Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering

Diploma Thesis

Microservices and Serverless architecture in web application

Apu Md Foyjur Rahman

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

MD FOYJUR RAHMAN APU

Systems Engineering and Informatics
Informatics

Thesis title

Microservices and Serverless Architecture in Web Applications

Objectives of thesis

The goal of this thesis is to explain the difference between monolithic approach and microservices in the web applications, describe the benefits of the serverless architecture in the web applications, single page applications in the Front-end development, the benefits and uses of the Containers, FaaS (Function as a Service). There will be also prepared an overview and the model of the proposed application in microservices and serverless architecture and the implementation of the proposed application

Methodology

In the beginning, the research of all the sources and prepare a good overview of the problem at hand will be performed. Next, the foundation by extracting all the relevant information from the sources, and creation comprehensive text with all the information mentioned above into literature overview. In the second part of the theoretical chapter, there will be description of all the services and technologies, which we will use in the implementation part, and description why do we choose them and what are their benefits including creation and design the model of the proposed simple web application. In the practical part, the implementation and steps needed to achieve the final working application will be documented.

The proposed extent of the thesis

60 – 100 pages

Keywords

microservices; serverless architecture; docker; web application; single page application

---

Recommended information sources

Nygard, Michael T. 2007. Release It! (Design and Deploy Production-Ready SoGware). 2007. ISBN-10: 0-9787392-1-3, ISBN-13: 978-0-9787392-1-8.
Rady, Ben. 2016. Serverless Single Page Apps. 2016. ISBN: 978-1-68050-149-0.
Richardson, Chris. Microservices Patterns. microservices.io. [Online] https://microservices.io/resources/index.html.

Expected date of thesis defence

2019/20 WS – FEM (February 2020)

The Diploma Thesis Supervisor

doc. Ing. Vojtěch Merunka, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 25. 11. 2019	Electronic approval: 25. 11. 2019
Ing. Martin Pelikán, Ph.D.	**Ing. Martin Pelikán, Ph.D.**
Head of department	Dean

Prague on 31. 03. 2020

Declaration

I declare that I have worked on my diploma thesis titled "**Microservices and Serverless architecture in web application**" by myself, and I have used only the sources mentioned at the end of the argument. As the author of the diploma thesis, I declare that the thesis does not break the copyrights of any person.

In Prague on ………………….

____

Apu Md Foyjur Rahman

Acknowledgment

I wish to thank my supervisor, doc. Ing. Vojtěch Merunka, Ph.D.for his support and guidance while working on this thesis. His invaluable insights about research have formed the basis of this thesis. I would also like to thanks our guest instructor Mr. Ing. Josef Pospíšil who also supported me patiently me throughout my dissertation work

Additionally, I would like to thank my family and close friends for their encouragement, support, and belief in me.

Microservices and Serverless architecture in web application

Abstract

The terminology "Microservice Architecture" has jumped over the decade a particular way of designing/architecting software applications as a bundle of independently deployable service. Microservice is not a new term, but it is an evolutionary term/architecture to find out a pathway to do revolutionary things that already took place in cloud, web, server virtualization, mobile, containerization. This is an explicit responsive architecture to this fantastic technology landscape that currently we live in. Alongside the term "Serverless Architecture" comes with a side by side to maximize and hassle-free automation in all life cycle of the application product. The Serverless architecture or computing offers the potential of application software in the cloud in the manner of auto-scaling, pay-as-you-go. This is a new trend in cloud computing, which enabled the business to get rid of underlying infrastructure issues.

My thesis aims to provide a more conclusive answer of how the Microservices and Serverless architecture reshape the current web application world.

In this study, we find out the difference between the classic monolithic and Microservices approach, Serverless architecture and standard backend approach, Single page and contemporary front-end approach, containers, and server hosting as well as FaaS and SaaS overview and some other third party services. We also conducted a single page application based on proposed architecture.

Keywords: Microservice, Serverless Architecture, Cloud, Docker, Virtualization, scalability, Single page application, Web Application, SPA.

Table of content

List of pictures

List of tables

List of abbreviations

List of abbreviations

HTTP	HyperText Transport Protocol
API	Application Programming Interface
SCM	Source Control Management
SBT	Simple Build Tool
OS	Operating System
SOA	Service Oriented Protocol
REST	Representational State Transfer
IDE	Integrated Development Environment
SMA	Scalable Microservice Architecture
IP	Internet Protocol
DNS	Dynamic Name System
IT	Information Technology
SAAS	Software AS A Service
SOAP	Simple Object Access Protocol
ESB	Enterprise Service Bus
LXC	LinuX Containers
VM	Virtual Machine
CPU	Central Processing Unit
WSDL	Web Service Description Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
IEEE	Institute of Electrical &Electronics Engineers
SLA	Service Level Agreements

NIST	National Institute of Standards
KVM	Kernel Virtual Machine
VMM	Virtual Machine Monitor
PAAS	Platform As A Service
NIST	Infrastructure As A Service
UTS	Unix Time Sharing
PID	Process ID number space
SDN	Software Defined Networking
CNM	Container Networking Model
BGP	Border Gateway Protocol
CQRS	Command Query Responsibility Segregation
SRV Record	SeRVice Record
LAN	Local Area Network
RPC	Remote Procedure Call
AMQP	Active Message Queuing Protocol
DSRM	Design Science Research Methodology
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
AUFS	Advanced multi layered Unification FileSystem
SHA	Secure Hash Algorithm
CI	Continuous Integration
CD	Continuous Delivery
SDK	Software Development Kit
UI	User Interface
POM	Project Object Model

NIST	National Institute of Standards
KVM	Kernel Virtual Machine
VMM	Virtual Machine Monitor
PAAS	Platform As A Service
NIST	Infrastructure As A Service
UTS	Unix Time Sharing
PID	Process ID number space
SDN	Software Defined Networking
CNM	Container Networking Model
BGP	Border Gateway Protocol
CQRS	Command Query Responsibility Segregation
SRV Record	SeRVice Record
LAN	Local Area Network
RPC	Remote Procedure Call
AMQP	Active Message Queuing Protocol
DSRM	Design Science Research Methodology
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
AUFS	Advanced multi layered Unification FileSystem
SHA	Secure Hash Algorithm
CI	Continuous Integration
CD	Continuous Delivery
SDK	Software Development Kit
UI	User Interface
POM	Project Object Model

1 Introduction

In this modern world, the infinite number of people and devices itself is gaining access to applications throughout the internet. This is massively increasing the demands on the system, which should be available and reliable at any time as they need. This is why many developers, architects, and Software companies started to research how to address and resolve these challenges.

As a result, the terms Microservices architecture came up. Microservices is a new architectural model that emerged in the past few years. There is no appropriate or official definition of the Microservice architecture model, but in short, Microservice is a collection of small pieces of autonomous services. It evolves a consensus over time in the industry.

Microservices architecture enables small teams of developers to focus on individual services, each of them has a specific task, and the Serverless computing facilitates these Microservices up and running with the least effort and time. The serverless meaning function of distributed computing and Serverless computing is not a technical term but refers to a system where software systems are deploying in a cloud platform. Here the people who are developing the software/application and use the service are not concern about the underlying infrastructure. So cloud computing is on-demand availability of computer systems resources, especially the underlying resources like the power of computing, storage, etc. without direct intervention by the consumer.

This dissertation aims to explore various architecture for web application and compare the capabilities, performance, agility, scalability between the monolithic and Microservice architecture.

In the modern era, the evolution of cloud computing excessively switches the strategy of developers to build and implement applications. In its Top 10 Strategical Technology Trends for 2016 (**Gartner Research, 2015**), states that **—** “The service architecture and mesh app are what enables delivery of apps and services to the flexible and dynamic environment of the digital mesh. This architecture can ensure users' requirements as their demand. It conducts together with the many information sources, devices, apps, services, and microservices into a flexible architecture in which apps extend across multiple endpoint devices and can coordinate with one another to produce a continuous digital experience. IT will increasingly deliver services as cloud services in the mesh app and service architecture, supported by software-defined application architectures, containers, and microservices. IT needs a DevOps mindset to bring together development and operations in support of continuous development, and continuous integration and delivery."

Software architecture has witnessed increasing interest from both the academia and the software industry. There has existed various software architecture for distributed systems including but not limited to Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), and Service-Oriented Architecture (SOA). Most of these designs were led by a consortium of large software Vendors and had little backing from the open-source community.

2 Objectives and Methodology

3.1 Objectives

- Explain the difference between the monolithic approach and Microservices in web applications.
- Which reference architecture can best serve as the base for scalable web services?
- The principal research question, we derived the following specific questions:
    1. What aspects are influencing the adoption of Microservice Architecture?
    2. To what extend can containerization enhance the design and implementation of Microservice Architecture?
    3. To what extend can Microservice architecture improve the scalability of web services?
    4. To what extend can Microservice testing be automated?
- Develop and test a scalable Microservice Architecture for web services
- Describe the benefits of Serverless architecture in web applications.
- Describe single-page applications in Front-end development.
- Describe the benefits and uses of the Containers.
- Describe the various virtualization technologies.
- Prepare an overview and the model of the proposed application in Microservices and Serverless architecture.
- Implement the proposed application

3.2 Methodology

In the study, we employ two types of methods that are conducted concurrently. The research methodology will, however, inform the system development methodology.

2.2.1 Research Methodology

In these studies, a research methodology called Design Science Research Methodology (DSRM) is employed. The object that we propose is a scalable Microservice Architecture for Web Service. Research phases that have to be carried out in DSRM are (**Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007**):

1) Problem denotation & study(evaluation of current practice)

2) Determine the objectives of a solution (what would a better artifact accomplish?)

3) Prototype design & development

4) Prototype demonstration (finding a suitable context then use the artifact to solve problems)

5) Prototype evaluation (observing how effective it is in solving the problem)

6) Communication.

Problem Definition and Analysis

After this point, usually, the process iterates back to step (2) or (3). Following this DSRM method, we first investigate the market to gain insight into state of the art relating to Microservice Architecture (step 1 in DSRM). Based on the findings of this market analysis, we will identify issues associated with the platforms concerning scalability, which motivates the need for a new platform. Current technology used, architecture components, and functionality gaps will be acknowledged as well. Step (1) will be covered in chapter 1 and 2 in this report.

Defining objectives of a solution

In the next step (2), we will propose requirements and architecture components that need to be incorporated into the platform design based on the literature study. This phase is necessary to illustrate the inadequacy of solutions in the market in achieving our project goals. We will carry out a literature study on the topics of web application architecture, system-level virtualization, and scalability. In the practical portion, describe our implementation and steps needed to achieve the final working application.

3 Literature Review

3.1 Introduction

The increasing number of connected devices is exponentially rising. According to (**Gartner Research, 2015**) the number of mobile devices will increase from the current six billion to twenty billion by the year 2020. In recent years the Microservice Architecture e has become famous for building web applications (**Adrian Cockcroft, 2014**), Twitter (**Jeremy Cloud, 2013**). Microservices is an Architectural style that realizes a single Software System as many small individual loosely coupled applications that communicate over a network. Each application has its own software development lifecycle. This decoupling allows many small teams to work on individual applications. All forms then converge to deliver one software product to the users, who perceive the whole architecture as one single system. The Microservice Architecture approach allows faster delivery of smaller incremental changes to an application.

On the one hand, the Microservice Architecture approach builds on Agile Methodology and DevOps principles. The DevOps philosophy is the realization that software development (Dev) and operations (Ops) teams need to communicate and collaborate to enable organizations to shorten the time it takes to transform developed software into running services. DevOps employs some practices that are well suited to the use of virtualization. Virtualization, OS-level virtualization, or Docker containerization, in particular, is key to automating most of the software deployment operations. In the following section, we define the terms that are used in this thesis.

3.2 Definition of Terms

3.2.1 Microservices

An architectural style that extends SOA principles by decomposing of an application into single-purpose, loosely coupled services managed by cross-functional teams (**Martin Fowler et al., 2014**).

Microservice Architecture

Microservice Architecture is being adopted by software as a Service (SaaS) and Function as a Service (FaaS) vendors due to the need to shorten software development cycles from several months to minutes. In this regard, Microservice Architecture is one of the prerequisites for agile methodologies based on DevOps. DevOps principles advocate for automation of most tasks of software deployment and are more inclined to use cloud computing technologies such as virtualization. Both Microservice Architecture introduces challenges such as increased inter-process communication, high fault rate, increased number of tests, and the need for consistency in the distributed data stores. Various desperate tools have been used to address these challenges. One mechanism based on OS-Level virtualization introduced by a Silicon Valley start-up called Docker is proposed as a means to simplify the realization of Microservice Architecture.

The Microservice Architecture was pioneered by web-scale companies (Netflix, Amazon, eBay, twitter), and is a paradigm shift for service development for fast-moving business needs. Microservice Architecture has accelerated innovation in these companies by enabling them to meet digital business challenges. (**Stackify, 2019**)

Continuous Delivery

Continuous Delivery (**Humble et al., 2010**) is a software development discipline that enables the on-demand deployment of software to any domain. With uninterrupted delivery, the software delivery life cycle will be automated as much as possible. It leverages techniques like Continuous Integration and Continuous Deployment and embraces DevOps.

On-demand Integration

On-demand Integration is a software development approach where members of a team integrate their work regularly, leading to multiple integrations per day. Each integration test is verified by an automated build to detect integration problems as quickly as possible **(Martin Fowler et al., 2014**).

Configuration Management

Configuration management or infrastructure automation -refers to monitoring and controlling changes to the software codebase. It‘s a constant practice for establishing and maintaining consistent product performance, especially in DevOps environments.

DevOps

DevOps is a set of exercises intended to minimize the execution period between committing a change to a system and the deployment being placed into average production while ensuring high quality. DevOps is a cultural and technical movement that focuses on building and operating high-velocity organizations (**Chef 2014, IBM 2014**).

DevOps is an IT organizational model in which system administrators work side-by-side with developers in a single, coordinated, agile environment. DevOps also breaks down corporate walls, and it promotes a fundamentally different way of solving IT problems (**Rackspace, 2015**).

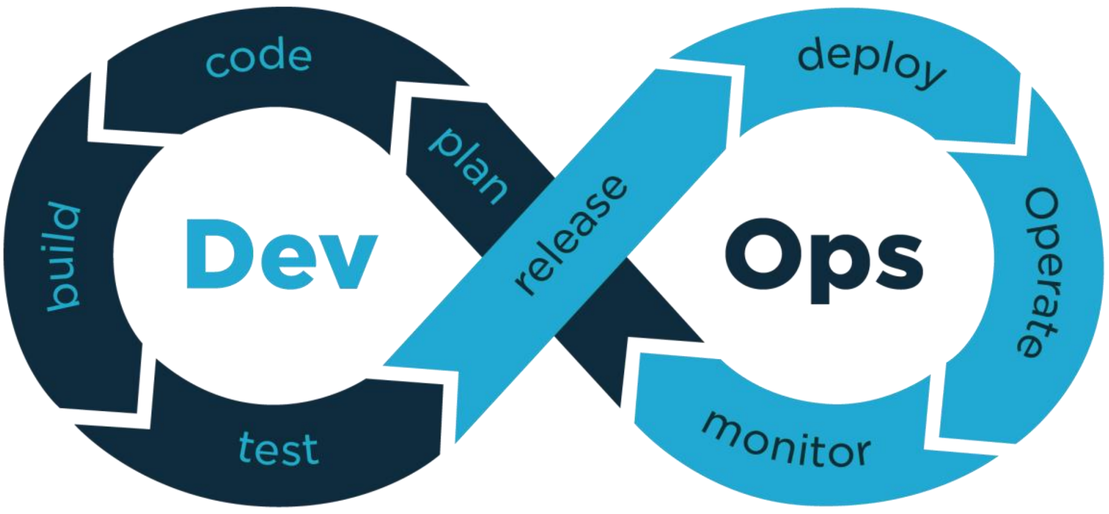


Figure 1: DevOps Cycle

3.3 OS-Level Virtualization

Operating System-level virtualization is a technology that partitions the operating system and creates multiple isolated Virtual Machines (VM). OS-level virtualization is a virtual execution environment that can be forked instantly from the base working environment (**Yang Yu, 2007; J. Lakshmi, 2010; Mathijs J.S, 2014**).

Operating System-level virtualization has been widely used to improve security, manageability, and availability of today‘s complex software environment, with small runtime and resource overhead, and with minimal changes to the existing computing infrastructure (**Pasa Maharjan, 2011**).

Docker

Docker is a free, open-source project that automates the packaging, shipping and deployment of applications using containers, by delivering an additional layer of abstraction and automation of OS-Level Virtualization on Linux (**Vladimír Jurenka, 2015**). Docker engine quickly wraps up any application and all its libraries and dependencies into a lightweight, portable, self-sufficient container that can run on any Linux based system (**Kavita Argarwal, 2015**).

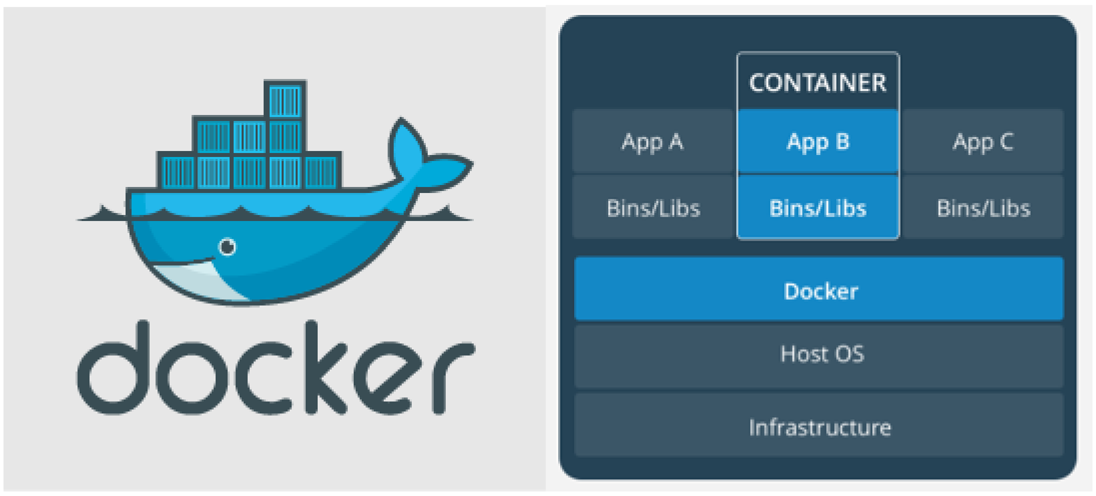


Figure 2: Docker Architecture

3.4 Monolithic Architecture

Monolithic architecture is a set up used for traditional server-side systems. The entire system function is based on a single application. A monolithic architecture consists of various kinds of components as required for the desired application that is all tightly coupled together and has to be developed, implements, and managed as one entity meaning application bundle all its functionality and service into a single workable unit. All the components are running on the same file system. This kind of system has various advantages. First, it is faster to develop, and it doesn't need to communicate via APIs. Monolithic by the name means composed of all functions in one piece.

