University of Nairobi

School of Computing and Informatics

Development of a Scalable Microservice Architecture for Web Services using OS-level Virtualization

Presented by : Khakame Peter Wamboko

P56/P/8256/2004

Supervisor : Robert Oboko

A Project Report Presented to the School of Computing and Informatics in Partial Fulfillment of the Requirements for the Award of Degree of Master of Science In Information Systems

University of Nairobi

2016

DECLARATION

I declare that this thesis is my original work and has not been submitted elsewhere for examination, award of degree or publication. Where other peoples work or my own work has been used, this has properly been acknowledged and referenced according to University of Nairobi's Requirements.

…… **……..**

Signature **Date**

Khakame Peter Wamboko
P56/P/8256/2004

This Thesis is submitted for examination with our approval as supervisor(s**)**

…… **……..**

Signature **Date**

Dr Robert Oboko
School of Computing & Informatics
University of Nairobi
P.O. BOX 30197-00100
NAIROBI
Robertoboko@uonbi.ac.ke

DEDICATION

This work is dedicated to mankind whose livelihoods can be improved by better use of emerging virtualization technologies.

ACKNOWLEDGEMENTS

I wish to thank my supervisor Dr Robert Oboko for his guidance and support. His invaluable insights about research have formed the basis of this thesis. I also wish to thank my colleagues whom we worked together under the supervision of Dr Oboko. The members of teaching and support staff at the Computing and Informatics department deserve special thanks for their support throughout the course.

I also wish to acknowledge the support I received from family members, my mum Resah Khakame, my brother Saul Wamocha and my friend Yona Eteneh.

My thanks to my Wife, Tabitha Adhiambo and my three daughters Milkah, Maria and Sara Joy who stood by me throughout my masters course. They tolerated my working late and my not being available to take them out on holidays.

Lastly I thank God for having given the life and good health to enable me completes this work despite the challenges that I faced.

TABLE OF CONTENTS

DECLARATION ii
DEDICATION iii
ACKNOWLEDGEMENTS iv
LIST OF TABLES xiv
LIST OF ACRONYMS xv
ABSTRACT xviii
CHAPTER ONE: INTRODUCTION 1
1.1 Background 1
1.1.1 The Universal Scalability law 1
1.1.2 Service Oriented Architecture (SOA) 2
1.1.3 Microservice Architecture 3
1.2 Problem Definition 3
1.3 The Solution to the Problem. 6
1.4 Purpose 6
1.5 Scope of the Study 7
1.6 Related Work 7
1.7 Organization of the Thesis 8
CHAPTER TWO: LITERATURE REVIEW 9
2.0 Introduction 9
2.1 Definition of Terms 9
2.1.1 Microservices 9
2.1.2 Continuous Delivery 9
2.1.3 Continuous Integration 10
2.1.4 Configuration Management 10
2.1.5 DevOps.. 10
2.1.6 OS-Level Virtualization 10
2.1.7 Docker 11
2.1.8 Monolithic Architecture 12
2.1.9 Software Application 13
2.1.10 Process 13

2.1.11 Web Service 13
2.1.12 Software Architecture 13
2.2 Principles of the Microservice Architecture 14
2.2.1 Microservice Architectural Constraints 15
2.3 Conceptual design 17
2.3.1 Software functional components 17
2.4 REST Architectural Styles and Architectural Constraints 17
2.5 Microservices Architectural Style 18
2.6 Characteristics of Microservices 19
2.6.1 Independent Technology Stacks 19
2.6.2 Independent Scaling 19
2.6.3 Independent Evolution of Features 21
2.6.4 Stable Interfaces – Standardized Communication 21
2.6.5 Componentization via Services 21
2.6.6 Favors Cross-Functional Teams 22
2.7 Challenges to a Microservice Architecture 23
2.8 Microservice Architecture , DevOps and Containers 24
2.8.1 DevOps is a Prerequisite to Successfully Adopting Microservices 24
2.9 Cloud Computing 24
2.9.1 Characteristics and Benefits of Cloud Computing 25
2.9.2 Cloud Computing Service Delivery Models 26
Table 2: Various cloud computing service delivery Models 27
2.9.3 Deployment Models 29
Figure 9: The docker Architecture and design principles 34
Docker API…… 34
2.11.1 Scaling up microservices with Docker compose 40
2.11.2 Scaling up microservices with Docker Swarm 41
2.11.3 Scaling up Microservices Kubernetes 47
2.11.4 Scaling up microservices with Apache Mesos 50
2.12 Containerized Application Management 51
2.13 Docker Plugin Architecture 54

2.14 Volume Plugins 56
2.15 Network Plugins 57
2.15.1 Types of Container Networking 57
2.15.2 Container Networking Standards 59
2.15.2.1 Container Networking Model 59
2.15.2.2 Container Networking Interface 62
2.15.2.3 Container Network Model and Container Networking Interface 62
2.15.2.4 Container Networking in OpenStack 63
2.15.3 Network Driver Plugins 63
2.15.3.1 Weave. 63
2.15.3.2 Calico 64
2.15.4 Microservices Discovery Techniques 64
CHAPTER THREE: METHODOLOGY 68
3.0 Introduction 68
3.1.1 Research Methodology 68
3.1.2 Problem Definition and Analysis 68
3.1.3 Defining objectives of a solution 68
3.1.4 Artifact Design & Development 69
3.1.5 Artifact Demonstration 69
3.1.6 Artifact Evaluation 69
3.1.7 System Development Methodology 69
3.1.8 Architectural Design 70
3.2 System Development Process 70
3.2.1 DevOps.. 70
3.2.2 The Software Deployment Pipeline 72
3.2.3 Faster Deployments 73
3.2.4 Docker Support for continuous Deployment Pipeline 73
3.3 Continuous Integration 74
3.3.1 Version Control System 74
3.3.2 CI server 74
3.3.3 Build Management 76

3.3.3.1 Maven… 76
3.3.3.2 Gradle…. 76
3.4 System Testing 77
3.4.1 Unit Testing 78
3.4.2 Integration testing 79
3.4.3 Automated Microservices Testing 79
3.5 Scalability Experiments 79
3.0 Data analysis 81
3.6 Scalability Model 81
3.7 System Evaluation 83
CHAPTER FOUR : MICROSERVICE SYSTEM DESIGN, IMPLEMENTATION AND TESTING 85
4.1 Principles of the Microservice Design 85
4.1.1 Responsible for One Single Capability 85
4.1.2 Individually Deployable 86
4.1.3 Consisting of One or More Processes 87
4.1.4 Owns its Own Data Store 87
4.2 Microservice Design Patterns 88
4.2.1 The Aggregator Pattern 88
4.2.2 Proxy Pattern 88
4.2.3 Pipeline Pattern 89
4.3 Scalable Microservice Design 90
4.4 Front-End Microservice Architecture 91
4.4.1 JavaScript 91
4.4.2 Node.JS 92
4.4.3 Angular 2 92
4.4.4 Angular 2 Architecture 93
4.5 Backend Microservice Architecture 95
4.5.1 Concurrency programming Models 95
4.5.1.1 Reactive Extensions 95
4.5.1.2 AKKA… 96

4.5.1.3 Futures… 98
4.5.1.4 Reactive Streams 98
4.6 Microservice Architecture Frontend and Backend Implementation 98
4.6.1 Frontend JavaScript Development Tools 98
4.6.2 JavaScript Based frameworks and libraries. 99
4.6.3 Microservice Development Using JHipter Framework 100
4.6.4 The Registry 101
4.6.5 The API Gateway 102
4.6.6 Creating Microservices 102
4.6.7 Running the Microservices Using Docker Compose 102
4.7 The Data Store Design 103
4.7.1 Data Consistency 103
4.7.2 Contention Free Access to Shared State 103
4.7.3 Implementation of Distributed Data Stores 106
4.7.3.1 Implementing Distributed Data Stores Using Containerization 106
4.7.3.1.1 Software Defined Storage 107
4.7.3.1.2 SDS Solution Providers 108
4.7.3.2 Testing Software Defined Storage 109
4.7.3.2.1 Crate… 109
4.8 Test Results for Validation of Scalable Architecture 111
4.9 Performance evaluation 115
4.10 Automated Testing 117
CHAPTER FIVE : RESULTS ANALYSIS 120
5.1 Introduction 120
5.2 What Factors are Influencing the Adoption of Microservices 120
5.2.1 Virtualization 120
5.2.2 Containerization 121
5.2.3 Internet of Things 121
5.3 Results Analysis 122
5.3.1 Model Validation 123
5.3.2 Scalability Testing 124

5.4 Automated Software Testing 124
CHAPTER SIX : CONCLUSIONS AND RECOMMENDATIONS 126
6.1 Introduction 126
6.2 What Factors are Influencing the Adoption of Microservice Architecture 127
6.3 To what Extend Can Containerization Enhance Design and Implementation of Microservice Architecture? 127
6.4 To What Extend Can Microservice Architecture Improve the Scalability of Web Services? 127
6.5 To What Extend Can Microservices Testing be Automated? 128
6.6 Recommendations for Future Work 129
REFERENCES.. 130
APPENDIX A1.. 134
KUBERNETES ORCHESTRATOR : 134
APPENDIX A2.. 137
SWARM ORCHESTRATOR: 137
APPENDIX C…. 140
CONTAINER START DELAY MEASUREMENTS 164

LIST OF FIGURES

Figure 1: Amdahl's and Gunther Laws governing how throughput varies with increasing no of processors (Gunther, 2007). 2

Figure 2: Monolithic architecture showing how various service are lumped together into a single process 12

Figure 3: Microservice Mechanisms for enabling loose coupling 15

Figure 4: Microservice Architecture Conceptual Model 17

Figure 5: Microservice Hierarchical Tree 18

Figure 6: Functional scalability i.e Y-axis scalability 20

Figure 7: Illustration showing how Conway's law applies to system design in a given organization (Martin Fowler et al , 2014) 22

Figure 8: Comparison of OS-Level Virtualization and Full virtualization (Source: Docker Inc.) 31

Figure 9: The docker Architecture and design principles 34

Figure 13: Docker public or private Registries. 39

Figure 14: Open Container layered architecture.[Docker Inc] 40

Figure 15: Swarm Orchestration Architecture [Docker Inc] 44

Figure 16: The components of Docker Swarm cluster based on Raft Consensus Algorithm 46

Figure 18: Kubernetes Master-slave design illustrating its microservices and container based architecture (Marko Lukŝa, 2016) 49

Figure 19: Mesos two level orchestration architecture 51

Figure 20: Docker plugin Architecture showing the extensions and interfaces to other systems .. 55

Figure 21: The Docker volumes plugin architecture 57

Figure 22: The Container Network Model (CNM) 61

Figure 23: A DevOps Based Software Development Cycle [Adoted from Software Testing] 71

Figure 24: DevOps is an inter displinary approach to software development. 72

Figure 25: Stages of a deployment pipeline 72

Figure 28: Gradle combines the best features from other build tools. 76

Figure 29: Gradle key features 77

Figure 30: Test Setup used to measure container start up times on Amazon Web Services (Jeff Nickoloff , 2016) 80

Figure 31: Data analysis steps using R 81

Figure 32: The Aggregator Design Pattern 88

Figure 33:The Proxy Pattern 89

Figure 34: The Pipeline Pattern 90

Figure 35: Microservice Architecture implementation 91

Figure 36: Angular 2 Architecture 93

Figure 37: Non-blocking interplay between actors and futures 98

Figure 38: Microservice Architecture implementation on the JVM using the JHipter 3.6 tooling 101

Figure 39: JHipster CLI for development of Microservices 102

Figure 40: Docker Swarm cluster creation automation script. 110

Figure 41: Validation of usability model using dataset1 (courtesy Nickoloff 2016) 112

Figure 42 : Validation of usability model using dataset2(courtesy Nickoloff 2016) 113

Figure 43: Determination of sigma and kamma coefficients for the scalability model based using regression analysis. 114

Figure 44: Layering of a cluster to abstract scalability away from the microservices to the orchestration layer 114

Figure 45: Obtained Results –Variation of throughput vs the number of containers 116

Figure 46: Test report integration tests conducted by the Continuous Integration server 119

Figure 47: Expected behavior of throughput vs no of containers. (courtesy Gunther, 2007). . 123

Figure 48: Mean container start up measurements for 10 containers showing the 90th percentile and 99th percentile 140

Figure 49: Mean container start up measurements for 20 containers 140

Figure 50: Mean container start up measurements for 30 containers 141

Figure 51: Mean container start up measurements for 40 containers 141

Figure 52: Mean container start up measurements for 50 containers 141

Figure 53: Mean container start up measurements for 60 container 142

Figure 54: Mean container start up measurements for 10 containers 142

LIST OF TABLES

Table 1: Characteristics of Cloud Computing..... 26

Table 2: Various cloud computing service delivery Models 27

Table 3: start delay vs Number of containers measurements..... 112

Table 4: Throughput vs Number of containers based container start delay time..... 115

Table 5: Measurements of throughput(tput) as the number of containers is increased from one to 414 134

Table 6: measurements of throughput(tput) as the number of containers is increased from one to 500 137

LIST OF ACRONYMS

HTTP	Hyper Text 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

XML	eXtented Markup Language
DSL	Domain Specific Language
EIP	Enterprise Integration Pattern
BDD	Behavior Driven Development
SVN	SubVersioN
VCS	Version Control System
IRC	Interactive Realtime Communication
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
TLS	Transport Layer Security
VXLAN	virtual extensible local area network
gRPC	Google Remote Procedure Call
MACvlan	Media Access Control virtual local area network
IPvlan	Internet Protocol virtual local area network
OPEX	Operation Expenditure
CAPEX	Capital Expenditure
SLA	Service Level Agreement

ABSTRACT

This thesis discusses Microservice Architecture and how it can facilitate web services scalability using containerization. The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating through standard API. The Microservice Architecture is used to foster separation of components that creates a more effective environment for building and maintaining highly scalable applications. Monolithic applications are made of modules that are all tightly coupled together and have to be developed, deployed and managed as one entity, since they all run as a single OS process. Changes to one part of the application require a redeployment of the whole application, and the lack of hard boundaries between the components, over time, results in the increase of complexity. Further monolithic applications are not designed to run in the cloud without making major redesign and modifications. Scalable cloud native applications on the other hand, may require big changes in the application code and which may not always possible within the timeframes of the fast changing business environments. Monolithic applications present a steep learning curve to new developers and may consume more man-hours before a change introduced by developers is reflected in production. This is mainly due to friction experienced between various development teams. For a scalable architecture every microservice is packaged as a container before deployment to any platform that supports containerized deployments. Each container is uniquely addressable using an IP. Docker architecture is extensible and works with other software development tools to realize a scalable build and deployment system for microservices. This thesis employs Docker Engine, Docker Machine, Docker Compose and Docker Swarm to realize the scalability of the Microservice Architecture. Docker Swarm is a middleware within the orchestration layer that abstracts the complexities arising from the Microservice Architecture. This abstraction simplifies the design and implementation of the microservices and enhances the scalability of the system by eliminating the contention delay and minimizes the coherency delay. From the results obtained by making measurements of container start up delay it was observed that Docker swarm scales linearly as the number of containers in increased. It was also noted that design of Docker Swarm orchestration software are based on Microservice Architecture hence their linear scalability. The main contributing factor of Docker Swarm scalability is the Raft consensus algorithm. This algorithm is the also playing a big role distributed databases. In order to investigate the scalability of the backend services we used the crate database running on cluster of machine managed by Docker swarm. It was observed that Docker swarm really simplifies the scalability of many online web services. This trend was also observed in Continuous Integration and Testing. Docker swarm based orchestration is and will remain to viable candidate for simplifying the scalability of microservices and web services.

CHAPTER ONE: INTRODUCTION

1.1 Background

The evolution of the modern cloud drastically changed the way developers build and deploy applications. In its Top 10 Strategic Technology Trends for 2016 (Gartner Research, 2015), states that "The mesh app and service architecture are what enables delivery of apps and services to the flexible and dynamic environment of the digital mesh. This architecture will serve users' requirements as they vary over time. It brings together 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 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 consortium of large software Vendors and had little backing from the open source community.

1.1.1 The Universal Scalability law

It's been proven that blocking of any kind, anywhere in the system will measurably impact scale due to:

- **Contention**: waiting for queues or shared resources.
- **Coherency (Crosstalk)**: the delay for data to become consistent.

Amdahl's Law tells us that the maximum speedup using *P* processors for a parallelizable fraction *F* of the program is limited by the remaining (1-*F*) fraction of the program that is running serially, on one processor or redundantly on all processors. According a study by (Neil J. Gunther, 2007) , blocking would actually reduce concurrency as a system is scaled and this holds true today. Gunther's Law is famously referred to as the Universal

Scalability Law and is graphically illustrated in figure 1 below. In thesis we shall investigate this law and use this model to improve scalability of web services using OS-Level Virtualization. The no. of processors can be interpreted to mean the number of servers, virtual machines or containers.

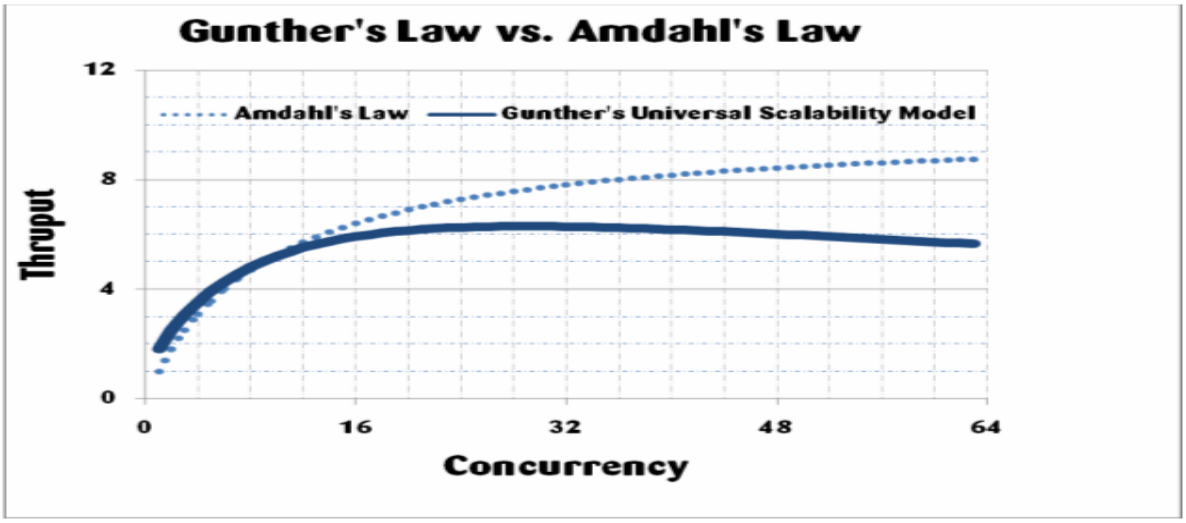


Figure 1: Amdahl's and Gunther Laws governing how throughput varies with increasing no of processors (Gunther, 2007).

1.1.2 Service Oriented Architecture (SOA)

SOA is based on good principles such loose coupling but lacks clear guidelines on how to decompose an application in smaller units. SOA is also dependent on complex communication mechanisms such as SOAP and Enterprise Service Bus (ESB). Iron.io, one of the software vendors using Microservice Architecture (Iron.io White Paper 2015) observed that while this pattern provided a framework for building effective application architectures, its practice has generally been ineffective due to unnecessarily complex abstractions and legacy protocols. Developers would attempt to use SOA to connect a wide range of applications that all spoke a different language, requiring an extra layer for an Enterprise Service Bus. This leads to archaic and costly configurations that cannot keep up as the technology and business landscape evolved.

1.1.3 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 tool based on OS-Level virtualization introduced by a Silicon Valley startup called Docker is proposed as a means to simplify 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 this companies by enabling them to meet digital business challenges. With flexible and fast-evolving architecture for service provisioning based on small team management and industrialized deployments, Microservices can be a fast path to capturing huge value from new technologies, thinking and philosophy.

1.2 Problem Definition.

Monolithic Architecture fails by increasing friction between teams due to centralized code base such as RDBMS

1. The Monolithic Architecture is a tightly coupled system. Every module in entire system is locked together because each component must use the centralized code base.
2. Centralized code bases are a high friction architecture hence the need to decouple and federate so that everyone can make changes independently of each other.

