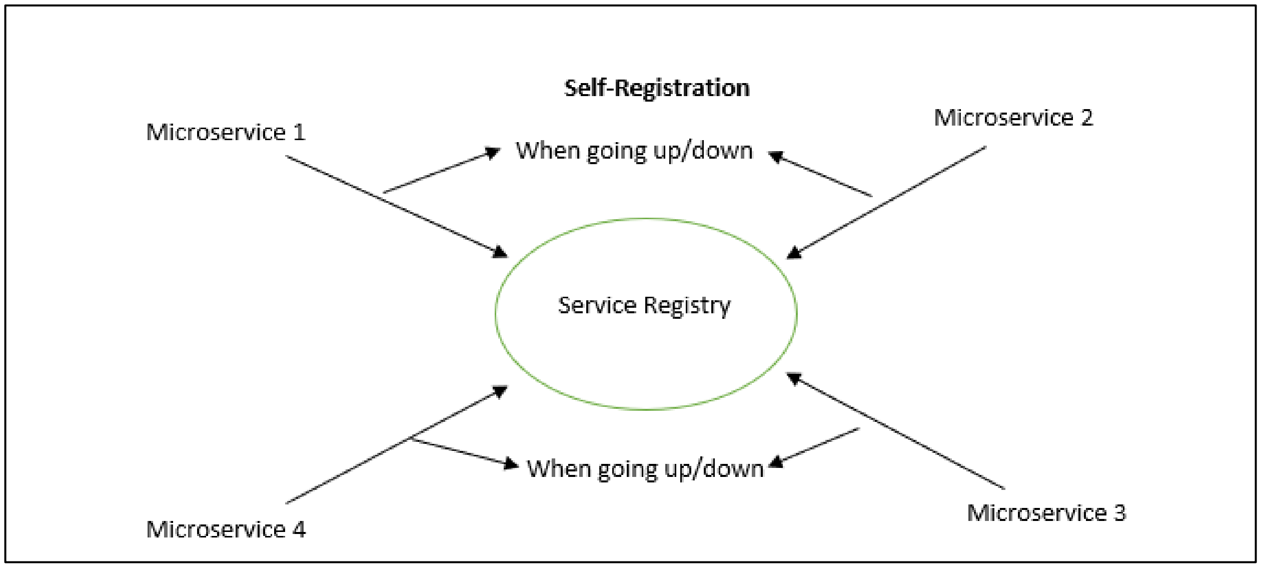

Figure 3. 3 Self-registration (Peyrott, The service registry, 2015).

Third-party: Third-party registration is typically utilized in the business. For this situation, there is a procedure or administration that deals with every single other Service. This procedure investigates or checks every now and then which instances of microservices work, and of course syncs the administration booth. Extra information may be given according to support config records (or approach), which the enlistment procedure uses to upgrade the database. Outsider registration is ordinary in models that utilization instruments, for example, apache zookeeper or Netflix Eureka and other administration administrators. Third-party registration likewise gives different advantages. For example, what happens when an administration goes down? A third-party registration administration may be designed to give safe fallbacks to administrations that fizzle. Different approaches may be executed for different cases. For example, the service - registration method may be notified of a high - load state and thereby include another endpoint by informing for another microservices - procedure or VM to be instantiated. These likely results, as you can imagine, are essential for huge structures (Peyrott, The service registry, 2015).

Discovery
Discovery is the partner to registration from the perspective of customers. At the point when a customer needs to get to a service, it must discover where the administration/service is found.

Client-end: Client-side discovery powers customers to inquiry a revelation administration before playing out the real demands. As occurs with self-registration, this expects customers to manage new concerns other than their principal objective. The discovery administration could be situated behind the API gateway. In the event that it isn't situated behind the gateway, adjusting, verification, and different cross-slicing concerns should be re-executed for the revelation administration. Moreover, every customer has to know the fixed endpoint (or endpoints) to contact the discovery administration. These are for the most part hindrances. The one major preferred standpoint isn't coding the vital rationale in the passage framework (Peyrott, The service registry, 2015).

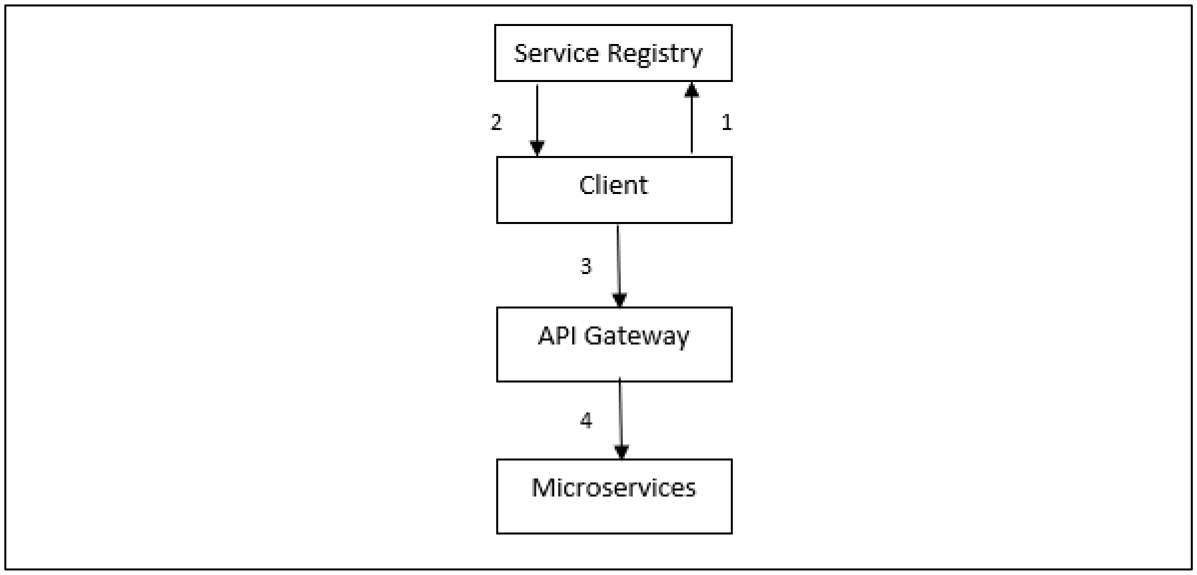

Figure 3. 4 Client-side discovery (Peyrott, The service registry, 2015).

Server-end: Server-side discovery urges the API gateway to handle the revelation of the correct endpoint (or endpoints) for a demand. This is ordinarily utilized in greater models. As all solicitations are specifically sent to the door, every one of the advantages talked about in connection to it apply (see API gateway). The gateway may likewise actualize revelation reserving, with the goal that numerous solicitations may have lower latencies. The rationale behind store refutation is explicit to a usage (Peyrott, The service registry, 2015).

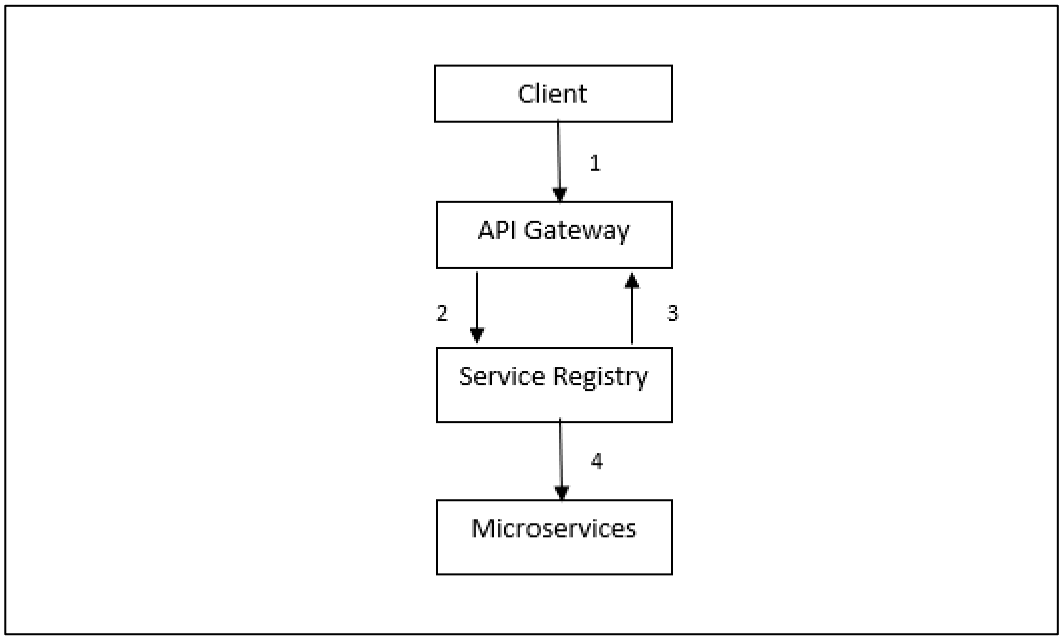

Figure 3. 5 Service-side discovery (Peyrott, The service registry, 2015).

3.5.6 Dependencies and Data Sharing

In a microservice-based design, administrations are demonstrated as disconnected units that deal with a decreased arrangement of issues. In any case, efficient frameworks depend on the collaboration and combination of its parts, and microservice models are not a particular case.

In a conventional monolithic application, conditions frequently show up as method calls. It is typically a matter of bringing in the correct pieces of the task to get to their usefulness. Basically, doing as such makes reliance on the various sections of the application. With microservices, every microservice is intended to work without anyone else. Nonetheless, now and again one may find that to give certain usefulness, access to some other piece of the framework is fundamental. In solid, some piece of the framework needs access to information overseen by another piece of the framework. This is what has usually known as data sharing: two separate parts of a framework share similar information. Nonetheless, rather than multithreaded applications, information sharing in a circulated design has its own arrangements of issues (Peyrott, Dependencies and Data Sharing, 2015):

- Can we share a separate database? Does this scale as we include service?
- Can we handle vast volumes of information?
- Can we give consistency ensures while lessening information get to conflict utilizing straightforward locks?
- What happens when an administration created by a group requires a difference in the pattern in a database shared by different administrations?

Unlike in a monolithic application, microservices can be simple Function or method calls are not made within the process. Views usually have to be done over the network.
In general, there are two possibilities (Newmann, 2015) which are described below:

Request / Response
Here, a microservice will send a call to another microservice and expect an answer. This communication mechanism has the advantage of giving the caller direct feedback receives, whether its call was successful or whether errors occurred. This concept is based on Instructions and allows the calling service to control and control the process flow to a degree.

Event-Based (Publish / Subscribe)
With this option, a microservice will send a message of a specific type for certain events. Further microservices can subscribe to different message types and take appropriate actions. For example, an enrollment service would redirect a user to a new one Course a message New enrollment of user x in course y will send on what others Then services can respond internally.

3.6 Advantages of microservices

Microservice architecture does not have any confirmed definition, and common characteristics instead describe that most microservice based systems share.

3.6.1 Single responsibility

Services usually run on different machines in a microservice architecture. Communication is conducted over the network between them. This approach adds overhead but, on the other hand, it eliminates tight attachments, maintains modularity, and makes understanding of different parts of a system generally easier. In the architecture of the microservice, each service manages its own instance compared with monolithic applications which traditionally use a single database for everything. This helps to maintain a reliable database, as no other service can access the data directly. The data management service knows very well its technicalities and represents as a gateway for other services that wish to use these data.

3.6.2 Autonomy

Due to the independence of the operation of microservices the technology stack is selected new opportunities. Each service can use entirely different technologies for microservices. It can be written in a particular language of programming and work on a certain platform that best suits its needs. It is inconvenient to provide language diagnostic API to enable you to communicate on different platforms with other services. However, it does not come without overhead using multiple technologies, and several organisations are trying to standardize on one single platform (such as JVM). In terms of storage of data, the previously described independent databases not only mean each service has its own case, but can also use completely different types of databases than other services. Each link in the chain the of the system (in this case an independent, independent service) can use the most appropriate storage technology.

3.6.3 Heterogeneity

The fact that microservices run independently creates new opportunities when it comes to the selection of the technology stack. Microservices each service can use completely different technologies. It may be written in a specific programming language and run on one particular platform, the one that suits its needs best. It comes with a drawback of providing language-agnostic API so it can communicate with other services on different platforms. However, using multiple technologies does not come without overhead, and many organizations try to standardize on a single platform (for example JVM). When it comes to data storage, independent databases described before do not only mean that each service has its own instance of the same database system, but it can also use entirely different kind of database compared to other services. This approach is called polyglot persistence. Each part of the system (in this case is an independent and autonomous service) can use a storage technology that is most suitable for it.

3.6.4 Resilience

Unlike components of monoliths, microservices are distributed among a large number of machines and make remote calls between each other. This introduces new kinds of failures developers (and administrators) Microservices need to deal. If one component of a system fails, this failure must stay isolated and not cascade to others. The service whose supplier has been unable should be able to respond as smoothly as possible.

3.6.5 Scaling

All needs to be scaled together with a massive monolithic application, even if only a small part of the system is limited in performance. This is not a problem for microservices and can be independently scaled on the basis of production requirement. Various services are utilized differently and more frequently duplicated and reproduced services are possible when only one instance of a less commonly used microservice can be used.

3.6.6 Easy deployment

The entire application must be constructed, evaluated, simulated, and dispatched for a minor modification in a large monolithic application. Even with such a big system it will be time consuming and it is likely that one component can be disrupted by a single scale in a specific section of the system. Individual services can be altered and independently deployed in case of a microservice architecture. In conjunction with different automation tools, including continuous integration (CI) and continuous delivery (CD), the entire software release process is speedier and there are no risks associated with changes.

3.6.7 Composability

All large systems are split into different reusable components. They are developed. This is usually done by utilizing existing libraries and having similar components of the app rather than writing the same code dozens of times. With remote interfaces, Microservices next step gets closer and offer common functionality. This approach enables the multiple use of a service by external Microservices parties in many various ways. Its methods may be referred to in the same data center by another service across the globe.

3.6.8 Replaceability

The service norms of microservices are strictly defined. Their code just is impossible to be reutilized but can operate via subsystem interfaces that are clearly defined. This and the small number of developers enables the codebase of whichever service to be rewritten completely, eventually resulting in the API remains unchanged. Since microservices have no dependencies on the compile-time (run-time only), they can be implemented separately. This significantly reduces change costs and speeds the entire release process along with fully automating the deployment.

3.6.9 Small teams

Business technologies organize microservices. Under Conway's law, "any system design organization produces a layout whose framework is the copy of the communication framework of the organisation" (Conway, 1968). This applies both to micro services and to other software components. In case of very small and responsibility of microservices, however, teams can and do usually have the complete lifespan of the services. This enables the teams to be more powerful and autonomous, while also making them accountable for their work, because they face the effects of their judgments and also the actual decisions to implement.

3.7 Disadvantages of Microservices

- Communication overhead
- Documentation overhead
- Diverse application
- Maintenance complexity and more initial investment
- Enlarge communication
- Gathering the data
- Security
- Testing and more monitoring cost.

3.8 Comparison between microservice and monolithic architecture

Microservice design could be a distinctive technique for developing applications that have adult up in the name in the previous few years. Several application developers adopt it as a primary alternative for making their applications. Because of high measurability, this design supports a good form of devices and platforms. The microservice design is most vital for developing an application during a cloud surrounding. During this design, the complete application is split into a variety of small services that are freelance of every different. Each of those tiny services is responsible for capital punishment some specific feature. The monolithic application handles an HTTP request. It conjointly takes care of retrieving the info, executes the domain logic and updates the info within the info. It selects the HTML views that need to be sent to the browser. In a monolithic application, a little amendment within the design ends up in to revamp and deploy the complete stone. Therefore, it's troublesome to preserve the planning and standard structure of the applying. This makes it troublesome to modify or change one module within the entire application. In microservice design, an application is often divided into multiple elements or elements, referred to as microservices. These elements are severally scaled. This design approach is incredibly helpful once the system has a very high load with a variety of reusable modules. The microservice architectures are easy and secure. The systems designed exploitation microservice design is loosely coupled because of the elements of a system work freelance of every other. This design permits several groups and developers to figure freelance for every different exploitation of this architecture. Below is the comparison obtained from google trends (Google, 2019).

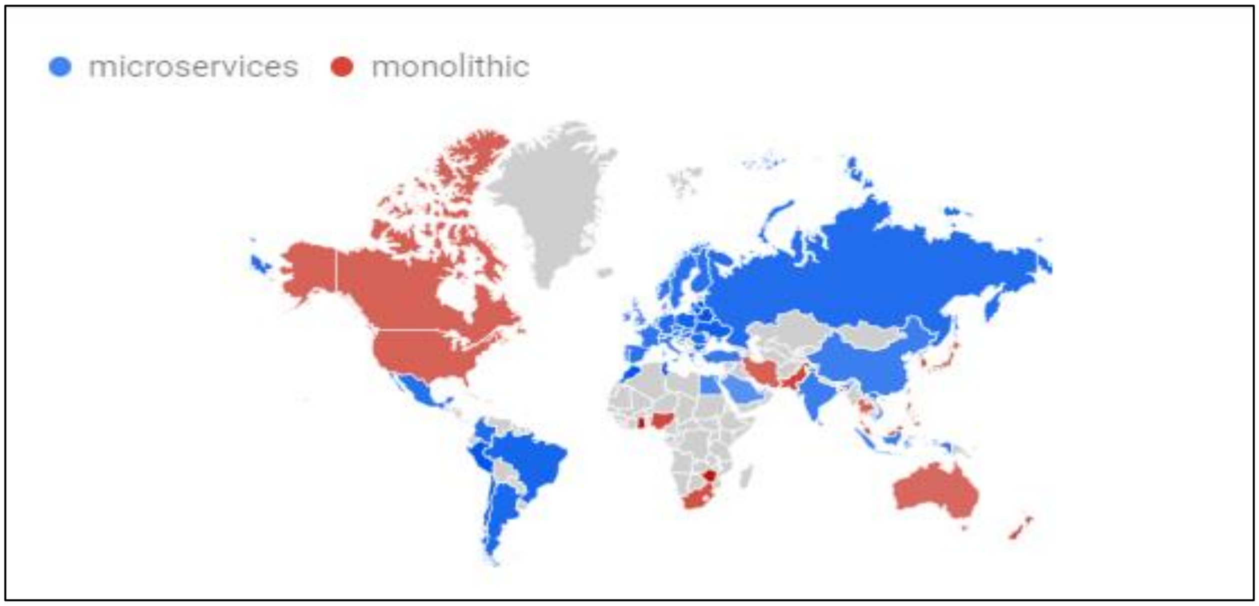

Figure 3. 6 Geographic Comparison between microservices and Monolithic (Google, 2019).