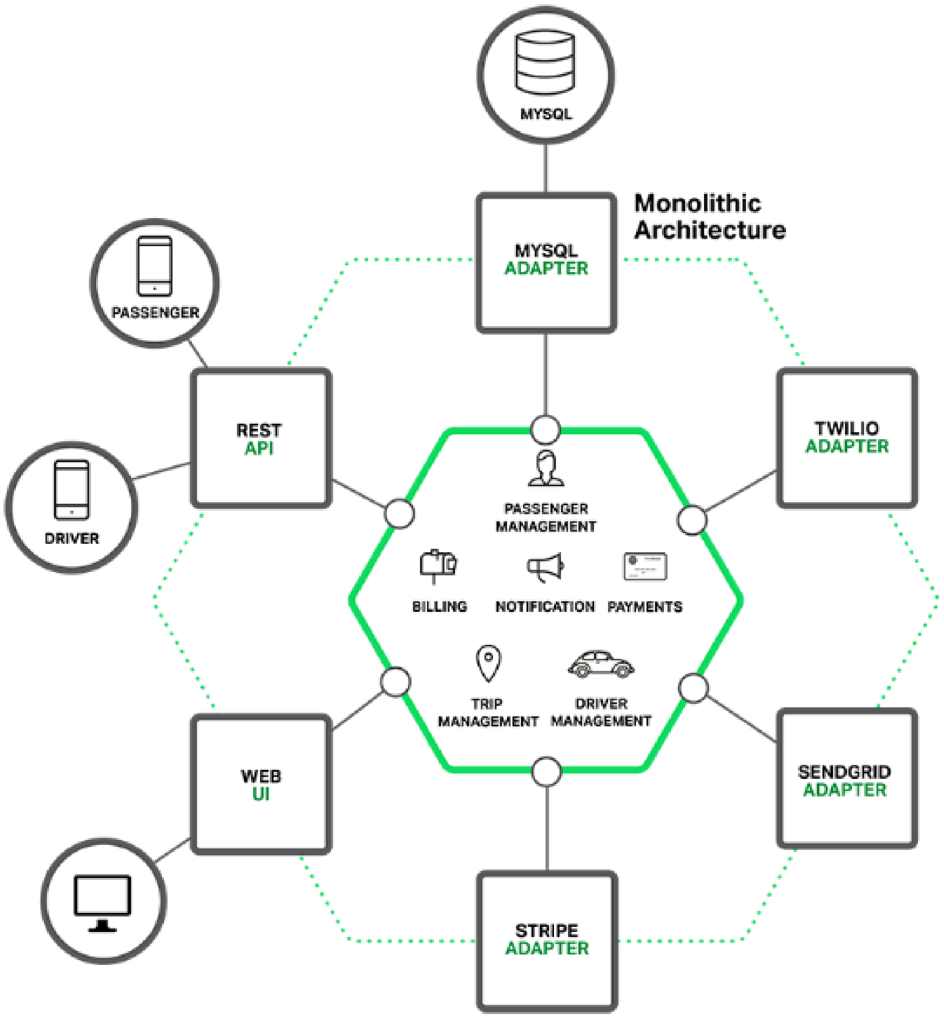


Figure 3: Monolithic Architecture

Benefits of Monolithic Architecture

When the terms come into account to develop a server-side application that consists of different types of components:

- Presentation – Subject to handling HTTP requests and responding with JSON/XML or either HTML (For web services API)
- Business Logic – The application of business logic
- Database access – Database object is responsible for accessing the database.
- Application integration – Possible to integration (e.g., via messaging or REST API)

Despite having a logically modular-based architecture, the application resides in the form of packaged and deployed as a monolith. The benefits of monolithic architecture as follows:

- Simple to develop
- Simple to conduct the testing. For example, to implement end-to-end testing by only launching the application and testing the UI.
- Simple to deploy. It just needs to copy of the packaged application to a server.
- It can be simple to scale horizontally by running multiple copies behind a load balancer.
- In the early stages, the monolithic approach works well, and most of the prominent and eminent applications which exist till the date were started as a monolith.

Drawbacks of Monolithic Architecture

- Monolithic is a simple approach but limitation in size and complexity
- An application becomes large and difficult to understand as well as not easy to fast changes and correctly.
- The system start-up time can be slower, depending on the size of the application.
- It must re-deploy the complete application on each change/update.
- Continuous expansion is challenging
- Scalability is very complicated when other modules have resource requirements.
- There is a significant impact on application reliability. The potential bug can bring down the entire process of application. Since all the parts of the application are a blend, that bug can cause the availability of the whole application.
- The monolithic application cannot embrace new technologies due to its limitations. Any changes in a component (framework) or language causes influence an entire system

3.5 Software Application

A Software application is the implementation of capabilities and virtualization by building and deploying a set of instruction through coding using a programming language. The users of an application can observe its real-world effects during operation. It is becoming a common trend to implement most capabilities through software. i.e., software-defined Networking, Network Function Virtualization, and Software-Defined Storage.

Process

An Operating System process is the concrete representation of a software application at runtime. One method always belongs to one app, and software applications may have many running processes. A process instance exists during execution, while a process type is a logical entity embodying the opportunity to execute process instances of it. A process type usually manifests itself in some way in the source code of an application.

Web Service

A web service facilitates the availability of capabilities of an application to other applications via a network. Web services enable communication between program functions, without the necessity of middleware. The possible ways of access are defined through a service interface.

3.6 Principles of the Microservice Architecture

The core principles of the Microservice Architecture are:

1. **Modeled Around Business Domain**: Domain-driven features enable us to find stable and reusable boundaries.
2. **Automation Culture**: Microservice is a compound of many moving parts means automation is a crucial factor.
3. **Hide implementation specification**: The drawbacks of the distributed system can often become tightly coupled with their provided services together.
4. **Decentralize features:** To achieve liberty, it can urge out of the center, architecturally, and organizationally.
5. **Deploy Autonomously/Independently**: The most significant characteristic of microservices know as.
6. **Consumer First**: Make the service available and easy to consume, as a creator of an API
7. **Failure Isolation:** Since microservice is a piece of service, it’s easier to isolate the affected part and spin-up the workable copy.
8. **Highly Manifest:** Microservice contains many moving parts, so what is happening in the system always can be challenging.

Because microservices are small, they are easier to produce, and it matters a great deal less what your implementation language is. Microservices may be entirely disposable, as rewriting your functionality is not that much work. Microservices communicate over the network using messages. Microservices are wholly defined by the messages they accept, and the messages they emit. From the point of view, an individual microservice instance, and from the developer writing that microservice, there is only content arriving, and content to send. In deployment, that microservice instance may be participating in a request/response configuration, or a publish/subscribe arrangement or any number of variants. How messages are distributed is not a defining characteristic of the Microservice Architecture. All distribution strategies are welcome without prejudice.

A network of microservices is dynamic. It consists of large numbers of independent processes running in parallel. You are free to add and remove an instance of services at will. This makes scaling, fault tolerance, and continuous delivery, and low risk. Naturally, you will need some automation to control the extensive network of services. This is where OS-level virtualization or containerization comes in. Containerization is becoming the preferred means of automating the building, packaging, and deployment of microservices. Containerization enables automation and gives you real control of your software development and deployment system, and immunizes you against human error. This is a barely low-risk procedure compared to big bang monolith deployments. (**SAM Newman, 2015**)

3.6.1 Microservice Architectural Constraints

Microservices is a contemporary software and delivery pattern, where applications are orchestrated several pieces of runtime services.

With Microservices, a software component is a hand over as an autonomous runtime service by a well-defined universal API. The microservices approach permits much quicker delivery of smaller steady changes to an application. On the other hand, Microservices needs expertise in spread programming and can become n functional nightmare without having proper tools in place. The following characteristics must-have in the microservices architecture style:

- Resilient
- Formulate
- Minimal
- Elastic, and
- Accomplish

Formulate

A microservice must provide an interface that is steady and is formulated to support service composition service.

Microservice APIs should be formulated with a popular way of identifying, manipulating resources, representing, and describing the schema and supported operations of API.

The identical interface obstacle of the REST API architectural design stated this in detail.

A microservice pattern should be designed to promote composition patterns like linking, aggregation, and high-level functions like as caching, gateways, and proxies.

Minimal

A Microservice should only contain highly conjunctive entities.

In software, conjunctive is a dimension of whether things belong together. A module has high cohesion if all the functions and objects in it are targeted on the same tasks. Higher cohesion leads to more maintainable software.

Elastic

A microservice should be able to scale, up and down, respectively, of other services within the same application.

This obligation implies based on load or other factors. It is possible to fine-tune application performance, resource usage, and availability. This obligation can be identified in different ways. Still, a well-known pattern is to design the system so that it can run multiple stateless representatives of each microservices, and there is a method for registration, Service naming, and discovery alongside with routing and load-balancing requests.

Resilient

A microservices might be failed without impacting other services within the same application.

A failure of an individual service instance must have minimal impact on the application. A breakdown of all the instances of a microservice should only bump a single application, and users must be able to continue with the rest of the application without collision.

Accomplish

A microservice should be functionally and operationally complete.

The creator of C++, Mr. Bjarne Stroustrup, said that a good interface should be "minimal but complete." For example: as small as possible but no smaller.

Likewise, a Microservice might offer a complete function, minimum dependencies (loose coupling) to other services in the application. It is crucial. Otherwise, it becomes unattainable to version control and upgrade separate services. (**Nirmata, 2015**)

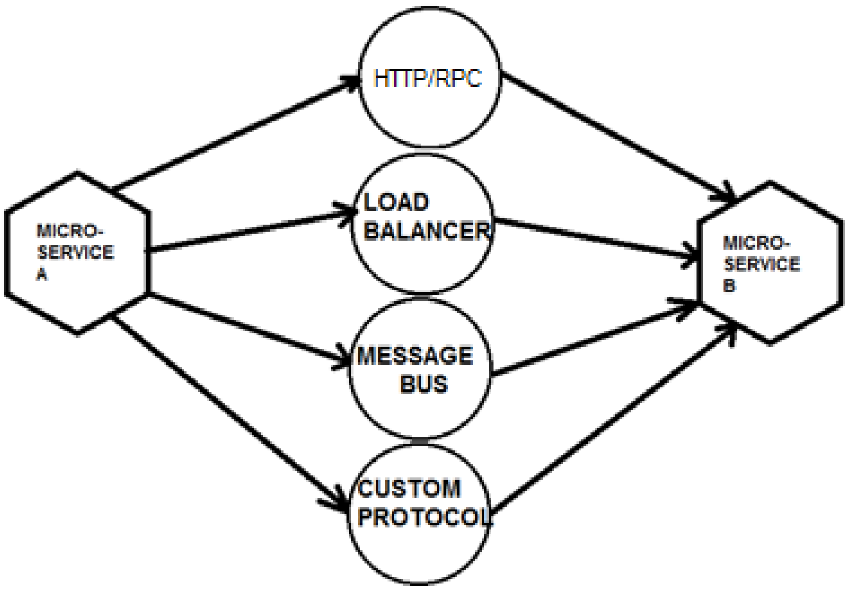


Figure 4: Microservice mechanisms for loose coupling

3.7 Conceptual design

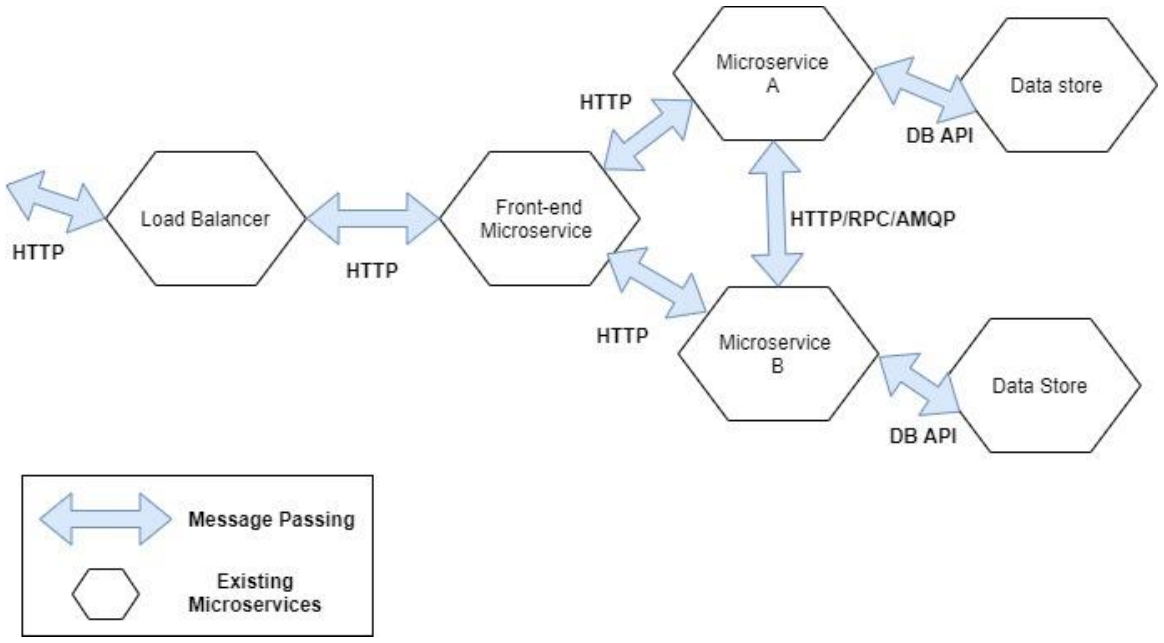


Figure 5: Microservice Architecture Conceptual Model

2.2.2 Software Functional components

Software functional component is reasonably big-scale code construction within an application, with an explicitly-defined API, that could potentially be exchanged out for another implementation. Microservice Architecture is an extension of the component-based software system and is distinguished by the fact that the code base is divided into discrete pieces that provide services through well-defined, limited interactions with other components.

3.8 REST Architectural Styles and Architectural Constraints

Fielding definition (**Roy T. Fielding, 2000**) of architectural style involves architectural constraints. Fielding defines Architectural style as a coordinated set of structural restraints that limited the roles and features of architectural components and the allowed relationships among those elements within any architecture that conform to that style. For this reason, consistent with his definition, he introduced REST through a set of constraints, i.e., client-server, stateless, cache, and uniform interface.

REST is a set of obstruction that informs the design of scalable hypermedia web applications. REST architectural style claims that these constraints will result in an architecture that works well in the areas of scalability, resiliency, usability, and accessibility. It seems to be accepted nowadays that REST indeed does lead to designs that are less tightly coupled than the more traditional architectures that have been informing the design of distributed systems and enterprise IT architectures.

3.9 Microservices Architectural Style

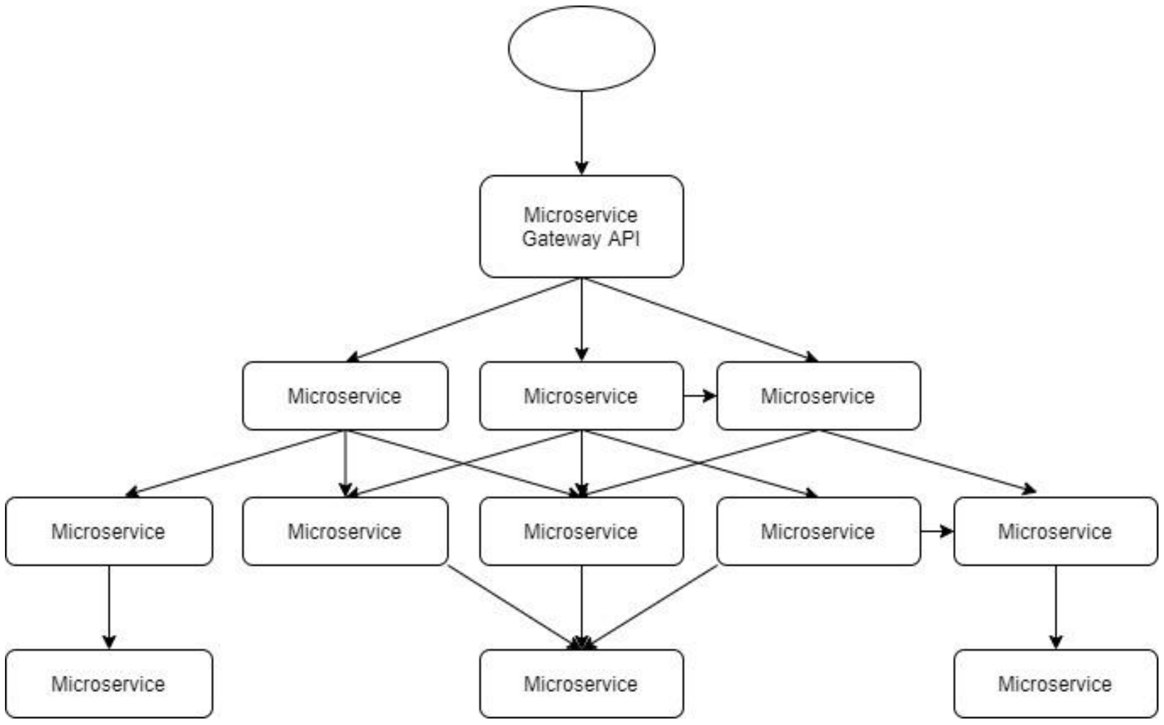


Figure 6: Microservice Hierarchical tree

Each user request is satisfied by a sequence of services

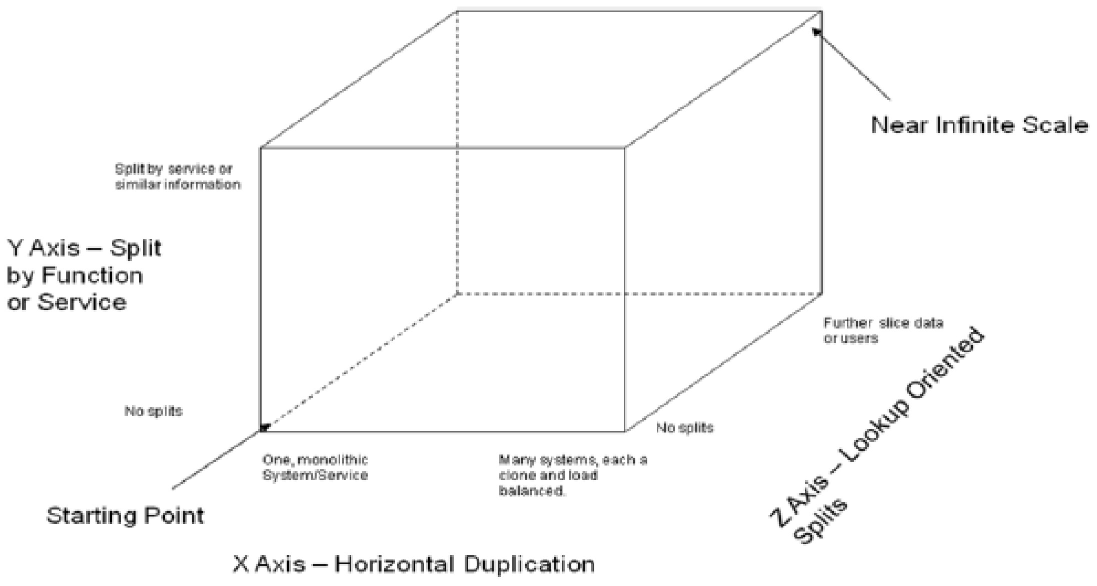
- Most services are internally available
- Each service communicates with other services through service interfaces

There is no particular definition of the Microservice architectural style, there are specific common characteristics such as automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

3.10 Characteristics of Microservices

Independent Scaling

Scalability is how a computer system, network, or process able to deal with a growing amount of workload capably or its ability to be enlarged to accommodate that growth (**Wikipedia**). According to Amazon —a service is said to be scalable if when we increase the resources in a system, it results in improved performance in a manner proportional to resources added. [**adopted by Martin L. Abbott et al., 2015**].