Monolithic applications are made of modules that are all tightly coupled together and have to be developed, deployed and managed as one entity, since they all run as a single OS process. Changes to one part of the application require a redeployment of the whole application, and the lack of hard boundaries between the components, over time, results in the increase of complexity. Further monolithic applications are not designed to run in the cloud without making major redesign and modifications. Scalable cloud native applications on the other hand, may require big changes in the application code and which may not always possible within the timeframes of the fast changing business environments. Monolithic applications present a steep learning curve to new developers and may consume more man-hours before a change introduced by developers is reflected in production. This is mainly due to friction experienced between various development teams.

Adopting Microservice Architecture quickly accelerates innovation while at the same time introducing complexity. The following problems are inherent in Microservice Architecture.

1. Given that every microservice will have its own data store, full consistency after a transaction cannot be achieved and one has to settle for eventual consistency or casual consistency.
2. Microservices introduces many dynamic parts that require constant testing, monitoring and configuration in order to maintain acceptable of quality of service.
3. There is need to choose and implement an inter-process communication mechanism between the microservices based asynchronous messaging.

This thesis seeks to address the problem of limited scalability inherent in monolithic architecture by using OS-Level virtualization or containerization. Though Monolithic systems are simpler to implement, their scalability is hampered by the inability to rapidly and iteratively introduce change to specific functional components. Since it is impossible to scale various functional components of a monolith independently, this architecture is inefficient and costly when web applications are hosted in the cloud. Monolithic architecture is not well suited for cloud environment and agile technologies. The main problem is how to reduce the time it takes for any change to the system to be reflected in

production. There is the need for smaller development teams working on smaller units of software that only encompass one function.

1.3 The Solution to the Problem.

The solution to the problem is can be achieved using two-pronged approach:

1. **Minimize friction**. This can be achieved by decentralizing the database and any other centralized code base by adopting microservices, containerization and DevOps.
2. **Lower risk**. Risk is detrimental in complex systems. Risk can be reduced by continuous testing and continuous delivery.

Microservice Architecture is proposed as means to decompose a large application with several functions into smaller single function units. Microservice Architecture enables independent scaling of different functional units while allowing iterative development cycles to handle the challenges posed by rapidly changing business requirements and environments. Each microservice has its own data store and can flexibly choose a suitable technology stack that provides the required elasticity, responsiveness and resiliency.

Microservice Architecture lends itself to the agile software development methodology and is more appropriate to use the DevOps approach. DevOps ensures that agility is extended to cover both software development and deployment. DevOps reduces the time it takes for change introduced in a system to be reflected in production. Hence DevOps puts a lot of emphasis on continuous and automated deployment. To address this requirement we intend to use OS-Level virtualization mechanism called containerization. Containerization is a light weight OS-level virtualization and is a software abstraction mechanism for making software portable across many platforms. Every software component is handled in a similar manner regardless of its functionality. Every Microservice should be build packaged, shipped and deployed using a container. Containers are more efficient and cost effective than traditional virtual machines hence it is highly recommended mechanism that facilitates elasticity and efficiency of web services.

1.4 Purpose

The main aim of this thesis is to develop and test a scalable Microservice Architecture for web services

Main Research Question

What reference architecture can best serve as the foundation for scalable web services?

From the main research question, we derived the following specific questions:

1. What factors are influencing the adoption of Microservice Architecture?
2. To what extend can containerization enhance 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?

1.5 Scope of the Study

This thesis will be limited to development and testing of scalability of Microservice Architecture using Docker Engine as the container platform and Docker Swarm as the Container Orchestration mechanism running on Linux platform.

1.6 Related Work

According to a study (Jeff Nickoloff, 2016) both Kubernetes and Docker Swarm were tested in clusters of 1000 nodes provided by Amazon Web Services. The study examined at the times it took both orchestration engines to start a new container when their respective clusters were 10 percent, 50 percent, 90 percent, 99 percent, and 100 percent full. According to Nickoloff, Kubernetes had longer start up times at the different percentiles. In contrast, Swarm did not start to suffer until its cluster was 90 percent full. He concluded that the reason for the difference is due to the algorithms and simplicity of thier architecture. He released the test results for third party inspection and reuse. In this thesis we shall use his raw test results to validate the performance of the Swarm orchestration layer using scalability model based on universal scalability law. In this thesis we shall use a workbench tool developed by Docker Inc.

In a study (Joab Jackson, 2016) conducted by Jewell while working for Codenvy Inc, Docker Swarm was embedded into its Che on-demand developer workspace software, after evaluating a number of different container orchestration providers, including Kubernetes. In the evaluation process, Codenvy looked at three essential

criteria: latency and speed of container activation, linear container scalability on physical nodes, and a low configuration footprint. Docker Swarm excelled in all three categories. The results showed that a custom Che software installation into a new account took less than 10 minutes and can scale to support thousands of nodes, or hundreds of thousands of workspace containers.

1.7 Organization of the Thesis

The thesis is organized as follows:

- **Chapter 2** gives background details on OS-level and Hardware-based virtualization and how Containers based on Docker Architecture can be used to realize a scalable Microservice Architecture. We discuss in the details the design of orchestration software including Docker Swarm, Kubernetes, Mesos, Kontena and Hypernetes.
- **Chapter 3** outlines the Methodology employed to realize the objective and address the questions that the study proposes to answer. The Research methodology is based on Design Science while the System Development methodology is Agile using the DevOps principles. The Software development methodology is highly influenced by containerization.
- **Chapter 4** splits the web services into two tiers namely the frontend and the backend. Design of the frontend and backend is discussed with a view of highlighting software patterns that are informing the development of scalable web services. Testing of the Microservice Architecture is done for the frontend, backend and database. In the testing we show how containerization can enhance the scalability of web services.
- **Chapter 5** analyzes of results and recommendations for future work. Given that Containerization, Microservices and DevOps are fast changing technologies, there is need for further investigation of these trends and thinking.
- **Chapter 6** draws conclusions based on our findings of this research based on the objectives of this research**.**

CHAPTER TWO: LITERATURE REVIEW

2.0 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 popular for building web applications (Adrian Cockcroft, 2014), Twitter (Jeremy Cloud, 2013). Microservices is Architectural style that realizes a single Software System as many small individual loosely coupled applications which communicate over a network. Each application has its own software development lifecycle. This decoupling allows many small teams to work on individual applications. All applications 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 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 with 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.

2.1 Definition of Terms

2.1.1 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).

2.1.2 Continuous Delivery

Continuous Delivery (Humble et al, 2010) is a software development discipline that enables on demand deployment of software to any environment. With Continuous 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.

2.1.3 Continuous Integration

Continuous 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).

2.1.4 Configuration Management

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

2.1.5 DevOps

DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal 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 organizational walls, and it promotes a fundamentally different way of solving IT problems (Rackspace, 2015).

2.1.6 OS-Level Virtualization

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

OS-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).

2.1.7 Docker

Docker is an open-source project that automates the packaging, shipping and deployment of applications using containers, by providing 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).

2.1.8 Monolithic Architecture

A monolithic Architecture dictates that an application consist of components that are all tightly coupled together and have to be developed, deployed and managed as one entity, since they all run as a single OS process and scales by replication of all functions on multiple servers (Martin Fowler et al, 2014).

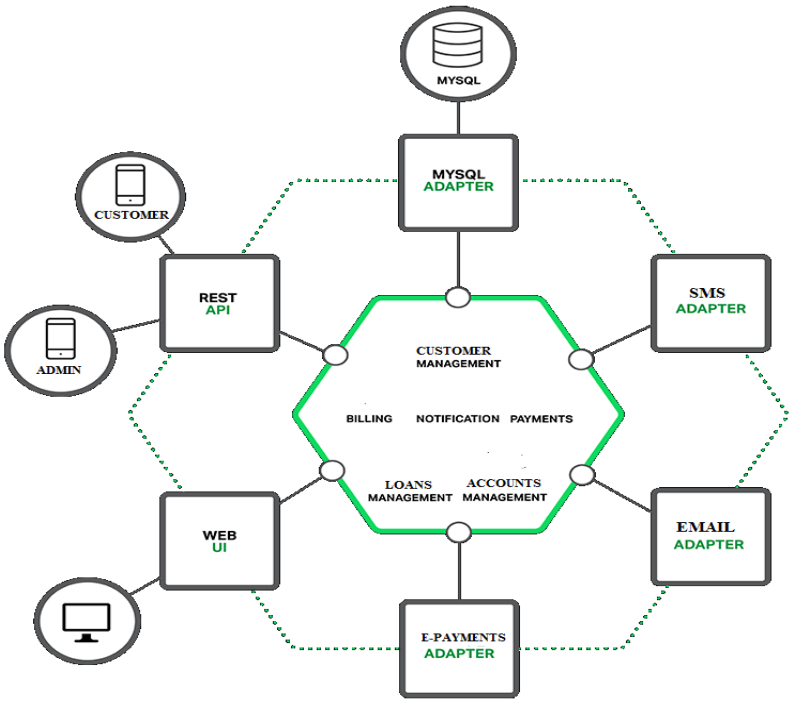


Figure 2: Monolithic architecture showing how various service are lumped together into a single process

2.1.9 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 are able to 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.

2.1.10 Process

An Operating System process is the concrete representation of an software application at runtime. One process always belongs to one application and software application 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.

2.1.11 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.

2.1.12 Software Architecture

According to the Institute of Electrical and Electronics Engineers (IEEE) Software Architecture is **"**the fundamental organization of a system embodied in its components or modules, their relationships to each other, and to the environment, and the principles guiding its design and evolution**".** This definition is fairly generalized and applies to other types systems. In His PhD thesis (Roy T. Fielding 2000) stated that "software architecture is an abstraction of the run-time behavior of a software system and not just a property of the static software source code".

(Wilde et al, 2011; Ian Gorton, 2011) define software architecture as "structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them".

2.2 Principles of the Microservice Architecture

The core principles of the Microservice Architecture are:

1. no components are privileged (privilege);
2. all components communicate in the same simple way (uniform communication);
3. components can be composed from other components (composition)

Because microservices are small they are easier to compose and it matters a great deal less what your implementation language is. In fact, microservices may be completely disposable, as rewriting your functionality is not that much work. Microservices communicate over the network using messages. Is it immaterial to the Microservice Architecture what data format the messages use, or the protocols by which they are transported. Microservices are entirely defined by the messages they accept, and the messages they emit. From the perspective of an individual microservice instance, and from the perspective of the developer writing that microservice, there are only messages arriving, and messages to send. In deployment, that microservice instance may be participating in a request/response configuration, or a publish/subscribe configuration, or any number of variants. The way in which 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 a services at will. This makes scaling, fault tolerance, and continuous delivery practical, and low risk. Naturally you will need some automation to control the large 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. Your default operational action is to add or remove a single microservice instance, and then to verify that the system is still healthy. This is a very low-risk procedure compared to big bang monolith deployments.

2.2.1 Microservice Architectural Constraints

A microservice network can deliver on the principles by meeting a small set of architectural constraints. These are transport independence, pattern matching, and additivity. Transport independence is the ability to move messages from one microservice to another without requiring microservices to know about each other. When one microservice needs to know about another microservice, in order to send it a message, this is a fatal flaw. It breaks the privilege principle, as the receiver is privileged from the perspective of the sender. You are no longer able to compose other microservices over the receiver, without also changing the sender.

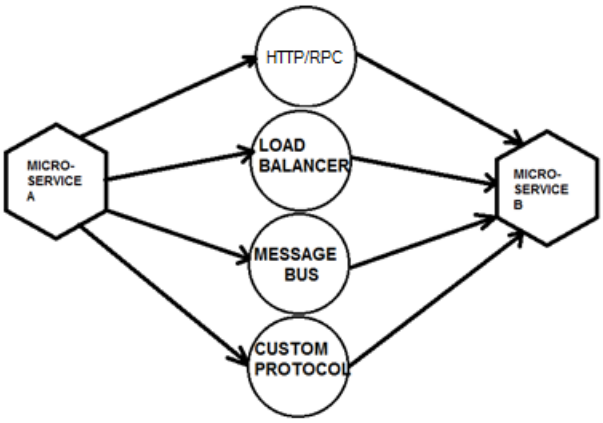


Figure 3: Microservice Mechanisms for enabling loose coupling

Pattern-matching is the ability to route messages based on the data inside the message. This is the capability that lets you dynamically define the network. It allows you to add and remove, on the fly, microservices, and to do so without affecting existing messages or microservices.

Pattern-matching is also subject to varying levels of application. The allocation of separate URL endpoints to separate microservices via a load-balancer matching HTTP

request path is an example of pattern matching. You need pattern-matching that is deep enough to express business requirements, yet simple enough to make composition workable. Hence to some extend one will be required to use service bus to achieve pattern marching that embodies business rules.

The additivity principle embodies the ability to change a system by adding new parts. The essential constraint is that other parts of the system must be immutable. Systems with this characteristic can deliver very complex functionality, and be very complex themselves, and yet maintain low levels of technical debt. Technical debt is a measure of how hard it is to add new functionality to a system. The more technical debt, the more effort it takes. A system that supports additivity is one that can support ongoing unpredictable changes to business requirements.

A Microservice Architecture that relies on the business logic of individual microservices to determine the destination of messages will require changes in both the sender and receiver when adding new functionality. Additivity is stronger if you use intelligent load-balancers, pattern-matching, and dynamic registration of new upstream receivers to shield senders from changes to the set of receivers. You can achieve near perfect additivity using peer-to-peer service discovery.

Microservices can plausibly address the needs of custom enterprise software. By aligning the software architecture of the system more closely with the real intent of the business, software projects can have far more successful outcomes. Real business value can be delivered faster. Microservice systems approach minimum viable product status faster, and thus can be put into production sooner. Once in production they make keep up with changing requirements easier.

Waste and rework, also known as refactoring, is reduced because complexity within each microservice cannot grow to dangerous levels. It is more efficient and easier to write new microservices to handle business change, than to modify old ones. As a software developer, the evolutionary approach to systems building offered by microservices allows you to make a bigger professional impact. It allows you to be successful by making the best use of your time.

2.3 Conceptual design

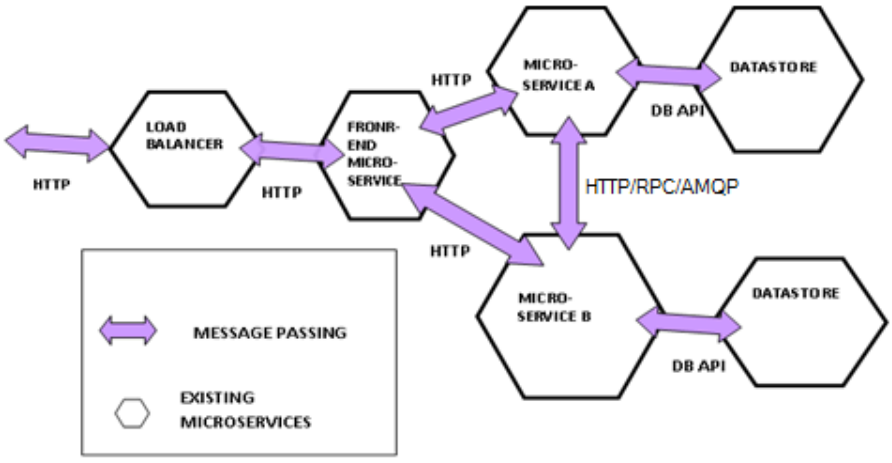


Figure 4: Microservice Architecture Conceptual Model

2.3.1 Software functional components

Software functional component is reasonably large-scale code structure within an application, with a well-defined API, that could potentially be swapped out for another implementation. Microservice Architecture is an extension of 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.

2.4 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 architectural constraints that restricts the roles/features of architectural elements 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 constraints that inform 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 established architectures that have been informing the design of distributed systems and enterprise IT architectures.

2.5 Microservices Architectural Style

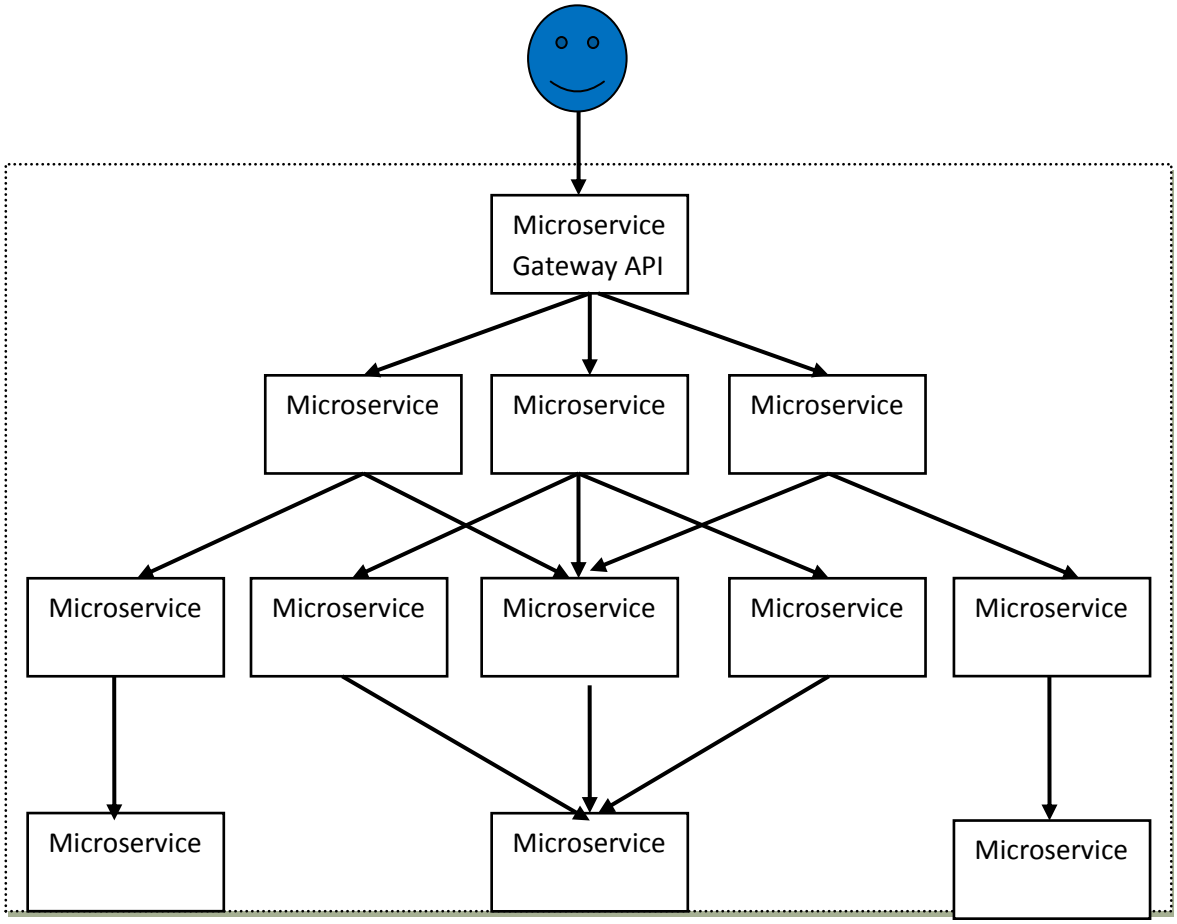


Figure 5: Microservice Hierarchical Tree

Each user request is satisfied by a sequence of services

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

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

2.6 Characteristics of Microservices

2.6.1 Independent Technology Stacks

Each service is implemented on its own technology stacks

- The technology stack can be selected to fit the task at hand
- Teams can also experiment with new technologies within a single Microservices
- No system-wide standardized technology stack also means
    - ❖ No struggle to get your technology introduced to the canon
    - ❖ No piggy-pack dependencies to unnecessary technologies or libraries
    - ❖ It‘s only your own dependency hell you need to struggle with

- Selected technology stacks are often very lightweight
    - ❖ A Microservices is often just a single process that is started via command line.

2.6.2 Independent Scaling

Scalability is the ability of a system, network, or process to handle a growing amount of work in a capable manner 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 increased performance in a manner proportional to resources added”.