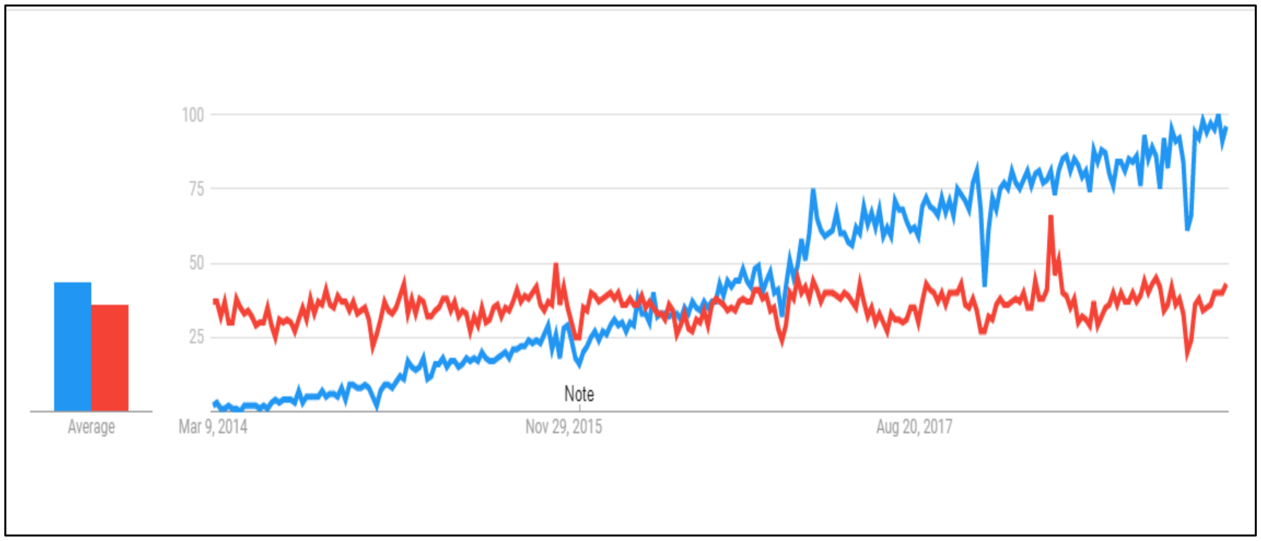

Figure 3. 7 Comparison between monolithic and microservices over the past 5 years (Google, 2019).

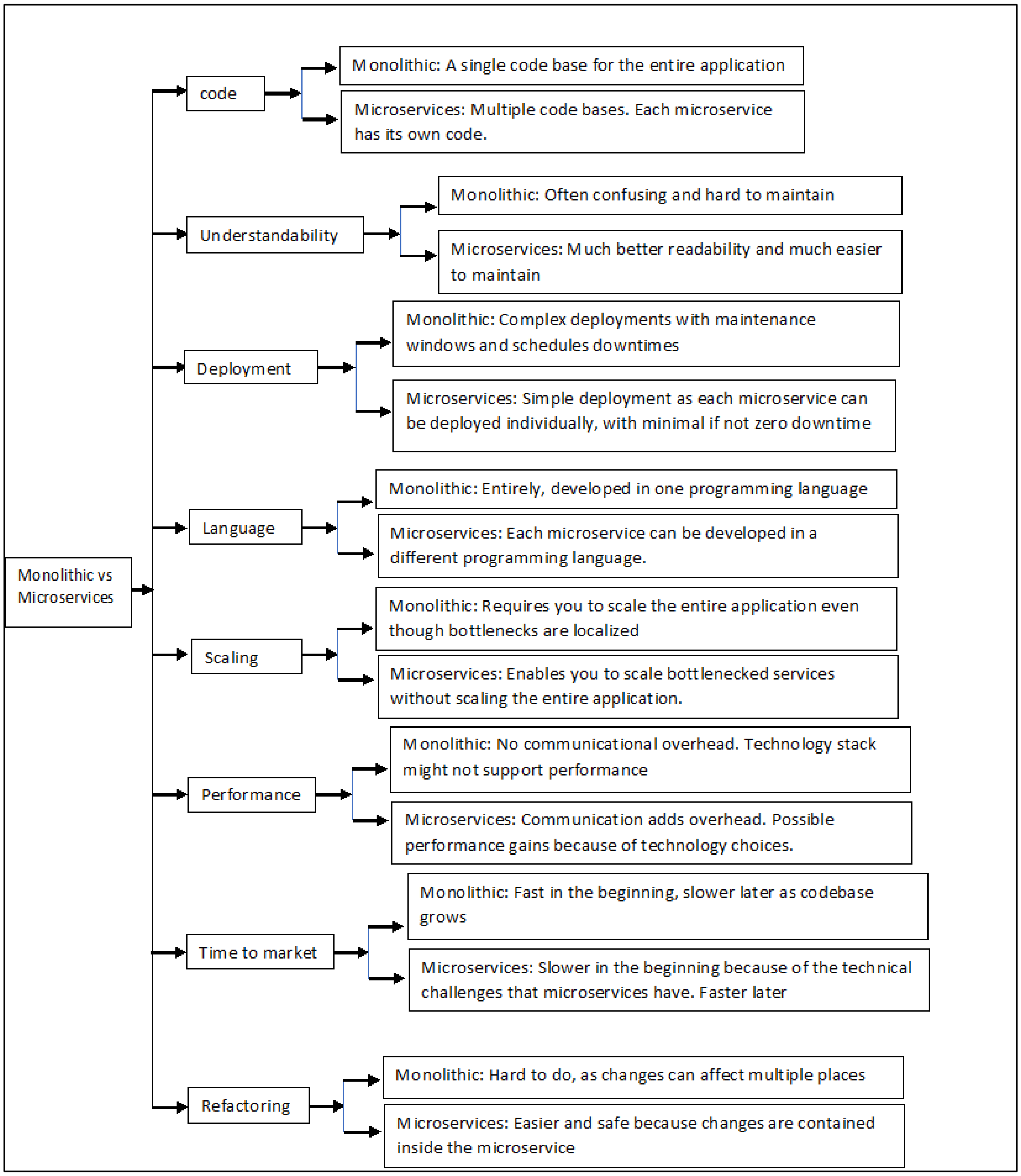
Monolithic vs Microservices
code
Monolithic: A single code base for the entire application
Microservices: Multiple code bases. Each microservice has its own code.
Understandability
Monolithic: Often confusing and hard to maintain
Microservices: Much better readability and much easier to maintain
Deployment
Monolithic: Complex deployments with maintenance windows and schedules downtimes
Microservices: Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime
Language
Monolithic: Entirely, developed in one programming language
Microservices: Each microservice can be developed in a different programming language.
Scaling
Monolithic: Requires you to scale the entire application even though bottlenecks are localized
Microservices: Enables you to scale bottlenecked services without scaling the entire application.
Performance
Monolithic: No communicational overhead. Technology stack might not support performance
Microservices: Communication adds overhead. Possible performance gains because of technology choices.
Time to market
Monolithic: Fast in the beginning, slower later as codebase grows
Microservices: Slower in the beginning because of the technical challenges that microservices have. Faster later
Refactoring
Monolithic: Hard to do, as changes can affect multiple places
Microservices: Easier and safe because changes are contained inside the microservice

Figure 3. 8 Comparison between characteristics of monolithic and microservice architecture (Author)

3.9 Fault tolerance in Microservices

Microservices are designed to run in a highly distributed environment. Such an environment brings a lot of benefits as well as many challenges. In this chapter, the essential characteristics of distributed systems which can negatively influence their behaviour are explained. It also gives an overview of the best practices on how to deal with these challenges and the most popular tools for the Java Platform that help to follow these practices. Fault tolerance can also be called as an adaptation to non-critical failure is a component of the framework that keeps a computer system or network device from notwithstanding any type of failure. Fault tolerance incorporates powerful strides to anticipate such failures in the system (Kaur & Kanu, 2015). Fault tolerance is like providing the system with an antidote against failure such that it does not lose its properties like availability and reliability. A fault tolerant system is a structure that can in case of a bug, endure it and keep on working.

Failure: A failure happens when a traditional framework isn't working accurately along these lines if the framework unfortunate behaviour bears the framework to bomb no less than one of its capacities appropriately, at that point the framework is in a breakdown.

Fault: The reason for this fault occurrence could be because of system failure. The fault is, therefore, a physical malfunction or failure of a component of hardware or software.

Bug: Occurrence of a bug in the system produces unexpected results from the running or executing application.

,,Not necessarily all bugs will lead to faults.,, (Madani & Jamali, 2018).

The system must be able to handle and continue to operate the fault. (Madani & Jamali, 2018).

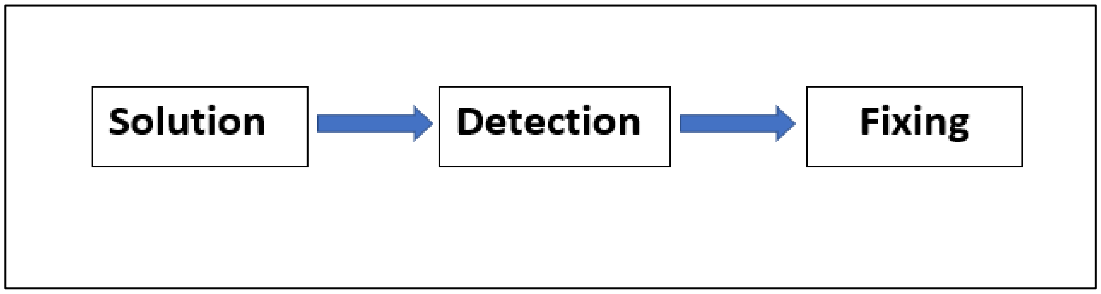

Figure 3. 9 Fault path (Madani & Jamali, 2018).

So, Tolerance can be achieved as follows:

Fault detection: In order to provide each assessment, the first step a system must take is to detect the functions of the fault.

Fault Repair: When a fault is detected by the system, the next step is to avoid or improve the fault (Madani & Jamali, 2018).

Characteristics of Distributed Systems

Distributed systems run in a very specific environment that introduces various kinds of new issues that need to be addressed by architects, developers, testers and administrators. People sometimes assume that these issues are very rare and that they can be ignored. The truth (Rotem-Gal-Oz, 2009) lies beyond, and it must always be taken into account.

3.9.1.1 Hardware

Distributed systems are not typically deployed on expensive high-performance machines, but instead, run on cheap commodity hardware. This can be accomplished because a single machine's performance doesn't matter and the overall performance of the entire system is more important. And when it comes to cost-effectiveness, having a lot of cheap machines is much more convenient than having a few expensive ones. Alongside, there is a slightly higher risk of failure with cheaper hardware. A hard disk drive (HDD) tends to be the most problematic part of the system. HDD failures occur quite frequently in large systems, where thousands of machines are used. They need to be taken into consideration and systems should be built in such a way that they can not only survive such an event but also hide it from the end user and continue to work without any noticeable outage.

Generally, the methods developed for this work include dividing a computational system into multiple modules. Each module in the system has been redundant, so the backup module will continue to work if the failure occurs in one of the modules. Tolerant faults include two types of error management and dynamic recovery (Madani & Jamali, 2018).

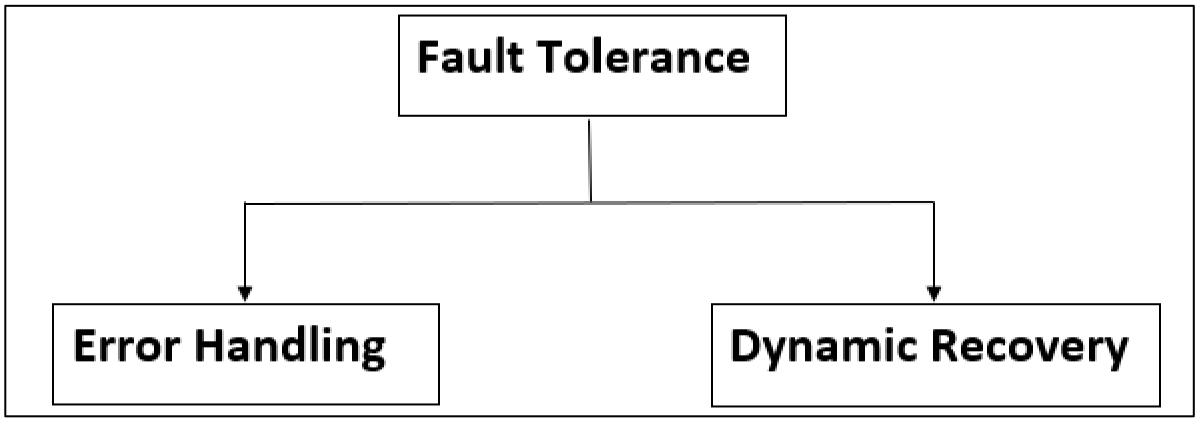

Figure 3. 10 Type of Fault Tolerance (Madani & Jamali, 2018).

Error Handling: Error Handling also called as fault coating is a technique of structural data loss that removes errors in a set of mixed components entirely.

Dynamic retrieval: A dynamic recovery technique is required when duplication of the work is performed at the same time. Self - repair is given by this technique.

3.9.1.2 Networks

They provide a channel of communication between multiple parts of a system and enable it to spread across multiple machines at various locations. But they are not very reliable and need to be addressed by distributed systems. Networks consist of a large number of interconnected devices such as switches, routers or bridges and can fail each of these devices. As a result of failures and unfortunate events such as congestion, there are chances that the message might be lost. They may also lead to a massive problem of network partitioning, particularly for data synchronization. Even if the messages are delivered to the right destination, other issues may still harm distributed system performance. It can significantly reduce an application's processing power if remote calls are treated in the same manner as local calls. Also, it is difficult to test an application behaviour for latency problems as a single machine, or a local network where tests are usually performed is not enough to emulate the actual latency in the production environment. Network bandwidth is another factor to consider when dealing with distributed system performance. Not only can the theoretical bandwidth be quite limited by itself, but the actual bandwidth could be much lower due to packet loss (Dykstra, 1999). In addition to all the issues mentioned above, there is also one critical issue that needs to be addressed in distributed systems, namely network security.

3.9.1.3 Software

Most software problems are rooted in the hardware mentioned above and network issues in distributed systems. Developers tend to underestimate the actual impact of these issues on an application's runtime behaviour that can and usually can have unpleasant consequences. Integration points are systems ' number one killer (Nygard, 2007, p. 46). Each remote connection presents a risk of stability. When making remote calls, connection failures, slow responses or application errors may occur on the other side. Applications may also have access to some shared resources that may make them stuck waiting to release locks. Well-designed systems should address all these situations. Otherwise, it can result in a variety of issues such as large numbers of blocked threads and memory leaks. Failures to cascade are even more severe than a single failure to call remotely. This is particularly true for systems with many interdependent components that use remote calls to communicate together. If no resilience mechanisms are applied and one component fails, it is quite likely that the components that depend on it will fail and their consumers will fail further. Such a failure in a very unimportant component can eventually turn the entire system down. There are some simple techniques described in the next section to help prevent such problems. Software faults can be manipulated using static and dynamic methods similar to those used for handling hardware faults. One of these methods is n - version programming, which in the form of independent programs uses static redundancy. They all do the same thing. A different approach also exists called design variation that incorporates software and hardware fault tolerance by using hardware and software in redundant channels to apply a fault tolerant computer system (Singh & Kinger, 2013). Hardware and software faults are tolerated, but the cost is very costly.

3.9.2 Stability Patterns

In order to reduce the impact of failures in distributed systems, several stability patterns have been created. The most critical microservices in this section are described in conjunction with the motivation to use them.

3.9.2.1 Timeouts

Networks are unreliable. Sometimes connections fail or tend to be somewhat sluggish. This could be an issue when synchronous remote calls are actively expecting an answer. They may end up waiting forever if they don't use any timeout mechanism. While timeouts are frequently used at lower abstraction levels (the networking part of operating systems), they are usually ignored at higher levels (books or applications themselves) (Nygard, 2007, p. 111). For all remote calls, applications should always set a timeout. Without it, the entire application could be hanged by a network failure or a system on the other side down. However, even if timeouts are used, to make a system work as expected, they need to be set carefully. Otherwise, additional problems may arise. If a remote call is waiting too long for an answer, the entire system can be slowed down. On the other hand, a response that would otherwise be received (Newmann, 2015, p. 211) may be ignored if a connection timeout is too fast.

3.9.2.2 Circuit breaker

A circuit breaker considered as a component which is found in electrical circuit works in the following way firstly identifying where the maximum usage is, immediately failing also termed as fail first, and then opening the circuit. Once the hazard is over, it is possible to reset the circuit breaker and return the entire system to a normal working state. In software systems, the circuit breaker is very similar to that in electrical circuits. It wraps hazardous calls and ensures that they do not harm the system. A circuit is closed under normal circumstances, and remote calls are executed as usual. When a failure occurs, it is noted by the circuit breaker. The circuit breaker tumbles as soon as the threshold previously set is crossed and results in failed request output. To check if the system is efficiently working again multiple request are sent back to back. If they succeed, they will close the circuit and execute all remote calls as before. Otherwise, the circuit will remain open, and after some time the same check will be done again (Nygard, 2007, p. 115). Circuit breakers are an effective way to prevent failures in cascades. They work closely with timeouts and serve good protection in the discrete components against unexpected failures. They can even be used when some part of the maintenance system is turned down. Manual tripping of the breaker can also be adjusted to protect the quality of a component (Newmann, 2015, p. 212). Because circuit breakers need to collect some data to function effectively, they are a valuable monitoring place. The circuit breaker design pattern allows your microservice to fail immediately to prevent repeated calls that are likely to fail. The software circuit breaker operates much like an electrical circuit breaker. A closed circuit represents a fully functional system, and an open circuit serves an incomplete system. These circuit breakers immediately trip the circuit open to remove the failure that has occurred. In a software circuit breaker, an additional half-open state exists. After the circuit is opened, it periodically changes to the half-open state, where it checks whether the failed component is restored and closes the circuit after it is considered safe and functional.

3.9.2.3 Bulkheads

One can prevent failure in a specific component by fragmenting a system from manipulating other components and which as a result brings down the entire system. Fragmenting the system or network into several segregated components ensures the continued operation of vital elements or components when one of the less vital elements causes an outage. For example, a flight status queries system will still be capable of providing reasonable latency for reservations or passenger check-in (Nygard, 2007, p. 119). For each connection, one Bulkhead framework uses separate establishment pools. In this case, the other establishment will not be influenced if one establishment/connection pool is exhausted. This ensures that a remote service problem does not affect components that are not connected (Newmann, 2015, p. 214).