Near Infinite Scale
Split by service or similar information
Y Axis – Split by Function or Service
No splits
Further slice data or users
Z Axis – Lookup Oriented Splits
No splits
One, monolithic System/Service
Many systems, each a clone and load balanced.
Starting Point
X Axis – Horizontal Duplication

X-Axis scalability

X-axis scaling composed of running manifold replication of an application beyond a load balancer. If there are N replicas, then each reproduction handles 1/N of the load. This is a conventional, commonly used approach of scaling an application.

Z-Axis scalability

Z-axis scalability is achieved widely used to make data stores more elastic. Data is divided (a.k.a. sharded) across a set of servers based on an attribute of each record.

Y-Axis scalability

Unlike the X-axis and Z-axis, which consist of running multiple, identical copies of the application, Y-axis axis scaling splits the form into various contrasting services. Every service is responsible for one or more closely related functions. Each Microservices can be scaled independently.

- Identified bottlenecks can be addressed directly
- Data sharding can be applied to Microservice as required
- Components of the System that do not represent bottlenecks can remain un-scaled and straightforward

Discrete Transformation of the Features

Microservices can be expanded without affecting other services

- For example, you can deploy a new version of service without re-deploying the whole system
- You can also step further as to replace the service by a complete rewrite. This is achieved through API versioning. A new API is introduced while the old can only be retired after the service consumers have migrated to the new API. In practice, the service may have several stable API versions.

Stable Interfaces – Standardized Communication

Communication between Microservices is often standardized using HTTP(S), gRPC, and AMQP – battle-tested and broadly available transport protocols. HTTP has been proven as the dominant protocol on the World Wide Web that is highly scalable.

REST – uniform interfaces on data as resources with known manipulation means

- Client-Server: Separation of logic from the user interface
- Stateless: no client context on the server
- Cacheable: reduce redundant interaction between client and server
- Layered system: intermediaries may relay communication between client and server (e.g., for load balancing)

- Code on demand: serve code to be executed on the client (e.g., JavaScript)
- Uniform interface

JSON – simple data representation format

REST and JSON are convenient because they simplify interface evolution

3.11 Challenges to a Microservice Architecture

Any application architecture that strives to solve issues of scaling does have several concerns, given the complicated nature of existing systems. Decoupling an application into discrete services means that there are now more moving parts to maintain.

Complex Orchestration

While a critical benefit of Microservice is its streamlined orchestration capabilities, more services mean maintaining more deployment flows.

Internal Service communication

Dissociate services need a reliable, effective way to communicate internally with various components of an entire application. Which hosted in the individually microservice instances. Delivering data over the network put forward latency and potential failure, which can interfere with the user experience. A common approach is to add API Gateway to coordinate all communication between users and services.

Data Reliability

As with any circulate architecture, ensuring compatibility is a challenge, both for data at rest and data in motion. Multiple replicated databases and continuous data delivery can easily lead to inconsistencies without the proper mechanisms in place.

Support High Availability

Ensuring high-availability is a necessity in any production system. Microservice provides more effective isolation and scalability; however, the uptime of each service contributes to the overall availability of applications. Each facility must then have its own distributed measures implemented to ensure application-wide availability.

Testing

While maintaining code and dependencies tight means a more straightforward development environment for particular services, it does inject challenges with examining as it delineates to the whole application. Services will often need to interact with each other or depend on a data origin or API. Testing one service individually would then require a complete test environment to be effective.

3.12 Microservice Architecture, DevOps and Containers

DevOps is a Pre-requisite to Successfully dramatize Microservices

Microservices, DevOps, and Containers are very interrelated and like birds of same feathers flock together. Monolithic application, when split into Microservices, enables higher modularity that leads to a more coherent set of functions that are independent of the rest of the system. DevOps is the practice each team uses to build and operate these Microservices, allowing each side to have a shared success story amongst the diverse set of roles of the entire system. Containers have become the way these Microservices are packaged, deployed, and released on infrastructure. This leads to better infrastructure utilization and simplifies the way a change is moved from a development environment to the production environment.

As monolithic applications are incrementally functionally decomposed into foundational platform services and vertical services, you no longer just have a single release team to build, deploy, and test your application. Microservice Architecture results in more frequent and more significant numbers of smaller applications being deployed. DevOps is what allows you to do more frequent deployments and scale to handle the growing number of new teams releasing frequent changes. Containerization facilitates and end-to-end pipeline for the software lifecycle. DevOps is a prerequisite for being able to successfully adopt Microservice at scale in a given organization (**IBM, 2015**).

3.13 Cloud Computing

The vast development of cloud computing technology in the recent past has a substantial impact on Service provisioning landscape as more and more enterprises begin to adopt this technology. The term "Cloud Computing" is currently a hot and highly discussed topic in both the technical, economic, and research world. It is used for describing what happens when applications and services hosted in remote data centers such as Amazon, Azure, or

Cloud Foundry. Cloud computing is not so new. However, more currently, though, cloud computing refers to many different types of services and applications being delivered in the internet cloud. Cloud computing definition remains unclear. Many people within the industrial and academic community have attempted to define what "Cloud Computing" really is, and what typical characteristics it presents.

A formal definition for cloud computing is given by (**Buyya et al. 2009**) as "Cloud is a parallel and distributed computing system composed of a collection of internally connected and virtualized systems which are dynamically catered and presented as one or more unified computing resources based on service-level-agreements (SLA) established through negotiation between the service provider and consumers."

According to National Institute of Standards and Technology (NIST) Cloud computing is defined as ―a model for facilitating ubiquitous, convenient, real-time network access to a shared pool of configurable computing resources (e.g., servers, networks, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction‖ (**P. Mell et al. 2011**). This cloud model is consists of five essential characteristics, three service models, and several evolving deployment models.

Characteristics and Benefits of Cloud Computing

One of the essential aspects possessed by cloud computing is elasticity, which is the ability to dynamically scale up or scale down computing resources whenever required to match with system workload within a compact time frame (typically within minutes). Through elasticity, users of cloud computing could avoid the risk of over-provisioning (underutilization) and under-provisioning (saturation). Other notable characteristics, including on-demand self-service, resource pooling, and multi-tenancy (multiple customers can use the same computing infrastructure and network, which results in an increase utilization rate). (**Jula, Sundararajan, & Othman, 2014**)

The essential characteristics of Cloud Computing are extracted from its definition (**Mell & Grance 2011, Mahmood & Hill 2011**) and are summarized in Table 1.

Table 1: Characteristics of Cloud Computing

On-demand self-service	Enables the users to access and consume computing capabilities with limited interaction between user and service provider.
Broad network access	The computing capabilities and resources are available online and can be used by different users through standardized mechanisms.
Resource pooling	*Heterogeneous computing resources can be combined and automatically assigned to serve different users based on a multi-tenant model.*
Highly Scalable	*Every computing capability and resource can be provisioned rapidly, elastically and/or automatically to scale horizontally or vertically.*
Metrics provision	*Provide monitoring, controlling and reporting for billing purpose and transparency between the service provider and the user automatically.*

The characteristics above can imply various benefits for potential customers. The most major benefits of Cloud Computing are the following: (**Mahmood & Hill, 2011**).

- ❖ **Cost Reduction** can be achieved by avoiding CAPEX for software and hardware acquisition. Massive cost reduction for OPEX and training.
- ❖ **Scalability** can achieve by adopting virtualization technology, resulting in innovation and change capacity for the organization
- ❖ **Access to various IT Services** for small and medium enterprises is available at per second/minute billing. Cloud Ecosystems provide **disaster recovery** and **business continuity** plans through the regional distribution of resources.
- ❖ **Availability** users can ubiquitously access their resources.

2.2.3 Cloud Computing Service Delivery Models

Three different cloud computing service delivery models could be distinguished (**Jula et al., 2014; Rimal, Jukan, Katsaros, & Goeleven, 2010; Mell & Grance, 2011**). Four more service models are emerging. These are Container as a Service [CaaS], Function as a Service [FaaS], DataBase as a Service [DaaS], and Docker Based PaaS (**Alex Williams et al., 2016**).

Table 2: Various cloud computing service delivery models

Service Model	**Description**	**Example Service**
Container as a Service	Encapsulates several components of the software development lifecycle such as Container orchestration and registries	Amazon EC2 Container Service, Docker cloud
Function as a Service or Backend as a Service	Based on –serverless‖ architecture where you don‘t have to manage the infrastructure used to execute your code: scaling, availability, patching and so on are taken care of by the service itself.	AWS Lambda, Google Cloud Function, IronWorker
Database as a Service	A shared, consolidated platform to provision database services using a self-service model for provisioning those resources with Elasticity to scale out and scale back database resources.	Amazon DynamoBD, Google‘s Firebase, Oracle 12c
Software as a Service (SaaS)	Service Providers) offer the computing capability which is deployed on a Cloud infrastructure. The consumers can access the applications through a web browser or a program interface. The software is installed in the Data centre where it can be managed, controlled and updated.	Gmail, Google Docs, YouTube, Facebook, SalesForce.com.
Platform as a Service (PaaS)	The consumers are provided with a platform where they can deploy their applications. The platform provides programming languages, tools, and libraries that can be used by the consumers to build and run their applications.	Elastic Beanstalk, Microsoft Azure.

Docker-based PAAS	This kind of infrastructure is built from the ground up using containerization platform such Docker	Deis, Flynn, Cloud Foundry and Openshift
Infrastructure as a Service (IaaS)	The consumer is provided with fundamental computing resources such as storage, networks, or processing. The consumer can use these computing resources to deploy and run applications or even operating systems.	(EC2, Windows Azure Virtual Machines, Google Compute Engine.

3.14 Serverless architecture

Serverless architecture is an emerging trend that is quickly gaining momentum. The idea is to be able to run server-side code without worrying about the messy details of provisioning and setting up servers. You write code, upload it, and it spin-up. All the challenges of managing the infrastructure, provisioning servers, auto-scaling, installing languages, and frameworks are eliminated and hidden away by the vendor. Examples include AWS Lambda and Google Cloud Function. According to (**Danilo Poccia, 2016**), the introduction of AWS Lambda, the abstraction layer is set higher, allowing developers to upload their code grouped in functions, and let the platform execute those functions.

Iron.io, a startup, has also introduced IronWorker as a serverless architecture that enables engineers to guide the machines to execute code in reaction to a particular context. It can facilitate in the operation as it comes in, like cleaning up data, providing notifications at scale, delivering out emails, or handling mobile check-ins quickly.

IronWorker can be used for a wide range of purposes. IronMQ message queue tools can be run on public clouds or in on-premises data centers. According to (**Ivan Dwyer, 2016**), whereas AWS Lambda is limited to Node.js, Java, and Python, IronWorker provides Clojure, Go, .NET, PHP, Python, Ruby, and binary code.

Google Cloud Function is a simplified approach for developers to create individual purpose, standalone operations that react to cloud functions without the need to manage a server or runtime environment.

Going serverless requires a slightly discrete action to software design and architecture. The backend service is broken down into small pieces of standalone functionality that execute a single task in react to a user action or event. In serverless architecture, the backend is composed of thin, single-purpose microservices that are event-driven, and the business logic moved from the backend to the client. It becomes the central orchestrator, calling various functions to perform some action for the user when needed.

Serverless architecture requires astute clients that know about and talk to a wide range of remote functions. While mobile app developers have had productive frameworks and platforms that allowed them to build complex logic on the client quickly, things weren‘t so simple for web applications. But thanks to productive client-side application frameworks like Angular 2 and a fast HTTP/2 protocol, it is now possible to build sophisticated applications seamlessly into the browser. This will help drive the serverless trend even further.

3.15 Deployment Models

In the previous section, the essential service models were discussed. These services can be deployed in various ways.

Private Cloud

The Cloud-based solution is provisioned and used by a single enterprise. Private Clouds inherit the characteristics of Cloud Computing (e.g., elastic service provisioning, virtualization, etcetera) and provide more benefits to the enterprise (**Armbrust et al., 2009**).

Community Cloud

The Community Cloud is similar to the private Cloud, but the community Cloud is owned and shared among a group of organizations. These organizations must share the same concerns, such as policy, mission, compliance considerations, and security requirements (**Mell & Grance, 2011**).

Public Cloud

The public Cloud is when the provisioning of Cloud-based solutions is publicly available for open use. The services are ubiquitous and accessible through an Internet connection, but this deployment model poses many security and privacy concerns (**Armbrust et al. 2009, Mell & Grance 2011**)

Hybrid Cloud

Hybrid Cloud blends at least two distinct deployment models (e.g., public, private, or community Cloud) which are tailored to provide data and application portability. In that way, some of the resources are residing on-premise, while others are outsourced (**Mahmood & Hill 2011**).

3.16 Virtualization Technologies

The virtualization techniques of interest in our study can be grouped into two categories: Full virtualization and Operating System-level (OS-level) virtualization. The foundation of Full virtualization is computer hardware mimicry. A host machine delivered mimics the hardware environments for its guests to run their separate OS like they are run as the independent machine. The worlds most virtualization solution provides are VMware and Kernel Virtual Machine (KVM). Conversely, in the container-based virtualization system, it shared the host operating system kernel to the entire guest system or containers. The host hardware isolates all the dependencies guests into different virtual machines, which mimic a new dedicated running environment for each guest and prevent them from accessing unrelated resources (**Tam Le Nhan, 2013**).

Full Virtualization Approach

Full virtualization aspects, also known as the original virtualization technology, refers to that whole virtual machine that replicates the entire underlying hardware, including processors, storage, memory, peripherals, etc. It is not compulsory to make any modification to run operating systems or other system software in a virtual machine. The Docker layered architecture diagram is shown on the left in figure 7.

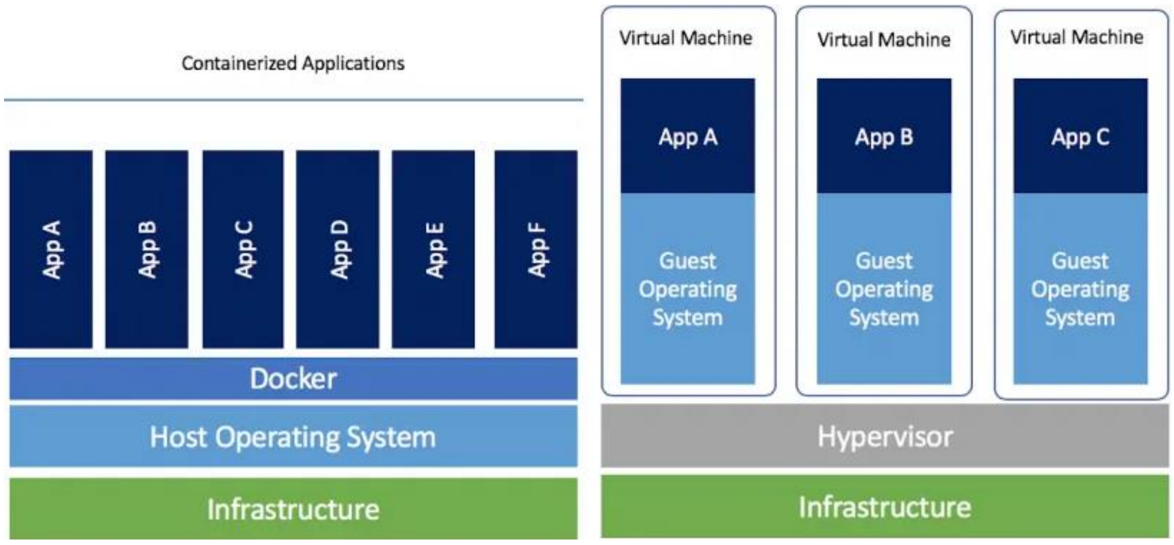


Figure 7: Comparison of OS-Level Virtualization and Full Virtualization (Source: Docker Inc.)

Linux Based Containers

LXC (Linux Container is a standard virtualization solution for Linux. LXC provides several kernel features to achieve the virtualization goal, such as kernel namespaces and control groups. Kernel namespaces facilitate a group of processes to have their own namespace and thereby isolating these processes from any means, not in the same namespace.

Choice of Virtualization Platform

Full virtualization approach offers the best isolation and resource protection mechanisms. OS-level virtualization provides the best performance and service density at the expense of isolation.

OS-Level virtualization was chosen due to its low overhead, small footprint, and resulting in higher container density. This is an essential consideration for Web services that are typically built for commodity hardware. Among different OS-Level virtualization applications, we choose Linux based Container called Docker.

Design Through Abstraction

A system may be orchestrate of many levels of notion and many phases of operation, each with its own software architecture. A web-based system consists of servers, databases, clients, load balancers, and gateways.