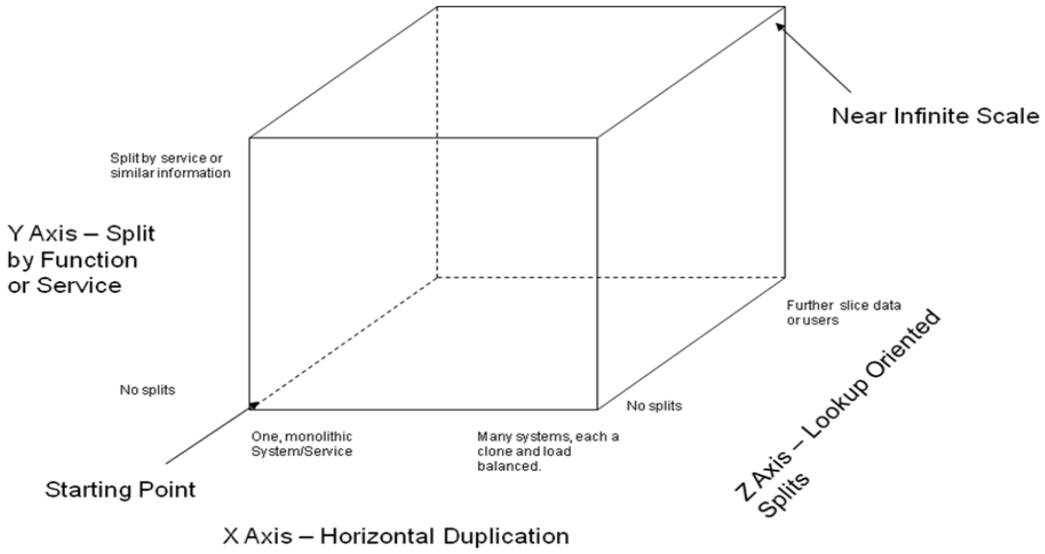


Figure 6: Functional scalability i.e Y-axis scalability

[adopted from Martin L. Abbott et al, 2015].

X-Axis scalability

X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application.

Z-Axis scalability

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

Y-Axis scalability

Unlike X-axis and Z-axis, which consist of running multiple, identical copies of the application, Y-axis axis scaling splits the application into multiple, different services. Each 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 needed
- Parts of the system that do not represent bottlenecks can remain simple and un-scaled

2.6.3 Independent Evolution of Features

Microservices can be extended without affecting other services

- For example, you can deploy a new version of a service without re-deploying the whole system
- You can also go so far 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.

2.6.4 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 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.

2.6.5 Componentization via Services

Interaction mode: share-nothing, cross-process communication

- Independently deployable (with all the benefits)
- Explicit, REST-based public interface
- Sized and designed for replaceability
- Upgrading technologies should not happen big-bang but in an incremental manner.

2.6.6 Favors Cross-Functional Teams

Conway's Law

"Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure" (Conway et al , 1968).

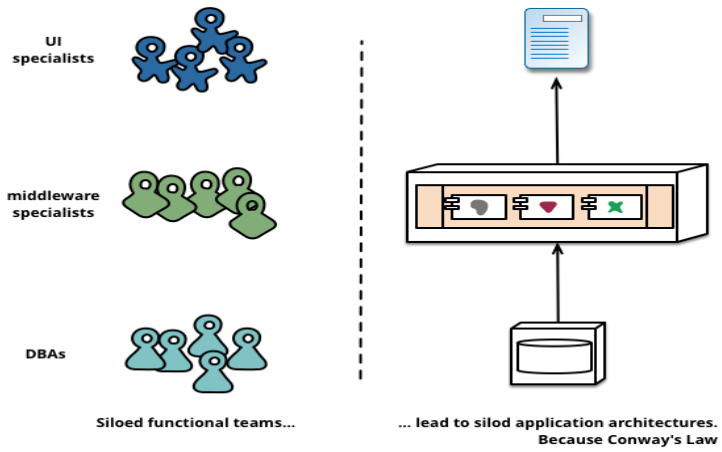


Figure 7: Illustration showing how Conway's law applies to system design in a given organization (Martin Fowler et al , 2014)

The Microservices employs a different approach, splitting up into services organized around business capability. Such services take a broad-stack implementation of software for that business area, including user-interface, persistent storage, and any external collaborations. Consequently the teams are cross-functional, including the full range of skills required for the development of user interface, database, and project management.

.

2.7 Challenges to a Microservice Architecture

Any application architecture that attempts to solve issues of scale does have a number of concerns, given the complex nature of distributed systems. Decoupling an application into independent services means that there are now more moving parts to maintain.

Complex Orchestration

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

Inter-Service Communication

Decoupled services need a reliable, effective way to communicate while not slowing down the whole application. Delivering data over the network introduces latency and potential failure, which can interfere with the user experience. A common approach is to introduce API Gateway to coordinate all communication between users and services.

Data Consistency

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

Maintaining High Availability

Ensuring high availability is a requirement 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 service must then have its own distributed measures implemented to ensure application wide availability.

Testing

While keeping code and dependencies tight means a simpler development environment for specific services, it does introduce challenges with testing as it relates to the entire application. Services will often need to communicate with each other or rely on a data

source or API. Testing one service independently would then require a complete test environment to be effective.

2.8 Microservice Architecture, DevOps and Containers

2.8.1 DevOps is a Prerequisite to Successfully Adopting 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 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 team 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 enviroment 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 greater 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 software lifecycle. DevOps is a prerequisite for being able to successfully adopt Microservice at scale in a given organization (IBM, 2015).

2.9 Cloud Computing

The vast development of cloud computing technology in recent past has substantial impact to 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 technical, economic, and research world. It is used for describing what happens when applications and services hosted in remote data centres such as Amazon, Azure or Cloud Foundry. Actually, 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 consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned 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 enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, 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 composed of five essential characteristics, three service models, and several evolving deployment models.

2.9.1 Characteristics and Benefits of Cloud Computing

One of the important characteristics 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 very short time frame (typically within minutes). Through elasticity, users of cloud computing could avoid 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 result in increase of 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 the 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 automaatically.*

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

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

2.9.2 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 , Function as a Service, DataBase as as Service 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 parts 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 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 platform where they can deploy their applications. The platform provides programming languages, tools and libraries which can be used by the consumers to build and run their own applications.	Elastic Beanstalk, Microsoft Azure.

Docker-based PAAS	This kind of infrastructure is build from 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.

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 starts running. All the complications 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 those functions be executed by the platform.

Iron.io a startup has also introduced IronWorker as a serverless architecture that allows engineers to program machines to execute code in reaction to certain circumstances. It can assist in the process of cleaning up data as it comes in, delivering notifications at scale, sending out emails, or handling mobile check-ins quickly.

IronWorker can be used for a wide range of purposes. IronWorker service and its IronMQ message queue service 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 can work with those as well as Clojure, Go, .NET, PHP, Python, Ruby, and binary code.

Google Cloud Function is a simplified approach for developers to create single-purpose, stand-alone functions that respond to Cloud events without the need to manage a server or runtime environment.

Going serverless requires a slightly different approach to application design. The backend service is broken down into stand-alone functions that perform a single task in response 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 shifts from the backend to the client. It becomes the main orchestrator, calling various functions to perform some action for the user when needed.

Serverless architecture requires very smart clients that know about and talk to a wide range of remote functions. While mobile app developers have had rich frameworks and platforms that allowed them to build complex logic on the client easily, things weren't so simple for web applications. But thanks to rich client-side application frameworks like Angular 2 and a fast HTTP/2 protocol, it is now possible to build complex applications seamlessly into the browser. This will help drive the serverless trend even further.

2.9.3 Deployment Models

In the previous section the most important 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. It is important that 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 available through an Internet connection but this deployment model is 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).

2.10 Virtualization Technologies

The virtualization techniques of interest to our study can be grouped into two categories: Full virtualization and Operating System-level (OS-level) virtualization. The foundation of the Full virtualization is hardware emulation. A host machine provides emulated hardware environments for its guests to run their individual operating systems as if they are running in a real machine. Most of the well-known virtualization solutions belong to those categories, VMware and Kernel Virtual Machine (KVM). Conversely, in a container-based virtualization, the OS kernel of the host machine is shared by the entire host. The host machine isolates 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).

2.10.1 Full Virtualization

Full virtualization, also known as the original virtualization technology, refers to that whole virtual machine that simulates the complete underlying hardware, including processors, physical memory, peripherals, etc. It is not necessary 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 8.

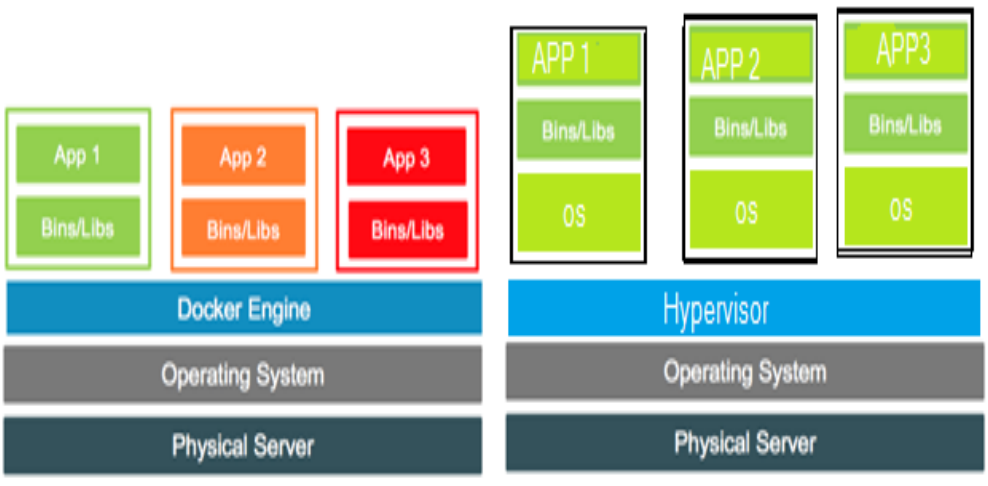


Figure 8: Comparison of OS-Level Virtualization and Full virtualization (Source: Docker Inc.)

2.10.2 Linux Based Containers

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

2.10.3 Choice of Virtualization Platform

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

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

2.10.4 Design Through Abstraction

A system may be composed of many levels of abstraction 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 elements at that level (L. Bass et al, 1998). Within each element 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 architectures continues down to the most basic system elements: those that cannot be decomposed into less abstract elements. 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 is 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 car 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 so as to drive the car.

2.10.5 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 off 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 the concept of virtualization. Containerization though not new concept 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).

2.10.6 Docker Architecture

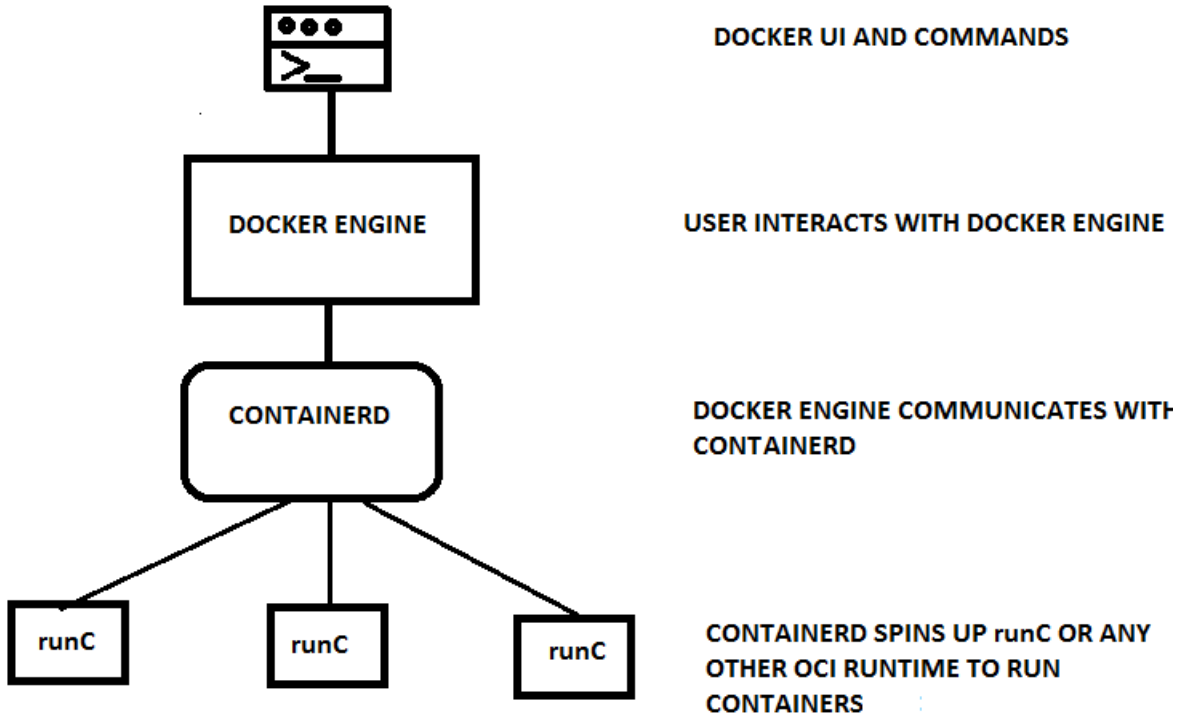


Figure 9: The docker Architecture and design principles

Docker consists of at least four parts: the command line interface (CLI), docker engine , containerd and runC.

Docker API

The Docker daemon has a remote API and this is in fact what the Docker command-line tool uses to communicate with the Docker engine. But because the API is documented and public, it's quite common for external tooling to use the API directly. This enables all manners of tooling, from mapping deployed Docker containers to servers, to automated deployments, to distributed schedulers. As you embrace Docker over time, it's likely that 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 natively supports 64-bit versions of Linux , Mac OS X and Windows due to the Unix underpinnings of these operating systems. To develop with Docker on non-Linux platforms, you will need to leverage virtual machines or remote 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 1.12+ consists of the following features

- **Orchestration:** comes with inbuild orchestration capability using Swarm. The inbuild Swarm is simple to use and enhances container security through 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 you can integrate better with 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 volumes, 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 inspect**: This is a little thing, but really handy if you use the Docker API. Docker inspect now has a Status field, a single consistent value to define a container's state (running, stopped, restarting, etc).

Containerd

This is Docker runtime for managing containers. Containerd improves on parallel container start times so if you need to launch multiple containers as fast as possible. containerd's API is very simple. It is build with gRPC for performance as well as the ability to easily create client libraries. The basic request to start a container via the API is to only provide the path to the OCI bundle and the ID for the container. By keeping the API simple we minimize the changes in the API when new features are added.

RunC

runC is the first implementation of the Open Containers Runtime specification and the default executor bundled with Docker Engine. Future versions of Engine will allow you to specify different executors, thus enabling the ecosystem of alternative execution backend without any changes to Docker itself. By separating out this piece, an ecosystem partner can build their own compliant executor to the specification, and make it available to the user community at any time – without being dependent on the Engine release schedule or wait to be reviewed and merged into the codebase. This loosely coupled design opens up new possibilities. This can facilitate Engine restarts/upgrades without restarting the containers, improving the availability of containers. In addition one is able to restart containerd and your containers will keep running.

Docker works with your operating system to package, ship, and run software. You can think of Docker like a software logistics provider. It is currently available for Linux-based operating and plans are underway to port it to most popular operating systems. Several software vendors at a Docker Conference held on 22nd June 2015 in San

Francisco agreed to form an alliance called Open Container Initiative to push for standard format for the container ecosystem. Declaring a standard container format, and providing reference software for running such a standardized container, is an important step.

Docker is not a programming language, and it is not a framework for building software. Docker is a tool that helps solve common problems installing, removing, upgrading, distributing, trusting, and managing software. Docker is open source, which means that anyone can contribute to it and it has benefited from a variety of perspectives.

It is common for companies to sponsor the development of open source projects. In this case, Docker Inc is the primary sponsor.

2.10.7 Benefits of Containerization .

Containers are Lightweight

Not only is Docker quicker than a traditional VM to spin up, it's more lightweight to move around, and due to its layered filesystem it's much easier and quicker to share changes with others.

Rapid 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 executed there without compatibility issues.

Version control and component reuse – you can track successive versions of a container, inspect differences, or roll-back to previous versions.

Sharing – you can use a remote repository to share your container with others.

Lightweight footprint and minimal overhead – Containerized images are typically very small, which facilitates rapid delivery and reduces the time to deploy new application.

Simplified maintenance – Containerization reduces effort and risk of problems with application dependencies.

2.10.8 Docker Containers and the Cloud Ecosystem

- **Images –** A Docker image is made up of filesystems layered over one another. At the base is a boot filesystem, bootfs, which resembles the typical Linux boot filesystem. An image contains the whole filesystem that will be available to then application, and other metadata, such as the path to the executable that should be executed when the image is run.
- R**egistries** – A Docker Registry is a service that stores your Docker images and facilitates easy sharing of those images between developers and machines. When you build your image, you can either run it on the same machine that you've built it on, or you can *push* (upload) the image to a registry and then *pull* (download) it on another computer and run it there. Some registries are public, allowing anyone to pull images from it, while others are private, only accessible to certain people or machines.
- **Containers** –A running Docker Image is called 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 amount of resources (CPU,RAM, etc.) that are allocated to it.

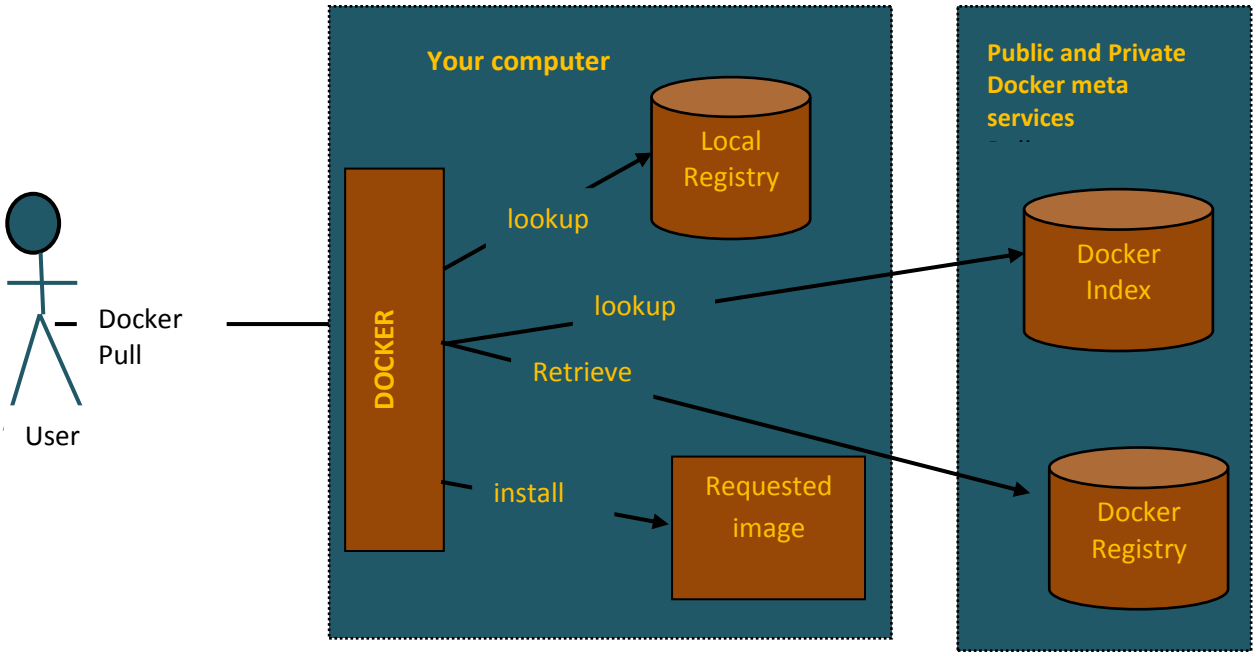


Figure 10: Docker public or private Registries.

2.10.9 Docker Machine

In early 2015, Docker announced the beta release of Docker Machine, a tool that makes it much easier to set up Docker hosts on bare metal, cloud, and virtual machine platforms. The easiest way to install Docker Machine is to visit the GitHub releases page and download the correct binary for your operating system and architecture. Currently, there are versions for 32- and 64-bit versions of Linux, Windows, and Mac OS X.

2.10.10 Open Container Ecosystem

There is a massive community aligning to use Docker mainly developers and system administrators. Like the DevOps movement, this has facilitated better tools by applying code to operations problems. Where there are gaps in the tooling provided 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.