3.9.2.4 Fail fast

When a process runs for more than a longer duration to eventually fail this can be summed up as time wastage. Therefore, it is necessary to make the applications smarter, so that they can detect such failures in advance. Making the system or the application smarter can improve its stability and efficiency. The service must try to get all the necessary resources or check their availability before starting the execution. As a result of smart services, the circuit breaker is informed during integration. It can also attempt to pre-allocate memory later during execution to avoid out-of-memory errors. It can fail quickly and save valuable time if any of the resources are not available. It is necessary for the system to identify between system failures caused by an unavailable resources and failures because of incorrect parameters or values. The 2 services should work in the following way where the first one trips the circuit breaker whereas, the second one must let is pass in case of invalid entrance (Nygard, 2007, p. 131).

3.9.2.5 Fallback

When the primary service fails, a fallback service run. It can provide the original service with graceful failure or continuous or partial operation. In our example, we use a circuit breaker policy fallback service. Upon opening the circuit breaker, subsequent service requests are routed to the fallback service immediately until the main service is restored.

3.9.3 Fault Tolerance Mechanism providers

3.9.3.1 JRugged

JRugged is a library that, together with some monitoring capabilities, provides simple circuit breaker implementation. It uses the design pattern of the decorator to wrap potential hazardous method calls. JRugged has mainly three important components namely,

- manage–initializers
- circuit breakers
- monitor performance.

Initialisers provide a way to detach service renovation from its initialisation. The service must incorporate an Initializable interface and forward it to a starter who continues the effort to set the service in in a thread running in the dark. This framework is useful for services with no resources to instantiate during building, but which are eventually supplied at some point in the future. To drive traffic to a tripped service, to avoid this tripping this component can be utilized. CircuitBreakerException is executed for some specific exceptions. Performance monitors are utilized for monitoring any service during execution time. This is used to figure out latency or throughput of a system. JRugged provides circuit breakers although it fails to provide any service for bulkhead and fallback cases.

3.9.3.2 Hystrix

Hystrix is a Netflix library. The definition provided at Github reads: Hystrix library provides fault tolerance and also detects latency. This library works for checking errors or faults of remote (Netflix, 2012).

Hystrix Circuit Breaker Overview

The client isolates the access point to the service by wrapping in a Hystrix Circuit Breaker all network call invocations (this is achieved through commands or annotations at the code level, more details below). The Circuit Breaker intercepts and monitors all invocations on an ongoing basis and acts on certain incorrect conditions.

Closed State: The Circuit Breaker is closed in the state when the service dependency is healthy, and no problems are detected.

Open State: The Circuit Breaker considers the following invocations to be failed and factors them in deciding whether to open the circuit:

- In case of examples like, cannot connect, or service returns HTTP error 404 some type of exception occurs.
- Hystrix's internal thread pool (or semaphore) for the command rejects execution (Hystrix uses thread).

Half-Open State: When a pre-set interval for example 5 or 10 secs is occurred, it indicates the half-open state. The following are the ways for using Hystrix:

- Direct Hystrix API implementation.
- Using Spring Cloud Netflix and the Javanica library–this is a less invasive way to introduce Hystrix to the codebase via annotations of the methods to be safeguarded by circuit breakers.

3.10 Monitoring of microservices

For many reasons, monitoring our application is necessary. First, we need to know if the application is executing, we need to know if any issues that need to be resolved and specifically where they are. Then we need to know the application's current load, how it works, what its bottlenecks are. Then we want to understand how users work with our application; we might be interested in different statistics about the use of individual components and so on. Our application also requires monitoring if it is needed to fulfil a service - level agreement. Monolith monitoring is not advanced. Usually, we have only one powerful hosting machine running our application on an application server. Therefore, we need just care about one hardware that we directly control, it's quality and specification, and we have simple access to various runtime information presented by the application server administration interface from our application. Any custom surveillance is also available in one place on this machine. While distributed architecture with microservices delivers substantial advantages, with many moving parts, it is a much more complex system. The system's internal functions are not as clear and accessible as monoliths. There is no single solitary observation point by default that provides a straightforward study of the application's various statistics. There may be thousands or even more microservices running simultaneously across different networks on different machines in large applications. What's more, in a matter of seconds, these microservices can quickly come up and die. It becomes difficult to collect overall system information in such an environment. So, we need some central point where we could view aggregated monitoring information from individual microservices transformed into some useful results.

3.10.1 Audit logging

Logging audit Concerns our users ' behaviour. Having a record of user actions for security, customer support or other reasons is helpful, sometimes necessary. Developers manage audit logging manually in business code.

3.10.2 Application metrics

Most metrics are usually managed by business code developers manually. Either container schedulers or separate utilities inside containers such as collectd (collectd, 2017) can provide some primary metrics such as processor usage, memory, and disk. We then add these metrics and create useful reports from them, fire alerts if something goes wrong and draw different monitoring graphs for the operations team. Metrics are also used to evaluate agreements at the service level. It must be taken into consideration that we cannot track everything; this could result in ten thousand metrics per second in large systems with many microservice instances and cause an overload of our system as well as our perception. The only concern is on metrics representing business value or vital performance statistics that can help with further development.

3.10.3 Distributed tracing

Tracking distributed helps us to understand our application's performance. Application requests often cover numerous micro-services and perform one or more operations, database requests and so on. External monitoring of responses or metrics for each microservice doesn't tell us why such an application is sometimes quick and sometimes slow. We have to track requests throughout the system and login time for each operation and other useful intermediate information. This is done by assigning a unique ID to each external request and passing it on to our application together with each subsequent call. Individual microservices can then record and forward related information to a centralized tool with the request ID. This tool adds all the data and displays them well arranged or even displays a reliability chart between our microservices. The tracking of calls between micro services is usually managed by the infrastructure or frame, but many times the developer must cooperate to add valuable, relevant information to the track.

3.10.4 Health check API

We could not suppose that our microservice works only because the container is in operation. Health check API. Moreover, we cannot even suppose that the microservice works and can perform its task simply because it is operational. Therefore, in our microservices, we need health inspections. Health controls are repeated audits of the ability to provide incoming demands for microservices. They usually repeat in seconds and expose the audit results to a microservice REST API for easy access, typical of this type of service is http:/192.168.122.85/health for example:

- Closed threads–there is any deadlock 2 in our microservice JVM will work on it.
- Database-available database, free database connections available.
- Connection-our microservice is connected, they respond to other microservices.
- Host-there is sufficient host disk space; the host doesn't run out of memory.
- Application logic-all business operations can be carried out; any resources are missing.

Some monitoring tools can be used to collect health check outcomes, but these primarily serve as an indication of how container schedulers are operating. The container planner can see a container running, but it doesn't know whether or not the instance of the microservice has been initiated alright. due to this we can set a container compiler for the API checks up and if the API is unavailable or faults have been returned, the container compiler knows that the incoming demand cannot be routed and will attempt to restart it.

3.10.5 Exception tracking

Tracking with exceptions Things should fail, at least temporarily, in the microservice architecture. Network connection times out, data are not synced, some services are not accessible for the moment. There can be a lot of exception to Microservices. Therefore, a tool can be used that adds these exceptions, deducts, filters traditional exemptions and shows the problems of the designers for further research. No support from developers in the business code is needed for this pattern.

3.10.6 Log aggregation

It's self-explaining. It is likely that our microservices will generate log files containing errors, warnings, information and debug messages. We need a tool to collect this log file at the same location with filter and search functions. If certain events are to be notified, the tool should also have a warning setting.

3.10.7 Monitoring solutions

Many proprietary solutions are available, which probably work better than versatile tools. These solutions are included in the APM category which represents all-inclusive universal analysis tools that offer functionalities like machine intelligence, main metrics and regulatory compliance, and fascinating data in many variables to help us identify a source of difficulties. Some of these closed source solutions offer unification with many new container schedules, compilers, paradigms, databases, network devices, alerting and other tools, Instana, Netsil, AppDynamics, DynaTrace.

3.10.7.1 OpenTracing

OpenTracking is probably the most useful model in the architecture of microservices. Other measurements cannot solve many problems because they do not follow the entire story of what happened step-by-step in our application. Many distributed tracking solutions are already available. Most solutions automatically create part of the traces by recording the functions invoked; developers, however, usually must add their steps in the trace and other relevant information. So they link their code to the selected tracking solutions and lock the application in the vendor, which is always good for long-term avoidance. Open Tracing (Tracing, n.d.) has been established for distributed tracing as a vendor-neutral open standard. It is a project that is very young. It resumes the tracking API from actual vendor implementation, which means that we are calling Universal Best Practice Functions, defined by the OpenTracing Standard. We can change the implementation quickly at any time. OpenTracing does not guarantee that various tracking implementations can be used by different components, but this is generally not necessary, because we usually want all tracking data in one tool. In Go, JavaScript, Java, Python, Objective-C, C++ Open Tracing has been defined. It has integrations in various stages and has a tracking program for Zipkin, Hawkular APM, Appdash, Instana, Jaeger, Light Step, in Springboot, Dropwizard, Django, Go Kit and other developers. Time and name of beginning and end are always the routes. They may hold further details in the form of tags, which are commonly understood by tracking implementation, and can hold key / value logs for the entire range which refer to specific time. They can also contain tags. Spans can start other spans, either in series or in parallel, by defining new span as a parent child. Thus, the span of your child inherits the transaction ID of your parent. Connected lengths produce a trace-building acyclic graph. The story of a transaction as it propagates across our system is told by traces. Traces recorded in the tracing tool are usually displayed in the form of a line graph showing the internal spans of all traces as shown in the image or in terms of time and serial or in parallel.

3.10.7.2 Zipkin

Zipkin (Zipkin, n.d.) is an OpenTracing distributed tracing tool. It collects the transmitted traces and manages the data collection and search. Applications, trace lengths, annotations or timelines are used to filters or sort traces. Zipkin can produce concrete tracks in the form of the chart described above and shows additional information like its tags after having clicked on spans. Zipkin can also produce a simple dependency graph based on recorded traces between components. You can either store the data in MySQL, Cassandra, or Elasticsearch.

3.10.7.3 Hawkular APM

The Hawkular APM is an open source tool that supports OpenTracing. The above-mentioned line graph does not display trace charts, but it can display a communicative reliance chart between full application components and individual trace charts. It can also generate numerous graphs from traced and measured business transactions (Hawkular, 2016).

3.10.7.4 Metrics library

Metrics (Metrics, 2010) is an open-source library for measurements, which is a part of the Dropwizard framework. It is probably the best metrics library for Java. It is effortless to use, has modules for common libraries like Jetty, Jersey, Apache HttpClient and can report metrics to JMX, Graphite or Ganglia and many other. It provides various metric types and also a health check API. The central part of the library is MetricRegistry class. We register all our metrics to created instance of this class, and it reports their values in given intervals to chosen reporters. The list of supported metrics follows (Metrics, 2010):

- **Gauge -** the simplest type, when reported it returns just a current integer value from a given function.
- **Counter** - a simple integer metric, provides inc(value) and dec(value) methods for incrementing and decrementing its value.
- **Histogram -** a histogram measures the distribution of values in a stream of data, it provides values like min, mean, max, standard deviation, and 75%, 95%, 99% quantiles. It provides an update(value) for adding a new value to its reservoir of cached values from which it computes results. Reservoirs can have different implementations, some cache results for a specified shorter time, some specified amount of ticks, others use more advanced algorithms to provide more long-term representative results.

3.10.7.5 Graphite

Graphite (Graphite, 2006) is an open-source tool for the monitoring of metrics (Graphite, 2006). Graphite has a huge number of collectors and integrations in other monitoring and visualization tools. Graphite is a push-based tool that requires parts to send their metrics to

- Carbon
- Whisper
- Graphite Web Graphite.

Under its unique name, metrics, along with value and time, are listed in Graphite.

3.10.7.6 Prometheus

Another open source monitoring tool is Prometheus (Prometheus, 2014). It has a pull-based approach as opposed to graphite. An interface that Prometheus scraps for values must provide Microservices. Prometheus scans a range of configured IP addresses to find running microservices. it is like Graphite a multivariate database, but it enables the information to be expanded by tags in addition to simple name-value time records. Some identifiers are implicit, such as the hostname that the record was scrapped from, and the programmer can add several. Because of these tags, users can then filter the values.

3.10.7.7 Grafana

Grafana (Grafana, n.d.) is an open - source data monitored dashboard. It has a highly elegant user interface and a large number of data source implementations for data collection. It also offers numerous notifying and codecs.

3.11 Related Work

3.11.1 CAP Theorem

Eric Brewer's states that consistency (C), availability (A) and cannot satisfy partition tolerance (P) simultaneously in a distributed system. It applies meanwhile occupied (Gilbert & Lynch, 2002). Consistency in this context means that all nodes of the system can work with the most up-to-date data and, in the case of queries, no answer based on old data return. Availability means that any request from a user to the system will make sense at some point is answered. From a practical point of view, meaningful availability is often gone given if the response times are several seconds. Partition tolerance refers to the distribution of the system across multiple network nodes. The system should also work if network partitions occur. That is, in case of communication errors within the system, allowing nodes to be in at least two groups divide that cannot communicate faultlessly with each other — for example, one broken connection between two data centers that share a system. It is well known that communication networks are not reliable. A system usually needs to meet the partition tolerance requirement to be a secure function. It ought to be notified that the trade-off among consistency and accessibility can be picked diversely for individual pieces of the framework. It thus results in Compromise for the whole system. So, some features require strict consistency and requests with errors answer instead of returning incorrect information. But it can work for other tasks be meaningful to give up the strict consistency and instead of less current or not provide complete data so as not to upset the user.

3.11.2 CAP Theorem and Microservices

As we are dealing with distributed software systems in microservice architectures, the Consideration of the CAP theorem in this context unavoidable. The classic relational Database system from a monolithic 3-tier architecture is a central point at which to ensure data consistency through transactions in the form of two-phase commit can. In a microservice architecture, every service is assigned its personal database with its data sovereignty, which means that not only business logic but also business data is distributed. Operations on distributed business data from different microservices require a trade-off between consistency of data or availability of the system.

3.11.3 Conway's Law

Frederick P. Brooks, Jr coined the term he has a publication of Melvin E. Conway quotes and calls them Conway's Law (Brooks, 1995). The law states that Organizations constructing and building systems are forced to produce creation that are versions of these Organizations ' communication structures (Conway, 1968).

Although this law applies to systems of various types, for software systems, we conclude that the communication structure of the evolving organization is reflected in the system design and thus in the software architecture.

Consider an organization of the development department divided into functional groups. Each team consists only of specialists for their area. If one were to assign to these teams the common task of developing a new system, one obtains according to Conway's Law, with high probability a software system in the classic 3-Tier Architecture. The database team would look at the individual data schemas and relational ones. Take care of connections. For example, the business logic developers would design the logic of the system as an application in a high-level object-oriented language, wherein the degree of modularization of the application of the communication within the Teams would be dependent. The team for the user interfaces would be suitable for a Layout and a fitting presentation. There will be specified interfaces between the database, logic, and UI, as the communication between the teams is not natural as pronounced as within a team.

Consider another structure in which there are several teams, each one individual Specialists from the functional areas included. Ideally, these teams are for your own responsibility for the technical fields. Now each team has full competence for developing a complete application, and the team can independently take care of their area and provide the software with the necessary features. Communication within the team can be done in shorter ways than via Department boundaries. Changes that affect all functionalities are more comfortable to perform. The global structure of the system now consists of encapsulated applications, because the communication between the technical teams is usually not informal, but should be limited to functions at the interface, which is a team of one other needed.

3.11.4 Conway's Law and Microservices

Microservices should be stand-alone applications and independent of others and can easily evolve. Following with a traditional business organization functional area, you will encounter difficulties in implementing each service. In the context of microservices, Conway's Law is more inversive. The Effects of Conway's Law can be used to more or less automatically preserve the desired microservices architecture by giving up the functional business organization for the benefit of small and multi-functional teams.

3.11.5 Domain Driven Design (DDD)

Domain Driven Design is a collection of patterns for the structured modeling and implementation of software for complex application domains. Generally, Domain Driven Design is actively concerned with looking at the functional relationships of the domain as crucial points for the structure of software. Essential in the DDD is the strategic design, whereby the overall complexity of an application domain is divided into different domain names, which form a bound context.

3.11.6 Bounded Context

The goal is to develop an all-encompassing language for each subject domain, which is equally understandable for subject matter experts and software developers, and as a basis for context-local Problems of the software should serve. From this, a domain model can be derived, which is valid exclusively for the demarcated context. It contains a representation of the actual professional pattern.

3.11.7 Domain Driven Design and Microservices

A CDM contradicts the principle of independent development of microservices. The Teams would have to agree on a globally valid domain model in a lengthy process. After that, any changes to the model would have to be made with all other users. This model can be agreed and integrated at all interfaces. Another problem is ambiguous words. Specific terms, such as Banks have different meanings in different subject domains. In the case of financial matters, the specialized domains are issued by the credit institution. In other Domains could also mean the building of the same institute or that the same piece of furniture. A differentiation based on bounded contexts is a good indication for the demarcation of microservices. In general, well-tailored bounded contexts are synonymous with a microservice, whereby contextual teams are left to their background into several microservices if they think service is too big or its functionality does not harbor any cohesion. This team solely manages Microservices. However, avoid a microservice that has responsibilities over multiple bounds Covered across contexts and thus handled by two teams. That would be through unclear responsibilities and an inconsistent domain model will sooner or later lead to misunderstandings or bloated data models.

3.11.8 Continuous Delivery

Continuous Delivery (CD) is an approach in software development to make changes to one Reliable and reproducible production of software (O'Reilly, Molesky, & Humble, 2014).
Jez Humble et al. have developed and described requirements for this successful CD must be fulfilled.

Security: CD is mainly based on a deployment pipeline of various tests. Most of them are automated and are supplemented by multiple manual tests. Going through this pipeline can be seen as a validation of the software and can be used as a test protocol in combination with the right log data. The key point for the Safety is the cover and quality of the tests. Each test works on the principle to refute the accuracy of the tested software. If this proves to be no test and there are still doubts about the production capability of the software, the test criteria must be extended become.

Speed: The primary measure of successful continuous delivery is the speed at which changes come into production. That's what CD aims for, with each new one Change to learn more about the quality and users of the software. For the responsible teams, quick feedback on changes made is critical. Additionally, CD attempts to avoid long integration tests by including Continuous Integration. Continuous improvement will shorten the time it requires to pass through the deployment pipeline. So, it is desirable that fixing a mistake by changing fewer lines of code within a few minutes can be validated as ready for production. This requires ever-increasing automation. DevOps is also closely linked with automated provisioning test environments for acceptance and capacity testing.

Sustainability: Behind CD is hidden at the beginning of much additional effort, by the automation of the deployment and the writing of the tests arises. Additionally, needed You also have an infrastructure and software architecture that supports a high degree of automation. The long-term goal, however, is to make economic sense in Thinking small steps and even making small changes directly to the factory bring. CD s therefore seen as a sustainable investment in more agility.