Software architecture represents an abstraction of system behavior at that level, such that architectural elements are delineated by the abstract interfaces they provide to other parts at that level (**L. Bass et al., 1998**). Within each component may be found another architecture, defining the system of sub-elements that implement the behavior represented by the parent element‘s abstract interface. This recursion of architecture continues down to the most basic system elements: those that cannot be decomposed into less obscure parts. The concept of containerization is a means of abstracting system behavior so that that a container may hide the inner components from a developer who intends to use this container in system design. A virtual machine is a container that can house several other containers that are at different levels of abstraction. All the details of how the container provides the required functionality and runtime behavior are hidden from the outside world. The container only exposes a uniform interface so that it can interact with other systems at that level. This is the principle on which Docker is based.

Mechanical systems follow similar abstractions. A car only exposes a uniform interface to the driver, but within a vehicle, there are several levels of system abstractions. Within the car, we have other systems such as the engine, the electronic control unit, and the Transmission system whose inner workings the driver requires not to know to drive the car.

Software Architecture Abstraction Through Containerization

Software architecture abstraction has been made possible by advances in virtualization. Running any software on any hardware platform was made possible by the introduction of Virtual machines. A virtual machine runs on top of a Virtual Machine Monitor (VMM) or

Hypervisor. The hypervisor acts as an interface between the virtual machine and the underlying Kernel or Hardware. The cloud computing paradigm is based on the concept of virtualization. Containerization though not a new idea is a subject of much debate in the last two years. This was after Docker Inc. popularized Containers in 2013. Within two years, Docker containers, which are based on Linux Containers (LXC) is promising to change the course of virtualization and the whole cloud computing ecosystem.

Docker containers go further, adding layers of abstraction and deployment management features. Among the benefits of this new infrastructure, technology is that containers that have these capabilities reduce coding, deployment time, and OS licensing costs. The VM model blends an application, a full guest OS, and disk emulation. In contrast, the container model uses just the application‘s dependencies and runs them directly on a host OS. Containers do not launch a separate OS for each application but share the host kernel while maintaining the isolation of resources and processes where required. A Docker application container takes the basic notion of LXCs, adds simplified ways of interacting with the underlying kernel, and makes the whole portable (or interoperable).

3.17 Docker Architecture

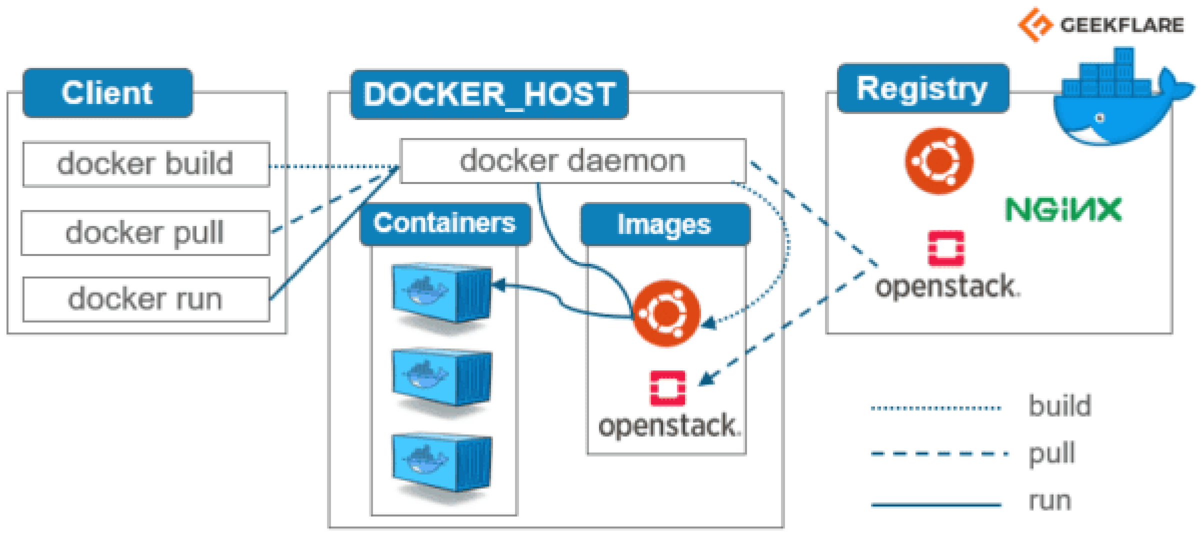


Figure 8: Docker Architecture (Source: geekflare)

Docker API

The Docker daemon has a remote API, and this is what the Docker command-line tool uses to communicate with the Docker engine. Since Docker API is well documented in the public repository, so it's very convenient to use an external means to access the API. This provides all manners of tooling, from mapping deployed Docker containers to servers, to automated deployments, to distributed schedulers. As you embrace Docker over time, likely, you will increasingly find the API to be a good integration point for this tooling. This API has made it possible to integrate Docker to other technologies such as Software Defined Networking and distributed Filing and storage.

Docker Client

The Docker client supports 64-bit versions of Linux kernel, Mac OS X, and Windows due to the Unix underpinnings of these operating systems. To develop an application or environment to the Docker on non-Linux platforms, It's necessary to leverage VMs or remote established Linux hosts to provide a Docker server. However, this state of affairs is changing with the introduction of docker for windows and OS.

Docker Engine

Docker engine release 18.09 + consists of the following features

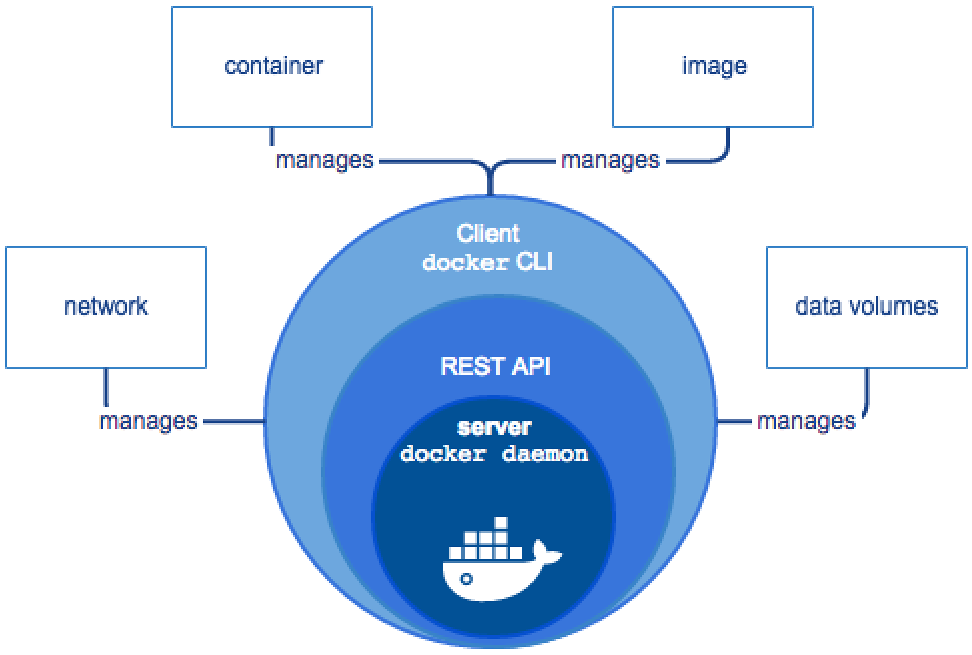


Figure 9: Docker Engine Architecture (Source: Docker Inc.)

- **Orchestration:** comes with inbuild orchestration capability using Swarm. The inbuild Swarm is simple to use and enhances container security through the use of TLS.
- **DNS round-robin load balancing**: It‘s now possible to load balance between containers with Docker‘s networking. If you give multiple containers the same alias, Docker‘s service discovery will return the addresses of all of the containers for round-robin DNS.
- **VLAN support**: VLAN support has been added for Docker networks so that you can integrate better with the existing networking infrastructure.
- **IPv6 service discovery**: Engine‘s DNS-based service discovery system can now return AAAA records.
- **Yubikey hardware image signing**: this is the ability to sign images with hardware Yubikeys.
- **Labels on networks and volumes**: You can now attach arbitrary key/value data to networks and sizes, in the same way, you could with containers and images.

- **Better handling of low disk space with device-mapper storage**:

The **dm.min_free_space** option has been added to make device-mapper fail more gracefully when running out of disk space.

- **Consistent status field in docker inspects**: This is a little thing, but handy if you use the Docker API. Docker examination now has a Status field, a single consistent value to define a container‘s state (running, stopped, restarting, etc.).

Containers

The container is a runtime instance for an image. Docker API or CLI enables to user/developer to create, move, start, stop, or delete functionality. A Container can connect one or more networks and possible to attach storage to it. A container by default well isolated from another container and its underlying host machine. Docker provides isolated functionality that enables the container's network, storage, and other subsystems isolation one from another.

Example `docker run` **command**

```
$ docker run -i -t centos /bin/bash
```

Service

Service provides the container scaling functionality across the multiple Docker daemons. It's working together as a swarm along with numerous managers and workers. All the members of the swarm is a daemon of docker, and communicating among them by using Docker API. A Service also maintains the number of replicas available at any given time.

By default, a Service doing load-balancing between all the worker nodes.

Benefits of Containerization.

- **Containers are Lightweight**

Not only is Docker quicker than a traditional VM to spin up, but it‘s also more lightweight to move around because it's shared the host OS kernel.

- **Fast application deployment** – containers include the minimal runtime requirements, libraries, and dependencies of the application, allowing them to be deployed quickly.
- **Portability across machines** – A container can be transferred to another machine that runs Docker and executes in a different machine without having any compatibility constraints.
- **Edition control and element reuse** – you can shadow following versions of a container, inspect varieties, or fall-back to the previous versions.
- **Sharing** – Docker using docker hub as a container registry or a remote repository to share your container with others.
- **Lightweight trail and insignificant burden** – Containerized images are typically minimal, which facilitates rapid delivery and reduces the time to deploy a new application.
- **Simplified maintenance** – Containerization reduces effort and risk of problems with application dependencies.

3.18 Docker Containers and the Cloud Ecosystem

- **Images –** A Docker image is made up of filesystems layered over one another. At the center is a boot filesystem, bootfs, which resembles the typical Linux boot filesystem. An image contains the whole filesystem that will be available to the application, and other metadata, such as the path to the executable that should be executed when the image is run
- **Registries** – A Docker Registry is a service that stores your Docker images and

  facilitates the secure sharing of those images between developers and machines. When you build your image, you can run within the same machine, or you can push (upload) the image to a docker hub registry and then pull (download) it on another computer and run it there. Some records are public, allowing anyone to pull images from it, while others are private, only accessible to specific people or machines.

- **Containers** –A running Docker Image is called a container and is a process running on the machine that has Docker daemon. A container is resource-constrained, meaning it can only access and use up the number of resources (CPU, RAM, etc.) that are allocated to it.

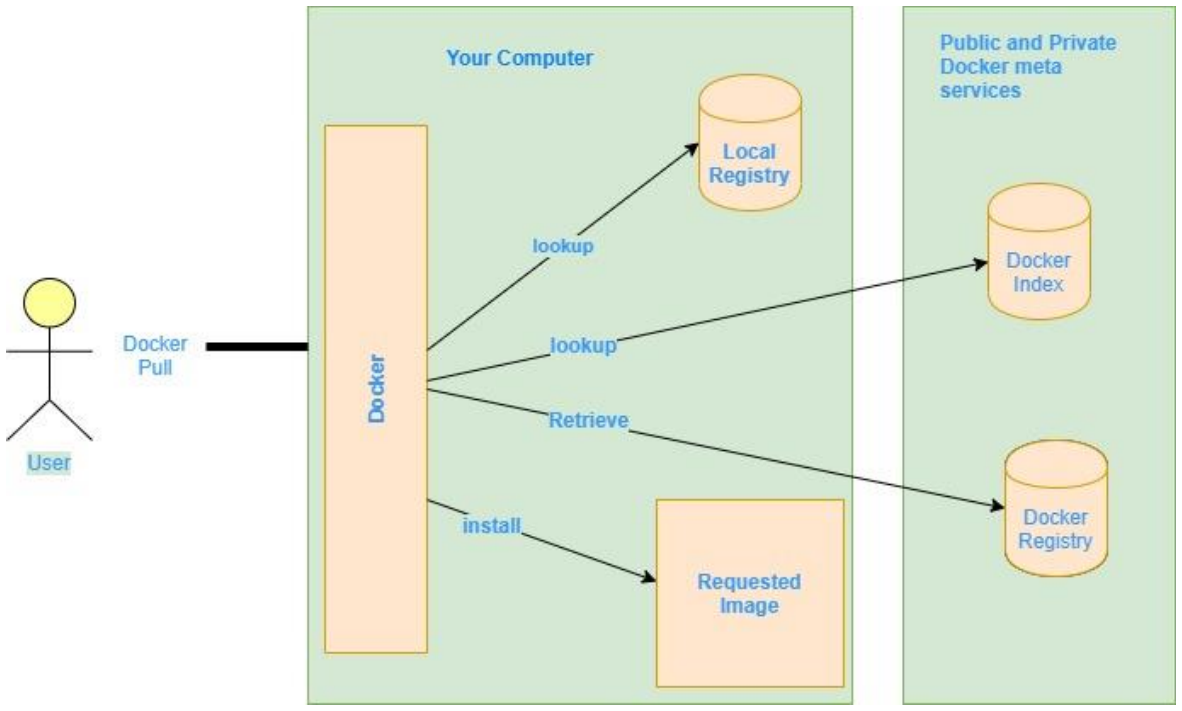


Figure 10: Docker Public or Private Registries

Docker Machine

Docker Machine enables the user to store Docker machines in diverse environments, alongside virtual machines that stay on the local system, cloud providers, or any physical computers. Docker Machine builds a Docker host, use the Docker Engine client as required to develop images and formulate containers on the host. Docker can be found 32 and 64-bit versions of Linux, Windows, and Mac OS X.

Open Container Ecosystem

There is a massive community aligning to use Docker, mainly developers and system administrators. Like the DevOps evolution, this has expedited more salutary tools by employing code to compelling problems. Wherever are rifts in the tooling given by Docker, other companies and individuals have stepped up to offer viable and open source solutions. That means they hosted on public repositories and can be modified by any others to fit their needs.

The figure below illustrates how the container architecture is layered. Note the layering of different services to simplify container management. As Docker enters its maturity, it will interact will the various layers of technologies.

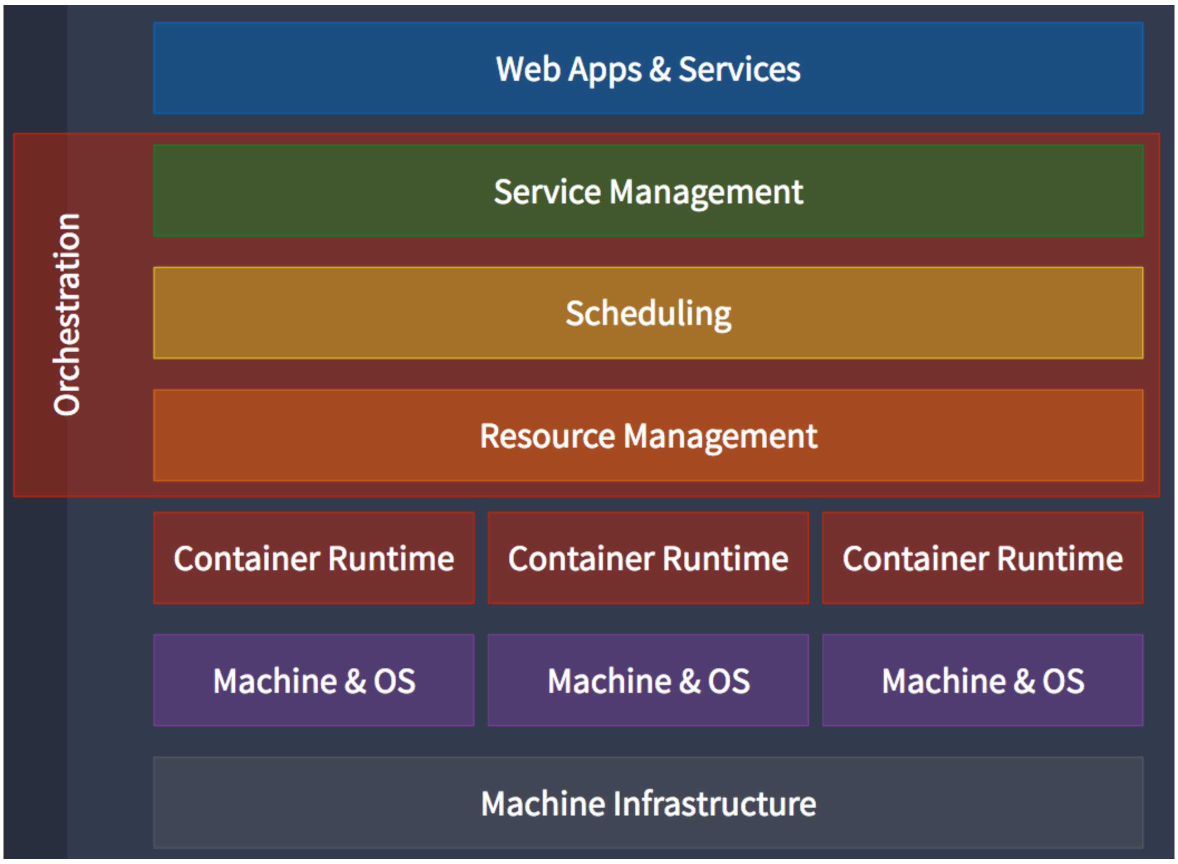


Figure 11: Open Container Layered Architecture [Docker Inc.]

2.2.4 Docker Extensions

For Docker to be useful in supporting the distributed application, it has to possess the following capabilities.

- **Portable beyond the environments:** Docker can operate any Linux based host machine where you can define how the application will run in a different context like development, testing, staging, and production seamlessly.
- **Portable beyond providers:** Docker enables us to move your application between different cloud providers and your own servers, or run it across several providers.

- **Composable:** You want to be able to split up your application into multiple services.

Scaling up microservices with Docker compose

Docker-compose is a tool for defining and running complex applications with Docker. With Compose, it is used yml file where multi-container application instructions in a single file, with a single the application, will be up and running, which does everything that needs to be done to get it running. Using Compose is a three-step process. Determine your app‘s environment with a Dockerfile so it can be duplicated anywhere ok system. [**Docker Inc.**]

1. Determine your app‘s environment with a Dockerfile so it can be replicated anywhere.
2. Determine the settings that make up your app in docker-compose.yml so they can be run together in an isolated environment:
3. Lastly, run docker-compose up, and will start and run your entire app.