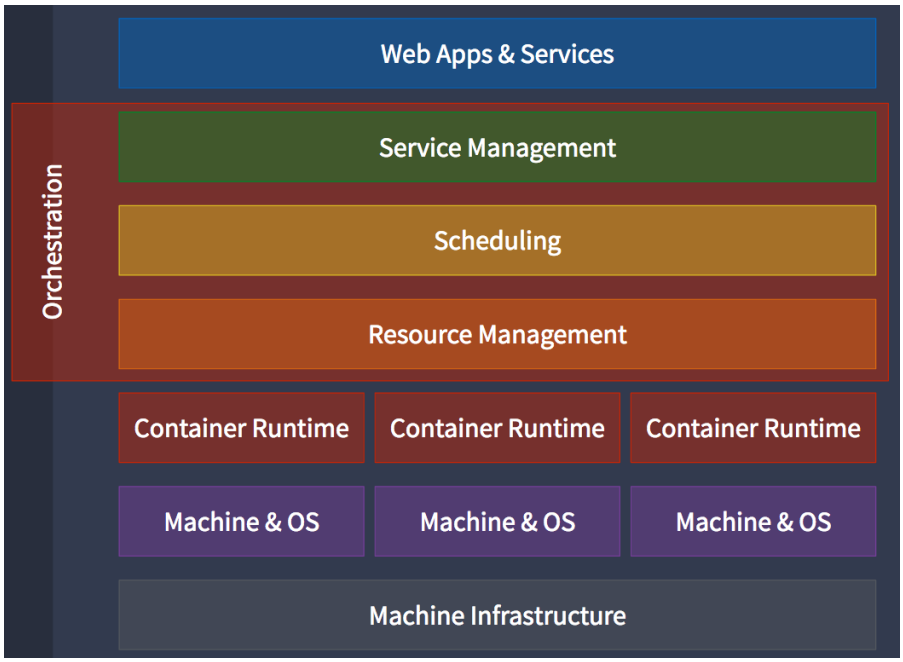


Figure 11: Open Container layered architecture.[Docker Inc]

2.11 Docker Extensions

For Docker to be really useful in supporting distributed application it has to posses the following capabilities.

Portable across environments: You want to be able to define how your application will run in development, and then run it seamlessly in testing, staging and production.

Portable across providers: You want to be able 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.

2.11.1 Scaling up microservices with Docker compose

Docker compose is a tool for defining and running complex applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running. Using Compose is basically a three-step process.

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services 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 Compose will start and run your entire app.

The main 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.

2.11.2 Scaling up microservices with Docker Swarm

Docker Swarm solves one of the fundamental limitations of Docker where the containers could only run on a single Docker host. Docker Swarm is native clustering for Docker. It turns 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 features embedded in the Docker Engine are built using SwarmKit. Engines participating 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 cluster 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 includes the commands 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 you run the Engine in Swarm mode, you orchestrate services.

Node

A node is an instance of the Docker Engine participating in the Swarm cluster. To deploy your application to a Swarm cluster, you submit a service definition to a manager node. The manager node dispatches units of work called tasks to worker nodes. Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the Swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes receive and execute tasks dispatched from manager nodes. By default manager nodes are also worker nodes, but you can configure managers to be manager-only nodes. The agent notifies the manager node of the current state of its assigned tasks so the manager can maintain the desired state.

Services and Tasks

A service is the definition of the tasks to execute on the worker nodes. It is the central structure of the Swarm system and the primary root of user interaction with the Swarm. When you create a service, you specify which container image to use and which commands to execute inside running containers.

In the replicated services model, the Swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state. For global services, the Swarm runs one task for the service on every available node in the cluster.

A task carries a Docker container and the commands to run inside the container. It is the atomic scheduling unit of Swarm. Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale. Once a task is assigned to a node, it cannot move to another node. It can only run on the assigned node or fail.

Load Balancing

The Swarm manager uses ingress load balancing to expose the services you want to make available externally to the Swarm. The Swarm manager can automatically assign the service PublishedPort or you can configure a PublishedPort for the service in the 30000-32767 range.

External components, such as cloud load balancers, can access the service on the PublishedPort of any node in the cluster whether or not the node is currently running the task for the service. All nodes in the Swarm cluster route ingress 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 is a decentralized and highly available group of Docker nodes. Each node is a self-contained orchestration subsystem that has all the inherent capabilities needed to create a pool of common resources to schedule Dockerized services.

A Swarm of Docker nodes creates a programmable topology, enabling the operator to choose which nodes are managers and which are workers. This includes common configurations like distributing managers across multiple availability zones. Because these roles are dynamic, they can be changed at any time through the API or CLI.

Managers are responsible for orchestrating the cluster, serving the Service API, scheduling tasks (containers), addressing containers that have failed health checks and much more. In contrast, worker nodes serve a much simpler function, which is executing the tasks to spawn containers and routing data traffic intended for specific containers. In production environments, we strongly recommend having nodes designated as either “managers” or “workers”. In this mode, managers do not execute containers, thus reducing their workload and attack surface. Separately, one of Swarm mode’s security

advances is that worker nodes do not have access to information in the datastore or the Service API. Worker nodes can only accept work and report on status. Thus, a compromised worker node is limited in the damage it can do to the system.

Managers and workers have different communication requirements in terms of consistency, speed and volume; therefore, they use two distinct communication methods. Raft is used to share data between managers for strong consistency (at the cost of write speed and limited volume) while gossip is used between workers for fast communication and high volume (albeit with only eventual consistency). And communication between managers and workers has separate requirements still. The one thing that they all have in common is that they have encrypted communication by default; mTLS.

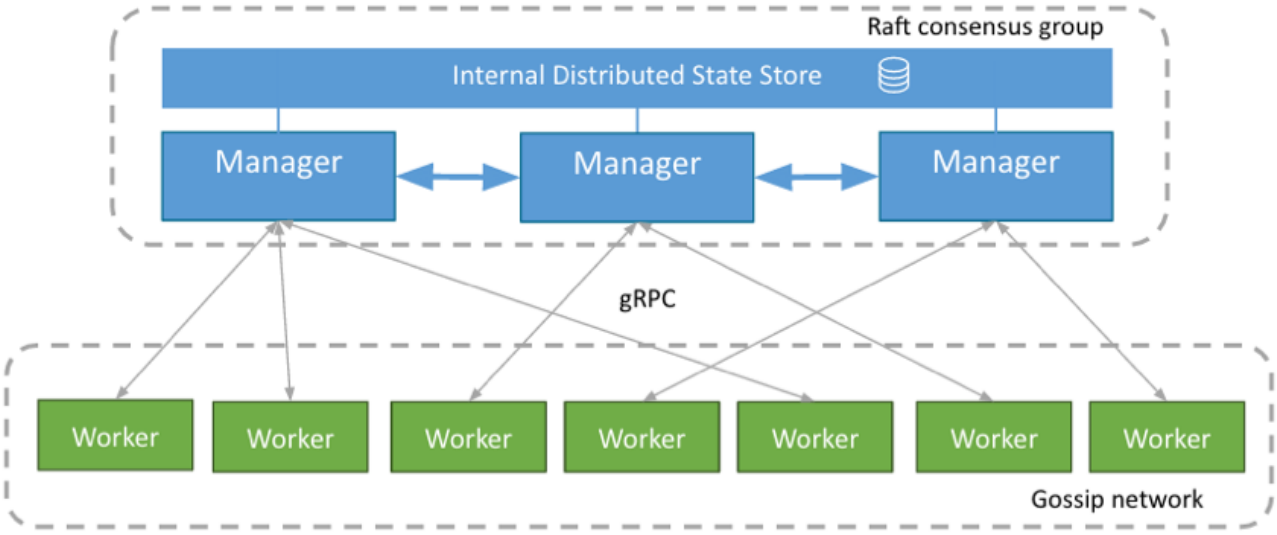


Figure 12: Swarm Orchestration Architecture [Docker Inc]

Raft Consensus algorithm

According to (Diego Ongaro et al , 2014) Raft is simple and understandable cluster consensus protocol. At any given time each server is in one of three states: *leader*, *follower*, or *candidate*. Under normal operation there is exactly one leader and all of the other servers are followers. Followers are passive: they issue no requests on their own but

simply respond to requests 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 node is given the role of manager, it joins a Raft consensus group to share information and perform leader election. The leader is the central authority maintaining the state, which includes lists of nodes, services and tasks across the Swarm in addition to making scheduling decisions. That state is distributed across each manager node through a built-in Raft store. Non-leader managers act as hot spares and forward API requests to the current leader. The system is therefore fault tolerant and highly available.

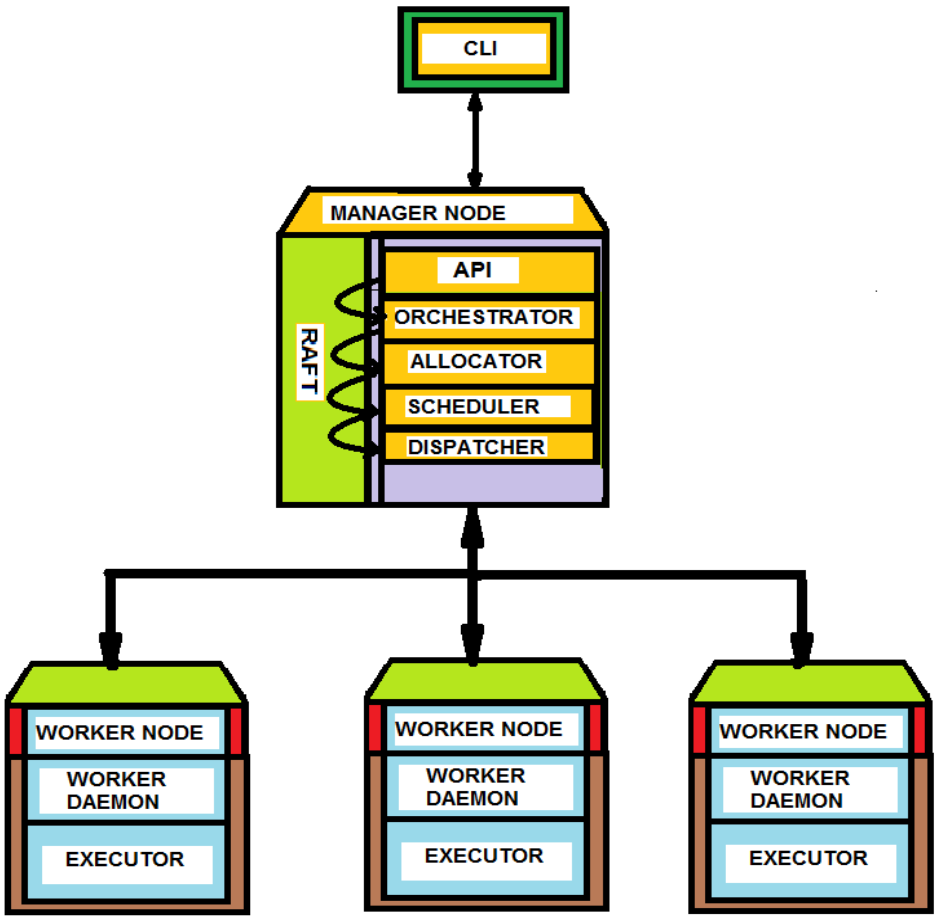
Having an integrated distributed data store allows for many optimizations that could have not been achieved using a generic store – this results in our built-in orchestration system being extremely fast. A major optimization is that the entire Swarm state is kept in-memory resulting in instant reads. This read optimization is highly beneficial to a critical orchestration; reconciling state which is a read-heavy workflow. Typically, a scheduler has to perform hundreds of reads: read the list of nodes, read what other tasks are running on those nodes, and so on. With the read optimization, there is an increase in velocity, which results from removing the need for hundreds of read network round-trips to the external database.

The final optimization is in how efficiently the data is persisted both in terms of size (protocol buffers) and performance (domain specific indexing). We can instantly query from memory the containers that are running on a given machine, the containers that are unhealthy for a specific service, etc.

Manager-Worker Communication

Worker nodes talk to manager nodes using gRPC, a fast protocol that works extremely well in harsh networking conditions, allows communication through internet links (built on HTTP/2) and has built-in versioning (so that different worker nodes running different versions of Engine can talk to the same manager node). Managers send workers sets of tasks to run. Workers send managers the status of the tasks in their assignment set, and a heartbeat so the managers can confirm that the worker is still alive.

As the diagram below illustrates, the dispatcher component of the manager code is what ultimately communicates with workers. It is responsible for dispatching tasks to each worker, while the worker (through it's executor component) is responsible for translating



those tasks into containers and creating them.

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

Based on the diagram above, let's briefly walk through what happens as a Docker service is created and ultimately spawns that set of containers:

Service creation

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

- **Orchestrator** reconciles desired state (as defined by the user) with the actual state (what's currently running on the Swarm). It will pick up the new service created by the API and respond to that by creating a task (assuming in this case, the user requested only one instance of the service)
- **Allocator** allocates resources for tasks. It will notice a brand new service (created by the API) and a new task (created by the orchestrator) and will allocate IP addresses for both.
- **Scheduler** is responsible to assign tasks to worker nodes. It will notice a task with no node assigned and therefore will start scheduling. It tries to find the best match (based on constraints, resources, …) and 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.

Service update

- Users update a service definition through the API (e.g. change from 1 to 3 instances). API accepts and stores.
- Orchestrator reconciles desired vs actual. It will notice that even though the user wants 3 instances, only 1 is running and will respond to that by creating two additional tasks.
- Allocator, Scheduler and Dispatcher will perform the same steps as explained above and the two new tasks will land on workers

2.11.3 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. It was started by Google and now it is used by several Software vendors. Kubernetes 1.4 has added more capabilities such as

- users will be able to set up services that span multiple clusters that can even be hosted across multiple clouds.
- support for stateful applications (think 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 a desired emergent behavior by combining the effects of separate, autonomous entities that collaborate. This is an improved design choice in contrast to a centralized *orchestration system*, which may be easier to construct 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:

1. **the Kubernetes Control Plane-** is highly scalable microservice based and loosely coupled components which controls and manages the whole Kubernetes system, and
2. **worker nodes**-Containerized hosts which run the actual applications you deploy in the Kubernetes cluster.

The components 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 actually runs your containers,
- Kubelet, which talks to the master node and controls Docker on that node,
- Kube Proxy, which proxies and load balances network traffic between your application

  components.

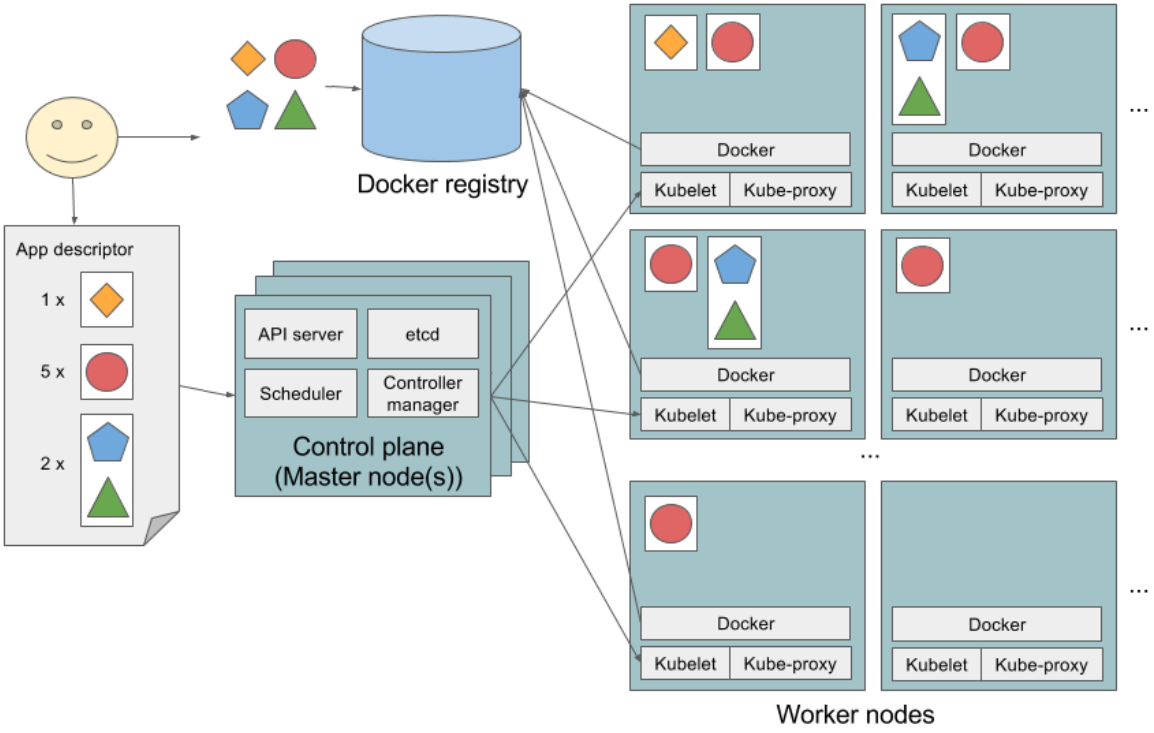


Figure 14: Kubernetes Master-slave design illustrating its microservices and container based architecture (Marko Lukŝa, 2016)

Thanks to the advent 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 experience of running every application inside container, Mirantis has decided to make OpenStack more scalable and manageable by running it using Kubernetes. By packaging OpenStack services so they can be managed by Kubernetes, Mirantis is addressing many of OpenStack's scaling, management and operational challenges, making it, in theory, as scalable as any microservice.

2.11.4 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. Delegating the actual scheduling of tasks 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 since there is no need to add in brand new code to the Mesos master and slaves modules every time a new framework is iterated. Instead, developers can focus on their application and framework

of choice.

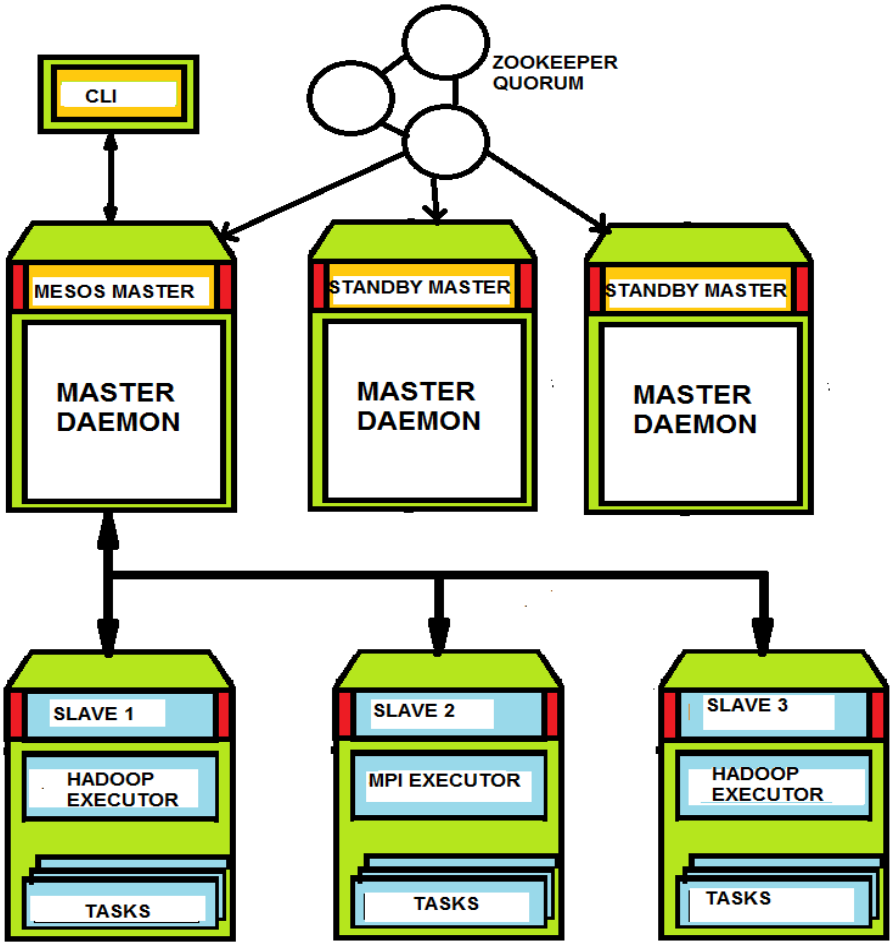


Figure 15: Mesos two level orchestration architecture

2.12 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 comes with varying user

cases and it is difficult to rely on one and omit others. Under 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 various such attempts.

Kontena

Kontena is an open-source system for deploying, managing, scaling and monitoring containerized applications across multiple hosts on any cloud infrastructure. It is primarily targeted for running applications composed of multiple containers, such as elastic, distributed micro-services.

The architecture of Kontena is influenced by both Docker Swarm and Kubernetes, so it had some opportunities to learn from those projects' mistakes and successes. Like Kubernetes, it works at a level of abstraction higher than containers; the primitive building components of Kontena are called services. The other main components 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 a master-slave architecture. Unlike some other container orchestration solutions, this master node doesn't provide any of the underlying processing power, and its purpose is purely for management and to provide audit logging. Kontena's next construct — called host nodes — are what provide the processing power and run the physical Docker containers.