Team: The team has the responsibility to have at any time a current and working version in the main branch, the conscience in the production plant can be brought. If there is a problem with a version within the pipeline, the responsible developers solve this within a few minutes or make changes to withdraw from the main branch. The definition of when a developer sees his work as finished shifts from the time when a change has been flawlessly integrated into the main branch at the time from which the difference ran without errors through all tests and are put into production can.

3.11.9 Continuous Integration

Continuous Integration (CI) is a practice that deals with integrating changes employed at the codebase of a software. She tries to integrate problems to minimize various modifications to software, to facilitate troubleshooting and to increase the reaction rate of changes. Suppose several developers are working on different features of the software. The features are designed independently and then after several weeks of working together in the main branch of the software versioning system to integrate. What a few developers and easy changes with Precise planning can still work well with hundreds of developers with high Probability to cause problems because the various features unforeseen Have dependencies. At best, these problems are manifested by a failure when compiling a current version of the software. In the worst case, they learn Developers but only weeks later from difficulties, since the dependencies just in manual Tests or occur in production. Troubleshooting a large amount of code is then added by the time distance of development and occurrence of the error difficult.

Continuous integration, therefore, requires developers to keep their changes in their own branches for weeks, then integrate the entire package. Instead, smaller partial pacts should be made at short intervals in the main branch to get integrated. This requires automation of compilation and testing processes as well as rethinking the way developers implement features. Developers need to make their changes in a way that they can do the best Daily a cool version of their current state with the stalls of the other developers to integrate. This results in a smaller distance between defect introduction and Detection, which minimizes risk and costs for troubleshooting. Also, CI scores facts. The development progress is better represented since the implemented features are all integrated immediately centrally and thus always a current state of the Development is present.

Overall, there is also better predictability when the central CI infrastructure the same platform versions as a production infrastructure. All changes are tested daily on a kind of reference infrastructure. So, the risk of libraries causing problems in the enterprise is much lower than if the developers only work on their own testing local development infrastructure (Glover, Matyas, & Duvall, June 2007), (Fowler, Continous Integration, 2006).

3.11.10 Deployment Pipeline

Essentially, Continuous Delivery (CD) consists of a deployment pipeline that performs multiple tests one after the other and gives feedback in due time. These tests are mostly automated, but can also contain human tests and manual releases. The modified software goes through different phases one after the other.

Commit phase: In this case, the change of the software is checked in to a versioning system and automatically compiled. Then unit tests are performed to create a to ensure first correctness. So crucial for this phase is an excellent Continuous Integration process to make it easier for multiple developers to integrate changes.

Acceptance tests: In this phase, the professional correctness is checked. Automated Tests check that the technical requirements of the software are entirely fulfilled.

Capacity Testing: The software is technically correct his. Therefore, after a change, the software is subjected to load tests to check if a change has caused a bottleneck. So, a software though can meet the technical requirements, but because of a lousy programming stick under load with large work data no longer has the necessary performance.

Exploratory tests: Only when the previous conditions are fulfilled, the purpose of the actual change tested. So here are mostly the features that led to the change tested. A well-known example here is A / B testing, with two identical ones apart from one different version of the software are operated in parallel in production. Load distribution can so the user behavior on the different versions are tested and analyzed.

Production: If the exploratory tests are also considered good, this version of the Software brought to production as the latest build. For safety, this can be so progressively through Canary releases (Sato, 2014).
Similar to the A / B testing, the new and old versions are operated in parallel and gradually on the new Version increased. Works satisfactorily and tune the corresponding Metrics, the old version can be turned off.

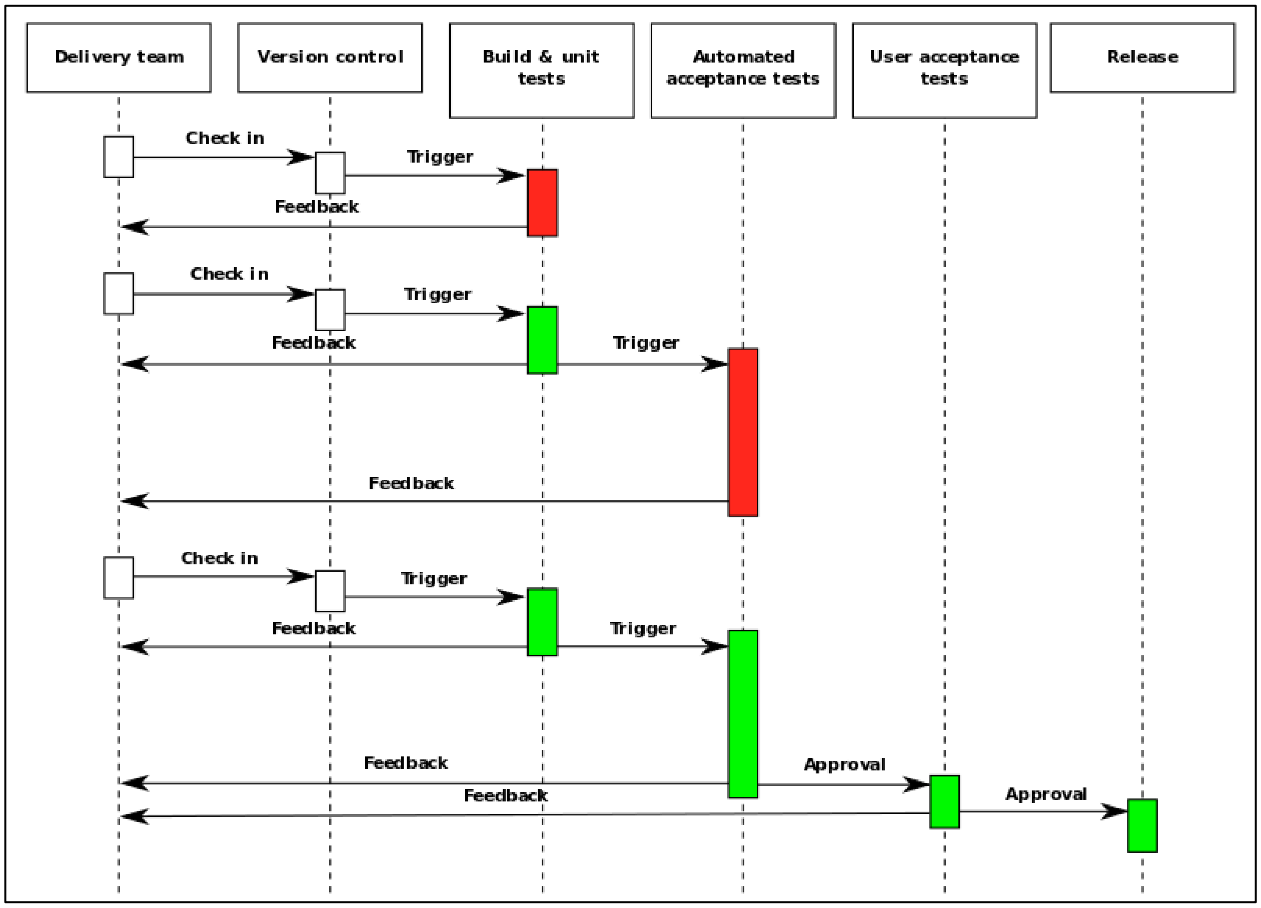

Figure 3. 11 Continuous delivery process diagram (Continuous delivery, n.d.)

3.11.11 Continuous Delivery and Microservices

Microservices and Continuous Delivery are very closely linked (Wolff, 2016).
They facilitate implementing a continuous delivery process. A small microservice can be a pipeline on the effect of their change within a reasonable amount of time. On the other hand, meaningful implementation of the microservice style relies on a working deployment pipeline. Automated execution of this pipeline makes it easy to bring many microservices into production. Manually managing and Testing the various versions of many small microservices would be unnecessary overhead and would severely limit the security gains for software changes.

3.11.12 DevOps

DevOps refers to development and operations. That describes a closer form of cooperation between developers and administrators. First and foremost, it is about the goals of both camps (development and operation) in order to achieve the strategic business goals in the operation of software systems realize that DevOps is an organizational approach that stresses empathy and cross-functional collaboration within and between teams especially development and IT operations in software development organizations, to operate resilient systems and accelerate delivery of changes (Dyck, Penners, & Lichter, 2015).

The company is responsible for this, increasingly sophisticated software systems reliable and as interested in making a few changes as possible. Faithfully Under the motto Never touch a running system, every change in the system brings new mistakes, or a failure of the system has to be planned. This circumstance is for customer-oriented and innovative companies too boring and unexhibited. Developers who want to be customer-focused Develop features and apply changes in the right places. The goal of Developers, therefore, accelerates the cycle of change and feedback. Fixed planned Release dates are extremely limiting for this cycle. DevOps seeks this contradiction through collaboration and blending of competences both sides dissolve. The application of DevOps is the consistent continuation of according to technical aspects changed business organization. Specifically, DevOps can show through an improved organization of development and operation. Thus, enhanced collaboration can be made by integrating operations into technical teams together responsible for a particular area. For example, competencies of company managers are already included in the development and thus one of the Productive environment similar development environment. At the same time, communication between developers and operators is one Teams easier than across departmental boundaries. From a technical point of view, DevOps shows itself through far-reaching automation. As a result, it can be concluded that key features of DevOps are Continuous integration, continuous delivery, and fully automated deployment being key points for DevOps (Brunnert, et al., 2015).

So, the administration of the infrastructure leaves through Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) continue to automate. Infrastructure as Code plays a central role in this Provision of standardized and suitable infrastructure for the respective software.

3.11.13 DevOps and Microservices

Some goals of DevOps and Microservices go in a similar direction. So, should DevOps be designed to make more comfortable and faster changes in the software; specifically, the cycle of change, deployment, feedback, and renewed change is accelerated too. An excellent structured microservice architecture supports this goal. At the same time, DevOps has a positive effect on the independence of technical teams, as each one Team thus receives competencies for the operation and thus autonomously over the release decide on changes. Like the microservice style, DevOps is very cloud shaped. Increasing use of IaaS, which is controlled mainly by API, requires that operations specialists so acquire competencies in software development. Without these skills, a great microservice architecture can be created not manage it properly. In the big technology companies that already have microservices As, a result, active DevOps community has emerged, developing practical deployment and monitoring tools in the form of open source. At the same time are developers the microservices but in the light, the automated monitoring by meaningful Support logging.

3.11.14 Scalability

Scalability describes the ability by adding appropriate resources to increase a system's performance of the same, compensate or improve the speed.

Scalability and Microservices

Efficient and good scaling is becoming increasingly important for companies. The cloud offers for it by flexible IaaS provides perfect conditions to computing performance within of a few seconds to provide. As in your own data center, the same applies in the cloud, that the vertical scaling limits are set and from a certain size without elastic horizontal scaling a reliable operation is not profitable. Microservices provide a way to scale individual components very quickly and accurately, enabling organizations to anticipate unexpected peak loads in individual components intercept. The rolling out of monolithic applications is due to the size and longer start time often not suitable. It will also be valuable resources for unused Components kept, because always the whole software must be duplicated.

3.11.15 History

The term microservice, for the primary time, was employed in might, 2011 in a very discussion at a software package design workshop close to the urban centre. The term microservice was recognized by the same folks in strength, 2012. In March 2012, a number of these concepts was conferred by James Lewis as a case study at thirty-third degree in Cracow in Microservices-Java, the operating system approach. In Gregorian calendar month 2012, the microservice design was conferred by Fred Saint George at Baruco. Andrian Cockroft, for the primary time, applied this approach as fine-grained Software Oriented Architecture.

Microservices are tiny services that accommodate the only single responsibility principle (SRP) (Newmann, 2015). Every service is concentrated solely on one practicality. Microservices are naturally loosely coupled (Richardson, 2018). Loose coupling offers developers an opportunity to form freelance changes to services while not moving the remainder of the codebase. As a result of microservices don't seem to be tied to every alternative, they'll be scaled and deployed severally.

Scaling of monolithic application is usually more robust than scaling microservices as a result of one needs to scale the entire application and deploy the whole codebase rather than scaling a part of the applying that demands a lot of resources (Villamizar, 2015). The present development of cloud services makes automatic scaling of resources straightforward and cost-effective. Microservices build most out of this automatic scaling.

As applications grow in size throughout the numerous years of development, it becomes tougher to keep up and build changes to them (Thones, 2014). It's doable to keep up and develop the monolithic software system; however, eventually, it becomes evident that changes to the design of the entire application need to be created. This type of trend was first noticed in corporations that have plenty of track, several developers and colossal codebase. Corporations like Amazon, Netflix, LinkedIn (Ihde & Parikh, 2015), (SoundCloud and lots of a lot of have created the transition to microservice design as a result of their existing monolithic application was too challenging to keep up, develop and scale.

Monolithic applications create it straightforward to develop, deploy, check and scale application once the scale of the codebase is comparatively tiny (Richardson, 2018). Monolithic approach is enough within the starting, and it's doable that the size of the codebase and want for fine-grained scaling is rarely required, which implies that it's higher to remain with the stone and avoid the technical and structural challenges that microservice design comes with. There are differing opinions. It is often argued that the refactoring of the prevailing monolith is just too hard to please task and instead the organization ought to pay longer at the beginning of the method to gauge the design and functionalities that are needed. It's a lot of easier to introduce accidental tight coupling in an exceedingly monolithic than in microservices. Breaking apart these tight couplings are often arduous and need lots of your time and understanding of the applying.

If the monolithic application for a few reasons, stops running in production, it's in no time to acknowledge that as a result of nothing is functioning. With microservices, if one service stops responding to alternative services still work and these quite error things must be handled properly. Communication between microservices is one in each of the massive inquiries to get right. Obtaining the communication wrong will cause a scenario wherever microservices lose their autonomy and therefore the most benefits of the full approach can diminish (Newmann, 2015). On prime of that, the communication between multiple microservices will introduce performance problems if the services are too fine-grained (Richards, July 2016). It looks to be an associate United accord that the quality ought to be inside services rather than electronic messaging pipes (Lewis & Fowler, 2014). One challenge is added to handle the orchestration of the microservices in production. Fortuitously within a previous couple of years, several new tools for supporting this are created like Kubernetes (Kubernetes, 2019) and Mesos (Mesos Documentation, n.d.).

3.12 Summary

In monolithic design, there's one code for all the elements of application. The complete application is predicated on one style, and therefore the entire application must be deployed, not the individual elements. However, there are some problems with monolithic applications. To resolve these problems, one will use microservices. In this, tiny services that are autonomous which work along with are often scaled and free severally with the potential of varied developers’ exploitation various languages across the world. The microservice design may be a technique that permits developing a personal application as a bunch of many tiny services. These services are implemented on their own method. Microservices offers higher quantifiability as compared to monolithic applications.

Because of high quantifiability, the microservices design supports for a large type of devices and platforms. In microservice design, these elements are severally scaled. This design approach is incredibly helpful once the system has a very high load with a variety of reusable modules. The microservice architectures are straightforward. Its aim is confined to a single element at that time. The systems designed exploitation microservice design is loosely coupled because of the elements of a system work freelance of every other. The communication among the services in microservice design uses Inter Process Communication (IPC). Asynchronous communication approach supported HTTP like REST is employed for IPC.

IV Docker

Docker container is similar to the shipping container. Shipping container allows everything (stocks) to be sent by ship, truck, and train. Docker can run, build and store the application with all its dependencies in a container. Its main benefit is that the application can run fast and reliably from one computing environment to another. Containerizing software allows developers to deploy them in zero - modified environments.
Docker is not a programming language and not a framework for creating software. It is a tool that solves common problems like installing, removing, upgrading, distributing, trusting, and managing software. It is open source Linux software also works on all operating systems. It means that everyone can contribute to it, and it has benefited from a variety of perspectives.
Docker is the most popular container technology. Docker runs each component in a separate container within its library and dependencies on all the same virtual machine and operating systems but within different environment or container.
Docker is a tool to easily encapsulate the process of creating a distributable artifact for any application. With it goes to the secure deployment of an application at scale into any environment and streamlining the work process and responsiveness of agile software development (Matthias & Kane, 2015).

When talking about Docker containers, many people connect them to virtual machines; however, this is neither completely right or completely wrong. Discerning these two concepts is challenging. While it is true that both are a specific type of virtualization, the difference between them is that containers are built to contain only the necessary libraries and dependencies inside them. Their bulkier counterpart, virtual machines, start with a full operating system and all included software that come with them (Docker, 2016).

One of Docker's most powerful things is the flexibility it manages IT associations. Choosing where to run your application can be put together 100% respectfully on the right side of your business. You can choose and mix and match your organization in whatever way it makes sense. You get a great mix of agility, portability and control with Docker containers (Coleman, 2016).

Compared to virtual machines, containers are reproducible, standardized environments that follow a particular ideology, create once – use many. After a container environment with the Dockerfile is manually crafted, it is available to be utilized when needed. The containers take only seconds to deploy, while virtual machines take significantly more time.
For instance, the Finnish railway company VR Group uses Docker to automate the deployment and testing process. Their problems were high operating costs, quality issues, and a slow time-to-market process. After implementing the Docker EE (Enterprise Edition), their average cost savings were increased by 50%. Logging and monitoring were easier for all user applications. Standardizing the apps on one platform enables the usage everywhere. A delivery pipeline was set up, which works for the entire platform. This allows for easy implementation of new items to the same environment (Fong, 2017).

4.1 Real-World Use Case for Docker

With Docker Enterprise, Paypal are expostulating 15 years worth of toolsets and building a reliable working model over multiple clouds (Bhattacharya, n.d.).

Application development: It is possible to share to the similar container from Developer to Quality Assurance and later to IT, putting portability to the development pipeline (Aquasec, 2015).

Running applications for microservices: Docker gives you the opportunity to run every microservice in your own container that makes up an application. It allows for a centralized architecture in this way (Aquasec, 2015).

4.2 Virtualization vs Containerization

4.2.1 Virtual Machines (VM)

By virtualizing the hardware, Infrastructure as a Service let you split the resources with other developers using the virtual machine. Each developer could build their application in their environment. They can use their operating system, access the hardware, Read Only Memory, networking interfaces and so on. This virtual machine consumes higher disk space as each virtual machine is massive. It is in gigabytes in size. The virtual machine takes minutes to boot up the entire operating system.