The primary function of Docker Compose is the creation of Microservice Architecture, meaning the containers and the links between them. But the tool is capable of much more.

Scaling up microservices with Docker Swarm

Docker Swarm unlocks one of the key limitations of Docker, where the containers could only run on a single Docker host. Docker Swarm is a native clustering mechanism for Docker. It spins a pool of Docker hosts into a single, virtual host.

Swarm terminology

This section introduces some of the concepts unique to the cluster management and orchestration features of Docker Engine 1.12.

SwarmKit

The cluster management and orchestration characteristic embedded in the Docker Engine and those are built by the SwarmKit. Docker Engines engage in a cluster are running in Swarm mode. You enable Swarm mode for the Engine by either initializing a swarm or joining an existing Swarm.

A Swarm is a bunch of Docker Engines where you deploy services. The Docker Engine CLI includes the commands for Swarm management, such as adding and removing nodes. The CLI also contains the controls you need to deploy services to the Swarm and manage service orchestration.

When you run Docker Engine outside of Swarm mode, you execute container commands. When enabling docker-engine in Swarm mode, you orchestrate services.

- **Node**

A worker node is an instance of the Docker Engine engaging in the Swarm cluster. Which sole purpose is to execute the containers in the docker. It is not possible to create swarm worker nodes without having at least one manager node. By default characteristics, all managers act as also workers node.

Worker nodes receive the request from the manager nodes and execute tasks according to that. The agent notifies the manager node of the current state of its assigned functions so the manager can maintain the desired state.

- **Services and Tasks**

A Service is the definition of the group of tasks to execute on the worker nodes by instruction from manager nodes. It is the interior architecture of the Swarm system, and only the root user can interact with the swarm system. When you create a job, it is necessary to specify which image from the container to use and commands to execute internally ongoing operational containers.

In the distributed services model, the Swarm manager replicates a specific number of replica tasks within the nodes based on to scale you set in the desired state. For global services, the Swarm runs one job for the tasks on every available node in a cluster.

- **Load Balancing**

The Swarm manager uses inbound load balancing to expose the services.

It enables swarm available externally. The Swarm manager can automatically allot the service PublishedPort, or you can configure a PublishedPort for the service in the 30000-32767 range.

External components, such as load balancers in the cloud able to access the service on the PublishedPort from any of the nodes in the cluster. All nodes in the Swarm cluster route inbound connections to a running task instance.

Swarm mode has an internal DNS component that automatically assigns each service in the Swarm a DNS entry. The Swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service

Swarm Orchestration Architecture

A Swarm orchestration is a not a centralized and highly available group of Docker nodes. Each node is an independent sub-system that has all the built-in capabilities, and it can create a pool of shared resources as per schedule Dockerized tasks.

A Swarm of Docker nodes can produce a programmable network topology. The operator can choose which nodes are performing as managers and which are as workers. This includes standard configurations like sharing managers node across multiple availability zones. Because these roles are compelling, they can be altered at any time through the API or CLI calls.

Managers are in charge of coordinating the cluster. It provides the service API, scheduling jobs (containers), undertakes containers that have failed health checks, and much more. On the other hand, worker nodes providing a much-unsophisticated function, that executing the tasks to reproduce containers and routing data traffic expected for specific boxes. In production environments, it is recommended to having nodes nominated as either managers or workers nodes. In this method, managers do not run containers, thus minimizing their workload and risky surface. In contrast, one of Swarm mode‘s security advances is that worker nodes do not have access to information in the data store or the API.

The raft is used to share data between managers for durable consistency (at the cost of write speed and limited volume). In contrast, gossip is used between workers for fast communication and high volume and transmission between managers and workers node has separate requirements still.

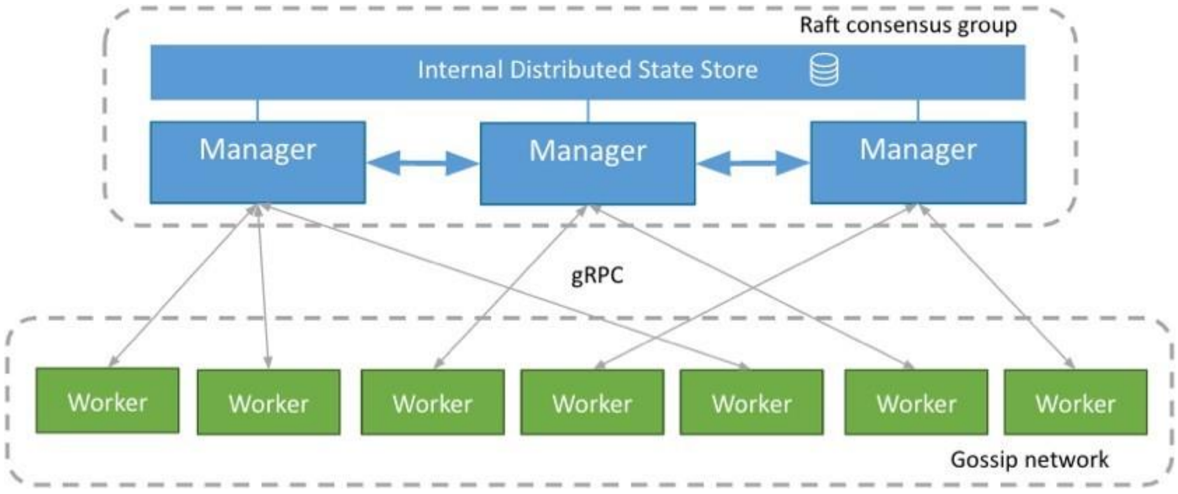


Figure 12: Swarm Orchestration Architecture [Docker Inc]

- **Raft Consensus algorithm**

According to (**Diego Ongaro et al., 2014**), Raft is an understandable and straightforward cluster consensus protocol. At any proposed time, the particular server is in one of three states: leader, follower, or candidate. Under regular operation, there is precisely one leader, and all of the other servers are followers. Followers are passive: they issue no requests on their own but simply respond to applications from leaders and candidates. The leader deals with all client requests, and if a client contacts a follower, the follower redirects it to the leader.

When a nodeact as a manager node, it joins a Raft concurrence group to yield information and execute the leader election. The centaral authority called leader, which making the scheduling decision across the swarm which including list of nodes, services and tasks. . That operation is distributed across each manager node through a built-in Raft store. [**Docker Inc.**]

The ultimate optimization is in how efficiently the data is ensured both in terms of size (protocol buffers) and performance (domain-specific indexing ok system). It is possible to run through instantly query from memory the containers that are running on a particular machine, the containers that are unhealthy for a specific service, etc.

- **Manager-Worker Communication**

Worker nodes communicate with manager nodes using gRPC. This fast protocol works exceptionally well in noisy networking conditions. Workers nodes continuously sending out their given task status and heartbeat state, so the managers can verify the worker node is still alive.

The following diagram illustrates the dispatcher parts of the manager node, which eventually communicates with workers nodes. It is liable for dispatching tasks to each worker, while the worker (though an executor parts) is in charge of translating those tasks into containers and creating them.

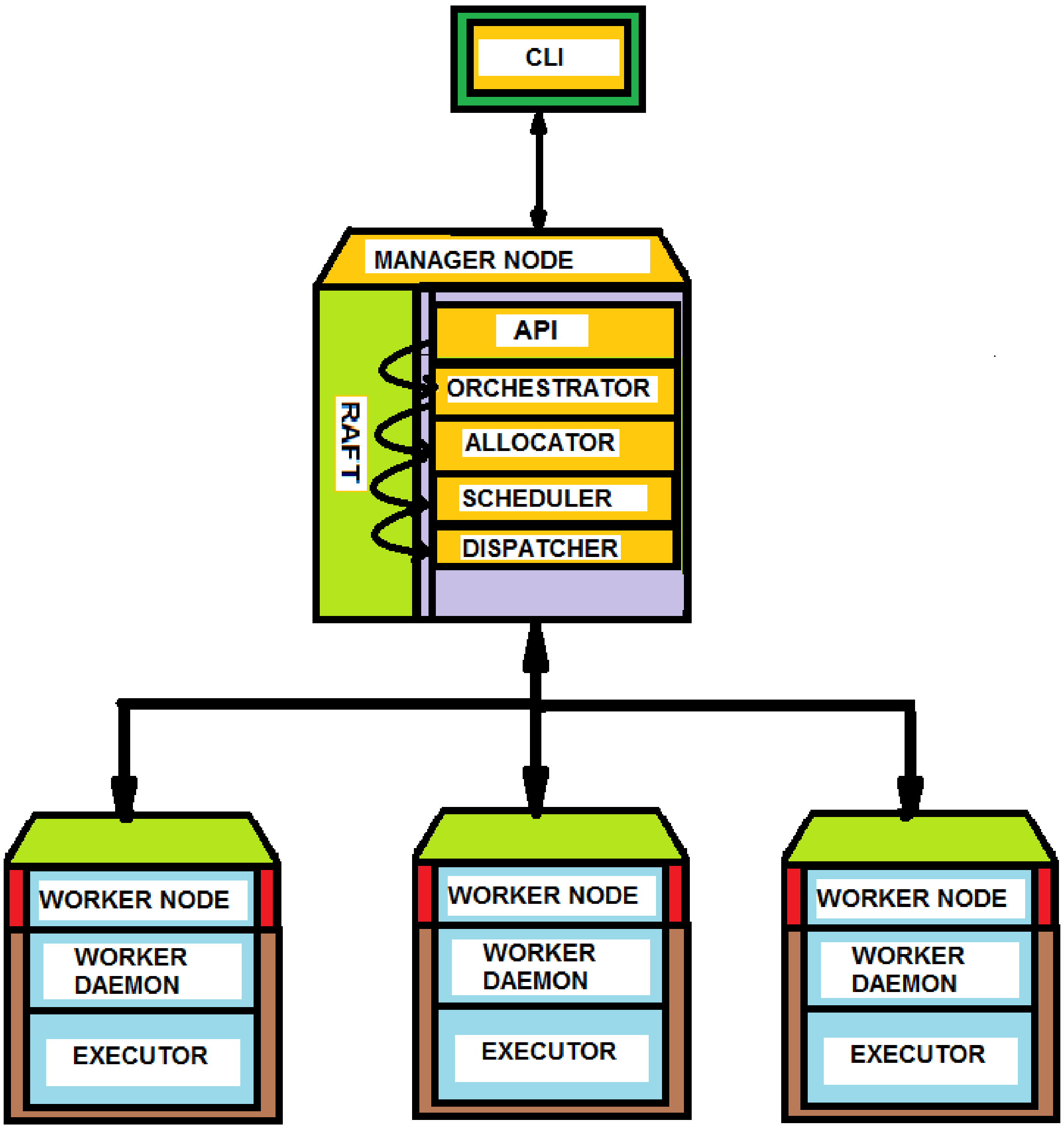


Figure 13: The elements of Docker Swarm cluster based on Raft Consensus Algorithm

Service initiation

The user sends the service definition to the API. The API accepts and stores the service state before forwarding the request to the Orchestrator.

- **Orchestrator** accommodates expected state(as defined by the system user) with the existent state (which is being operated on the Swarm). It will catch the latest service created by the API and react to that by the start of a task (assuming in this case, the user requested only one instance of the service)

- **Allocator** allocates resources for jobs. It will notice a brand new facility (generated by API) and the latest task (bring up by the orchestrator) and will allocate IP addresses for both.
- **Scheduler** is responsible for assigning tasks to worker nodes. It will inform a task to the unused node. For that reason, it starts scheduling. The scheduler tries to find the best match based on some criteria su.ch as constraints, resources. Finally, it will assign the task to one of the nodes
- **Dispatcher** is where workers connect to. Once workers are connected to the Dispatcher, they wait for instructions. In this way, a task assigned by the scheduler will eventually flow down to the worker.

Allocator, Scheduler, and Dispatcher will perform the same steps as explained above, and the two new tasks will land on workers.

3.19 Scaling up Microservices Kubernetes

Kubernetes is an open-source project to manage a cluster of Linux containers (Docker and **rkt**) as a single system, managing and running containers across multiple hosts, offering co-location of containers, service discovery and replication control. Google started it and now it is used by several software vendors. Kubernetes 1.4 has added more capabilities such as

- The Kubernetes allows users to set up services that span multiple clusters that can even be hosted across multiple clouds.
- Support stateful applications like databases. The project now also features improved autoscaling support.
- Support for **rkt** as an alternative container runtime to Docker‘s runtime.

support for twice as many nodes in a cluster as before (up to 2,000) and services can now span different availability zones

The design of Kubernetes is a combination of microservices and small control loops, and this achieves the desired emergent behavior by combining the effects of separate, autonomous entities that collaborate. This is an improved design selection in divergence to a consolidate orchestration system, which may be simpler to build at first but tends to become brittle and rigid over time, especially in the presence of unanticipated errors or state changes.

A Kubernetes cluster is composed of two parts:

- **The Kubernetes Control Plane -** is highly scalable microservice-based and loosely coupled components which controls and manages the whole Kubernetes system.
- **Worker nodes -** Containerized hosts who run the actual applications you deploy in the Kubernetes cluster. The elements of the control plane are (**Marko Lukŝa, 2016**): The API Server. Which you use to communicate with and perform operations on the Kubernetes cluster the Scheduler, which is responsible for scheduling your apps (assigning a worker node to each deployable component of your application), the Replication Controller, which performs cluster-level functions, such as replicating components, keeping track of worker nodes, etc.
- **etcd**, a reliable distributed store that stores the whole cluster configuration persistently. The worker nodes, on the other hand, run. Docker, which runs your containers.
- **Kubelet**: which talks to the master node and controls Docker on that node.
- **Kube Proxy**, which proxies and load balance network traffic between your application components.

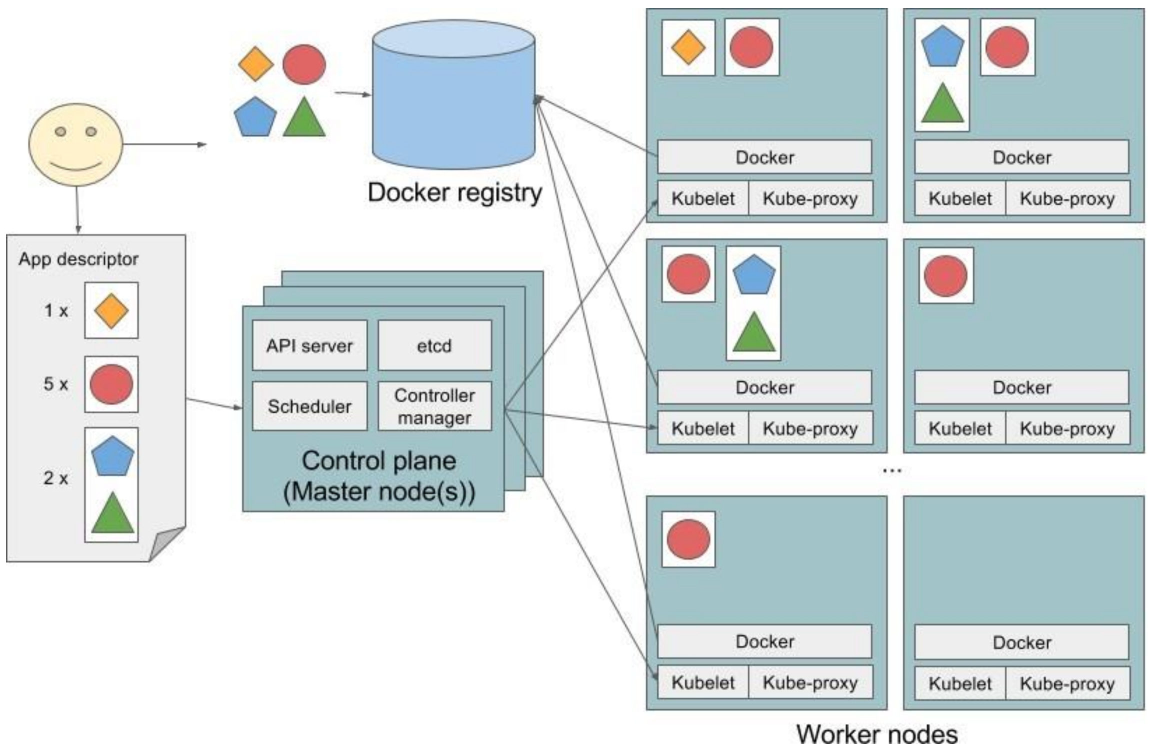


Figure 14: Kubernetes Master-slave design illustrating its microservices and container-based architecture (Marko Lukŝa, 2016)

Thanks to the development of Linux namespaces, VMs, IPv6, and software-defined networking. Kubernetes can take a more user-friendly approach that eliminates these complications: every service gets its own IP address, allowing developers to choose ports rather than requiring their software to adapt to the ones chosen by the infrastructure, and removes the infrastructure complexity of managing ports.

Kubernetes derivatives

Asian telecommunications giant Huawei Technologies has released its own container orchestration engine, the Cloud Container Engine (CCE). CCE is based on Kubernetes. CoreOS launched a project called Stackanetes, which was designed to run OpenStack as an application on your infrastructure, just like any other application. In effect, Stackanetes uses the Kubernetes orchestration engine to manage a distributed OpenStack deployment.

Learning from Google‘s over ten years of experience of running every application inside a container, Mirantis has decided to make OpenStack more scalable and manageable by running it using Kubernetes. By packaging OpenStack services so Kubernetes can manage them, Mirantis is addressing many of OpenStack‘s scaling, management, and operational challenges, making it, in theory, as scalable as any microservice.

3.20 Scaling up microservices with Apache Mesos

Mesos architecture is quite different from Kubernetes. The main difference in the design is that Mesos employs two-level scheduler architecture. The actual scheduling tasks delegate to frameworks. The master can be a very scalable light-weight piece of code. It enables rapid growth in the number of frameworks that Mesos supports. In Mesos, there is no need to add brand new code to the master and slave modules every time a new framework during the iteration. Instead, the developer can focus on their application and framework without worrying about the underlayer system.