The Host Nodes

Each host node is an actual bare metal or virtual Linux machine, such as an AWS EC2 instance or Digital Ocean Droplet. Each host node is assigned to one — and only one — grid, and communicates with the master node over a secure web socket. Kontena can be configured to automatically provision and assign new nodes to grids when additional capacity is necessary. Host nodes can also be deallocated by Kontena when traffic slows down. All logging and statistics are saved by the master node, as host nodes are also potentially ephemeral.

Kontena CLI

If you have played with docker-compose, you will find configuring Kontena very intuitive. While not a one-to-one match, Kontena's kontena.yml files are very close to docker-compose.yml files. In fact, you can even have a base docker-compose.yml file that you can then extend and reference using a kontena.yml file. Kontena's strengths lies in

- **Easy installation and ease of use:** Kontena works off the shelf—on any public cloud, on-premises or hybrid—and requires minimum effort to install. The platform requires no maintenance and includes automatic updates, which enables developers to spend more time working on what matters the most – their own software and applications.
- **Scalability:** Unlike many other platforms, Kontena is feasible for running even the smallest container workloads and it may be scaled up when needed. This scalability means developers have just the right-sized tool for their unique organization needs.
- **Open source:** Kontena is open source and integrates with other, complementary open source software 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 plugins.
  - **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 DNS challenge.
  - **Load Balancer Sticky Sessions** - Now it's possible to configure load balancer to use sticky sessions.

Kontena proposes a more complete and automated container that includes more functionality — such as scheduling, orchestration, network overlay, load balancing and

secrets management — so that developers using containers do not have to search for and bond together the many individual components on their own for enterprise use.

2.13 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 special requirements of each user's applications.

- **Volume Plugins**, which allow third-party container data management solutions to provide data volumes for containers which operate on data, such as databases, queues and key-value stores and other stateful applications that use the file system.
- **Network Plugins**, which 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.

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 functionality in cool, new and interesting ways.

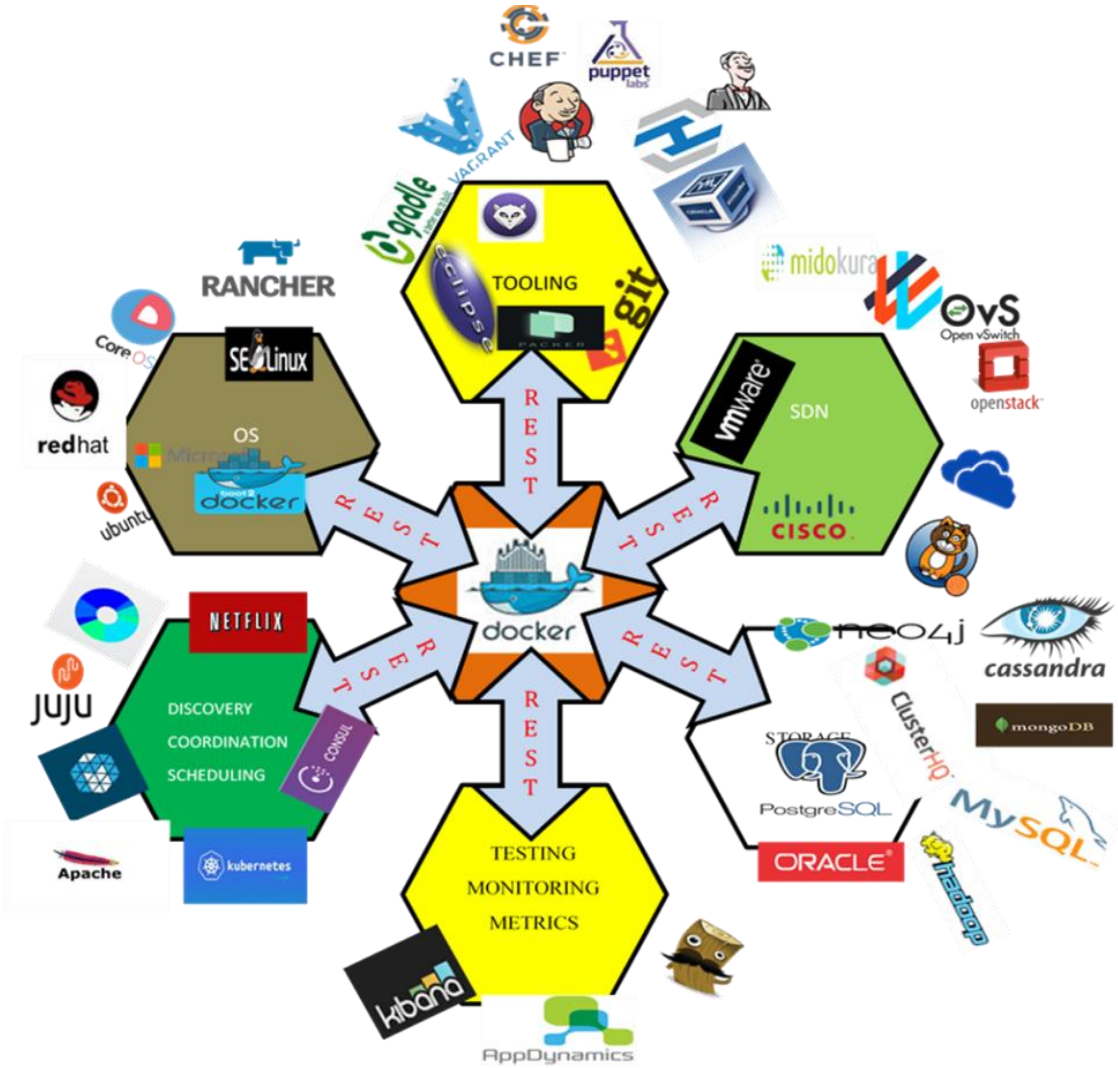


Figure 16: Docker plugin Architecture showing the extensions and interfaces to other systems

2.14 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+ has also good 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 enable 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 are 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.

Docker Volume Plugin Architecture

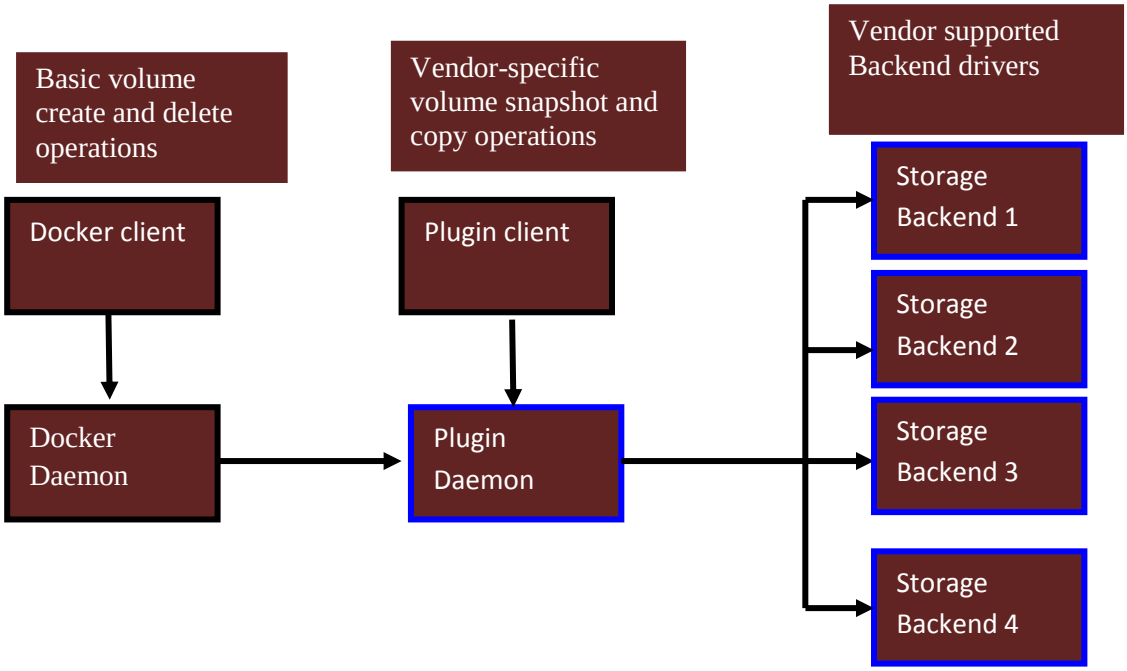


Figure 17: The Docker volumes plugin architecture

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.

2.15 Network Plugins

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

- Overlay
- Underlay

2.15.1 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 translation 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 simpler and eliminates the need for port mapping and are more efficient.

MACvlan

MACvlan allows 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 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. 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 sub interfaces 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 scale is a primary concern, L3 has the potential for massive scale.

2.15.2 Container Networking Standards

There are two container networking specification initiatives namely

- Container Networking Model
- Container Networking Interface

2.15.2.1 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. Libnework 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 provides 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 a number of different constructs

- Endpoint
- Network
- Sandbox

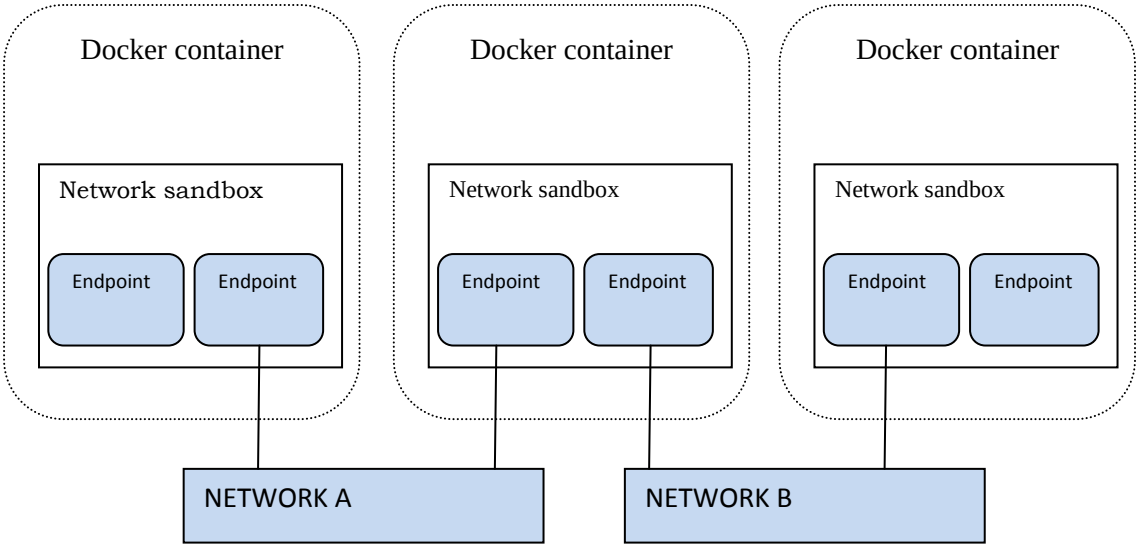


Figure 18: The Container Network Model (CNM)

Endpoint

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

Network

A Network is a uniquely identifiable group of Endpoints that are able to 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 network A and B that are completely isolated.

Sandbox

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

2.15.2.2 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 a number of 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.

2.15.2.3 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 the 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.

2.15.2.4 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.

2.15.3 Network Driver Plugins

2.15.3.1 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).

2.15.3.2 Calico

Calico employs the underlay solution for interconnecting Virtual Machines or Linux Containers. Instead of a vSwitch, Calico employs a vRouter function in each compute 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 exact same principles as the Internet, using Border Gateway Protocol (BGP) at the control plane. With well-known implementations BGP is able to comfortably handle tens of thousands of distinct routes. 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.

2.15.4 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 knowing 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 a number of 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 a number of 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. 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, and it will coordinate with the others to agree on which order state changes happen in.

Zookeeper was originally 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 a number of 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 key use cases. For

example, it exposes an HTTP interface for service discovery, and one of Consul's killer features is that it actually 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 system.

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-off, hence the need for finding an optimal communication framework for microservices.

gRPC

gRPC, is based on *client- server architecture whereby* 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 a heterogenous 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.

CHAPTER THREE: METHODOLOGY

3.0 Introduction

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

3.1.1 Research Methodology

In this research, a research methodology called Design Science Research Methodology (DSRM) is employed. DSRM is about solving problems by introducing artifacts in a context. The artifact 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 definition & analysis (evaluation of current practice)
2) Defining 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 problem)
6) Communication.

3.1.2 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 on the 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 with respect to scalability which provides motivation for the need of a new platform. Common technology used, architecture components and functionality gaps will be acknowledged as well. Step (1) will be covered by chapter 1 and 2 in this report.

3.1.3 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 literature study. This phase is

necessary to illustrate the inadequacy of solutions in the market in achieving our project goals. We will carry out literature study in the topics of web application architecture, system level virtualization and scalability.

3.1.4 Artifact Design & Development

Subsequent step (3) in DSRM is about artifact design and development. Based on the findings of steps 1, 2 and 3, we will construct an architecture design of our platform in this chapter 4. We will first study what architectural design principles to implement and accordingly, its design specification and requirement.

3.1.5 Artifact Demonstration

Chapter 4 in this report corresponds with Step (4) which deals with artifact demonstration. We will find a suitable context in order to demonstrate the feasibility of our architecture design by means of a prototype. The case selection will be based on consideration from literature.

3.1.6 Artifact Evaluation

In addition, we will select a tool to test the principles of the architecture design. After constructing the prototype using appropriate tools and components, the prototype is validated using one microservice. In the last section is Chapter (6) which corresponds to Step (6) in DSRM research phase, we conclude this thesis with discussion of results then point out recommendations for future researchers.

3.1.7 System Development Methodology

The system development methodology used will be Agile development methodology. This methodology is preferred because it is iterative and incremental and so allows the assessment of the projects direction throughout the development lifecycle. The methodology also supports addition of features to a system incrementally which is important especially when dealing with web applications. The overall model of the system will be created and the features added incrementally until all the objectives are met.

Agile development methodology will follow iteratively through the following steps:

- Requirements analysis
- Architecture and design (features design).
- Development and evaluation of the features.

The steps will be carried out as a continuous and iterative process as defined in the agile development methodology.

3.1.8 Architectural Design

This stage is involves understanding of the problem by studying an organizational setting and identify the different services offered by an organization and its partners. Each service is designed independently and integrated to form a complete system. The components of the system include

- Messaging services
- Discovery services
- Service directory
- Data security service

3.2 System Development Process

Most of the software development processes (also known as the software life cycle), are being replaced by more agile methodologies.

3.2.1 DevOps

At the core of DevOps thinking is the realization that software development (Dev) and operations teams need to collaborate closely to enable organizations shorten the time it takes to transform developed software into running services

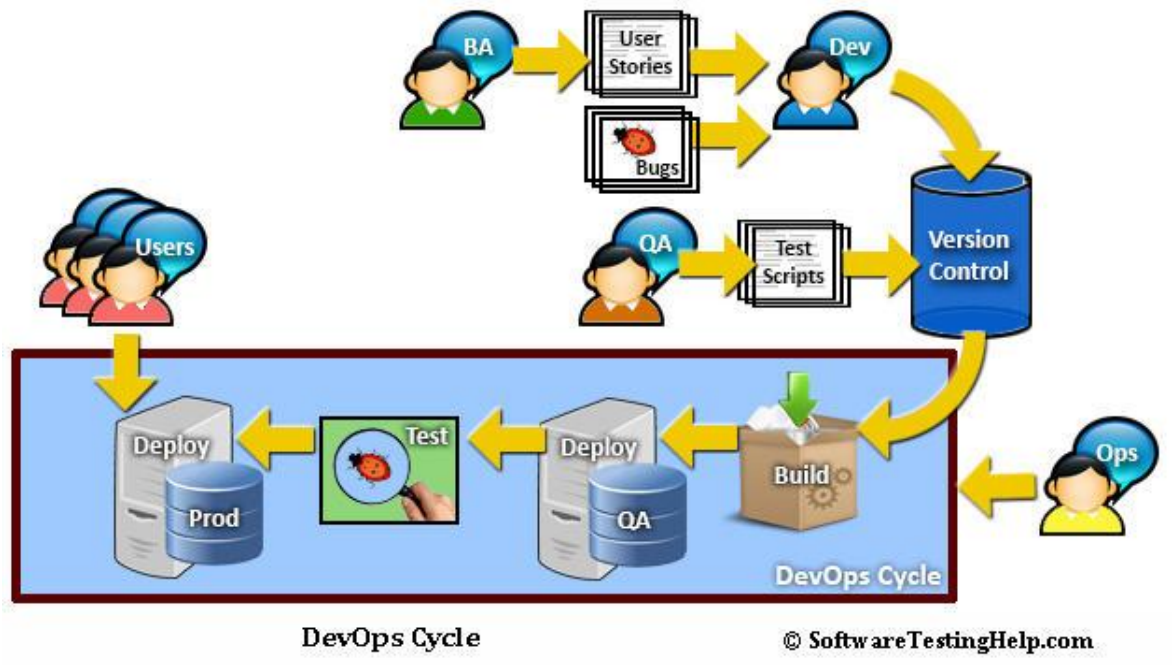


Figure 19: A DevOps Based Software Development Cycle [Adoted from Software Testing]

DevOps is associated with agile and goes further to employ continuous integration, configuration management, virtualization, and cloud computing. It puts emphasis on automated and repeatable allocation and configuration of execution environments. In an effort to break the barriers between development and operations, speed up delivery, and enhance the supplied solutions, DevOps employs some approaches that are well suited with use of virtualized infrastructure.

DevOps is an extension of agile development methodology that emphasizes on the automation of packaging, deployment and testing using appropriate tools. The specific goals of a DevOps approach span the entire delivery pipeline including improved deployment frequency, which can lead to faster time to market, lower failure rate of new releases, shortened lead time between fixes, and faster mean time to recovery in the event of a new release crashing or otherwise disabling the current system. Simple processes become increasingly programmable and dynamic, using a DevOps approach, which aims to maximize the predictability, efficiency, security, and maintainability of operational processes. Very often, automation supports this objective.

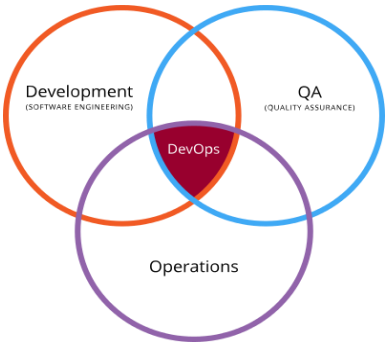


Figure 20: DevOps is an inter displinary approach to software development.

3.2.2 The Software Deployment Pipeline

Continuous delivery introduces the concept of a deployment pipeline, also referred to as the build pipeline. A deployment pipeline represents the technical implementation of the process for getting software from version control into your production environment.

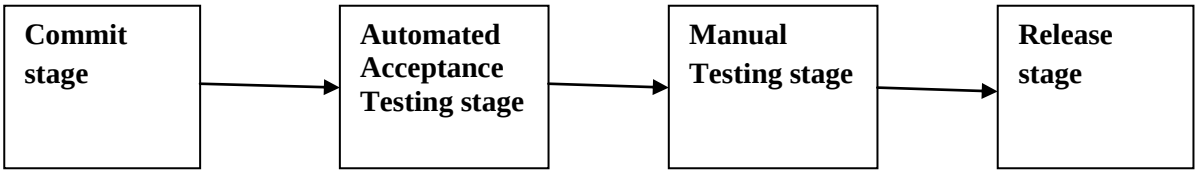


Figure 21: Stages of a deployment pipeline

Commit stage

The main stakeholder of this phase is the development team as it provides feedback about broken code and finds "bugs." The job of this stage is to compile the code, run tests, perform code analysis, and prepare the distribution.

Automated acceptance test stage

Functional and nonfunctional requirements are met by running automated tests.

Manual test stage

Verifies that the system is actually usable in a test environment. Usually, this stage involves QA personnel to verify requirements on the level of user stories or use cases.

Release stage

Either delivers the software to the end user as a packaged distribution or deploys it to the production environment.

The following tasks can be automated using build tool

- Compiling the code

- Running unit and integration tests
- Performing static code analysis and generating test coverage
- Creating the distribution
- Provisioning the target environment
- Deploying the deliverable
- Performing smoke and automated functional tests

Docker has been chosen as a tool to facilitate faster and simplified deployments because it gives a useful abstraction layer on which to build, deploy and manage a software system.