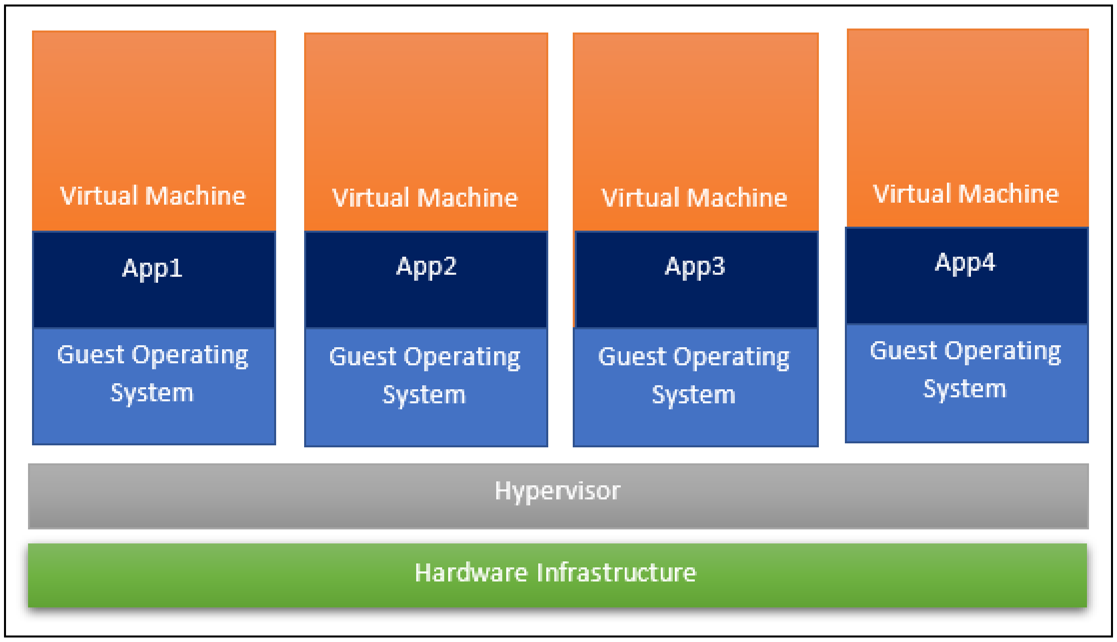

Figure 4. 1 Virtual Machine (Wagner, Torre, & Rousos, 2018)

Developers can choose the tool and install the web server, database or middleware and build as they like. When an application needed to be scaled up, developers have to copy a complete virtual machine and run the guest operating system of each copy or each instance of that specific application. This will eventually make the system slow and to increase the operating speed, and storage space will turn out to be expensive.

- The tool is associated with a virtual machine are more comfortable to access and more straightforward to work with. Docker has complicated tooling ecosystem, which includes tools managed by both Docker and third parties (Aquasec, 2015).

- When virtual machine fully working, one can start a Docker example in the same virtual machine, and run containers within the virtual machine. As a result, Containers and virtual machines are somehow related and can work side by side (Aquasec, 2015).

4.2.2 Containers

What container does is that developers can scale the application up and down in Platform as a Service and it provides an abstraction layer of the operating system and hardware in Infrastructure as a Service. Here abstraction layer implies working with complicated things in the simplified format (Nickoloff, 2016, p. 11). It is an invisible box where it contains the code, its libraries, and its dependencies.

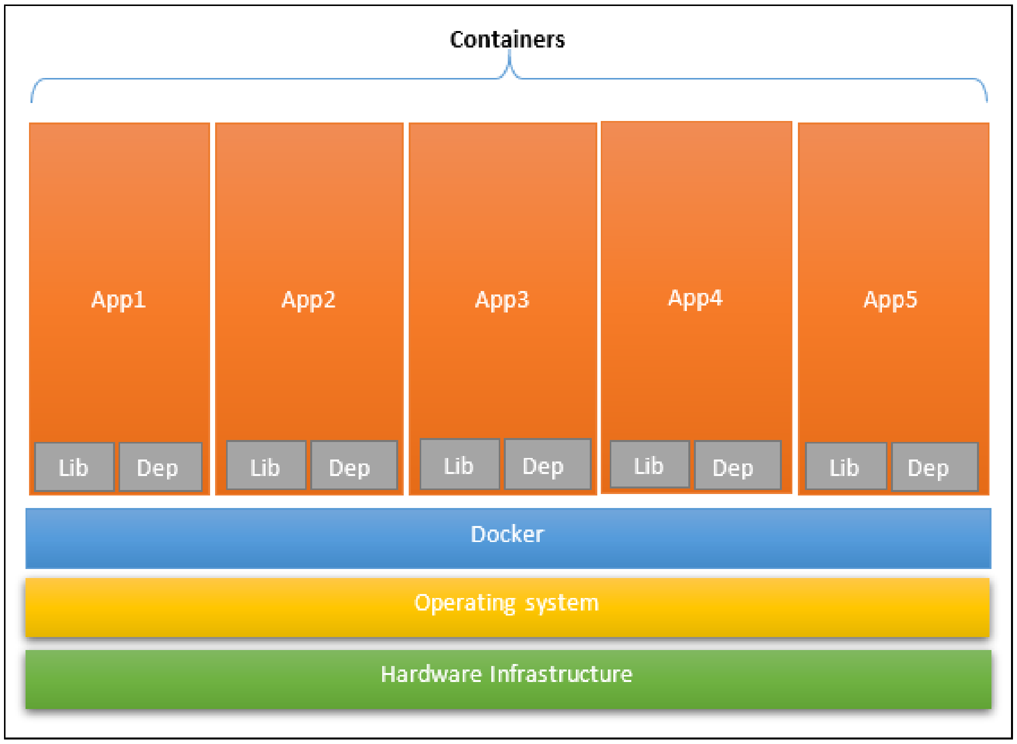

Figure 4. 2 Docker Container (Wagner, Torre, & Rousos, 2018).

The container sits on the top of the physical server, and each container shares the host operating system kernel and binaries and libraries too. Containers contain the properties of the host operating system along with all the dependencies thereby reducing the need for another system to create the necessary environment to run a specific application. Containers are lightweight and megabyte in size. Containers bootup faster in seconds. Here the code is ultra-portable, and without modifying or rebuilding it, it goes from development or from the laptop or the system to stage, to production to the cloud.

- Process-isolated docker containers do not require a hypervisor hardware. Docker containers are smaller in size than a virtual machine and require far less resources (Aquasec, 2015).

- The docker is fast. While a virtual machine can take at least a few minutes to boot and be ready for development, starting a Docker container from a container image takes between a few milliseconds and a few seconds. (Aquasec, 2015).

VM	**Container**
It is Gigabyte in size	It is Megabyte in size
VM is heavyweight	Container is lightweight
Bootup time is minutes	Bootup time is seconds
Virtualization is done on hardware level	Virtualization is done on the operating system
Memory allotted depending on requirement	Less memory is required
Virtual machine provides required operating system	Containers uses the host operating system

Table 4. 1 Difference in VM vs Containers (Bauer, 2018)

4.3 Why we need docker?

This is the most common problem that many industries were facing. There is a developer who has built an application that works fine in his environment, but when it reaches production, there were specific issues with that application. Why does that happen? That happened because of difference is the computing environment between developer and production.

4.3.1 Workflow before Docker

Before docker, the cycle of sending an application to the production looks like as follows:
A traditional deployment workflow

1. Development team request resources from the production team.
2. Production team provides resources and give it to developers.
3. Developers code the application and prepare the tool for deployment.
4. Developers and the production team changed the code again and again.
5. Developers found the more application dependencies.
6. Production team works and install the same dependencies as developers.
7. Repeat steps 5 and 6 more times
8. After repetitions, the application is deployed.

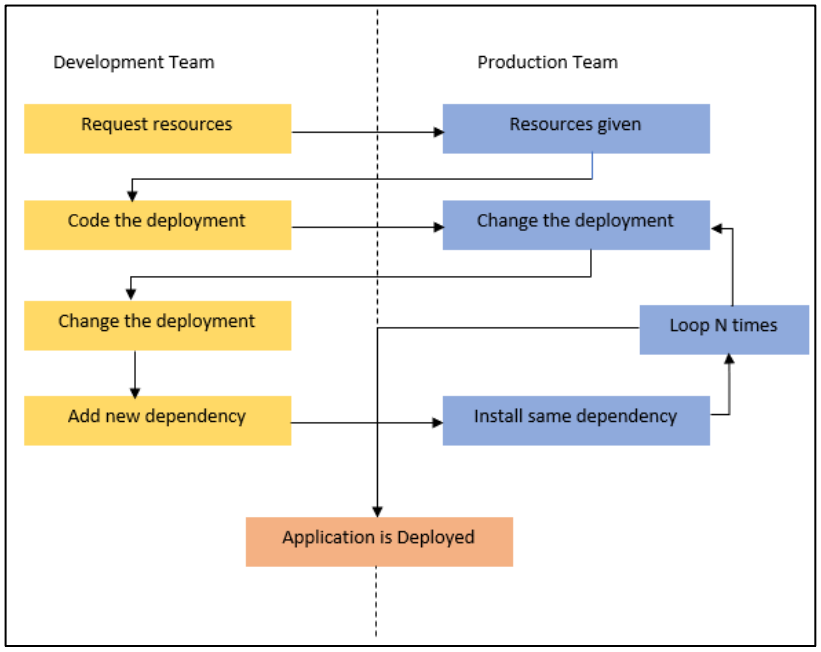

Figure 4. 3 Traditional deployment workflow (without Docker) (Matthias & Kane, 2015).

4.3.2 Workflow after Docker

A Docker deployment workflow

1. The developer codes the application and builds the docker image.
2. Send it to the docker hub registry.
3. Production team provide configuration information to the container and provide resources.
4. Developers deploy the application.

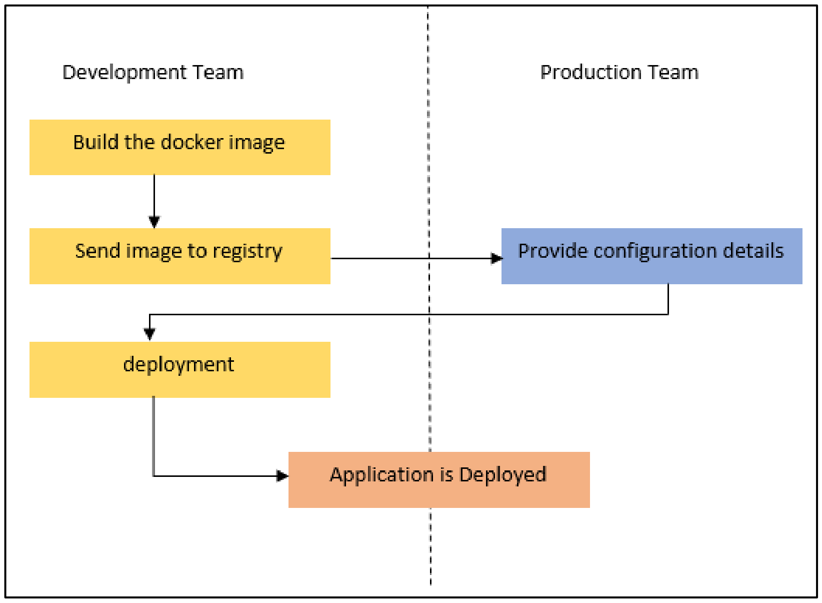

Figure 4. 4 Docker deployment workflow (Matthias & Kane, 2015).

4.4 The Docker Engine

Docker Engine lets you to use the specific components to develop, assemble, ship and run applications (Aquasec, DockerArchitecture, 2015):

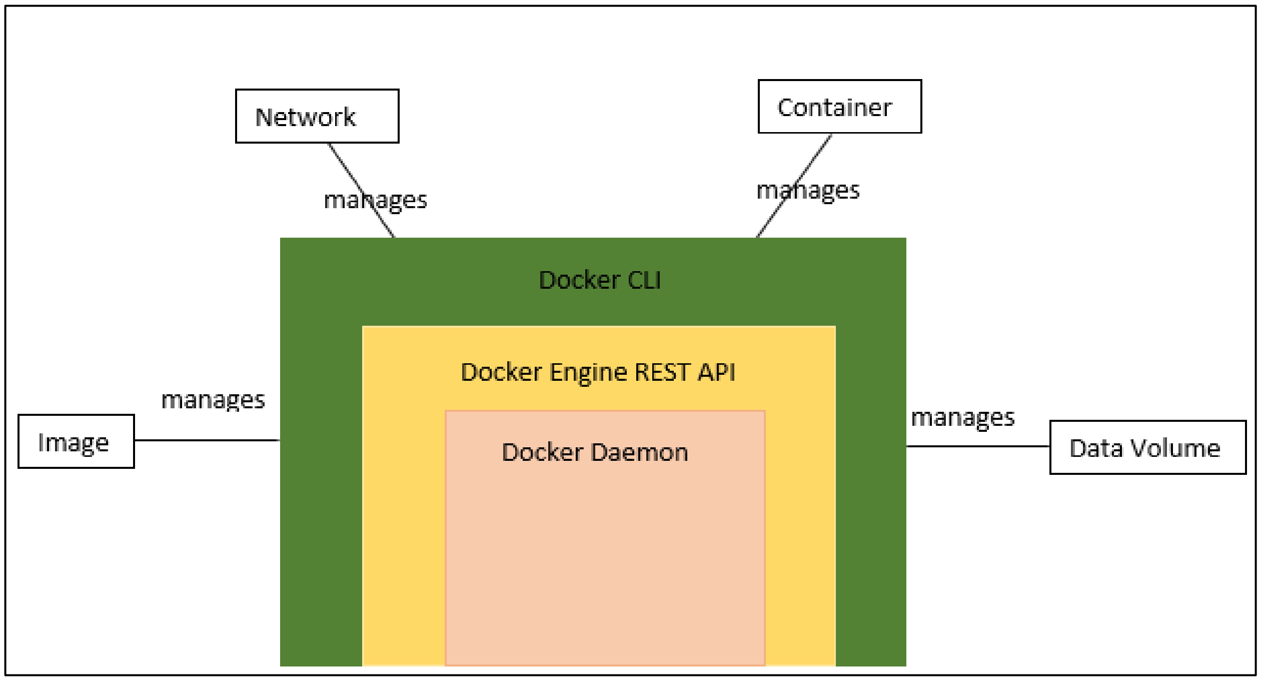

Figure 4. 5 Docker engine (Nickoloff, 2016).

1. **Docker Daemon**: Docker daemon look over Docker images, containers, networks, and storage volumes. This tool helps I handling API request from Docker and responding to these request (Aquasec, DockerArchitecture, 2015).
2. **Docker Engine REST API**: An API is used to interact applications with the Docker daemon; an HTTP client can access it (Aquasec, DockerArchitecture, 2015).
3. **Docker CLI**: A command line interface client is for interact to Docker daemon. It clarifies how developer maintain container instances and is one of the main reasons why developers like Docker (Aquasec, DockerArchitecture, 2015).

4.5 Docker Architecture

This architecture relies on both the client model and the server model. Docker architecture includes the following parameters namely (Aquasec, DockerArchitecture, 2015),

- Docker Client
- Docker Host
- Network and Storage components
- Docker Registry

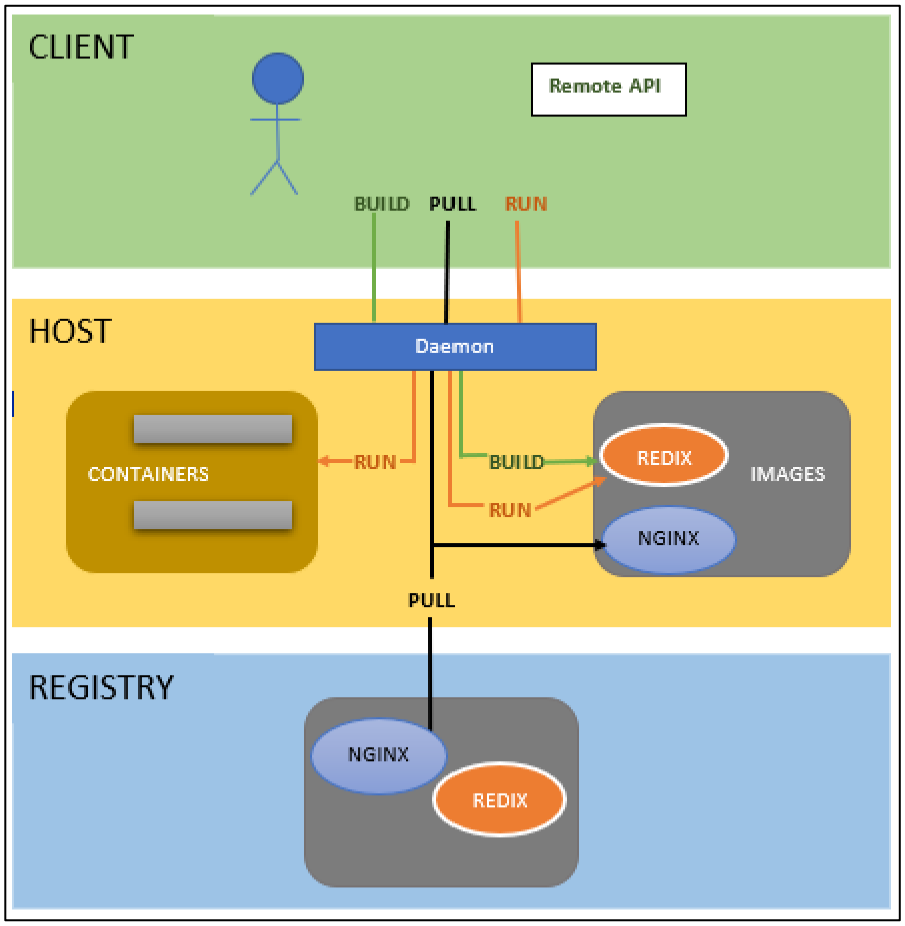

Figure 4. 6 Docker Architecture (Nickoloff, 2016).

4.5.1 Docker client

This service offers interaction with the Docker. The docker client can establish a connection with the same host or a remote host using Docker daemon. This service is used to build, run and stop interaction between the Docker daemon and the application.
The goal of the docker client can allow the pull of images from a docker hub and to execute/run on any docker host.
Following are the client Commands:

```
Docker build
Docker pull
Docker run
```

4.5.2 Dockerhost

The docker host gives an entire environment to perform and run applications. The Docker host includes the following parameters namely docker images, docker containers, docker daemon, networks, and storage. The docker daemon receives commands via the CLI or the rest API. The daemon can likewise speak with other daemons to deal with its administrations. The docker daemon tool extracts and produces docker container images as per the customers demand. Once the demanded image is extracted, the next step is to make use of all the pre-requisite set by the build file for creating a working model. To build the container the build file includes criteria for the daemon to run the container by firstly loading.

Docker objects

For grouping of an application different docker objects are utilized. The main docker objects are as follows:

Images

Images are a read-only binary template used to build containers additionally contain metadata that depict the containers abilities and necessities. Storing and sending the application is done by Images. An image can be utilized without anyone else to construct a container or altered to add extra components to broaden the present setup. Container images can be shared worldwide using a private container docker hub/registry. Images are a centerpiece of the docker experience as they empower cooperation between engineers such that was impractical previously.