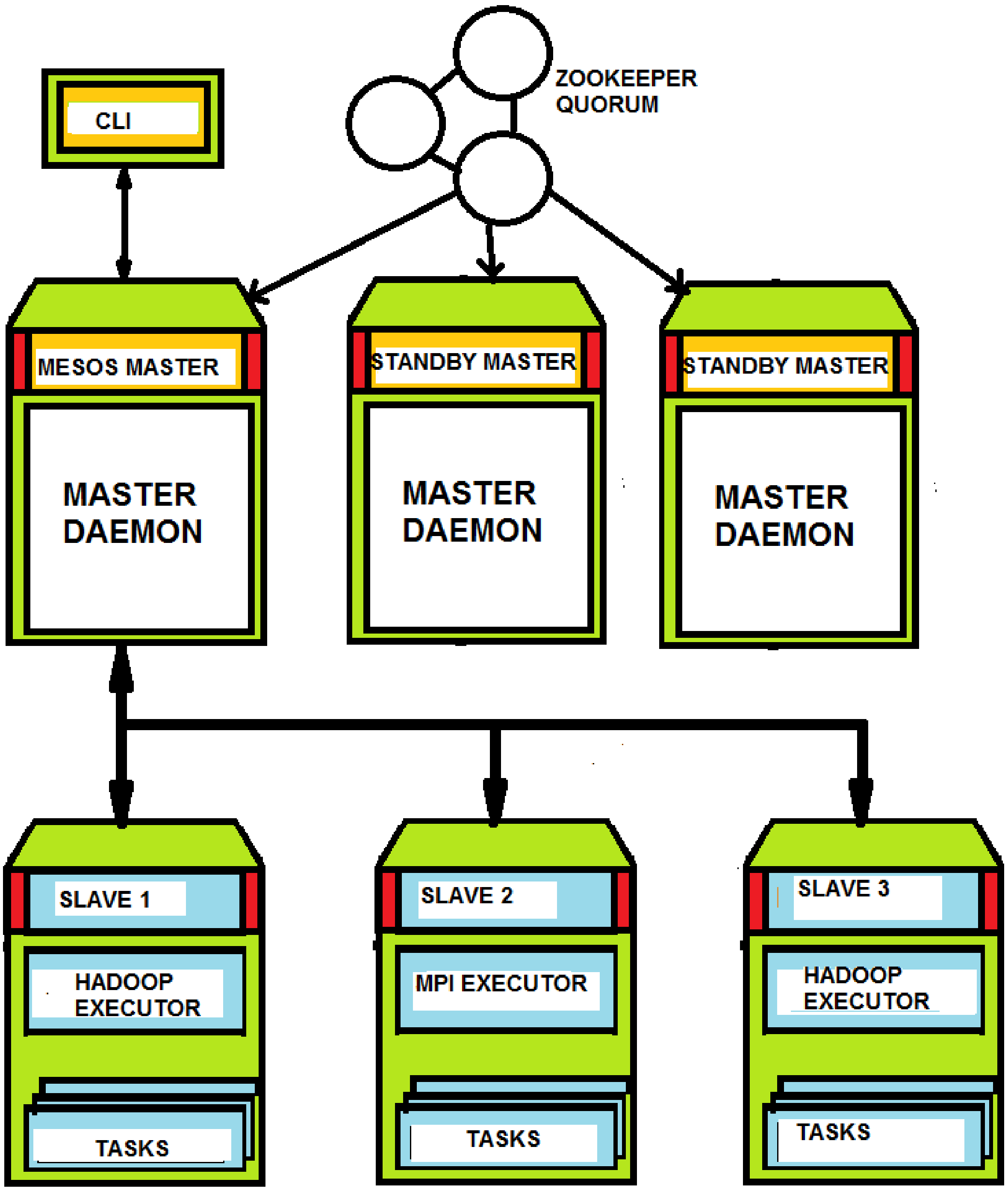


Figure 15: Mesos Two Level Orchestration Architecture

3.21 Containerized Application Management

According (**Alex Williams, 2016**), it takes roughly 16 man-hours per year per VM for patching, updating the OS, antivirus, etc. And this is for infrastructures that are reasonably automated. If they‘re not automated, it probably takes longer. Multiply that by the number of VMs in your environment, that‘s a lot of OPEX. Kubernetes, on the other hand, is a great system, but it requires a lot of manpower to install and maintain. It requires a lot of resources. When you look at reasons why it‘s not taking off faster, it‘s that there‘s a learning curve.

The cluster management tools that are dominating the container orchestration scene include Docker Swarm, Kubernetes, and Mesos. All these tools come with different use cases, and it is difficult to rely on one and omit others. Under the production environment, it may be possible to have containerized management tools that fully automates the cluster management tasks using various orchestration tools. In the following subsections, we look at numerous such attempts.

Kontena

Kontena is a free, open-source system for deploying, scaling, managing, and monitoring containerized applications beyond multiple instances on any type of cloud infrastructure. Initially, it's targeted for running applications composed of various containers, such as elastic, distributed micro-services.

Both Docker Swarm and Kubernetes influence the architecture of Kontena, so it had some opportunities to learn from those projects‘mistakes and successes. It's working like Kubernetes at a level of abstraction but higher than containers. The first building components of Kontena are called services. The other main parts of Kontena‘s architecture are the grid, services, the master node, host nodes, and the Kontena CLI.

The Master Node

Similar to Kubernetes, Kontena works on master-slave architecture. The Master node doesn't like other orchestration solutions. It doesn't provide any additional underlying processing power. The primary role is to provide audit logging and management of containers purely.. Kontena‘s other component is called host nodes — are what offer the processing power and run the physical Docker containers.

Kontena CLI

If you have played with docker-compose, it can find Kontena very inherent. When it is not a one-one match, Kontena‘s kontena.yml files are very close to the docker-compose.yml file. It is possible to have a base docker-compose.yml file that can be extended and reference using a kontena.yml file. Kontena‘s strengths lie in

· **Easy installation**: Kontena works on any public cloud, on-premises, or hybrid and requires minimal effort to install. The platform doesn't require maintenance and includes automatic updates. Instead, developers spend more time working on their own projects or applications.

· **Scalability**: Compare to the other platforms, Kontena is suitable for running even the smallest container, and it may be scaled up when it required. This scalability means developers have a tool to define the right size for their unique organizational needs.

· **Open source**: Kontena is a free, open-source, and it can integrate with other orchestration, as well as leading software-as-a-service offerings aimed at monitoring and logging. This minimizes vendor lock-in and ensures developers have a wide array of options available to them.

Kontena version 0.15 supports the following features.

· **CLI plugins** - CLI functionality can be extended with plugins. All provisioning features are now available as a separate plugin.

· **Health checks -** It's now possible to configure health checks for each service.

· **Let's Encrypt support** - Support for issuing Let's Encrypt certificates using the DNS challenge.

· **Load Balancer Sticky Sessions -** Now, it's possible to configure the load balancer to use sticky sessions.

Kontena able to proposes a more complete and automated container that includes more functionality such as orchestration, scheduling, network overlay, load balancing, and secrets management.

3.22 Docker Plugin Architecture

During the Docker Conference held on 22nd June, 2015 Docker announced a plugin architecture. The Docker ecosystem of tool-makers is growing exponentially. The value of plugins is to integrate this ecosystem seamlessly with the Docker Engine. Customization leads to applications that fit the end-users‘ needs better. This extensibility must retain Docker‘s portability, consistency, and ease of use. That is the idea behind Docker plugins: one set of interchangeable tools via one Docker open platform. A user can swap out a plugin and replace with another without having to modify their application. You can swap in different volumes, networking, composition, or scheduling framework, depending on user preferences and the individual requirements of each user‘s requests.

- **Volume Plugins**: It allows third-party container data management solutions to provide data volumes for containers that operate on data, such as databases, queues, and key-value stores and other stateful applications that use the file system.
- **Network Plugins**: This allows third-party container networking solutions to connect containers to container networks, making it easier for vessels to talk to each other even if they are running on different machines.

In both cases, the plugin mechanism takes a piece of core functionality that Docker already provides and allows users and tool-makers to load, or write, plugins that extend that feature is crisp, new, and exciting ways.

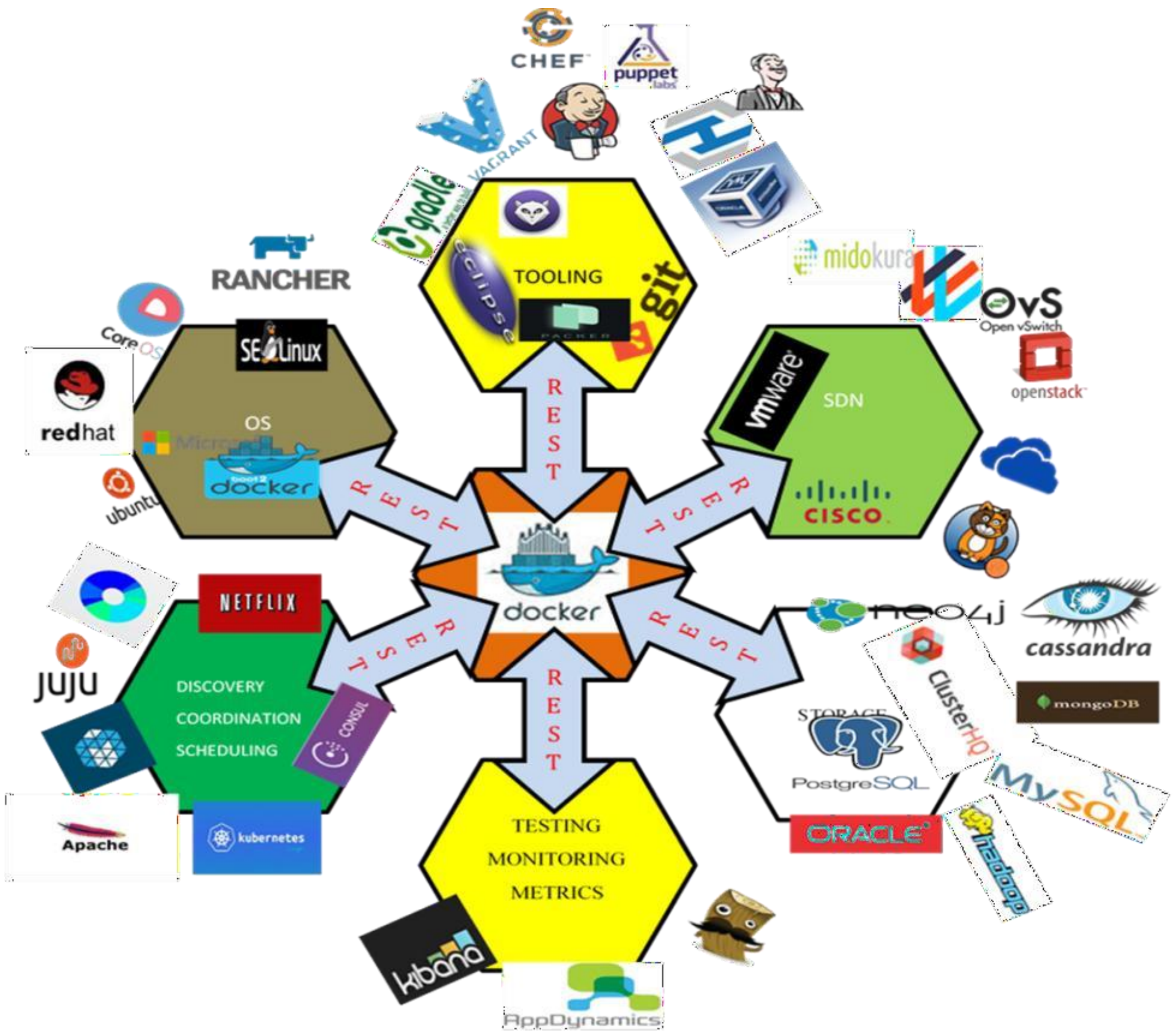


Figure 16: Docker plugin Architecture showing the extensions and interfaces to other systems

3.22.1 Volume Plugins

Starting with version 1.8, Docker introduced support for third-party volume plugins. Existing tools, including Docker command-line interface (CLI), Compose and Swarm, work seamlessly with plugins. Kubernetes 1.3+ also has excellent support for volume plugins (databases).

According to Docker, volume plugins enable engine deployments to be integrated with external storage systems and data volumes to persist beyond the lifetime of a single-engine host. Customers can start with the default local driver that ships along with Docker, and move to a third-party plugin to meet specific user storage requirements. Further volume plugin enables containerized applications to interface with filesystems, block storage, object storage, software-defined storage.

Currently, Docker supports more than a dozen third-party volume plugins for use with Azure File Storage, Google Compute Engine persistent disks, NetApp Storage, and vSphere.

Basics of Volume Plugin Architecture

Docker ships with a default driver that supports local, host-based volumes. When additional plugin is available, the same workflow can be extended to support new backends.

The third-party volume plugins are installed separately, which typically ship with their own command-line tools to manage the lifecycle of storage volumes. Docker‘s volume plugins can support multiple backend drivers that interface with popular filesystems, block storage devices, object storage services, and distributed filesystems storage.

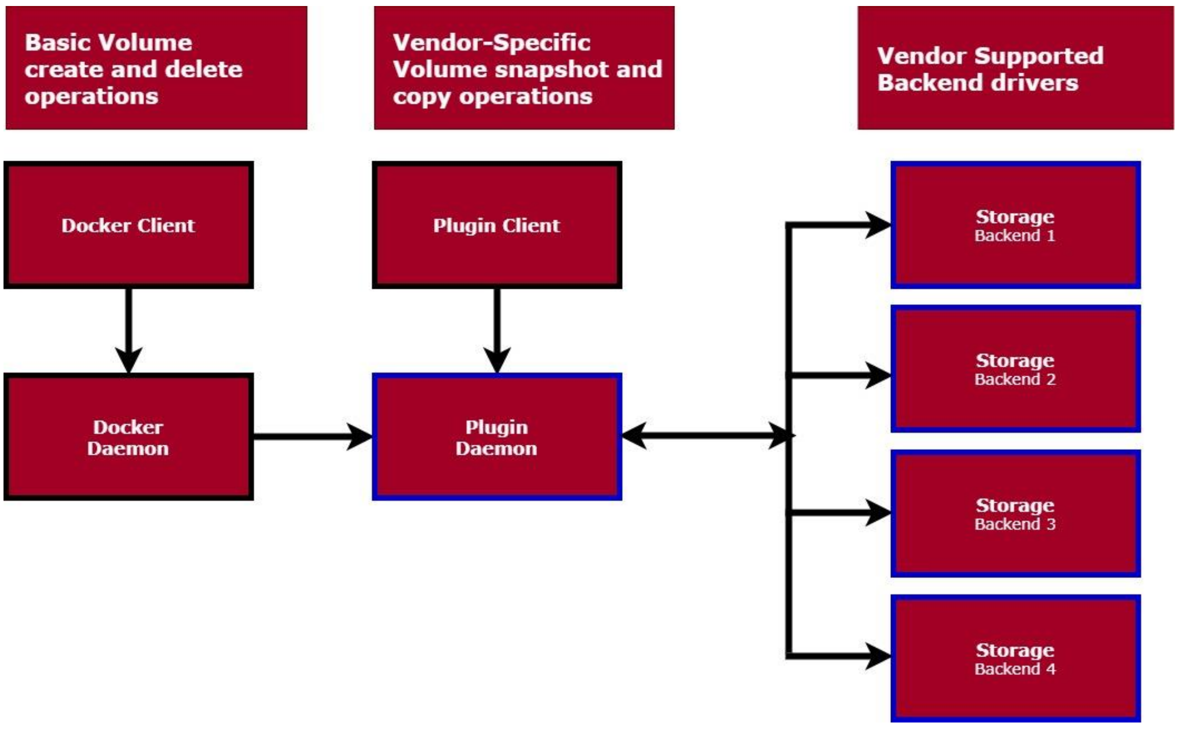

Basic Volume create and delete operations
Vendor-Specific Volume snapshot and copy operations
Vendor Supported Backend drivers
Docker Client
Plugin Client
Docker Daemon
Plugin Daemon
Storage Backend 1
Storage Backend 2
Storage Backend 3
Storage Backend 4

Figure 17: The Docker Volumes Plugin Architecture

Flocker works with mainstream orchestration engines such as Docker Swarm, Kubernetes, and Mesos. It supports storage environments ranging from Amazon Elastic Block Store (EBS), GCE persistent disk, OpenStack Cinder, vSAN, vSphere, and more.

3.22.2 Network Plugins

Container networking technology is an excellent enabler for scalable microservices. Container networking types of concern to us include:

- Overlay
- Underlay

2.2.5 Types of Container Networking

Container networking types can be categorized based on IP-per-container versus IP-per-pod models and the requirement of network address translation (NAT) versus no interpretation needed.

Overlay

Overlays employ tunnels to deliver communication across and between hosts. Many tunneling technologies exist, such as virtual extensible local area network (VXLAN). VXLAN has been the tunneling technology of choice for Docker libnetwork, whose multi-host networking entered as a native capability in the 1.9 Docker engine release. Multi-host networking requires additional parameters when launching the Docker daemon, as well as a key-value store. Some overlays rely on a distributed key-value store. If you‘re doing container orchestration, you‘ll already have distributed key-value store lying around. Docker Swarm has inbuilt support for overlay networking.

Underlays

There are two types of underlay networking based, namely media access control virtual local area network (MACvlan) and internet protocol VLAN (IPvlan). Both network drivers are conceptually more straightforward and eliminate the need for port mapping and are more efficient.

MACvlan

MACvlan allows the creation of multiple virtual network interfaces behind the host‘s single physical interface. Each virtual interface has unique MAC and IP addresses assigned, with a restriction: the IP addresses need to be in the same broadcast domain as the physical interface. MACvlan networking is a way of eliminating the need for the Linux bridge, NAT, and port-mapping, allowing you to connect directly to the physical interface.

The host cannot reach the containers. The container is isolated from the host. This is useful for service providers or multi-tenant scenarios and has more isolation than the bridge model.

IPvlan

IPvlan is similar to MACvlan in that it creates a new virtual network interface and assigns each a unique IP address. The difference is that the same MAC address is used for all pods or containers on a host, i.e: the same MAC address of the physical interface. Best run on kernels 4.2 or newer, IPvlan may operate in either L2 or L3 modes. Like MACvlan, IPvlan L2 mode requires that IP addresses assigned to subinterfaces be in the same subnet as the physical interface. IPvlan L3 mode, however, requires that container networks and IP addresses be on a different subnet than the parent physical interface.

MACvlan and IPvlan

When choosing between these two underlay types, consider whether or not you need the network to be able to see the MAC address of the individual container. In this sense, IPvlan L3 mode operates as you would expect an L3 router to behave.

Docker is experimenting with Border Gateway Protocol (BGP). While static routes can be created on top of the rack switch, projects like goBGP have sprouted up as a container ecosystem-friendly way to provide neighbor peering and route exchange functionality.