3.2.3 Faster Deployments

If you are pushing an application or release candidates between environments frequently, as is typical in a Continuous Delivery world, this is a huge win. This is because only the application code and binaries that has changed in that build need to be shipped onto your servers, even though you are benefiting from pushing a lightweight 'virtual machine' with all of the repeatability benefits that that brings. If your Continuous Integration server is pushing out hundreds of release candidates into environments per day, this quickly becomes a win.

3.2.4 Docker Support for continuous Deployment Pipeline

Continuous deployment is a set of automation practices that reduces lead time and improves the reliability, quality, and overhead of software releases. Implementing continuous deployment requires some work, but it has a very positive impact on a project. By establishing a sound deployment pipeline and keeping all of your environments—from development to test to production—as similar as possible, you can drastically reduce risks in the development process and make innovation, experimentation, and sustained productivity easier to achieve.

With such an array of environments, repeatability again becomes important. Efficiency in shipping the containers between environments also becomes much an issue. Docker to a large extend simplifies this, making moving images through a pipeline easy and fast.

3.3 Continuous Integration

3.3.1 Version Control System

A tool that manages and tracks different versions of software or other content is referred to generically as a version control system (VCS) or a source code manager (SCM). The aim is to develop and maintain a repository of content, provide access to historical editions of each datum, and record all changes in a log. In this thesis, the term *version control system* (VCS) is used to refer generically to any form of revision control (Scott Chacon et al, 2010**).**

Git is a powerful, flexible, and low-overhead version control tool that makes collaborative development a pleasure. Linus Torvalds designed Git to initially support the development of the Linux Kernel. Currently Git used for diverse range of projects and works well across all operating systems. For example two or more individuals can write a book to completion without ever meeting using Git. For main characteristics that make Git suitable for scalable development you are referred to a book titled version control with Git (Jon Loeliger, 2009).

3.3.2 CI server

Continuous Integration, in its simplest form, involves a tool that monitors your version control system for changes. Whenever a change is detected, this tool automatically compiles and tests your application.

If something goes wrong, the tool immediately notifies the developers so that they can fix the issue immediately. Continuous Integration can also help you keep tabs on the health of your code base, automatically monitoring code quality and code coverage metrics, and helping keep technical debt down and maintenance costs low. The publicly-visible code quality metrics can also encourage developers to take pride in the quality of their code and strive to improve it. Combined with automated end-to-end acceptance tests, CI can also act as a communication tool, publishing a clear picture of the current state of development efforts. And it can simplify and accelerate delivery by helping you automate

the deployment process, letting you deploy the latest version of your application either automatically or as a one-click process.

The CI/CD workflow for most CI servers is as described below

- Developer commits changed code to a repository
- Repository notifies CI server via a webhook of the change
- CI server initiates a CI/CD build, executing the following:
    - ❖ Spin up a new container for the CI/CD build
    - ❖ CI server prepares the container based on specifications provided in a Dockerfile that exists in the repository (if provided)
    - ❖ Upon success, CI server creates a Docker image based on the final state of the CI/CD build container
    - ❖ CI server then pushes the Docker image to Docker Hub in the designated repo with the generated tag.
- Upon pushing the changed image to Docker Hub, the new image is then deployed into a full-topology test environment (this can be automatically deployed when the new image is created or manually deployed).
- Functional/Integration tests are then executed automatically upon successful deployment.
- Software versions ready for user acceptance will be deployed manually for smoke testing by users.
- Upon successful testing, the immutable Docker images that have been tested can then be deployed to any production environment running the Docker Engine.

3.3.3 Build Management

3.3.3.1 Maven

Maven is more than a project management tool since it encompasses a project object model, a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle.

Maven is based on the central concept of a build life cycle. The process for building and distributing a particular artifact or project is clearly defined. For developers to use Maven, they must learn a small set of commands that enable them to build any Maven project. The Maven Project Object Model (POM) ensures that the project is built correctly.

3.3.3.2 Gradle

Microservice Architecture proposes multiple programming languages, each of which is best suited to implement a specific problem domain. This makes Gradle the best tool for Microservices polyglot programming which gives developers the freedom to choose the best programming tools for the job at hand. Compared to other tools, Gradle build scripts are declarative and readable. Writing code in Groovy instead of XML significantly cuts down the size of a build script and is far more readable.

As shown in figure 28 Gradle has combined the best techniques from older tools such as Ant, Ivy, Maven, Gant to produce a better software build automation too.

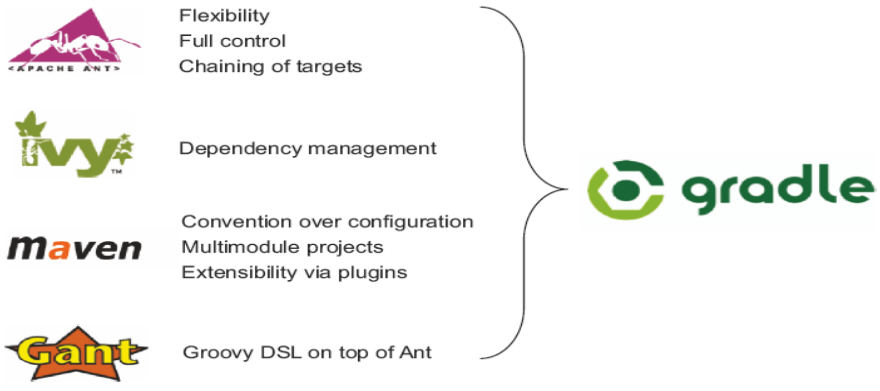


Figure 22: Gradle combines the best features from other build tools.

Scalable Builds

Web applications with several microservices are becoming a reality. Building and testing minor code changes can be cumbersome. What you need is a tool that's capable of rebuilding the parts of your software that actually changed. It reliably determines for you the tasks need to be skipped, built, or partially rebuilt. Because your build clearly defines the dependencies between microservices, Gradle takes care of rebuilding only the necessary parts.

Gradle command forks a daemon process, which not only executes your build, but also keeps running in the background. Subsequent build invocations will piggyback on the existing daemon process to avoid the startup costs.

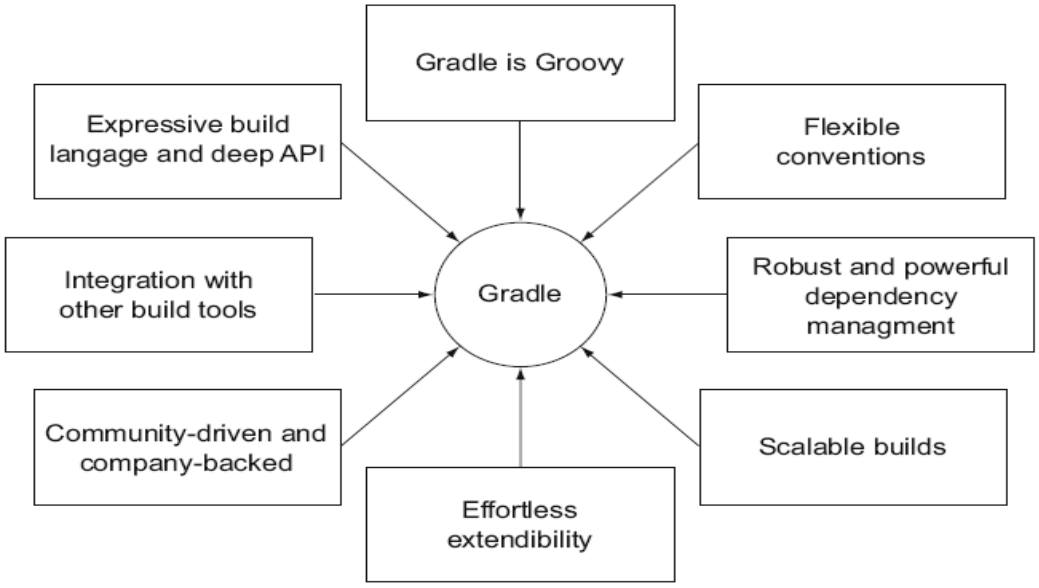


Figure 23: Gradle key features

3.4 System Testing

Due to introduction of many dynamic parts after adopting Microservice Architecture, Software testing should be given a lot of attention and support. This should be supplemented with monitoring tools to foster visibility into the system.

3.4.1 Unit Testing

A *unit test* in microservice is just like in other systems, can be understood as running the smallest piece of testable software to determine whether it behaves as expected. This testing is where we hope to catch most of the bugs in the system and it is the most frequently executed testing compared to other kinds of tests.

Reasons for unit testing

To give a developer very fast feedback about whether the implemented piece of code is good in isolation. This test is also critical in the code refactoring activities where small-scoped tests can timely help ensure quality of the code restructuring as we go.

3.4.2 Integration testing

Integration testing is a testing method which collects relevant modules together and verifies they collaborate as expected, by exercising communication paths among them to detect any incorrect assumptions each module should interact with other modules under test. In Microservice Architectures this testing is typically used to verify interactions between the layers of integration code and any external components that they integrate to. The external components could be data stores, other microservices and so on.

Another important external integration test is the test against communication between two microservices. Very often a proxy component is used to encapsulate message passing between two remote services, marshalling requests and responses from and to the modules are actually process the request. Containerization based on Docker is an important tool in ensuring that test data in one environment can be used in another environment.

3.4.3 Automated Microservices Testing

Testing your code is an important activity of the software development lifecycle. It ensures the quality of your software by checking that it works as expected. In this study we chosen to use Gradle build management tool because it integrates with a wide range of Java and Groovy unit testing frameworks.

3.5 Scalability Experiments

In order to measure the performance and scalability of the Docker Swarm and Kubernetes orchestration mechanisms, the test bench was set up on Amazon Web Services Cloud . In each case the container start delay times were measured.

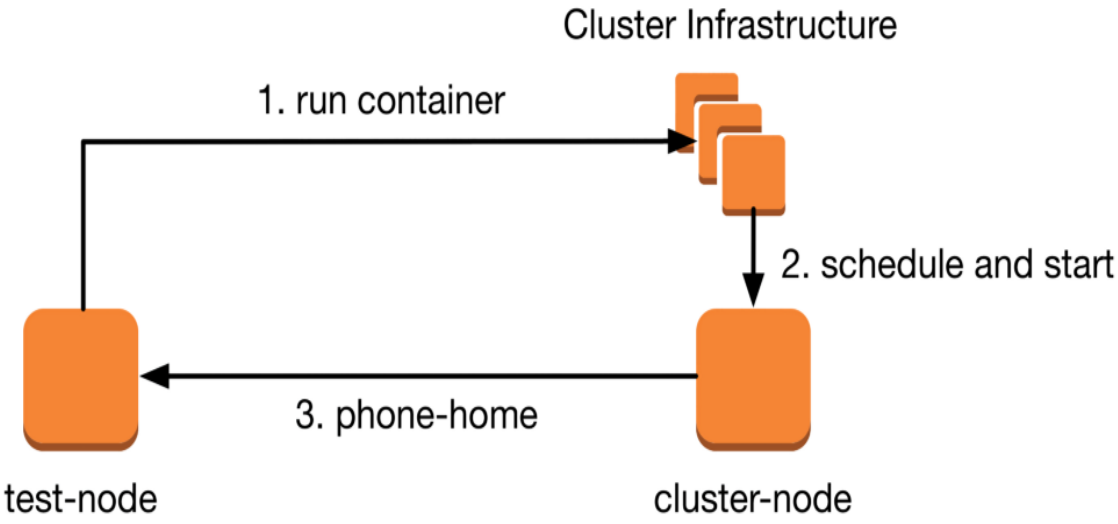


Figure 24: Test Setup used to measure container start up times on Amazon Web Services (Jeff Nickoloff , 2016)

Both Kubernetes and Swarm were tested in clusters of 1000 nodes on Amazon Web Services.

We repeated this experiment on premise setup using five nodes running on Intel core 3 8GB RAM. Our results are as captured by the benchmarking tool as shown in appendix B. The test cluster consisted of five machines created using Docker machine using the VirtualBox Driver. Docker Swarm was used to create the cluster of five machines. Two of the machines were designated as cluster managers wherby one was the cluster leader and the other manager was on standby mode to take-over in the event that the leader failed. The whole process of virtual machine creation was automed using

3.0 Data analysis

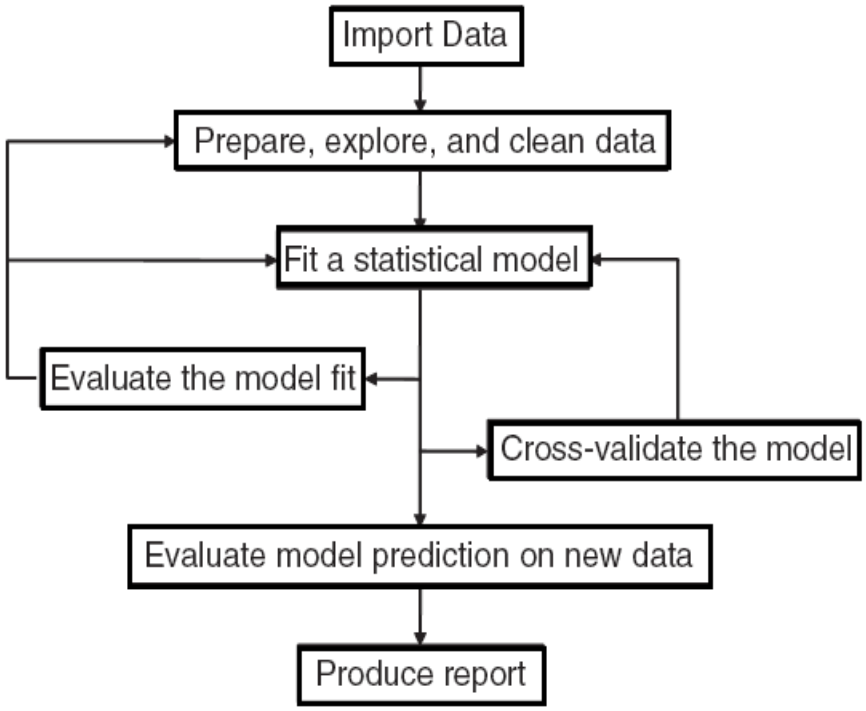


Figure 25: Data analysis steps using R

3.6 Scalability Model

Scalability can be defined as a mathematical function, a relationship between independent and dependent variables (input and output). This is the type of formal definition you need to model and analyze scalability.

The most important part of understanding such a scalability model is choosing the correct variables to describe the way systems really operate. According to (Neil J. Gunther,) definition, *work* is the driving factor of scalability. Useful ways to think about work include, to mention a few,

- Units of work (requests).
- The rate of requests over time (arrival rate).
- The number of units of work in a system at a time (concurrency).
- The number of customers, users, or driver processes sending requests.

Neil Gunther's Universal Scalability Law (USL) provides a formal definition of scalability, and a conceptual framework for understanding, evaluating, comparing, and

improving scalability. It does this by modeling the effects of linear speedup, contention delay, and coherency delay due to crosstalk.

An ideal system of size 1 achieves some amount λ of throughput X, in completed requests per second. Because the system is ideal, the throughput doubles at size N=2, and so on. This is perfect linear scaling:

$$X(N) = \frac{\lambda N}{1} \quad \text{..... Equation 1}$$

The λ parameter defines the slope of the line. I call it the *coefficient of performance*. It's how fast the system performs in the special case when there's no contention or crosstalk penalty. Here are two ideal systems, with λ of 1800 and 800, respectively.

Contention appears in most systems at some point, for example as a final stage in scatter-gather processing when assembling multiple intermediate results into a single output. As parallelization increases, contention becomes the limiting factor. This is codified in Amdahl's Law, which states that the maximum speedup possible is the reciprocal of the serialized (non-parallelizable) portion of the work. If I add a term to the denominator expressing the serialized fraction of the work, and multiply it by the coefficient of serialization σ, it becomes Amdahl's Law:

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1)} \quad \text{..... Equation 2}$$

The last bit is the crosstalk penalty, also called the consistency or coherency penalty. Crosstalk potentially happens between each pair of workers in the system (threads, CPUs, servers, containers, virtual machines etc). You probably remember that the number of edges in a fully connected directed graph is n(n − 1). The USL adds a term to represent the amount of crosstalk, multiplied by the coefficient κ:

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \kappa N(N-1)} \quad \text{..... Equation 3}$$

According to Little's law, the mean number of requests resident in a system (N) is equal to the throughput (X) times the mean response time (R).

$N = XR$..... Equation 4

This relationship is valid for stable systems, in which all requests eventually complete. If you use Little's Law to solve the USL for response time as a function of concurrency, the result is a quadratic function:

$$R(N) = \frac{1 + \sigma(N-1) + \kappa N(N-1)}{\lambda}$$ ..... Equation 5

Determination of the constants using regression

The R software was used to test the scalability model and using this software it was possible to determine the constants lambda (λ), sigma(σ) and kamma(κ) using regression.

Because the USL is a *model*, it can assist you extrapolate system behaviour under load beyond what you can observe. The USL's maxima predicts the system's highest throughput, so it's a way to assess a system's capacity. It can help you get a better idea of how close you are to the system's maximum capacity.

The USL has a few nice properties that make it suitable for this type of capacity planning:

- It's uses "black box" approach in whichs data that's usually easy to get.
- Gathering data and using regression to analyze it is also easy.
- USL is a simple model, without complex mathematics.
- USL is highly intuitive when compared to most other approaches.

3.7 System Evaluation

The system will be evaluated using one microservice application. Based on this microservice demonstration will be carried out on how to build the microservice and package it appropriately for unit and integration Testing. Finally the microservice will be packaged as Docker container. Using this container a demonstration illustrating scalable Microservice Architecture will be conducted.

Quantitative Scalability Experimental Measurements (QSEM) an approach to quantitatively evaluate the scalability of Web-based applications and other distributed systems. The analysis uses readily-obtained data from straightforward measurements of

throughput at different numbers of microservices or nodes. The results provide an understanding of the application's scalability that makes it possible to extrapolate behavior to higher numbers of microservices or nodes with confidence. The QSEM method consists of seven steps.

1. Identify critical Use Cases

2. Select representative scalability scenarios

3. **Determine scalability requirements**

4. **Plan measurement studies**

5. **Perform measurements**

6. **Evaluate data**

7. **Present results**

CHAPTER FOUR : MICROSERVICE SYSTEM DESIGN, IMPLEMENTATION AND TESTING

4.1 Principles of the Microservice Design

The following design specification will guide the development and implementation of a scalable Microservice Architecture.

1. A microservice is responsible for one single capability
2. A microservice is individually deployable
3. A microservice consists of one or more processes
4. A microservice owns its own data store

This list of characteristics should help you recognize a well-formed microservice when you see one, and it will also help you scope and implement your own. By incorporating these design specifications, you'll be on your way to getting the very best from your microservices and producing *a composable, scalable, and resilient system* as a result.

Throughout this chapter, I'll be showing how these specifications should drive the design and development of microservices. Now let's look briefly at each specification in turn.

4.1.1 Responsible for One Single Capability

The statement "do one thing and do it well" has guided the design of Unix Modules and has proved very successful. This principle was borrowed by Object Oriented Programming and renamed Single Responsibility Principle (Martin C. Roberts et al, 2006). A microservice should implement exactly one capability. That way the microservice will have to change only when there is a change to that capability. Furthermore, we should strive to have the microservice fully implement the capability, so that only one microservice has to change when the capability is changed. There are two types of capabilities in a microservice system:

A business capability is something the system does that contributes to purpose of the system, like keeping track of users' shopping preferences or calculating prices. A very

good way to tease apart which separate business capabilities a system has is to use Domain Driven Design.

A technical capability is one that several other microservices need to use to integrate to some third-party system for instance. Technical capabilities are not the main drivers for breaking down a system to microservices. They are only identified when you find several business-capability microservices needing the same technical capability.

4.1.2 Individually Deployable

A microservice should be *individually deployable*. You should be able to deploy a change in a microservice to the production environment without deploying (or touching) any other part of your system. The other microservices in the system should continue running and working during the deployment of the changed microservice and continue running once the new version is deployed.

Being able to deploy each microservice individually is important for several reasons. For one, in a microservice system, there are many microservices, and each one will collaborate with several others. At the same time, development work is done on all or many of the microservices in parallel. If we had to deploy all or groups of them in lock step, managing the deployments would quickly become unwieldy, typically resulting in infrequent and big, risky deployments. This is something we very much want to avoid.