Containers

Containers are epitomized conditions in which you run applications. The image characterizes the container and any new setup alternatives gave on beginning the container, including and not restricted to the system associations and capacity choices. Container approach resource that is characterized in the image, except if other access is described when incorporating the image with a container. You can likewise make another image dependent on the present condition of a container. Since holders are a lot smaller than a virtual machine, they can be spun up in only seconds, and result in much better server thickness.

Networking

Docker executes organizing in an application-driven way and gives different alternatives while keeping up enough abstraction for application designers. There are essentially two sorts of systems accessible - the default docker network and client characterized networks. As a matter of course, you get three unique systems on the establishment of docker - none, bridge, and host. The none and host systems are a piece of the network stack in docker. The bridge network arrange naturally makes a gateway and IP subnet, and all containers that have a place with this system can converse with one another using IP tending to. This system isn't usually utilized as it doesn't scale well and has requirements regarding system ease of use and administration revelation.

The other kind of systems is client characterized systems. Administrators can design numerous clients characterized systems.

There are three types:

- **Bridge network**: similar to the default bridge network, a client characterized bridge network contrasts in that there is no requirement for port sending for containers inside the network to speak with one another. The other contrast is that it has full help for programmed network discovery.
- **Overlay network**: an overlay network is utilized when you need containers on isolated hosts to almost certainly speak with one another, as on account of a distributed network.

- **Macvlan network**: when utilizing bridge and overlay arranges a bridge dwells between the container and the host. A macvlan network expels this bridge, giving the advantage of presenting container assets to the outer network without managing port sending. This is acknowledged by utilizing mac addresses rather than IP addresses.

Storage

You can store information inside the writable layer of a container; however, it requires a capacity driver. Being non-diligent, it perishes at whatever point the container isn't running. Additionally, it is difficult to exchange this information. As far as persistent storage, docker offers four choices:

- **Data volumes**: data volumes give the capacity to make relentless capacity, with the capability to rename volumes, list volumes, and furthermore list the holder that is related to the volume. Information volumes sit on the host record framework, outside the containers duplicate on compose component and are genuinely proficient.
- **Data volume container**: a data volume container is an elective methodology wherein a devoted container has a volume and to mount that volume to different containers. For this situation, the volume container is free of the application container and hence can be shared crosswise over more than one container.
- **Directory mounts**: another alternative is to mount a host's nearby registry into a container. The volumes would need to be inside the docker volumes envelope, though with regards to directory mounts any registry on the host machine can be utilized as a hotspot for the volume.
- **Storage plugins**: storage plugins give the capacity to associate with outer capacity stages. These plugin maps are stockpiling from the host to an external source like a capacity cluster or a machine. A rundown of capacity modules can be found on docker's plugin page.

4.5.3 Docker registries

Docker registries are administrations that give areas from where you can store and download images. A docker registry contains repositories that have at least one docker images. Public registries incorporate docker hub, and docker cloud and private registries can likewise be utilized. Normal directions when working with registries include:

```
docker push

docker pull

docker run
```

4.6 Dockerfile

To build an image from an application a Dockerfile is needed. The purpose of the Dockerfile is to automate the image building process, in which all the necessary dependencies and libraries are installed.

Although premade images are available for use from the Docker Hub (https://hub.docker.com/), it is sometimes better to make a specific Dockerfile. This way, you know how the final image is built, what licenses and/or properties it contains. The downside of this is that the responsibility to keep the Dockerfile updated falls on the organization. It is also possible to combine prepared images with self-made Dockerfiles to maximize efficiency.
The Dockerfile supports 13 different commands, as seen in Table 4.2, which tell the Dockerfile to build the image and to run it inside a container. There are two phases in the building process: Build and Run. In the BUILD -phase you determine the commands which are executed during the build process. In the RUN- phase the commands specified are run when the container is run from the image. The two commands WORKDIR and USER can be used by both phases (Janmyr, 2015).

BUILD	**Both**	**RUN**
FROM	WORKDIR	CMD
MAINTAINER	USER	ENV
COPY	LABEL	EXPOSE
ADD		VOLUME
RUN		ENTRYPOINT

Table 4. 2 Important Docker Commands

4.6.1 BUILD phase

FROM command is used to get the image for the build. The Dockerfile can begin FROM command, it does not need operating system to start, it starts the container. An operating system is extracted from the Docker registry to act as a basis for the last image (Dockerfile, 2017).

MAINTAINER command is used to get the name and e-mail of the author. The purpose of this command is to clarify the end-user to contact in case of problem created (Dockerfile, 2017).

COPY command is utilized to copy items from the sources system to the target destination inside the container (Janetakis, 2017).

ADD command is the same as COPY, but with ADD, the extraction of a single file is possible from the source to the destination. Also, an URL- address can be used instead of a local file (Janetakis, 2017).

The RUN command is used to executes a new parameter on the current image. It enters inside the Dockerfile. This is mainly used for installation of dependencies or updates (Dockerfile, 2017).

ONBUILD command adds a trigger instruction into the image, which is executed later. This is useful when building an image which is used later as a base for other images (Dockerfile, 2017).

4.6.2 RUN phase

The CMD command is executed when the container is construct from the final image. There can only be one CMD instruction (Matthias & Kane, 2015, p. 44).

The ENV command is to define the environment of the application running inside the container. (Matthias & Kane, 2015, p. 43).

The EXPOSE command informs docker which ports the container listens during runtime (Dockerfile, 2017).

The VOLUME command makes a mount point with the predetermined name and marks it as holding remotely mounted VOLUMES from localhost or different CONTAINERS (Dockerfile, 2017).

The ENTRYPOINT command is used to create the container which can runs as an executable process (Dockerfile, 2017) (DeHmaer, 2015).

4.7 Docker-compose

Docker Compose is a designing tool for executing multi-container Docker applications (Podviaznikov, 2017). Docker Compose and Dockerfile have unique instructions for building images and running containers (Podviaznikov, 2017). When the docker image is build, the `docker-compose.yaml` file can be accessed. The purpose of the file is to build and link the containers together. This is used, for example, when making a multi-container application such as linking an application to one or multiple databases.

4.8 Docker terminology

Container image: The image of the container is a package of all the information and dependencies. An image includes all the interface deployment and execution dependencies that a container runtime will use. A multi - base image that is pieces that are stacked on top of each other to form the container's file system. Once it's made, an image is permanent (Wagner, Torre, & Rousos, 2018).
Dockerfile: Docker file is a text file that contains directions on how to create a Docker image. The first line expresses the base image and obeys the rules for installing required projects and copying all files (Wagner, Torre, & Rousos, 2018).
Build: Build a container image activity depending on the rules and trying to set provided by its Docker file, as well as additional files in the directory where the image is constructed (Wagner, Torre, & Rousos, 2018).
Container: Container is a Docker image example. A container creates a single application, service or process being implemented. It includes a Docker image, an environment for implementation, and a new standard set of guidelines. Many cases of a container can be crated from the same image when a provider is scaled (Wagner, Torre, & Rousos, 2018).

Volumes: Volume indicates a filesystem which can be used by the container. Images are read-only and most programs have to write to the filesystem, volumes include a writable layer over the image of the container so that the programs view a filesystem that can be published. The program does not really know that a layered filesystem is accessed; it is only the filesystem. Volumes are regulated by Docker and reside in the host structure (Wagner, Torre, & Rousos, 2018).
Tag: A label which can be used to understand various images or versions of that same image (Wagner, Torre, & Rousos, 2018).
Repository (repo): a group of images connected to the Docker, indicated with a tag displaying the image version. Some repository includes many exclusions to a particular image, such as an image that includes SDKs (heavier), an image that produces only runtime (lighter), etc. It is feasible to mark these exclusions with tags (Wagner, Torre, & Rousos, 2018).
Registry: Registry is a repository access service. Docker Hub is the default public image registry. Organization has its private databases to store and establish the images it has produced (Wagner, Torre, & Rousos, 2018).
Docker Hub: Uploading images is a public database and developers are capable of working with images. Docker Hub gives hosting of images for Docker, public registries, private registries, and GitHub and Bitbucket full integration (Wagner, Torre, & Rousos, 2018).
Docker Community Edition (CE): Windows Docker Community Edition offers both Windows Container and Linux Container development environments. Windows development tools to build, run and test containers locally. The Windows Linux Docker host is largely focused on a Hyper-Visor VM. Windows-centered host for Windows Containers (Wagner, Torre, & Rousos, 2018).
Compose: One can encompass a single application that can override values depending on the model based on several images with.yml files. With a single docker-compose command, one should deploy the entire multi-new container application, which generates a container per image on the Docker host. .yaml is the extension of compose file (Wagner, Torre, & Rousos, 2018).

4.9 The Container in Cloud Computing

The Container in cloud computing is an approach to OS virtualization. Containerizing packages all the resources for the user. To build blocks, a container in cloud computing is used, which help in producing operational efficiency, application version control, productivity developer and environmental consistency. This container guarantees the user that no matter which platform all the resources are made available. The container usage in online services benefits storage with cloud computing information security, availability, and elasticity.

4.9.1 Benefit of Containers in Cloud Computing

These benefits of Containers in Cloud Computing provided by CASB solution (CASB, 2018).

The Unity in Cloud Storage: The container increases portability. It removes the technical frictions, and the program moves through the entire process cycle. It is a group of key files of an application and software server and dependencies. This can be separated from any resource. The manual configuration of every server is along these lines entirely abstained from enabling the clients to declare another component.

Application Version Control: The users can take a look at the application code of the current version and their dependencies, through the container in cloud computing. The Docker containers operate a plain file. The users can quickly check and follow the version of the container.

Operational Activity Efficiency: Users can use the cloud computing container to obtain more assets. It is necessary to identify the necessary memory, disk space and CPU consumed by the container. The container is an operating system strategy that works on an app and code, the containers have a super-fast boot time. Users can enter and leave the app readily and mark it up and down. The applications are separated by the confinement operation from each other. There are no shared dependencies to this idea.

The productivity of the Developers: The containers delete the inter-service dependencies and friction, thus increasing efficiency. The program element is subdivided into different parts running a different micro-service. There is no pressure over the libraries and dependencies synchronized for each service due to the independence of the containers. It is feasible to upgrade each service individually.

4.10 Summary

Containers in the docker are much better than virtual machines because they are lighter, faster and portable. You can start a Docker instance and run containers on the virtual machine inside the virtual machine, but it slows down the system. Eventually, to get the system to work quickly, upgrading the system configuration would need to increase the cost. The containers for the docks are very fast. It is possible to ship containers over multiple team members. Containers reduce errors plaguing teams from DevOps. Containers package all application code, libraries and dependencies to be shipped anywhere.

V Kubernetes

Kubernetes is useful for automating, scaling and deploying or executing of containerized applications (Kubernetes, 2018). Kubernetes is the leading container orchestration engine in the field of containerization. Developed by Google, it uses the Docker images as a basis to deploy applications into the containers. With Kubernetes, the containers are easily scaled up, destroyed and remade. Compared to normal virtual machines, they are deployed faster, more efficiently and reliably. Docker creates the image, which is used in Kubernetes. In the world of growing virtualization and the Internet of Things, applications and services need to be deployed quickly and efficiently.

Instead of operating at the hardware level and rather in the container level, Kubernetes provides some features related to a Platform as a Service (PaaS). PaaS usually refers to a cloud service. In the cloud service, the user can develop, deploy and run applications using several environments provided by the service provider. Essentially, the provider takes all the responsibility in installing and configuring the environment. This way the customers are free to apply the application code to the cloud. This is what Kubernetes basically does, except that the cluster is managed by the developer or cluster administrator and it can be done locally. These features include for example, deployment, scaling, load balancing, logging and monitoring. Kubernetes comprises of a set of independent control processes that drives the current state of the deployment to the wanted result (Kubernetes, 2018) (Meegan, 2016).

The reason Kubernetes was chosen instead of the native Docker cluster, Docker Swarm, is its scalability, portability and self-healing attributes. Kubernetes has been around longer than Docker Swarm and therefore has much more documentation. It also has a more widespread 3rd party application support available (Kubernetes, 2018).

The popular game Pokémon GO, uses Kubernetes containers. They were especially critical during the launch, where millions of users connected almost simultaneously. The actual traffic that had to be handled was 50 times larger compared to the expected traffic. This is where the scalability of Kubernetes kicks in. When the traffic increases rapidly, Kubernetes sees this and automatically deploys more containers to adapt. Deployed on the Google Cloud, it was a great success and tens of thousands of cores were provided dynamically to enable an enjoyable user experience to the end user (Stone, 2016).

Container orchestration definition

As we have studied microservices are loosely coupled, this container orchestration helps to organize these services so that they work hand in hand. Container orchestration mentions the process of organizing the work of individual parts and application layers (Hewlett, n.d.).

5.1 Understanding the Core of What Kubernetes Does

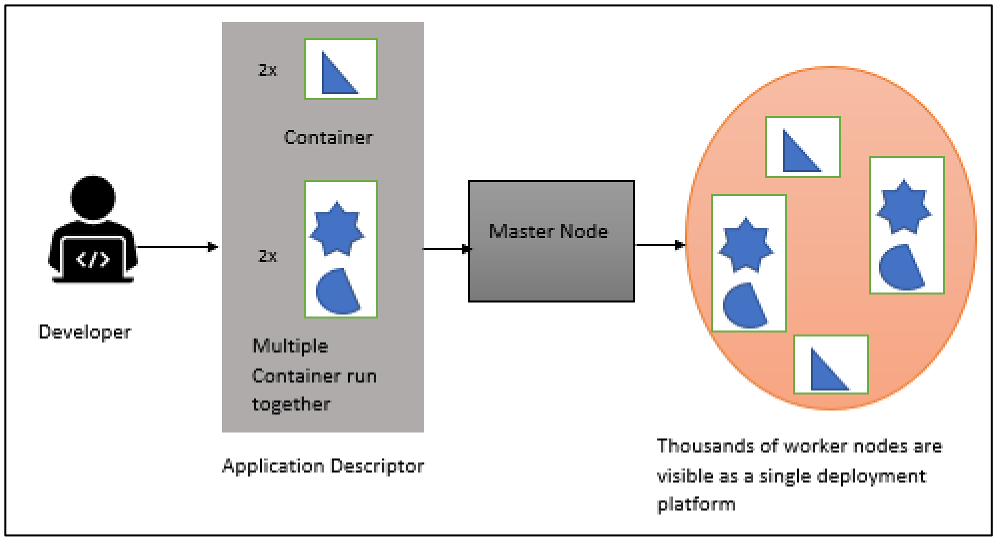
2x
Container
2x
Multiple Container run together
Application Descriptor
Developer
Master Node
Thousands of worker nodes are visible as a single deployment platform

Figure 5. 1 Core of what Kubernetes does (Lukša, 2017).

The system includes one master node and multiple worker nodes. Once an application is handed over to the master node it further passed on worker node by Kubernetes.
The developer can specify that certain application for example, as shown in the fig. must run together, and Kubernetes will deploy the application on the same worker node. Communication between nodes happens regardless of their deployment locations.

Supporting developers focus on the main app features
Kubernetes can be thought of as an OS for the cluster. It reduces application developers from having to implement certain infrastructure-related services into their applications; rather, they depend on Kubernetes to provide these services. This includes things such as service discovery, load-balancing, scaling, and self-healing. Application developers can, therefore, focus on implementing the attributes of the apps and not waste time making sense of how to combine them with the infrastructure (Lukša, 2017).

Supporting production teams achieve better resource utilization
Containerized application can be executed in the cluster. Kubernetes has a feature where each component can find each other and run as well. Because implementation doesn't care which node it runs on, Kubernetes can migrate the application at any time and, by mixing and matching apps, make far better use of resources than manual scheduling is possible (Lukša, 2017).

5.2 Understanding the Parameters architecture of a Kubernetes cluster

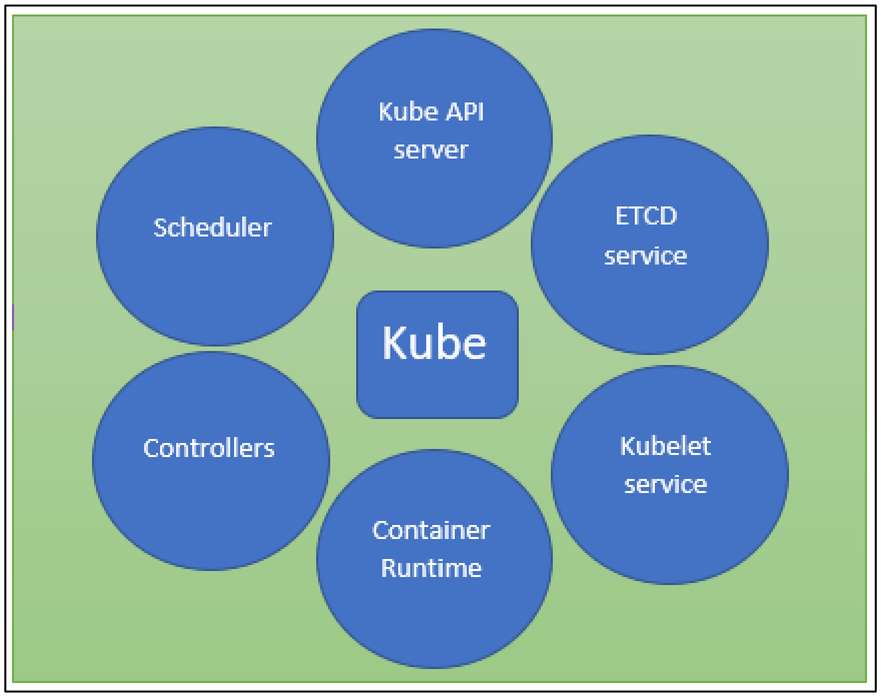

Figure 5. 2 Kubernetes Parameters (Author).