Although multiple modes of networking are supported on a given host, MACvlan and IPvlan can‘t be used on the same physical interface concurrently. In short, if you‘re used to running trunks down to hosts, L2 mode is for you. If the scale is a primary concern, L3 has the potential for a massive scale.

3.23 Container Networking Model

Container Network Model (CNM) formalizes the steps required to provide networking for containers while providing an abstraction that can be used to support multiple network drivers. Libnetwork is the canonical implementation of the CNM. Libnetwork provides an interface between the Docker daemon and network drivers. The network controller is responsible for pairing a driver to a network. Each driver is responsible for managing the network it owns, including services provided to that network like IPAM. With one driver per network, multiple drivers can be used concurrently with containers.

Libnetwork

Libnetwork implements Container Network Model (CNM), which formalizes the steps required to provide networking for containers while providing an abstraction that can be used to support multiple network drivers. Libnetwork delivers a unified API for integrating networking solutions from Weave, Nuage, Cisco, Microsoft, Calico, Midokura, and VMware into Docker. Finally, Libnetwork implements the Container Network Model (CNM).

The CNM contains several different constructs

- Endpoint
- Network
- Sandbox

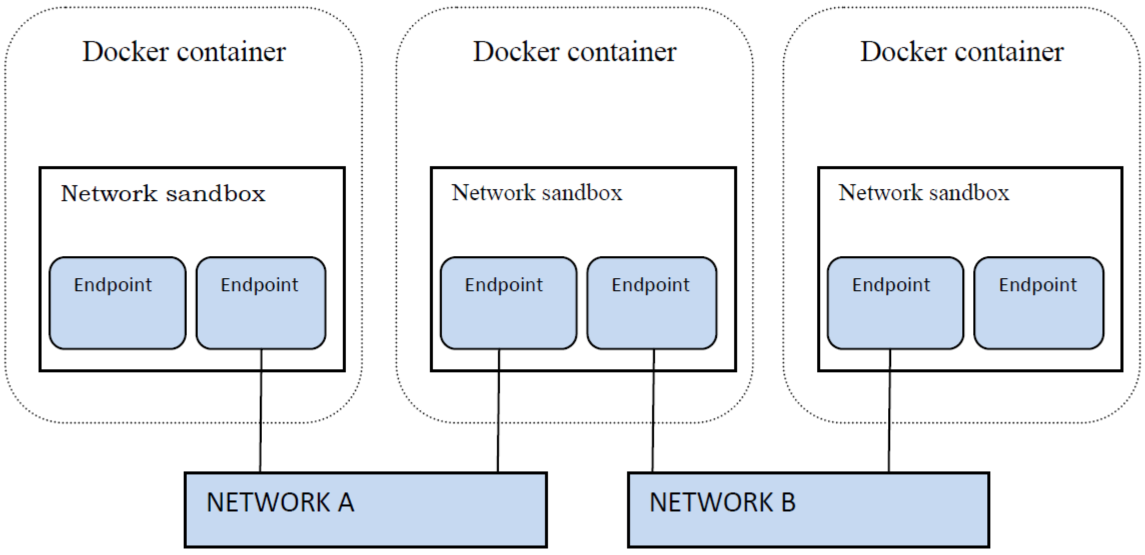


Figure 18: The Container Network Model (CNM)

Endpoint

A network interface can be used for communication over a specific network. Endpoints join exactly one network, and multiple parameters can exist within a single Network Sandbox.

Network

A Network is a uniquely identifiable group of Endpoints that can communicate with each other directly. An implementation of a Network could be a Linux bridge, a VLAN, VPN, etc. A network consists of many endpoints. You could create networks A and B that are entirely isolated.

Sandbox

An isolated environment that allows Network configuration for a Docker Container. This includes the management of the container's interfaces, routing table, and DNS settings. A Sandbox may contain many endpoints from multiple networks.

Container Networking Interface

The CNI (Container Network Interface) project consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with several supported plugins. CNI concerns itself only with network connectivity of containers and removing allocated resources when the container is deleted. Multiple plugins may be run at one time with a container joining networks driven by different plugins. CNI plugins support two commands to add and remove container network interfaces to and from networks. Add gets invoked by the container runtime when it creates a container. Delete gets invoked by the container runtime when it tears down a container instance.

Container Network Model and Container Networking Interface

Both container standardization models democratize the selection of which type of container networking may be used for creating and managing network stacks for containers. Both models allow containers to join one or more networks. And each allows he container runtime to launch the network in its own namespace, segregating the application/business logic of the container to the network to the network driver.

CNI supports integration with third-party IPAM and can be used with any container runtime while CNM is designed to support the Docker runtime engine only. With CNI‘s simplistic approach, it‘s been argued that it‘s comparatively easier to create a CNI plugin than a CNM plugin.

These models promote modularity, composability and choice by fostering an ecosystem of innovation by third-party vendors who deliver advanced networking capabilities. The orchestration of network micro-segmentation can become simple API calls to attach, detach, and swap networks.

Container Networking in OpenStack

OpenStack is a framework for managing, defining, and utilizing cloud resources. The official OpenStack website (www.openstack.org) describes the framework as open-source software for building private and public clouds.‖ According to (V.K. Cody Bumgardner, 2015), OpenStack Software delivers a massively scalable cloud operating system.

According to the online publication (**Alex Willams et al. 2016**), OpenStack is rapidly becoming a core building block for companies such as AT&T, Verizon, BMW, Volkswagen, and Walmart, that are building private cloud infrastructures. OpenStack has become an integration engine that bridges the union of containers, bare metal, and virtual machines. OpenStack brings these resources together in one platform and supports a variety of networking and scaling approaches and storage options.

OpenStack delivers choice, scalability, and the flexibility to adopt new technologies. Initially focused on infrastructure automation for virtual machines, OpenStack supports container networking through two projects, namely Kuryr and Magnum.

Kuryr

Kuryr, a project providing container networking, currently works as a remote driver for libnetwork to provide networking for Docker using Neutron as a backend network engine. Support for CNM has been delivered, and the roadmap for this project includes support for CNI.

Magnum

Magnum, a project providing Containers as a Service (CaaS) and leveraging Heat to instantiate clusters running other container orchestration engines, currently uses non-Neutron networking options for containers.

Network Driver Plugins

Weave

Weave uses open vSwitch architecture, and containerized applications are interlinked and appear to be plugged into the same network switch, with no need to configure port mappings, links, etc. Services provided by application containers on the weave network are accessible to the outside world, regardless of where those containers are running (**Weaveworks, 2015**).

Calico

Calico employs the underlay solution for interconnecting Virtual Machines or Linux Containers. Instead of a vSwitch, Calico operates a vRouter function in each computes node. The vRouter uses the existing L3 forwarding capabilities of the Linux kernel, which are configured by a local agent (Felix) that programs the L3 Forwarding Information Base with details of IP addresses assigned to the workloads hosted in that compute node (**Cloudsoft, 2015**).

Calico provides high scalability because it‘s based on the same principles as the Internet, using the Border Gateway Protocol (BGP) at the control plane. With well-known implementations, BGP can handle tens of thousands of distinct routes comfortably. And because Calico connects virtual machines or containers directly via IP, it scales beyond the data center and natively supports cloud connectivity across any geographic distribution.

Microservices Discovery Techniques

Service discovery is a mechanism for locating where the Microservices are hosted. Once you have several microservices forming your application, your attention inevitably turns to know where on earth everything is. Perhaps you want to know what is running in a given environment so you know what you should be monitoring. Maybe it‘s as simple as knowing where your customer service is so that those Microservices that use it know where to find it. Or perhaps you just want to make it easy for developers in your organization to know what APIs are available so they don‘t reinvent the wheel.

DNS

DNS has a host of advantages, the main one being it is such a well-understood and well-used standard that almost any technology stack will support. Unfortunately, while several services exist for managing DNS inside an organization, few of them seem designed for an environment where we are dealing with highly disposable hosts, making updating DNS entries somewhat painful.

Dynamic Service Discovery

The downsides of DNS as a way of finding nodes in a highly dynamic environment have led to several alternative systems, most of which involve the service registering itself with some central registry, which in turn offers the ability to look up these services later on. Often, these systems do more than just providing service registration and discovery.

Ectd

Etcd is an open-source distributed key-value store that serves as the backbone of distributed systems by providing a canonical hub for cluster coordination and state management.

Etcd is written in Go and uses the Raft Consensus protocol. The raft is a protocol for multiple nodes to maintain identical logs of state-changing commands, and any node in a raft node may be treated as the master. It will coordinate with the others to agree on which order state changes happen in.

Zookeeper was initially developed as part of the Hadoop project. It is used for an almost bewildering array of use cases, including configuration management, synchronizing data between services, leader election, message queues, and as a naming service.

Like many similar types of systems, Zookeeper relies on running several nodes in a cluster to provide various guarantees. This means you should expect to be running at least three Zookeeper nodes. Most of the smarts in Zookeeper are around ensuring that data is replicated safely between these nodes and that things remain consistent when nodes fail. Zookeeper is often used as a general configuration store, so you could also store service-specific configuration in it, allowing you to do tasks like dynamically changing log levels or turning off features of a running system.

Consul

Like Zookeeper, Consul supports both configuration management and service discovery. But it goes further than Zookeeper in providing more support for these critical use cases.

For example, it exposes an HTTP interface for service discovery, and one of Consul‘s killer features is that it provides a DNS server out of the box; specifically, it can serve SRV records, which give you both an IP and port for a given name. This means if part of your system uses DNS already and can support SRV records, you can just drop in Consul and start using it without any changes to your existing policy.

Consul also builds in other capabilities that you might find useful, such as the ability to perform health checks on nodes. This means that Consul could well overlap the capabilities provided by other dedicated monitoring tools, although you would more likely use Consul as a source of this information and then pull it into a more comprehensive dashboard or alerting system.

Consul heavily relies on a RESTful HTTP interface for everything from registering a service, querying the key/value store, or inserting health checks. This makes integration with different technology stacks very straight forward.

Microservices Interprocess Communication

Each microservice instance is housed in its own container; hence there must exist a mechanism for inter-container communication. The lethal combination of HTTP and JSON resulted in a new Architectural style called REST. REST has become wildly popular among web developers. Many applications rely on REST, even for internal serialization and communication patterns. But HTTP is not the most efficient protocol for exchanging messages across services running in the same context, same network, and possibly the same machine. HTTP‘s convenience comes with a huge performance trade-offf, hence the need for finding an optimal communication framework for microservices.

gRPC

gRPC, is based on client-server architecture whereby an application can directly call methods on a server application on a different machine as if it was a local object, making it easier for you to create distributed applications and services. gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. On the server-side, the server implements this interface and runs a gRPC server to handle client calls. On the client-side, the client provides the same methods as the server.

When compared to REST, gRPC offers better performance and security. It heavily promotes the use of SSL/TLS to authenticate the server and to encrypt all the data exchanged between the client and the server. gRPC uses HTTP/2 to support highly scalable APIs. The use of binary rather than text minimizes the payload. HTTP/2 requests are multiplexed over a single TCP connection, allowing multiple concurrent messages to be in flight without compromising network resource usage. It uses header compression to reduce the size of requests and responses.

Multi-Language Support

gRPC clients and servers can run and talk to each other in heterogeneous environments. For example, you can easily create a gRPC server in Java with clients in Go, Python, or Ruby.

gRPC uses protocol buffers, Google‘s open-source mechanism for serializing structured data. Proto3 is the latest version of protocol buffers and is recommended because it has a slightly simplified syntax, some useful new features, and supports lots more languages. This is currently available in Java, C++, Python, Objective-C, C#, JavaNano (Android Java), Ruby, JavaScript, and Go language generator with more languages in development.

4 Practical Part

The aim of this section describes the contingent implementation of the application.

It includes various technologies, data-flow, front-end, and back-end terminology.

In terms of

2.2.6 Infrastructure

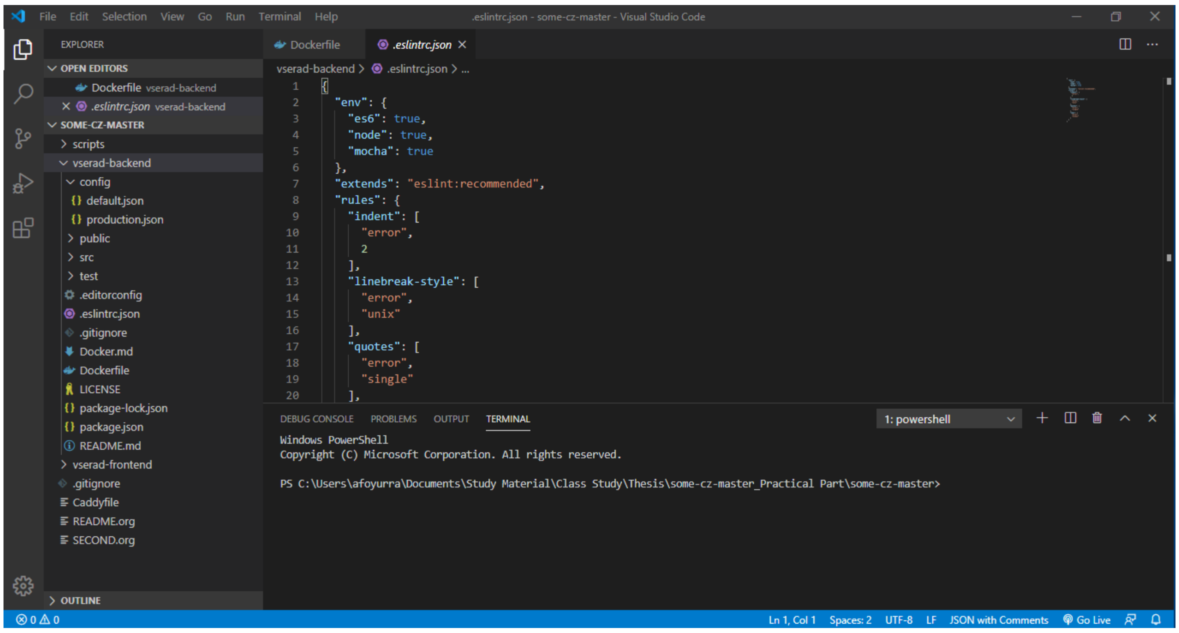
All infrastructure software is Open Source, which is under a GPL license meaning free to use, and we can both view and edit its code, which we often do. All of the system's infrastructure will be in the cloud. All communication between the backend and front end will be a secure HTTPS protocol.

2.2.7 Backend

All background operations such as calculating, storing, and versioning data will be implemented as a Feathers.js application providing a documented API for the frontend application.

Feathers is a very lightweight web framework that is very useful for creating real-time applications. It supports REST APIs using TypeScript or JavaScript.

Feathers support almost every backend technology out there. It also can interact with any frontend technology like Angular, React Native, React, VueJS, Android or ios, supports over a dozen database.



```
{
  "env": {
    "es6": true,
    "node": true,
    "mocha": true
  },
  "extends": "eslint:recommended",
  "rules": {
    "indent": [
      "error",
      2
    ],
    "linebreak-style": [
      "error",
      "unix"
    ],
    "quotes": [
      "error",
      "single"
    ],
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\afoyurra\Documents\Study Material\Class Study\Thesis\some-cz-master_Practical Part\some-cz-master>
```

Figure 19: Apps Backend Repository

2.2.8 Frontend

The frontend of the system will be developed as a so-called single-page application using the React.js JavaScript framework, where all interactions take place only on the browser side, and business operations are performed via the API, as mentioned in the backend section.

React.js is a framework developed by Facebook, which is also its largest user. We believe that what is suitable for FaceBook, as the most visited site in the world, is ideal for your system.

Visually, the system will be built on Google's Material Design.

Material design is a design language. Material design is a new perspective on what the human and device relationship can be. It’s a way for designers to collaborate with users. It is a cross-platform design framework developed by google in 2014.

The list of tools used to develop the application frontend:

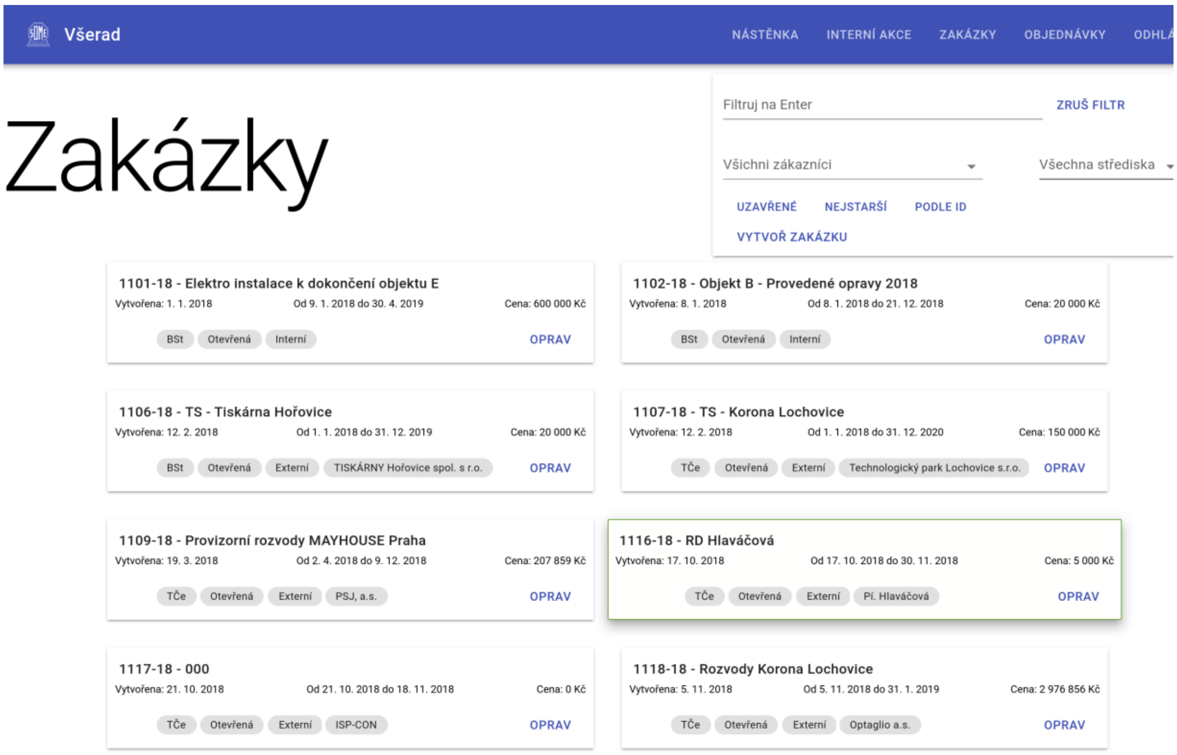
ClojureScript

React.js

Rum

Potok

Material Design Components



1101-18 - Elektro instalace k dokončení objektu E
Vytvořena: 1. 1. 2018 Od 9. 1. 2018 do 30. 4. 2019 Cena: 600 000 Kč
BSt Otevřená Interní OPRAV

1102-18 - Objekt B - Provedené opravy 2018
Vytvořena: 8. 1. 2018 Od 8. 1. 2018 do 21. 12. 2018 Cena: 20 000 Kč
BSt Otevřená Interní OPRAV

1106-18 - TS - Tiskárna Hořovice
Vytvořena: 12. 2. 2018 Od 1. 1. 2018 do 31. 12. 2019 Cena: 20 000 Kč
BSt Otevřená Externí TISKÁRNY Hořovice spol. s r.o. OPRAV

1107-18 - TS - Korona Lochovice
Vytvořena: 12. 2. 2018 Od 1. 1. 2018 do 31. 12. 2020 Cena: 150 000 Kč
TČe Otevřená Externí Technologický park Lochovice s.r.o. OPRAV

1109-18 - Provizorní rozvody MAYHOUSE Praha
Vytvořena: 19. 3. 2018 Od 2. 4. 2018 do 9. 12. 2018 Cena: 207 859 Kč
TČe Otevřená Externí PSJ, a.s. OPRAV

1116-18 - RD Hlaváčová
Vytvořena: 17. 10. 2018 Od 17. 10. 2018 do 30. 11. 2018 Cena: 5 000 Kč
TČe Otevřená Externí Pí. Hlaváčová OPRAV

1117-18 - 000
Vytvořena: 21. 10. 2018 Od 21. 10. 2018 do 18. 11. 2018 Cena: 0 Kč
TČe Otevřená Externí ISP-CON OPRAV

1118-18 - Rozvody Korona Lochovice
Vytvořena: 5. 11. 2018 Od 5. 11. 2018 do 31. 1. 2019 Cena: 2 976 856 Kč
TČe Otevřená Externí Optaglio a.s. OPRAV

Figure 20: Apps Frontend

3.1 Authentication

Different users with different levels of authority over data will log on to the system. A third-party service will provide this login that the system will use, namely Auth0. http://auth0.com.