Instead, we want to be able to deploy small changes to each microservice very frequently, resulting in small, low-risk deployments. To be able to deploy a single microservice while the rest of the system continues to function, the build process must be set up with this in mind: Each microservice has to be built into separate artifacts or packages.

The deployment process must also be set up to support deploying microservices individually while other microservices continue running. For instance, you might use a rolling deployment process where the microservice is deployed to one server at a time, in order to reduce downtime.

The fact that we want to deploy microservices individually affects the way they interact. Changes to the interface of a microservice usually must be backwards compatible so that other existing microservices can continue to collaborate with the new version the same way they did with the old. Furthermore, the way microservices interact must be robust in the sense that each microservice must expect other services to fail once in a while and must continue working as best it can. One instance of microservice failing – for instance, because of a short period of downtime during deployment – must not result in other microservices failing, only in reduced functionality or in slightly longer processing time.

4.1.3 Consisting of One or More Processes

A microservice must run in a separate process, or indeed in separate processes, if it's to remain as independent of other microservices in the same system as possible. The same is true if a microservice is to remain individually deployable. This is achied through containerization.

4.1.4 Owns its Own Data Store

A microservice owns the data store where it stores the data it needs. This is another consequence of wanting the scope of a microservice to be a complete capability. For most business capabilities, some data storage is needed.

The fact that each microservice owns its own data store makes it possible to use different database technologies for different microservices depending on the needs of each microservice. One microservice for example, one might use MySQL Server to store product information, whereas the Product Pricing Microservice might store each product prices in Redis, and the Recommendations Microservice might use an ElasticSearch index to provide recommendations. The database technology chosen for a microservice is part of the implementation and is hidden from the view of other microservices.

This approach allows each microservice to use whichever database is best suited for the job, which can also lead to benefits in terms of development time, performance, and scalability. But one consequence of a microservice owning its own data store is that you can swap out one database for another later on.

4.2 Microservice Design Patterns

Several design patterns for microservices have emerged but three are becoming popular.

4.2.1 The Aggregator Pattern

The most simplistic pattern used with microservices is the aggregator pattern (). It is already well known from the Enterprise Integration pattern catalog and has proven to be useful outside Microservice Architecture. The primary goal of this pattern is to act as a special filter that receives a stream of responses from service calls and identifies or recognizes the responses that are correlated.

Once all the responses have been collected, the aggregator correlates them and publishes a single response to the client for further processing. In its most basic form, aggregator is a simple, single-page application (e.g., JavaScript, Angular 2) that invokes multiple services to achieve the functionality required by a certain use case.

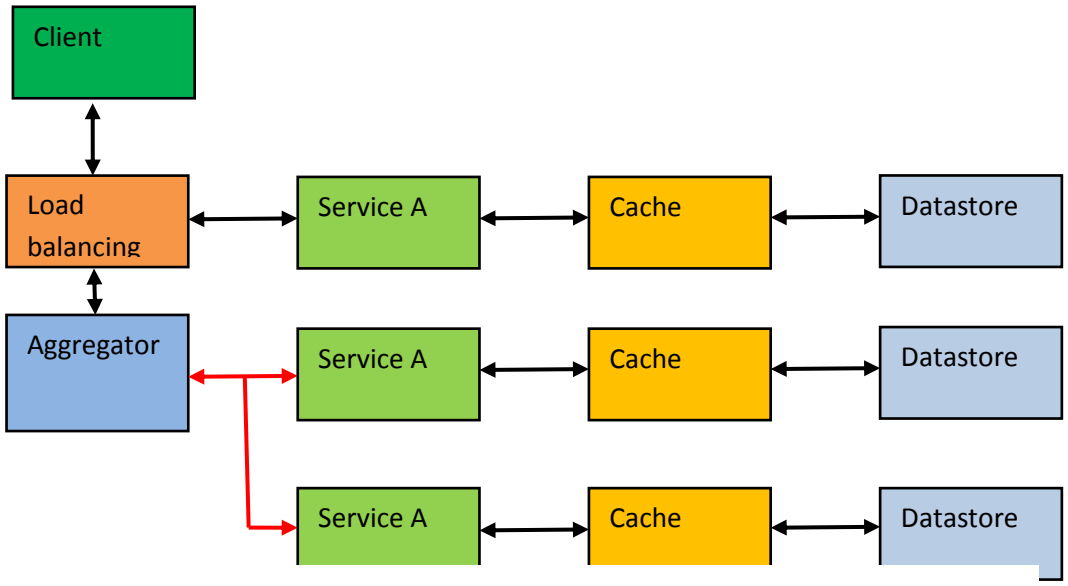


Figure 26: The Aggregator Design Pattern

4.2.2 Proxy Pattern

The proxy pattern allows you to provide additional interfaces to services by creating a wrapper service as the proxy(Fig. 22). The wrapper service can add additional functionality to the service of interest without changing its code. The proxy may be a simple pass-through proxy, in which case it just delegates the request to one of the

proxied services. It is usually called a smart proxy when additional logic is happening inside the proxy service. The applicable logic varies in complexity and can range from simple logging to adding a transaction.

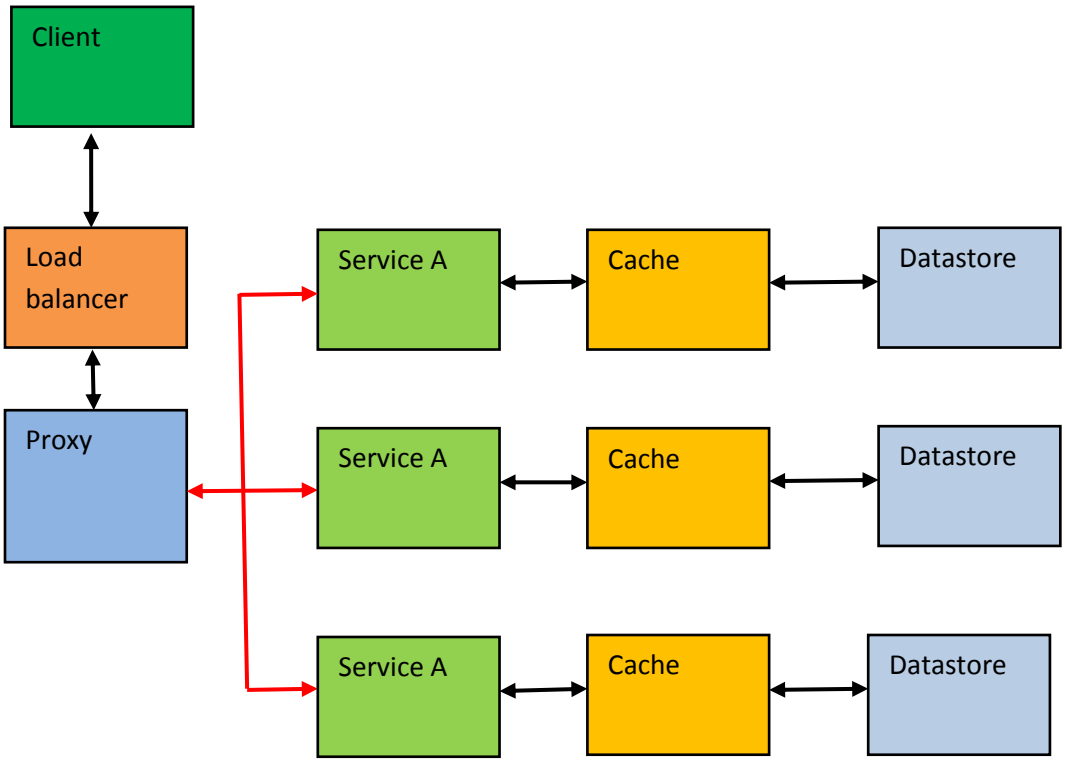


Figure 27:The Proxy Pattern

4.2.3 Pipeline Pattern

In more complex scenarios, a single request triggers a complete series of steps to be executed. In this case, the number of services that have to be called for a single response is larger than one. Using a pipeline of services allows the execution of different operations on the incoming request (Fig. 23). A pipeline can be triggered synchronously or asynchronously, although the processing steps are most likely synchronous and rely on each other. But if the services are using synchronous requests, the client will have to wait for the last step in the pipeline to be finished.

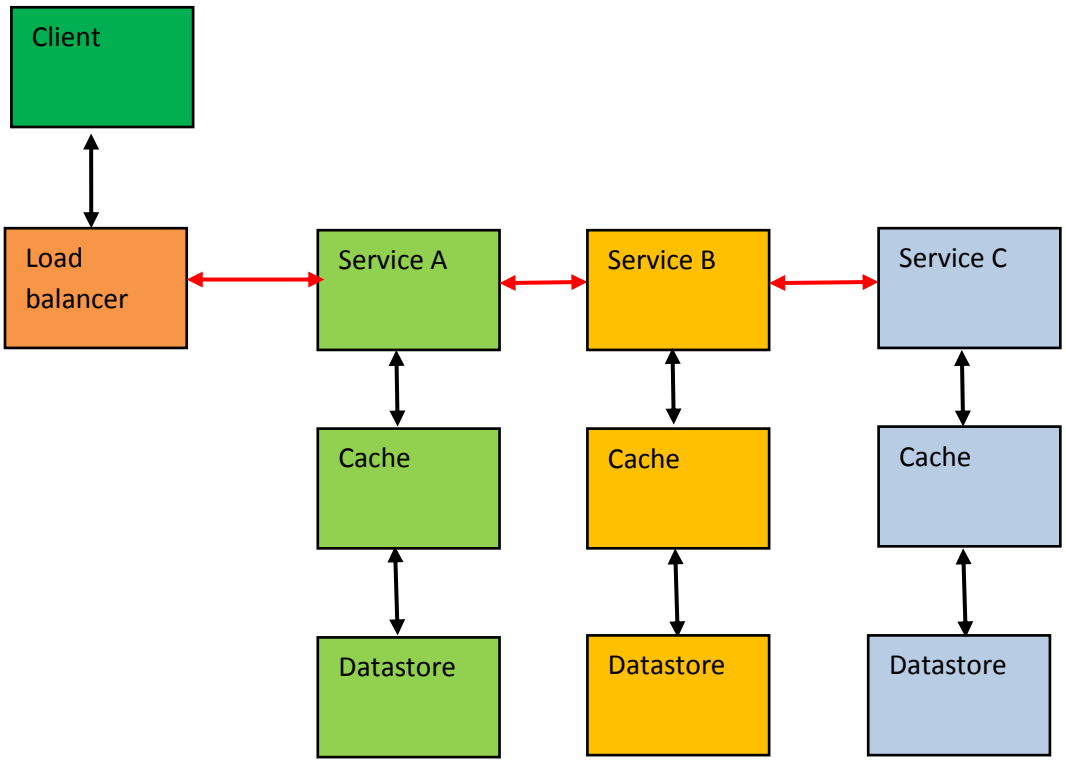


Figure 28: The Pipeline Pattern

4.3 Scalable Microservice Design

Many web application development frameworks have evolved to support concurrent programming techniques. In this thesis we have chosen to use Sails due to its in build capabilities to simplify web application development through auto-configuration and other features as explained in the following section.

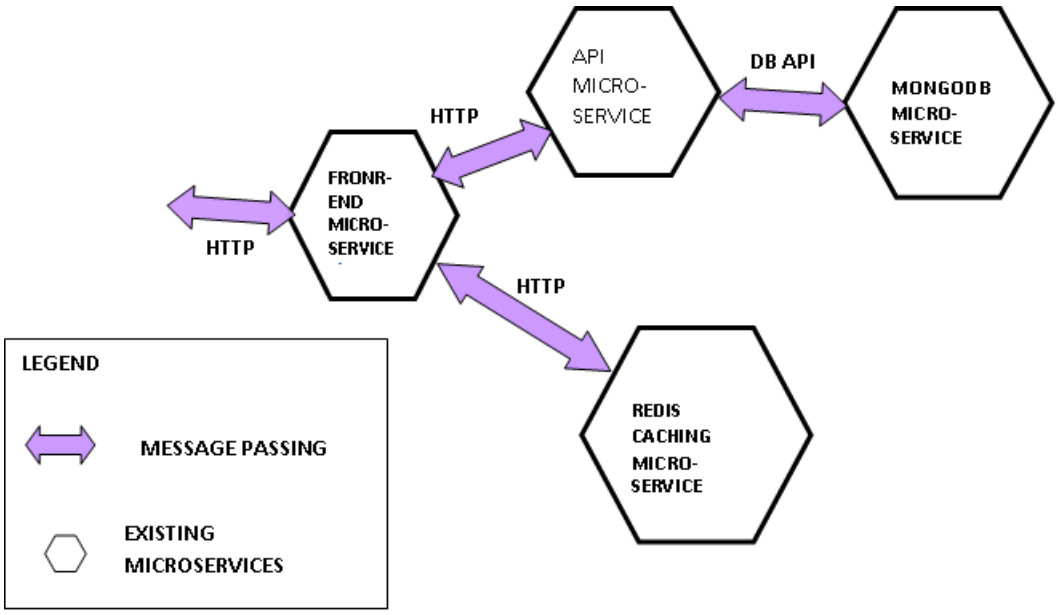


Figure 29: Microservice Architecture implementation

4.4 Front-End Microservice Architecture

The user interface is currently a subject under very serious discussion and research with many frameworks being development to improve on responsiveness, resiliency and scalability of the frontend to web applications. In this section we look at various front end tooling and frameworks that work together with many backends hence their suitability for polygot programming. The web user interface is dominated by JavaScript that run with browsers. However this trend is shifting to have Javacript run natively on mobile phones without the need for browsers. Therefore JavaScript remains the common denominator for any user interface for web applicatios.

4.4.1 JavaScript

Since 2005 after the Ajax revolution, Javacript is a language of choice for running web applications in browsers (e.g. Gmail and Google Maps). JavaScript comes with its own limitations and frameworks such Angular, Google Web ToolKit (GWT) have been engineered with workarounds to circumvent this limitations. GWT for examples employs Java to develop the rich user interface while Angular 2 uses Typescript. The code in either Java or Typescript which are statically typed languages capable of catching must errors at compile time is the transcompiled into JavaScript the lingua franca for many

browsers. TypeScript is a superset of JavaScript but like Java it allows you to define new types. Declaring variables with types rather than the generic *var* opens the door to new tooling support, which you will find to be a great productivity enhancer. TypeScript comes with a static code analyzer, and as you enter code in your TypeScript-aware IDE (Eclipse, WebStorm/IntelliJ Idea, Visual Studio Code, Sublime Text, etc.) you're guided by context sensitive help suggesting the available methods in the object or types of the function argument. If you accidentally use an incorrect type, the IDE will highlight the erroneous code.

4.4.2 Node.JS

Node is predominantly a platform for JavaScript applications. All JavaScript based web frameworks are build on Node. Such frameworks include Express, Sails, Angular 2 and react.js just to mention a few. There are several JavaScript Libraries such jQuery, Bootstrap, Knockout that requires Node to run.

Node is a platform build on build on Chrome's JavaScript runtime for easily building scalable network application that can run on resource constrained devices. Node uses V8, a virtual machine that powers chrome, for server-side programming. V8 has high performance because it cuts out the middleware because it compiles straight into native machine code instead of using JVM or an interpreter.

4.4.3 Angular 2

Angular 2 is a second generation of lightweight JavaScript based frameworks that can support writing of large, maintainable and extensible web applications. Angular 2 is an open source web application framework that offers unique tools for web application developers due to a well supported code base, vibrant community, and rich ecosystem of web components for extension.

Angular 2 has the following features that promises a more scalable architecture for web applications and services.

Mobiles Fast- Inbuilt support for mobile features such as touch events and smartphone memory constraints.

New standards support-Support for future browsers has been taken care off

Performance-JavaScript was not initially intended for writing sophisticated applications and neither did browsers. Hence Angular 2 introduces workarounds to alleviate these shortcomings.

Web components- Angular 2 introduces extensibility and Modularity via use of web components

Dependency Injection-supports a popular re-usability mechanisms that are used in Enterprise web application frameworks such as Spring and .NET.

Routing-Routing is a system, method or convention for getting your user to the right place in your application based on the URL they request.

Angular 2 is written in Typescript, a super-Script of JavaScript that adds optional sophisticated languages features to JavaScript. Use of Typescript instead of Pure old JavaScript enhances Code Maintenance and makes detection of Bugs easier during compile time**.**

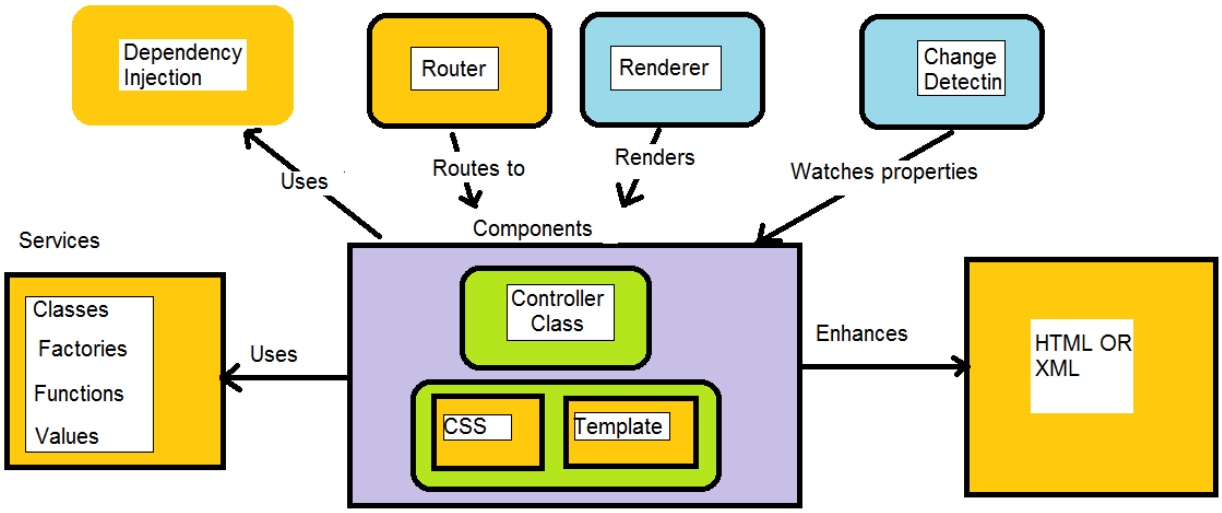


Figure 30: Angular 2 Architecture

4.4.4 Angular 2 Architecture

In addition, Angular 2 includes several in build services:

Dependency Injection: This is a way to create and manage the services included in Angular or which you write. One you have defined a service, the dependency injection system is responsible for finding it when needed by a component and making it available where and when you need it. Dependency Injection is a design pattern that inverts the way of creating objects your code depends on. Instead of explicitly creating object instances (such as with new) the framework will create and inject them into your code.

Router: This is responsible for associating URL paths with parts of your application. So, when the user requests http://www.myapp.com/api/product/{id}, the application will display the product page view whose id is 123 and when a user requests http://www.myapp.com/offer/{id} the application will display the offers page relating to a given product id.

Renderer: As a developer you don't often deal directly with the renderer but it is responsible for rendering your component's view to the user while hooking up the code defined in the component's controller class.

Keeping the rendering engine in a separate module allows third-party vendors to replace the default DOM renderer with one that targets non browser-based platforms. For example, this allows reusing the application code across devices, with the UI renderers for mobile devices that use native components. This means that Angular 2 component code is decoupled from the actual runtime environment, which gives you more flexibility since the code isn't necessarily tied to a particular environment (browser, mobile, server, web worker, etc). A custom Angular 2 2+ renderer is already implemented in the NativeScript framework, which serves as a bridge between JavaScript and native iOS and Android UI components. With NativeScript you can reuse the component's code by just replacing the HTML in the template with XML. Another custom UI renderer allows to use Angular 2 with React Native, which is an alternative way of creating native (not hybrid) UI's for iOS and Android.

Change detection: This is responsible for watching your application for changes caused by the user entering data, information arriving from a database request or when some other event occurs. When it detects a change, it kicks of any needed updates or follow on

actions. This is what allows you to type information into an Angular 2 form and have it instantly update on another part of the page.