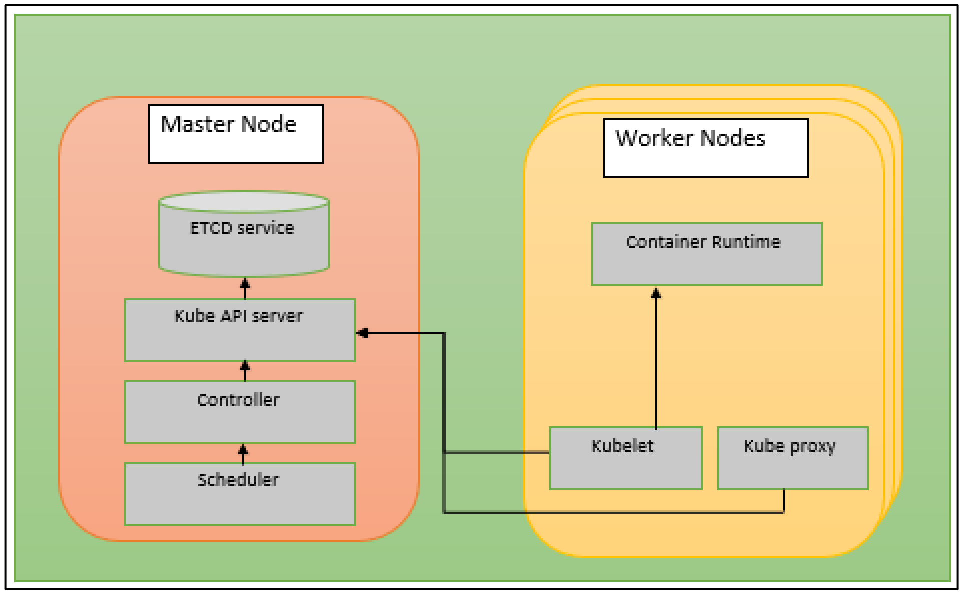

Figure 5. 3 Architecture of Kubernetes cluster (Lukša, 2017).

The **master node** watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes.

A **node** is a machine – physical or virtual – on which kubernetes is installed. A node is a worker machine, and this is where kubernetes will launch containers.
It was also known as Minions in the past. So, you might hear these terms used interchangeably.
But what if the node on which our application is running fails? Our application goes down. So, you need to have more than one nodes.

5.2.1 Components

Following are the main components found on the master node:

ETCD cluster: ETCD cluster is a straightforward, decentralized key-value storage that stores Kubernetes cluster information (e.g. number of pods, status, namespace, etc.), API objects, and service discovery specific details. For security purposes, the ETCD cluster can be accessed only from the API server. With the help of viewers, ETCD informs the cluster of interface changes. Alerts are API queries for any cluster node etc. to activate data updates in the storage of the node (Aquasec, Kubernetes Architecture, 2015).

Kube-apiserver: Kubernetes (Kube) API server is the main management entity that recognizes all REST queries for modifications (to pods, services, replication sets / controllers and others) that represent as a cluster frontend. Also, the only other part communicating with the ETCD cluster is the API server. It assumes that information is stored in ETCD and is consistent with the deployment service particulars (Aquasec, Kubernetes Architecture, 2015).

Kube-controller-manager: Kube controller manager executes in the background a number of different controller mechanisms to control the mutual cluster and execute mundane tasks. The replication controller, e.g., manages replicas in a pod, endpoints controller populates endpoint objects such as providers and pods, and so on. When a service setup changes, the controller marks (Aquasec, Kubernetes Architecture, 2015).

kube-scheduler: Kube scheduler helps to schedule the pods on the different nodes depends on resource utilization. It studies the service's operational requirements and schedules it on the node. For example, if the application requires 1GB of memory as well as 2 CPU cores. Then the pods for that specific application will be scheduled on a node with at least those resources. The scheduler runs every time there is a need to schedule pods. The scheduler should know the total number of resources available and resources allocated to existing workloads on each node (Aquasec, Kubernetes Architecture, 2015).

5.2.2 Node (worker) parameters

Listed are the main parameters found on a (worker) node:

Kubelet is the main node-based service and frequently uses new or modified pod specifications (mainly via the Kube API server) to help make sure that pods and containers are productive to work, even when they are in preferred condition. This component also informs the master node about the host's health.

Kube-proxy is a proxy service running at every worker node to distribute services to the outside world using individual host subnetting. It sends requests through multiple insular networks in a cluster to the right vats / container.

The **container runtime** is the underlying software that is used to run containers. This way even if one node fails you have your application still accessible from the other nodes. Moreover, having multiple nodes helps in sharing the load as well. Now we have a cluster, but who is responsible for managing the cluster? Where is the information about the members of the cluster stored? How are the nodes monitored? When a node fails how do you move the workload of the failed node to another worker node? That's where the Master comes in. The master is another node with Kubernetes installed in it, and is configured as a Master.

5.3 Studying an application in Kubernetes

To run an application in Kubernetes, you first need to package it up into one or more container images, push those images to an image registry, and then post a description of your app to the Kubernetes API server. The description includes information such as the container image or images that contain your application components, and which ones need to be run co-located (together on the same node) and which don't. For each component, you can also specify how many copies (or replicas) you want to run. Additionally, the description also includes which of those components provide a service to either internal or external clients and should be exposed through a single IP address and made discoverable to the other components (Lukša, 2017).

Realizing how to characterize a container: The Scheduler will schedule the stipulated groups of containers on the working nodes available based on computer resources that will be needed for each group and the unassigned assets for each node at that time. On these nodes the Kubelet commands the container runtime (for instance, the container docker) to take images and run the containers. Test Figure 5.4 to comprehend how applications in Kubernetes are deployed. Four containers are mentioned in the app descriptor, clumped in three. There is only one container in the first two pods, and two containers in the last. This means that both containers must be adjacent-located and not isolated. Next, see the number of reproductions that have to work in tandem to each pod. After the descriptor has been submitted to Kubernetes, the stipulated amount of reproductions of each pod will be scheduled for the available nodes. The kubelet at the nodes then tells Docker to draw container pictures and run containers from the image register (Lukša, 2017).

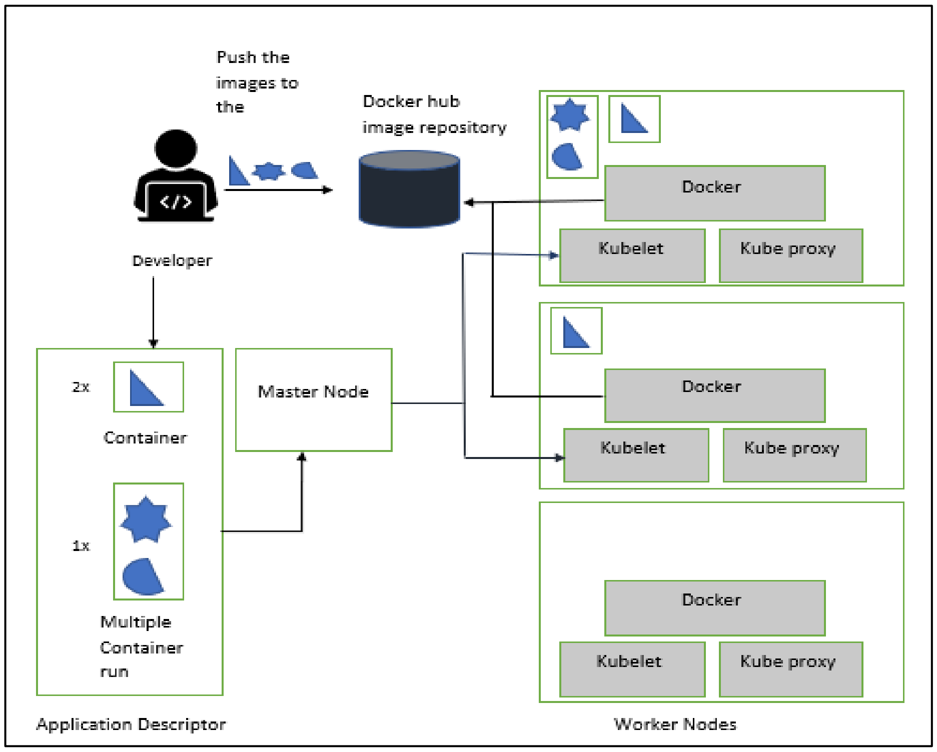

Figure 5. 4 Functioning inside a container (Lukša, 2017).

Executing the packaging: Kubernetes ensures that the application's dispatched status always corresponds with the synopsis you have provided when you run the application. If, for example, you always want 5 instances of a running Web server, Kubernetes will always run five instances. by any means it stops functioning appropriately, such as when its process breaks or stops reacting, Kubernetes will instantaneously restart it all by itself. Likewise, if an entire operator node perishes or become inaccessible, Kubernetes selects new nodes from the newly selected nodes for all containers executing on the node (Lukša, 2017).

Scaling the copy number: You can choose to scale up or scale down the count of replicas while an application is being run, and Kubernetes will rev up extra replicas or stop the exceed. This can recalibrate the amount instantaneously on real-time basis, such as CPU load, memory consumption, queries per second or any other benchmarks exposed by your app (Lukša, 2017).

A Shifting Attack: Kubernetes may require your containers to relocate across the cluster. This can happen if the node at which they are run has failed or are expelled from a node for other containers. This may happen. How can you use the container correctly if it constantly moves around the cluster when it provides a service to external clients or other containers in this cluster? And how can customers connect to containers that provide a service when they replicate them and distribute them all over the cluster? To make it easy for clients to find containers that offer a particular service, Kubernetes can tell which containers offer the same service, and Kubernetes will display them in a single static IP address, which will be used for all the cluster applications. This is done via environment variables, but customers can also view the service IP via good old DNS. The cube proxy ensures that the connectivity to the service is balanced in load across all service containers.

The IP address of the service remains constant, so that customers can always connect to their containers while moving around the cluster (Lukša, 2017).

5.4 Kubernetes Concepts

Using Kubernetes calls for the perception of the distinguishing reflections which it uses to talk about framework conditions such as programs, pods, volumes, namespace and deployments.

Pod: Pod refers to a container or more that can be monitored as a single program. A pod collects containers for applications, storage assets, a special network ID and a different container management setup (see in next topic).

Service: The pods are unstable, meaning that kubernetes do not assure that the physical pod is kept alive. (For instance, the duplication console could decapitate and begin a new pod set). In fact, a service characterizes and acts as a conduit to a sensible set of pods. Then (customer) pods can send queries without having to keep track of what the service is physical.

Volume: The volume of Kubernetes is identical to the volume of docker. The density Kubernetes covers a full jacket and is installed on all the jacket containers. Kubernetes guarantees protection of the data through reboots of containers. The density will be erased once the pot is destroyed. A pod can also be attached to different volumes.

Namespace: For circumstances with multiple customers, a virtual cluster (a solitary physical cluster can run different virtual cluster) was suggested for segregation of problems. Assets within a namespace must be exemplary and in an entirely new namespace they cannot access the resources. Furthermore, a namespace can be allocated an asset amount to prevent more than many resources of the physical cluster.

Deployment: The deployment highlights the absurdity you want to see in a.yaml file of a pod or replica set. The deployment monitor then upgrades the surrounding continuously until the the present status matches the status specified in the deployment file (e.g., creating or deleting replicas). If the, for example. Yaml document characterizes two replicas, but only one is running from now, another one will be made. Note that replicas supervised by means of an entity, only through new implementation, should not be explicitly controlled. The command-line tool is Kubectl. Kubectl is interacting with the API server kube and forwarding master node commands. Every command becomes an API call. The cube control method is used to activate and handle applications on a cluster, to obtain data about the cluster, to obtain the designation of the cluster nodes, and much more. The kubectl execution instruction is used to activate a cluster application. The kubectl cluster-info command is used for viewing cluster information and for listing all the nodes within the cluster, the kubectl receive pod command is used.

5.5 What is Pod?

Accessing web application

We said that every pod has its own IP address, but that address is internally available to the cluster and not outside it. You will show it through a Service Object in order to make the pod reachable from outside. You will develop a special LoadBalancer service, because you can only access the service from the cluster if you create a regular service (the ClusterIP service), such as the pod. The LoadBalancer service creates an external load balancer and allows you to connect to the pot through the public IP of the load balancer (Lukša, 2017).

Knowledge of the pod and its container: the pod is the system's most important component. It only contains one container, but usually a container can contain as many containers as you like. The Node.js process inside the container is bound to 8080 port and waiting for HTTP requests. The pod is provided with its own unique IP and hostname (Lukša, 2017).

Knowing the replicationcontroller's function: ReplicationController is the example next component. It ensures that one instance of your pod is always running exactly. ReplicationControllers are generally utilized to replicate and retain pods (that is, create multiple copies of a pod). You did not specify how many replicas you would like, so the replication Controller created one. The ReplianceController would create a new pod to replace the missing one if your pod were to disappear for any reason (Lukša, 2017).

Knowing why you need a service: Knowing why you need a service: e.g.-http service is the third aspect of the system. You have to learn a crucial detail about pods in order to know why you would want services. They're unfulfilling. A pod may vanish at any time the node on which it runs has failed, because somebody has removed the pod from a healthy node otherwise. When any of these happens, the ReplicationController will remove a missing pod only with a new one, as described earlier. This new pod gets from the pod it replaces a dynamic IP address. That is where services come in — to fix the issue of constantly shifting pod IP addresses and also presenting multiple pods to a single constant IP and port pair. It receives a static IP when a service is produced, which never really changes during the service's lifetime. Rather than connecting directly to pods, customers should use their constant IP address to connect to the service. The service ensures that the connection is received by one of the pods irrespective of where the pod currently runs (and what its IP address is). Services for a group of one or more pods that all provide the same service comprise a static location. Queries that arrive at the service's IP and port will be submitted to the IP and port of one of the service's pods at that time (Lukša, 2017).

5.6 Features of Kubernetes

The following are the features offered by Kubernetes:

- **Automates various manual processes**: Kubernetes will control for user which server will host the container; how will it be launched.
- **Interact with several group of containers**: Kubernetes is able to manage more cluster at same time.
- **Provides additional services**: Management of containers, Kubernetes offers security, networking and storage device.
- **Self-monitoring**: Kubernetes checks constantly the health of nodes and containers.
- **Horizontal scaling**: Kubernetes allows you scaling resources not only vertically but horizontally, easily and quickly.
- **Storage orchestration**: Kubernetes mounts and add storage system of user's choice to run application.
- **Automate rollouts and rollbacks**: If after a change to user's application goes wrong, Kubernetes will rollback for you.
- **Container balancing**: Kubernetes knows where to place containers by calculating the best location for user.
- **Run everywhere**: Kubernetes is an open source tool that allows users to choose to use on-premises, hybrid or public cloud infrastructure to move their workloads wherever they want.

5.7 Summary

Kubernetes provides automation for deploying, scaling of application packages over wide clusters of hosts. Developer can deploy application on Kubernetes. There is no need of system administration, because Kubernetes work automatically with the failed nodes. The chapter focuses on how to deploy and study an application on Kubernetes. It also takes a deep dive into the basic concepts on Kubernetes. At the end, this chapter discusses the features of Kubernetes.

VI Implementation of proposed architecture

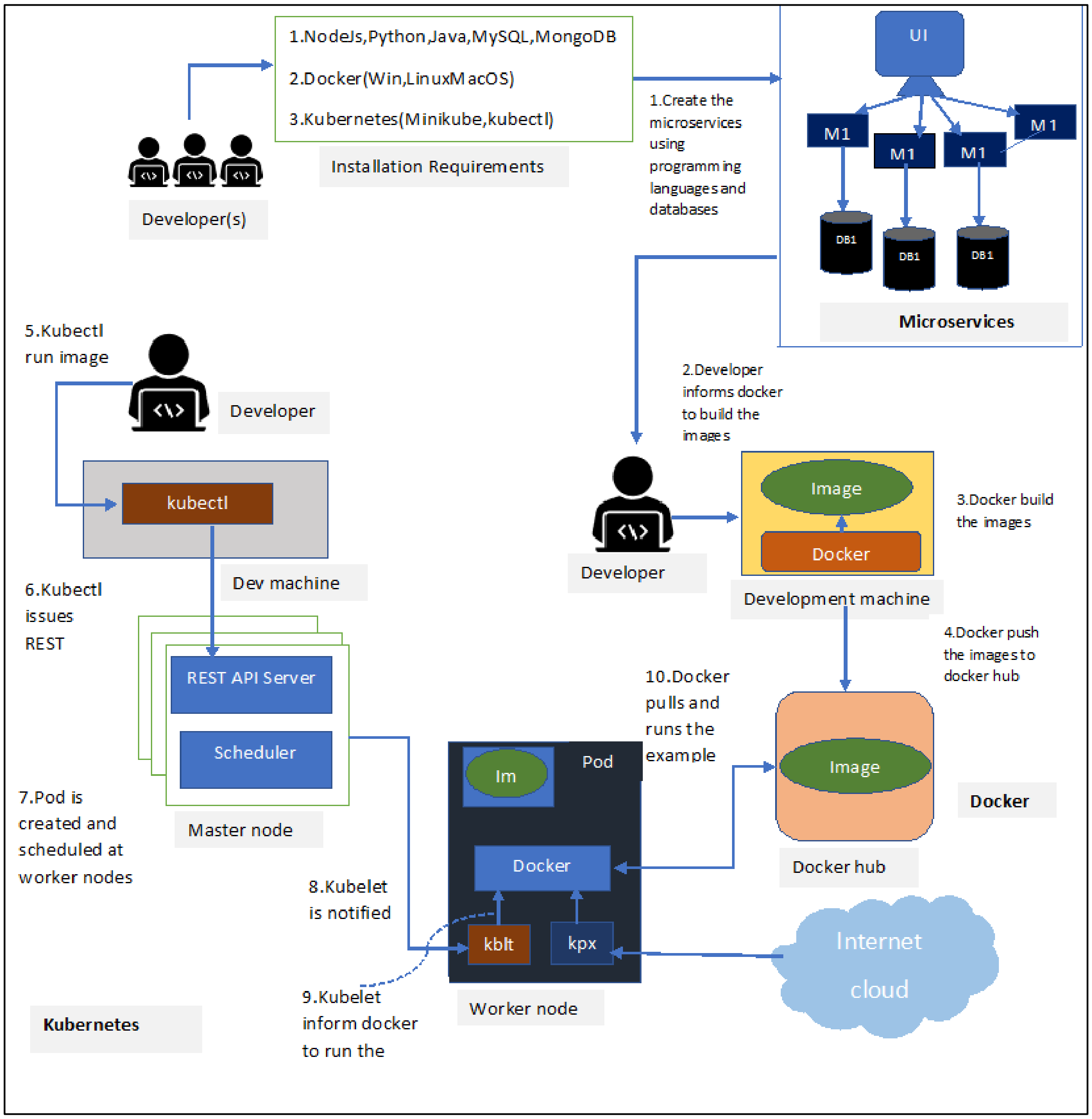
1.NodeJs,Python,Java,MySQL,MongoDB
2.Docker(Win,LinuxMacOS)
3.Kubernetes(Minikube,kubectl)
Installation Requirements
Developer(s)
1.Create the microservices using programming languages and databases
UI
M1
M1
M1
M1
DB1
DB1
DB1
Microservices
5.Kubectl run image
Developer
kubectl
Dev machine
2.Developer informs docker to build the images
Developer
Image
Docker
Development machine
3.Docker build the images
6.Kubectl issues REST
REST API Server
Scheduler
Master node
4.Docker push the images to docker hub
10.Docker pulls and runs the example
Pod
Im
Image
Docker
Docker
Docker hub
7.Pod is created and scheduled at worker nodes
8.Kubelet is notified
kblt
kpx
Internet cloud
9.Kubelet inform docker to run the
Worker node
Kubernetes

Figure 6. 1 Proposed architecture on cloud (PaaS) (Author)

Kblt: kubelet, kpx: kube-proxy, Im: Image

6.1 Working

Developer needs to understand the requirement of this thesis. We need to install Docker and Kubernetes.