Auth0 is the most accessible and most straightforward tool to enable administrators to manage user identities, including creating and provisioning, password resets, blocking, and deleting users. It's effortless and quickly connect to the application. Auth0 also provide to set up rules and access analytics dashboard and possible to customize the login page. It has an extensive list of features:

- **Universal Login (**Social logins, MFA**)**

- **Single Sign-on**
- **Multifactor Authentication (**SMS, Email, Google Authenticator, etc.)
- **Breached Password Detection and Brute force protection**
- **User Management**
- **Passwordless** (SMS/Magic link)
- **Machine to Machine** (Internal/External APIs and Applications)

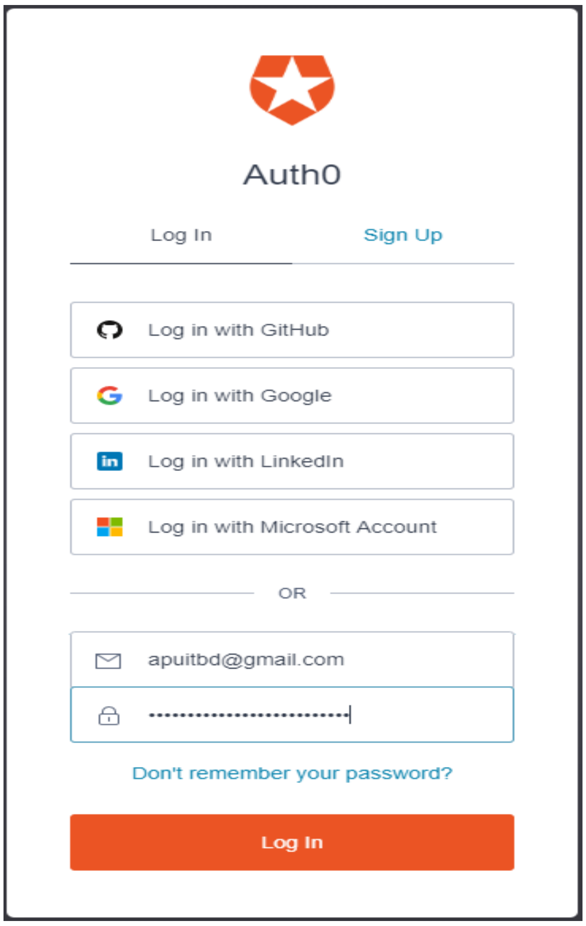


Figure 21: Auth0 Login page

2.2.9 Authorization levels

- **Administrator**

The administrator is a superuser of the system. The administrator has full access rights to create, modify, delete all the information carried in the system.

- **Executive**

This role has similar rights to the administrator, only limited to its part of the company (center, economic, workshop) department without the possibility to change or delete data concerning the whole company. Some processes will also be subject to administrator approval.

- **Ordinary Employee**

It only uses the information contained in the system, or adds some related to its (hours worked, costs). Most actions on data will be subject to approval by the manager or administrator.

3.2 Dashboard

The dashboard of the whole system, where every user will see the most relevant information for him at the moment. Initially in text form, later graphically rendered. There will also be links to frequently used events.

3.3 Order Life Cycle

The first and most important plan will be the life cycle of the order. This plan will be run in time, from its inception, through changes during its implementation, to its closure.

Furthermore, there will be a list of all orders in the system in which it will be possible to search and filter. Reports and visual reports can also be created from selections and filters.

3.4 Order attributes

Creator

The employee who created the order is pre-filled according to the logged-in user.

Date of establishment

The date on which the order is created automatically pre-filled according to the current date/time.

Order number

The order number consists of four parts, which together make up its full value.

- **center number** selection from the list or according to logged in user
- **section number** selection from the list according to the selected center number
- **order sequence** automatically followed by a number by center number and order number
- **year of establishment** separated by a dash, automatically according to the current date

Type of order

Internal or external order.

Name of the contract

For internal, the name is selected from the domestic order codebook. A new name is created for the external job.

Planned launch

Select a date from the calendar

Scheduled completion

Select a date from the calendar.

Subscriber

For internal Internal for external selection from the subscriber list.

Comment

List of dated text notes to which information about the course of the event can be added.

Brief description of the contract

For external order only.

Order price

For internal order price set to zero. For external, it contains value without VAT, percentage of VAT, and the final price with VAT.

3.5 Dials

There will be a part of the system for the management and management of all code list items existing in the company. Codebooks are understood as information that does not change primarily in time and has a registration identifier.

It will be possible to view all items, to see and edit their data for individual items or to mark them as deleted. Repairs and deletions will never mean loss of information. All historical data will be stored in the system for future reference.

The code lists will be primarily filled before the system is put into operation, according to data already existing in the company.

3.6 Resorts

Individual parts of the company, according to the standard division currently used.
Standard information such as its identification number and manager is kept at the center.

3.7 Employees

List of all employees in the company. For every employee, there is kept standard information necessary for personnel management of the company:

- Title before the name
- Name
- Surname
- Title after name
- Center
- Birthdate
- Place of birth
- Place of residence
- Identification number
- Employed by
- Employed in
- Position according to the code list
- Basic salary CZK / hour
- Premium CZK / hour
- Skills

3.8 Subscribers

List of all customers with which the company cooperates. They will be kept:

- The number generated at creation
- Name
- Billing address
- IČ
- VAT no
- Bank account
- Contact person
    - Name
    - Position
    - Center
    - Telephone
    - Email
    - Note

3.9 Suppliers

List of all suppliers with which the company cooperates. They will be kept:

- The number generated at creation
- Name
- Billing address
- IČ
- VAT no
- Offered services
- Bank account
- Contact person
    - Name
    - Position
    - Center
    - Telephone
    - Email
    - Note

5 Results and Discussion

In the thesis, we employ the trend of technology in web application. Today in the digital era business mastered agile, cost effective, responsive and scalable. To improve the scalability of web services using OS-Level Virtualization. The no. n of processors can be interpreted to mean the number of containers. Scalability is handled at the orchestration layer. The design of three cluster orchestration, namely Docker Swarm, Kubernetes, and Mesos, was examined in sections 3.14 and 3.15 respectively. The orchestration layer has the role of minimizing contention and crosstalk as the system is scaled up. According to the model that is based on USL, the expected behavior as the system is scaled up is shown below. However, our results showed that for well-designed orchestration layer contention delay is wholly eliminated, and coherency delay is minimal. The coefficient of contention delay sigma was zero while the coefficient of coherency delay was found to be 0.0004475

In section 3.15, we discussed the design of the orchestration layer for the three cluster orchestration software. Docker Swarm 3.12.18 has a simpler architecture based on the Raft consensus algorithm to handle contention delay arising from the need to coordinate the state of the containers that are constantly changing as the microservices are scaled up and down. Looking keenly at the Docker Swarm architecture, one will notice that it is based on a Microservice Architecture. Inter-container communication is achieved using the gRPC protocol that is built on the fast and high efficient HTTP/2 and protocol buffers protocol.

Kubernetes is based on a very complex architecture, which is an improvement of the battle-tested Borg and Omega. Both Borg and Omega have been in use by Google for over several years to run Google data centers. It was noted that as Google moved from Borg to Omega and to open-source Kubernetes, the architecture was gradually changed from monolithic to microservices.

Model Validation

Using the scalability model and test results generated by Jeff Nickoloff, we managed to measure the performance of the Docker Swarm and Kubernetes architecture. Our finding showed that both the two software are capable of scaling of a cluster to over 1000 nodes. Within this range, both systems scaled linearly as the number of the container were increased from 1 to 30000. Regression analysis done using the R software found out that the sigma coefficient (σ) and Kamma (κ) factor was found to be zero, meaning that the effect of contention for shared resources and coherency delay for data to become consistent was non-existent.

6 Recommendations for Future Work

Microservices is a promising architecture, and Containerization abstracts the complexity introduced by splitting an application into independent and composable functional units. However, given that microservices dictate that DevOps as the best development methodology, there is a need to investigate and research on the strategies for organizations to use to transition from monolithic to Microservice Architecture.

The introduction of containers in data centers to supplement Virtual machines should be investigated further to devise mechanisms of integrating all Orchestration technologies to improve the isolation of processes running on a single host. This will enhance security in data centers and hasten the adoption of containers in the cloud. There is a need to develop the microservices development and deployment framework that is extensible and can use the popular orchestration software as plugins.

Microservices development frameworks should be designed to exploit the functions that are being offered at the orchestration layer. For example, developers should be able to create microservices endpoints by importing this information from the Docker Compose files.

To simplify software development, there is a need to enhance the Docker Compose files to be capable of having enough information for creating microservices without the need to use other development tools. JHipster is JVM based development tool that can be enhanced to exploit the Docker Compose information to simplify the generation of microservices code.

The size of images also poses challenges because they consume a lot of network capacity as they are moved from registries to development and production environment. Further research based on Unikernels, light-weight Virtual Machines, and Serverless Computing is required to enhance cloud security and reduce the size of images.

7 Conclusion

The introduction of Microservice Architecture has reinforced the resolve for agility in the software industry. Containerization is promising to transform IT in a more profound way than full virtualization. The scalability and the associated cost reduction and energy saving that can be achieved when the two technologies are applied concurrently is enormous.

The most crucial resource in IT is Humanware. However, it was discovered that for large project teams working on monolithic software, the man-month law works in reverse as you increase the number of developers. This means that the performance of the team is not proportional to the increased man-hours. Microservice Architecture enables scalable development of software since small manageable cross-functional teams can handle each Microservice.

Similarly, a monolithic application will scale horizontally by consuming more virtual machines or servers. Given that the monolith comprises software components whose functionality has varying demand from the users, it becomes wasteful to assign more resources to all software components equally. The Microservice Architecture addresses this problem by enabling scalability on a functional dimension. By splitting the application into smaller units, the workforce per unit can be resized accordingly to enhance productivity. The number of technologies supported may be scaled accordingly as the need arises.

By employing distributed data stores in Microservice Architecture, scalability is enhanced by eliminating coherency delay that is prevalent in relational databases. The developer has the flexibility to choose the right database technology. Factors influencing the adoption of Microservice Architecture include virtualization, containerization, and the Internet of things.

To What Extent Can Containerization Enhance Design and Implementation of Microservice Architecture?

Containerization abstracts the complexity that is introduced by Microservice Architecture. Most functions that arise due to splitting a monolith into microservices such load balancing, health checks, etc. are handled at the orchestration layer. Similarly, the features such as service discovery, scheduling, inter-container communication are hidden from the developer and managed at the orchestration layer. The Docker Architecture is extensible through the use of plugins. Volume plugins allow third-party container data management solutions to provide data volumes for containers that operate on data, such as databases, queues, and key-value stores and other stateful applications that use the file system. Network plugins allow third-party container networking solutions to connect containers to container networks, making it easier for containers to talk to each other even if they are running on different machines. Docker Swarm is a powerful cluster management tool that is capable of handling over 1000 hosts and scheduling up to 30000 containers.

To What Extent Can Microservice Architecture Improve the Scalability of Web Services?

Scalability of web applications can be achieved through vertical scaling, horizontal scaling and functional scaling. By using Microservice Architecture it becomes possible to split a web application into smaller units that can be packaged as containers for efficient utilization of the hardware and software resources. With containerized microservices you can achieve a high software density in a data centre than using a virtual machine. This means that scaling up and down a given function in a web application is easier, faster and less costly.

Scalability is multi-dimensional. Containerized Microservices makes functional scalability possible. Using Docker Compose, we managed to demonstrate how you can scale up service by specifying the number of containers using Command Line Interface. By varying the number of containers for a given microservice, we were able to achieve the desired scalability.

Microservice Architecture reduces friction amongst developer teams, operations team and quality assurance teams. As you scale up by adding developers to monolithic system the man-months increases. The coordination effort for a team of n members is proportional to n(n-1). By splitting the monolith into microservices you are able to assign 3-5 people to one microservice and this reduces friction though reduced coordination effort.

On the infrastructure side, it has been found the only 15-30 % of the servers are used in data centers while the rest of servers are on standby but consuming power. With microservices, you only scale those microservices that are receiving higher user requests. One is able to back more containers per host machine and considerably reduce the OPEX by eliminating the need for operating system per virtual machine. With the proper mix of containers and virtual machines, you are can achieve high isolation and security, making multi-tenancy in data centers. With the orchestration layer forming an abstraction layer, the complexity arising from microservices is hidden from the developer. Based on the analysis of the results using the scalability model, Docker Swarm was found to scale linearly with the increase in the number of containers up to thirty thousand containers placed over one thousand hosts in the AWS cloud. We repeated this test using our own data measured using a cluster of five machines, and it was found that throughput increased linearly with the number of the container. This means that the contention delay is eliminated and has no effect on scalability. Coherency delay that is caused process waiting for data to be consistent is minimal; This delay can be handled by using ACID 2.0 design principles.

8 References

Adrian Cockcroft, (2014), –Migrating to Microservice‖. In: QCon London.

Agile Manifesto (Accessed on 3/07/2015) URL: http://www.agilemanifesto.org/

Alex Williams,(2016), –The Docker and Container Ecosystem eBook Series‖, The New Stack.

Alex Williams, (2014), –Flocker, A Nascent Docker Service For Making Containers Portable, Data and All.

Bob Williamson et al,(2015) –Akka in Action Manning‖.

Brendan et al.(2015), ― Borg, Omega, and Kubernetes, Lessons learned from three container-management systems over a decade‖, Google Inc.

Cloudsoft , ―Container Networking plugin‖, (accessed 17/07/2015) URL: http://www.projectcalico.org/learn

Cody Bumgardner, (2015), ― Openstack in action, Manning publications‖

Conway et al , (1968), –How do Committees Invent?‖, Datamation 14 (5): 28–31.

D. Ongaro , J. Ousterhout, (2014), ― In Search of an Understandable Consensus Algorithm In 2014 USENIX Annual Technical Conference, pages 305-319, Philadelphia,PA.

Damon Edwards, –What is DevOps?‖ (Accessed on 3/7/2015) http://dev2ops.org/blog/2010/2/22/what-is-DevOps.html

Danilo Poccia, (2016), –AWS Lambda in Action, Event-Driven Serverless Applications‖ First Edition, Manning .

David Hilley et al.(2009), –Cloud computing: A taxonomy of platform and infrastructure-level offerings‖.

Edward A. Lee, (2006), –The Problem with Threads‖ Technical Report No. UCB/EECS-2006-1

L. Bass, P. Clements, and R. Kazman, (1998), –*Software Architecture in Practice" ,* Addison Wesley, Reading, Mass.

L. G. Williams and C. U. Smith, (2004), ―Web Application Scalability: A Model-Based Approach,‖ Proceedings of the Computer Measurement Group, Las Vegas.

L. Qian, Z. Luo, Y. Du, and L. Guo, (2009), ― Cloud computing: An overview. In Proceedings of the 1st International Conference on Cloud Computing, CloudCom '09, pages 626–631, Berlin, Heidelberg, Springer-Verlag.

Leonard Richardson, Sam Ruby, (2007), ―RESTful Web Services‖, O'Relly.

M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A Patterson, A. Rabkin, I. Stoica, and M. Zaharia. (2009), ―Above the clouds: A Berkeley view of cloud computing‖, Technical Report UCB/EECS-2009-28, EECS Department, Universityof California, Berkeley.

Mahmood, Z., Hill, R. (2011). Cloud Computing for Enterprise Architectures. Computer Communications and Networks, Springer

Martin C. Roberts et (2016), ―Agile Principles, Patterns, and Practices in C#‖, First Edition Prentice Hall .

Martin Fouler http://martinfowler.com/articles/Microservice.html accessed on 19-6-2015

Martin Fowler and James Lewis. Microservice (Accessed on 25/06/2015). 2014. URL:http://martinfowler.com/articles/Microservice.html

Mathijs Jeroen Scheepers, (2014), ―Virtualization and Containerization of Application Infrastructure: A comparison‖ University of Twende.

Neil J. Gunther, (2007), ―A Review of "Guerilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services, Performance Dynamics Company.

P. Mell and T. Grance. (2011), ―The NIST definition of cloud computing‖, Technical report, National Institute of Standard and Technology - NIST.

Pasa Maharjan (2016), ―Comparing and Measuring Network Event Dispatch Mechanisms in Virtual Hosts‖ Mater of Science Thesis, Tampere University of Technology.