1. Microservices: microservices is the technique of breaking down an application into smaller services performing different task for the application. The interaction mechanism is explained in the chapter 2.
2. Docker: Docker containers reduce the efforts faced during deployment thereby making the work easy. Containers enable a developer to bundle application with all of the elements it needs, such as libraries and other dependencies, and send it all out as one package.
3. Kubernetes: Kubernetes provides automation for deploying, scaling of application packages over wide clusters of hosts. Kubernetes revolves around infrastructure which is container-centric. Kubernetes is an open source platform.

Concepts and guidelines of proposed architecture (Lukša, 2017) (Docker, 2018).

1. Developer can freely use any programming language such as C#, NodeJS, Java, and so on and databases such as MongoDB, MySQL, etc.
2. Developer must separate the domains and clear with the functionality while building the application.
3. Every designed microservice should focus on one microservice at a time.
4. All microservice deploy separately.
5. Communication between one microservice to other microservice done by stateless server.
6. Microservice architecture consist the client, API Gateway, Messaging Service, Databases, Management, Service Discovery (see the microservice chapter 2).
7. Once the microservices created developer should create docker file (Docker file does not contain extension) for each service with the required instructions.
   a. Docker file is text document that contain commands such as FROM, COPY, RUN CMD and many more (see the docker chapter 2).
   b. Use command prompt or power shell to write docker commands. Some important commands:
      - `docker build .`
        to create an image through Docker file.
      - `docker exec -it <container id> sh`
        execute command in the container
      - `docker build -t <username>/<image name>.`
        to tag the image.
      - `docker run <image name or container id>`:
        created the tagged image
      - `docker create <image name or container id >`
        to create the container
      - `docker start <image name or container id >`
        to start the container
      - `docker stop <image name or container id >`
        to stop the container

- `docker ps –all`
  to show the list of running container
- `docker run -p <request port no.>:<container port no.> <username>/<image name>`
  docker run with port mapping

c. Docker-compose is use to run multiple docker container as a single service (application). It means developer have an application which required two different images, developer could create one docker-compose file which would start both the containers as a service without the need to start each one individually. It has. yaml extension.
- `docker-compose up -d`
  it runs the entire application
- `docker-compose down`
  to stop the container

8. Push the docker container/image to the docker hub or registry. So that Kubernetes can pull these images. For example, developer push `example` image.
9. After that, Kubernetes run to get the container image.
   - `Kubectl run example`
10. Once the command run, in the cluster it made a new Replication Controller object by passing a REST HTTP request to the Kubernetes API server as shown in the Fig 5.1. Replication Controller watches over pods or group of pods is always available.
11. Here it creates a new pod and assigned to one of the worker nodes by scheduler. The Kubelet on that node watch that the pod was assigned to it and tells Docker to pull the particular image from the Docker hub/registry. After getting the `example` image, docker is created and run the container.
    - `kubectl get pods`
      it shows name of container, running/pending status of pod
12. The proposed architecture is based on microservice architecture, docker architecture and Kubernetes architecture.

6.2 Features of the proposed architecture

The following features can be concluded from the above architecture:

- Once the application developed using this proposed architecture is on cloud storage it is independent of the geographic region.
- Monitoring, logging, and diagnostics is managed by cloud providers (for example Google Cloud's Stack driver tool).
- Eliminates cost of hardware and its implementation.
- Migration from monolithic to microservice, or implementation of microservice architecture reduces the difficulties during debugging.
- Unlike Monolithic, the modifications in a specific service will not affect the entire application.
- Developers responsible for their specific service but also maintain communication with the other departments responsible for that specific application.
- Since the proposed architecture is developed using microservices; therefore, all the characteristics of microservices is achieved.
- Fast deployment is achievable.
- Deployment is not dependent on the version of Docker or Kubernetes platform.
- The proposed architecture makes scaling and automation easy.
- Incorporating fault tolerant mechanism makes microservices more efficient.

Conclusion

A new architecture for developing cloud-based applications is proposed. As large code based monolithic architecture are difficult to scale and debug, so the proposed architecture shows the utilization of microservices. Using microservices reduces the effort of scaling the entire application and focuses on the specific service that needs to be altered. The architecture makes use of Docker for containerizing the application. Containerized services or docker images are portable. Containers are more efficient, light weight and fast compared virtual machines. The architecture further makes use of Kubernetes for deploying the containerized application. The architecture implements docker and kubernetes together which improves security, fault tolerance and monitoring. The architecture provides efficient scaling depending on the requirement of the application.

The new proposed architecture goes from three different steps which demonstrates microservice architecture, docker and kubernetes. Having this architecture incorporated in the real time scenario will give the developers to focus on one specific task of their concern. This broadens the selection criteria of resources (programming languages, databases) for the developer. The implementation of docker along with its basic components is covered in detailed. To achieve automation which is an ever-evolving topic the architecture makes use of kubernetes. The final step of the thesis shows the release of the application created using this architecture over the cloud.

As the architecture depends on kubernetes allows deploying and running of software components without the need of server underneath. A systematic approach in the form of a workflow of this architecture is shown in this thesis.

The developers who will benefits the most from this architecture are developers concerned only with microservices, the ones responsible only for containerization of application and those concerned with automation and monitoring can make use of kubernetes. All these three units can work hand in hand.

The future scope of the project will be to implement the proposed architecture practically. Doing so, will increase the efficiency of the developer and on production (Cloud) level.

References

Aquasec. (2015). Aquasec Containers: Docker Containers vs Virtual Machines. Retrieved February 04, 2019, from Aquasec: https://www.aquasec.com/wiki/display/containers/Docker+Containers+vs.+Virtual+Machines (Online Available)

Bauer, R. (2018, June 28). Backblaze Blog: What's the Diff: VMs vs Containers. Retrieved February 02, 2019, from BackBlaze: https://www.backblaze.com/blog/vm-vs-containers/

Bhattacharya, M. (n.d.). Docker Customers PayPal. Retrieved February 01, 2019, from Docker Customers: https://www.docker.com/customers/paypal (Online Available)

Bokhari, M. U., Shallal, Q. M., & Tamandan, Y. K. (2016). Cloud computing service models: A comparative study. 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom). New Delhi, India: IEEE.

Brooks, F. P. (1995). The mythical man-month: essay on software engineering. Addison-Wesley Publication Co.

Brunnert, A., Hoorn, A. V., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., . . . Walter, J. (2015). Performance-oriented DevOps: A Research Agenda. arXiv:1508.04752v1 [cs.SE].

CASB. (2018, August 07). Knowing About Container In Cloud Computing. Retrieved February 15, 2019, from CloudCodes: https://www.cloudcodes.com/blog/container-in-cloud-computing.html

Cloud Strategy Partners, L. (n.d.). Cloud Service and Deployment Models. IEEE Educational Activities and IEEE Cloud Computing. Retrieved from https://cloudcomputing.ieee.org/images/files/education/studygroup/Cloud_Service_and_Deployment_Models.pdf (Online Available)

Coleman, M. (2016). CONTAINERS AND VMS TOGETHER. Docker blog. Retrieved Febraury 01, 2019

collectd. (2017, November 21). The system statistics collection daemon. Retrieved January 28, 2019, from collectd: https://collectd.org/ (Online Available)

Continuous delivery. (n.d.). Retrieved from wikipedia: https://en.wikipedia.org/wiki/Continuous_delivery (Online Available)

Conway, M. E. (1968). How do committees invent. Boston: Datamation 14(4), 2831.

DeHmaer, B. (2015, July 16). Dockerfile: ENTRYPOINT vs CMD. Retrieved February 05, 2019, from CenturyLink Developer Blog: https://www.ctl.io/developers/blog/post/dockerfile-entrypoint-vs-cmd/ (Online Available)

Docker. (2016). Docker for the Virtualization Admin.

Docker. (2018). Retrieved March 02, 2019, from https://docs.docker.com/ (Online Available)

Dockerfile. (2017). Docker Documents: Dockerfile Reference. Retrieved February 05, 2019, from Docker Docs: https://docs.docker.com/engine/reference/builder/ (Online Available)

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2016). Microservices: yesterday, today, and tomorrow. arXiv:1606.04036v4 [cs.SE], 2-2.

Dyck, A., Penners, R., & Lichter, H. (2015). Towards Definitions for Release Engineering and DevOps. In the third preceedings of the Third International Workshop on Release Engineering, (pp. 3-4). New Jersey.

Dykstra, P. (1999, December 20). Gigabit Ethernet Jumbo Frames. Retrieved January 07, 2019, from http://proj.sunet.se/lanng/lanng2000/gigabit_jumbo_frames.html

Feng, X., Shen, J., & Fan, Y. (2009). REST：An Alternative to RPC for Web Services. First International Conference on Future Information Networks, (p. 4).

Fielding, R., & Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. Internet Engineering Steering Group (IESG).

Fong, J. (2017, October 26). Finnish Railways and Accenture Partner to Modernize Key Transportation APPS. Retrieved Febraury 01, 2019, from Docker Blog: https://blog.docker.com/2017/10/finnish-railways-accenture-partner-modernize-key-transportation-apps/ (Online Available)

Foundation, A. S. (2018, December 10). Apache Thrift Libraries. Retrieved from Apache Thrift (online available): https://thrift.apache.org/lib/ (Online Available)

Fowler, M. (2006, May 01). Continous Integration. Retrieved from MARTINFOWLER.COM: https://martinfowler.com/articles/continuousIntegration.html (Online Available)

Fowler, M. (2014, March 6). Circuit Breaker. Retrieved January 09, 2019, from MARTINFOWLER.COM: https://martinfowler.com/bliki/CircuitBreaker.html (Online Available)

Gilbert, S., & Lynch, N. (2002). Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services. In ACM SIGACT News. doi:10.1.1.20.1495

Glover, A., Matyas, S., & Duvall, P. M. (June 2007). Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional.

Google. (2019). Google trends compare. Retrieved march 03, 2019, from Google trends: https://trends.google.de/trends/explore?date=today%205-y&q=microservices,monolithic (Online Available)

Grafana. (n.d.). Grafana. Retrieved January 29, 2019, from https://grafana.com/ (Online Available)

Graphite. (2006). Graphite getting started. Retrieved January 29, 2019, from http://graphiteapp.org/ (Online Available)

Hawkular. (2016). Hawkular. Retrieved January 29, 2019, from http://www.hawkular.org/hawkular-apm/ (Online Available)

Hewlett, P. E. (n.d.). What is Container Orchestration? Retrieved March 01, 2019, from Hewlett Packard Enterprise: https://www.hpe.com/emea_europe/en/what-is/container-orchestration.html

Ihde, S., & Parikh, K. (2015). From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture., (p. 38).

Janetakis, N. (2017, May 5). Docker Tip #2: The Difference between COPY and ADD in a Dockerfile. Retrieved from Nick Janetakis: https://nickjanetakis.com/blog/docker-tip-2-the-difference-between-copy-and-add-in-a-dockerile

Janmyr, A. (2015, March 21). Jayway Blog: A not very short introduction to Docker. Retrieved February 05, 2019, from Jayway: https://blog.jayway.com/2015/03/21/a-not-very-short-introduction-to-docker/ (Online Available)

Joomla. (2018). Cloud Auditor. Retrieved September 28, 2018, from Trusting the cloud: https://www.trustingthecloud.eu/joomla/index.php/cloud-auditor (Online Available)

Joomla. (2018). Cloud Broker. Retrieved September 28, 2018, from Trusting the cloud: https://www.trustingthecloud.eu/joomla/index.php/cloud-broker (Online Available)

Joomla. (2018). Cloud Carrier. Retrieved September 28, 2018, from Trusting the cloud: https://www.trustingthecloud.eu/joomla/index.php/cloud-carrier (Online Available)

Joomla. (2018). Cloud Consumer. Retrieved September 28, 2018, from Trusting the cloud: https://www.trustingthecloud.eu/joomla/index.php/cloud-consumer (Online Available)

Joomla. (2018). Cloud provider. Retrieved September 28, 2018, from trusting the cloud: https://www.trustingthecloud.eu/joomla/index.php/cloud-provider (Online Available)

Kaur, P. D., & Kanu, P. (2015). Fault Tolerance Techniques and Architectures in Cloud Computing -A. International Conference on Green Computing and Internet of Things (ICGCIoT).

Kubernetes. (2018). Kubernetes Documentation Concepts. Retrieved February 20, 2019, from Kubernetes Official Webpage: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (Online Available)

Kubernetes. (2019, Feb 26). Retrieved Feb 26, 2019, from Kubernetes official homepage: https://kubernetes.io/ (Online Available)

Lewis, J. a. (2014, March 25). Microservices a definition of this new architectural term. Retrieved from MARTINFOWLER.COM: https://martinfowler.com/articles/microservices.html (Online Available)

Lukša, M. (2017). Kubernetes in Action. Manning Publications.

Madani, S. S., & Jamali, S. (2018). A COMPARATIVE STUDY OF FAULT TOLERANCE TECHNIQUES IN CLOUD COMPUTING. INTERNATIONAL JOURNAL OF RESEARCH IN COMPUTER APPLICATIONS AND ROBOTICS - ISSN 2320-7345, (p. 9).

Matthias, K., & Kane, S. (2015). Docker: Up & Running. O'Reilly Media.

Meegan, J. (2016, August 10). A practical guide to platform as a service: What is PaaS? Retrieved February 20, 2019, from IBM blogs cloud computing: https://www.ibm.com/blogs/cloud-computing/2016/08/10/practical-guide-paas/ (Online Available)

Mell, P. M., & Grance, T. (2011). The NIST Definition of Cloud Computing. Special Publication (NIST SP). Retrieved from https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

Mesos Documentation. (n.d.). Retrieved feb 26, 2018, from Apache Mesos: http://mesos.apache.org/

Metrics. (2010). Metrics. Retrieved January 29, 2019, from http://metrics.dropwizard.io

Netflix. (n.d.). Netflix Hystrix. Retrieved January 28, 2019, from Github: https://github.com/Netflix/Hystrix

Newmann, S. (2015). Microservices: Konzeption und Design.

Nickoloff, J. (2016). Docker in Action. New York: Manning Publications Co.

Nygard, M. T. (2007). Release It!: Design and Deploy Production-Ready. Pragmatic Programmers. Pragmatic Bookshelf.

O'Reilly, B., Molesky, J., & Humble, J. (2014). Lean Enterprise How High Performance Organizations Innovate at Scale. O'Reilly Media.

Peyrott, S. (2015, September 13). API Gateway. An Introduction to Microservices, Part 2. Retrieved November 15, 2018, from auth0 blog: https://auth0.com/blog/an-introduction-to-microservices-part-2-API-gateway/

Peyrott, S. (2015, November 09). Dependencies and Data Sharing. Retrieved November 16, 2018, from auth0 blog series: https://auth0.com/blog/introduction-to-microservices-part-4-dependencies/

Peyrott, S. (2015, October 02). The service registry. Retrieved November 15, 2018, from auth0 blog series: https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/

Podviaznikov, A. (2017). The Versatility of Docker Compose. Retrieved February 05, 2019, from Runnable: https://runnable.com/blog/the-versatility-of-docker-compose

Prometheus. (2014). Prometheus. Retrieved January 29, 2019, from https://prometheus.io/

Raymond, E. S. (2003). The Art of UNIX Programming. Addison-Wesley Professional.

Richards, M. (July 2016). Microservices AntiPatterns and Pitfalls. O'Reilly Media, Inc.

Richardson, C. (2018). Pattern: Microservice Architecture. Retrieved from Microservice.io: https://microservices.io/patterns/microservices.html

Rotem-Gal-Oz, A. (2009). Fallacies of Distributed Computing. Retrieved January 02, 2019, from : http://www.rgoarchitects.com/Files/fallacies

Sato, D. (2014, June 25). Canary Release. Retrieved from MARTINFOWLER.COM: https://martinfowler.com/bliki/CanaryRelease.html

Singh, A., & Kinger, S. (2013). An Efficient Fault Tolerance Mechanism Based on Moving Averages Algorithm. International Journal of Advanced Research in Computer Science and Software Engineering. Retrieved January 08, 2019, from https://pdfs.semanticscholar.org/f754/fe3cf1ba5b519b5de47ebb6c330d68630fdc.pdf

Sivagami, V. M., & EaswaraKumar, K. (2015). Survey on Fault Tolerance Techniques in Cloud Computing Environment. International Journal of Scientific Engineering and Applied Science (IJSEAS), (p. 7). Retrieved January 02, 2019, from http://ijseas.com/volume1/v1i9/ijseas20150952.pdf

Slee, M., Agarwal, A., & Kwiatkowski, M. (2007). Thrift: Scalable Cross-Language Services Implementation . Palo Alto, CA: Facebook White Paper, Vol. 5 .

Stone, L. (2016, September 29). Bringing Pokémon GO to life on Google Cloud. Retrieved February 15, 2019, from Google Cloud: https://cloud.google.com/blog/products/gcp/bringing-pokemon-go-to-life-on-google-cloud (Online Available)

Strijkers, R., Makkes, M. X., & Demchenko, Y. (2012). Intercloud Architecture for Interoperability and Integration. The 4th IEEE Conf. on Cloud Computing Technologies and Science (CloudCom2012) (p. 9). Taipei, Taiwan: IEEE.

Sumaray, A., & Makki, K. S. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. ICUIMC '12 Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication. Kuala Lumpur, Malaysia: doi>10.1145/2184751.2184810.

Thones, J. (2014). Microservices. IEEE Software, 116-116.

Tracing, O. (n.d.). Open Tracing Documentation. Retrieved January 28, 2019, from Open Tracing: https://opentracing.io/docs/

Villamizar, M. G. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. 10th Colombian Computing Conference, 10CCC 2015 (p. 8). Bogota, Colombia: Institute of Electrical and Electronics Engineers (IEEE).

Wagner, B., Torre, C. d., & Rousos, M. (2018). .NET Microservices: Architecture for Containerized .NET Applications. Washington: Microsoft Developer Division, .NET and Visual Studio product teams.

Warner, W. P. (1982). What is a software engineering environment (SEE)? (Desired Characteristics). Virginia: NSWC TR-82-465 .

Wolff, E. (2016). Microservices: Flexible Software Architecture. Addison-Wesley Professional.

Zipkin. (n.d.). Zipkin. Retrieved January 29, 2019, from Zipkin: https://zipkin.io/