4.5 Backend Microservice Architecture

Most software houses, open source communities and research institutions are converging on common software designs that are scalable. It is becoming clear that polygot programming is being embraced. However new programming languages are emerging particularly suitable for the clouding computing or cloud native. In this design we identified two critical barriers to scalability as contention and crosstalk. Contention degrades scalability because parts of the work can't be parallelized and queue up, so speedup is limited. Crosstalk introduces a coherency penalty as workers (threads, CPUs, Containers, Virtual Machines, Servers etc) communicate to share and synchronize mutable state. To address these two issues using employ software abstraction techniques based on containerization. This abstraction layer is added between you application layer and operating system/Machine layer as illustrated in fig. This layer is called Orchestration. Many web application development frameworks have evolved to support concurrent programming techniques. In this following section we discuss some common software patterns that are informing the evolution of Microservice Architecture.

4.5.1 Concurrency programming Models

4.5.1.1 Reactive Extensions

Reactive Extensions is a library that was developed by Microsoft to make it easy to work with streams of events and data. In a way, a time-variant value is by itself a stream of events; each value change is a type of event that we subscribe to and updates the values that depend on it.

Rx makes it easy to work with the streams of events by abstracting them as **observable sequences,** which are also the way Rx represents the time-variant values. *Observable* means that you as a user can observe the values it carries, and *sequence* means an order exists to what's carried. Rx was architected by Erik Meijer and Brian Beckman and drew its inspiration from the functional programming style. In Rx a stream is represented by

observables that you can create from events, tasks, collections, or create by yourself from another source.

Using Rx you can query the observables with **LINQ** operators and control the concurrency with **schedulers;** that's why Rx is often defined as

Rx = Observables + LINQ + Schedulers.

Rx makes the events handling code simpler and expressive by using declarative operations (in LINQ style) to create queries over a single sequence of events. Rx also provides methods that are called combinators (combining operations) that allow joining different sequences of events to handle patterns of event occurrence or correlation between them. There are more than 600 operations (with overloads) in the Rx library—each one encapsulates a recurring event processing code that otherwise you'd have to write yourself.

Observables

Observables are how the time-variant values (which we defined as observable sequences) are implemented in Rx. They represent the "push" model in which new data is pushed (or notified) to the observers. In one sentence observables are defined as the source of the events (or notifications) or, if you prefer, the publishers of a stream of data. And the push model means that instead of having the observers fetch data from the source and always checking if there's a new data that wasn't already taken (the "pull" model), the data is delivered to the observers when it's available.

4.5.1.2 AKKA

AKKA is a domain-neutral concurrency toolkit designed for the purposes of building scalable, fault-tolerant applications on the JVM and .NET Common Language Runtime (CLR). AKKA is written in Scala (Rob Williamson et al 2015) and is usable from both JVM and has recently been ported to .NET Runtime. It's primary goal is to make the achievement of performance, reliability, and scalability simpler. AKKA is based on the Actor Model allowing software developers to just focus on how to most efficiently implement solutions. Actors let the programmer just focus on getting the work done; the

system provides means outside the code for scaling it when the demand curve grows (or shifts). Akka Toolkit can be integrated with Spring, .NET, Camel and ZeroMQ. Akka provides more tools that we can use, including non-blocking IO, interaction with Akka deployments on remote hosts, distributed transactions, finite-state-machines, fault tolerance, performance tuning, and others.

Actor

An actor within the actor model encapsulates three key concepts namely storage, processing and communication.

Communication: One of the key aspects of the actor model is the underlying means of communication between actors. An actor has a mailbox with an associated address, whenever it receives a message, this is added to the end of the queue. Actors are able to communicate with absolutely any actor within the actor system or even in other actor systems.

Behaviour

In order to handle any messages, actors have specific behaviour associated with them. Actors are then scheduled to process messages in the queue at any time in the future. In order to ensure the safety of any state within an actor, the messages are processed one by one in first in, first-out order until the mailbox is empty.

Storage

Actors are also capable of storing some data within the confines of the actor boundary. Any data within this actor is unable to be accessed by anything else outside of this actor. This is one of the key components makes applications to scale out not only past thread boundaries but also across network boundaries.

4.5.1.3 Futures

This refers to a computation that may or may not have yet finished. This means that you can start a computation that's expected to take a while—because it's processor-intensive or because it calls a web service—and not have it block the current computation.

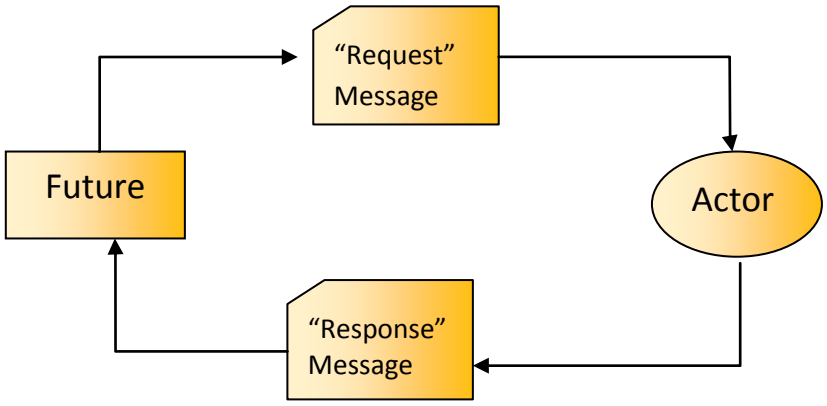


Figure 31: Non-blocking interplay between actors and futures

Futures are composable while actors are not. Functional composition in particular, gives us a level of expressiveness that brings a large amount of power and flexibility to our daily coding. What if we could bring that level of expressiveness to our daily coding while at the same time mixing in concurrency? Figure 34 below shows how actors can be combined with futures.

4.5.1.4 Reactive Streams

Reactive programming is about non-blocking, event-driven applications that scale with a small number of threads with back-pressure - a feedback mechanism that aims to ensure producers do not overwhelm consumers.

4.6 Microservice Architecture Frontend and Backend Implementation

4.6.1 Frontend JavaScript Development Tools

JavaScript is a de facto standard programming language for the internet and frontend web applications.

Node.js is a platform built on Chrome's JavaScript engine. Node includes both a framework and a runtime environment for running JavaScript code outside of the browser.

npm is a package manager that allows you to download tools as well as JavaScript libraries and frameworks. This package manager has a repository of thousands of items, and we'll use it for installing pretty much everything.

Bower used to be a popular package manager for resolving application dependencies (such as for Angular 2 and jQuery).

Grunt is a task runner. Lots of steps need to be performed between developing and deploying the code, and all these steps must be automated. You may need to transpile the code written in TypeScript into widely supported ES5 syntax, and the code, images, and CSS files need to be minimized.

4.6.2 JavaScript Based frameworks and libraries.

The frontend of web application is now dominated by JavaScript based frameworks such as angular, ReactJs and Ember.JS.

Angular is an open source framework for developing web applications. The framework makes it simpler to create custom components that can be added to HTML documents and to implement application logic. Angular uses data binding extensively, includes a dependency injection module, supports modularization, and offers a routing mechanism.

Ember.js is an open source MVC-based framework for developing web applications. It includes a routing mechanism and supports two-way data binding. This framework uses a lot of code conventions, which increases the productivity of software developers.

Jasmine is an open source framework for testing JavaScript code. Jasmine doesn't require a DOM object. It includes a set of functions that test whether certain parts of your

application behave as expected. Jasmine is often used with Karma, which is a test runner that allows you to run tests in different browsers.

Polymer is a library created by Google for building custom components based on the WebComponents standard. It comes with a set of nice-looking customizable UI components that can be included in the HTML markup as tags. Polymer also includes components for applications that need to work offline, as well as components that use various Google APIs (such as calendar, maps, and others).

RxJS is a set of libraries for composing asynchronous and event-based programs using observable collections. It allows applications to work with asynchronous data streams, such as the server-side stream of stock price quotes or mouse move events. With RxJS, the data streams are represented as observable sequences. This library can be used with or without any other JavaScript framework.

Bootstrap is an open source library of UI components developed by Twitter. The components are built using the responsive web design principles, which makes this library extremely valuable if your web application needs to automatically adjust its layout depending on the screen size of the user's device.

4.6.3 Microservice Development Using JHipter Framework

There are several microservices development frameworks. Popular software development frameworks such as .NET and JAVA are redesigning their platforms to comply with Microservice Architecture.

JHipster combines both spring boot, Yoeman and Angular 2 to make web application developers more productive. Any microservice development and deployment framework has to use the underlying orchestration layer such as Docker Swarm, Kubernetes and Kontena.

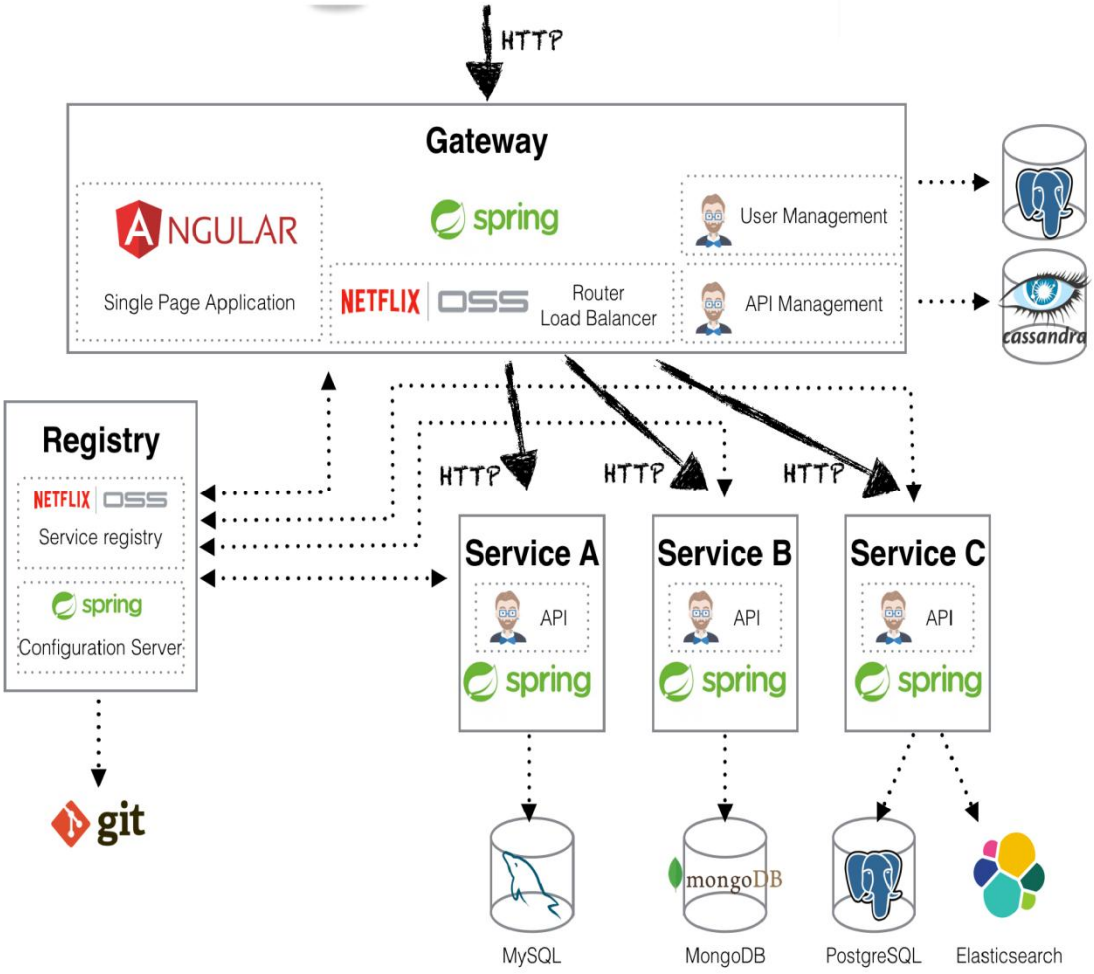


Figure 32: Microservice Architecture implementation on the JVM using the JHipter 3.6 tooling

4.6.4 The Registry

The registry allows the microservices to be discovered by the gateway, which will handle routing requests to the correct microservice.

Two options exist. Either we can clone the registry repository, or we can use the JHipster Docker image. If we clone the repository, we only need to navigate to the directory and run mvn. To use the docker image, we can use the following command:

Docker service create --name registry --publish 8761:8761/tcp jhipster/jhipster-registry

4.6.5 The API Gateway

Ounce we have a running registry, we can create our gateway, which will handle routing requests to the correct microservice. It will also be our entryway into the application. Generating the gateway is simply done by running yo jhipster command, and choose microservice gateway.

Starting the gateway only requires the running the mvn command after being generated. If we access the registry again, we will see the following:

Figure 33: JHipster CLI for development of Microservices

4.6.6 Creating Microservices

Now we can finally create and use some microservices. We are asked some questions about the options we want to include/exclude in the application, and it generates the application. From here, we can create entities and begin to develop the business logic for our microservice.

4.6.7 Running the Microservices Using Docker Compose

We simply answer a few questions, then set everything up with one simple command. For our example, we just choose projects from our current working directory, which

include gateway, and other microservices in your application. You also have the option to configure ELK to monitor your microservices and once you finish answering these questions, we can simply and quickly start up our entire application. Before running **docker-compose up** command, make sure the **run mvn package docker: build** coammand in the root of each project you want to be involved in your docker-compose. After running **docker-compose up -d**, all microservices in the application will be started. The resulting docker container also includes the JHipster Console (assuming you chose to monitor your application).

4.7 The Data Store Design

Monolithic Architecture is not scalable because there is a lot of friction amongst developments and operations teams due to centralized databases

4.7.1 Data Consistency

The centralized RDBMS is not suitable design for a Microservice Architecture. Strong consistency requires coordination, which is extremely expensive in a distributed system, and puts an upper bound on scalability, throughput, low latency and availability.
The need for coordination—adding to the costs of contention and coherency, as defined in the Universal Scalability Law—means that individual services can't make progress individually but has to wait for consensus. When designing Microservices-based systems we should therefore strive to minimize the service-to-service coordination of state, to allow the Microservices to comfortably share state through well-defined API (Jonas Bonér, 2016).

There are reasonable ways of coordinating data changes in a scalable and resilient fashion, but it requires that your operations on the data are composable. *Composability* in this context means that changes to data can be made available to other services without stalling them and without waiting on coordination to take place.

4.7.2 Contention Free Access to Shared State

Apology-Oriented Programming

According to (Jonas Boner, 2016), the idea of Apology-Oriented Programming is built on the premise that it is easier to ask for forgiveness than permission. If you can't coordinate (and be sure about something), then take an educated guess, a bet that a condition will hold, and if you were wrong, apologize and perform a compensating action. This model works very well with an Event-Driven Architecture that leverages asynchronous message-passing and Event Sourcing.

ACID 2.0

According to (Pat Helland, 2011), the term the acronym ACID 2.0 has a different meaning from the traditional ACID (Atomic Consistent Isolated and Durable).

The "A" in the acronym stands for Associative, which means that grouping of messages does not matter—and allows for batching. The "C" is for Commutative, which means that ordering of messages does not matter. The "I" stands for Idempotent, which means that duplication of messages does not matter. The "D" could stand for Distributed, but is probably included just to make the ACID acronym work.

Casual Consistency

Relying on eventual consistency is sometimes not permissible, since it can force us to give up too much of the high-level business semantics (Jonas Boner, 2006). If that is the case then using *causal consistency* can be a good trade-off. Semantics based on causality is what humans expect and find intuitive. Causal consistency can be made both scalable and available and is even proven to be the best we can do in an always available system (Prince Mahajan et al, 2011). Causal consistency is usually implemented using logical time and is available in many NoSQL databases, Event Logging and Distributed Event Streaming products (products allowing use of logical time to implement causal consistency include Riak and Red Bull Eventuate).

Distributed Transactions

Historically, *distributed transactions* have been used to coordinate changes across a distributed system. They provide the illusion that you are the only person using the

datastore. This is not true, and upholding this illusion is extremely costly, making systems slow, unscalable, and brittle.

The *Saga Pattern* is a scalable and resilient alternative to distributed transactions . It is a way to manage long running business transactions based on the discovery that long running business transactions often comprise multiple transactional steps in which overall consistency of the whole transaction can be achieved by grouping these steps into an overall distributed transaction.

The technique is to pair every stage's transaction with a compensating reversing transaction, so that the whole distributed transaction can be reversed (in reverse order) if one of the stage's transactions fails.

A distributed storage system has multiple, sometimes competing, goals: availability, low latency, and partition tolerance and scalability. Though strong consistency model simplifies programming and provide users with the system behavior that they expect, it is not well suited to distributed databases.

The first four of these properties are described as the 'ALPS' properties: **A**vailability, **L**ow-Latency, **P**artition-tolerance, and **S**calability.

Availability: all operations complete successfully and no operation can block indefinitely or return an error indicating that data is unavailable.

Low-latency: target response times on the order of a few milliseconds

Partition tolerance: the data store continues to operate under network partitions

Scalability: the data store scales out linearly

In this model popularly known as ALPS we sacrifice strong consistence for Causal consistency. Causal consistency does not impose any order on concurrent operations.

Most database vendors are revising their database system design to adhere to these principles of Apology Oriented Programming, ACID 2.0, Saga Pattern and Casual Consistency.

Data replication

A replica may execute an operation without synchronising *a priori* with other replicas. The operation is sent asynchronously to other replicas; every replica eventually applies all updates, possibly in different orders. A background consensus algorithm such as Raft reconciles any conflicting updates. This approach ensures that data remains available despite network partitions.

One tool that embraces these ideas is CRDTs (**Conflict-Free Replicated Data Types**), as they are eventually consistent, rich data-structures (including counters, sets, maps and even graphs) that compose, and that converge without coordination. The ordering of the updates does not matter, and can always be automatically merged safely. CRDTs are fairly recent, but have been hardened in production for quite some years, and there are production-grade libraries that you can leverage directly (for example in Akka and Riak). CRDTs is one of the most interesting ideas coming out of distributed systems research in recent years, giving us rich, eventually consistent, composable data-structures that are guaranteed to converge consistently without the need for coordination.

4.7.3 Implementation of Distributed Data Stores

The NoSQL movement is based some good design principles that are being also borrowed by the traditional SQL based databases. Currently most RDBMS vendors have moved to claim that they are capable of offering didtributed databases through Database as a Service (DBaaS). DBaaS is a big improvement over the traditional RDBMS however they have several limitations. One of the limitations of the DBaaS is inability to scale and unsuitable for Microservice Architecture.

4.7.3.1 Implementing Distributed Data Stores Using Containerization

4.7.3.1.1 Software Defined Storage

According to Wikipedia, Software-defined storage (SDS) is an evolving concept for computer data storage software to manage policy-based provisioning and management of data storage independent of the underlying hardware. Software-defined storage definitions typically include a form of storage virtualization to separate the storage hardware from the software that manages the storage infrastructure. The software enabling a software-defined storage environment may also provide policy management for feature options such as deduplication, replication, thin provisioning, snapshots and backup.

To put it simply SDS is a class of storage solutions that can be used with commodity storage media and compute hardware; where storage media and compute hardware have no special intelligence embedded in them. All the intelligence of data management and access is provided by a software layer. The solution may provide some or all the feature of modern enterprise storage systems like scale up and out architecture, reliability and fault tolerance, high availability, unified storage management and provisioning, geographically distributed data center awareness and handling, disaster recovery, QoS, resource pooling, integration with existing storage infrastructure, etc. It may provide some or all data access methods like file, block and object.

Containers were originally designed to be stateless – they do not natively support requirements for databases, message queues, application state and instrumentation data. Traditional storage architectures are complex and lack API functionality to support DevOps. Storage doesn't scale with apps and performance is unpredictable. It's also very difficult to move data securely between locations and/or cloud providers and management and performance tool sets are lacking. Finally, the cost model is geared towards fork lift CAPEX spikes and complex refresh cycles.

The rise of containers in enterprise has led to the creation of a new class of storage optimized for containerized workloads. Existing storage technologies, such as network-

attached storage (NAS) and storage area network (SAN), are not designed to run containerized applications. A good storage solution should entail the following characteristics

- Use of commodity x86 hardware
- Scale-out, shared-nothing, design
- Strong API-based management interface

4.7.3.1.2 SDS Solution Providers

Hedvig Distributed Storage Platform

Software defined storage solution that virtualizes and aggregates flash and spinning disk in a commodity-based server cluster or cloud, presenting it as a single, elastic storage system that can be accessed as block, file, or object storage. Hedvig improves storage flexibility and economics for container based environments. There are two main components in the Hedvig architecture:

- Hedvig Storage Service – a distributed systems engine that scales storage performance and capacity with x86 and ARM servers – including cloud instances.
- Hedvig Storage Proxy – a lightweight VM, container, or physical node that delivers read/write access to the Hedvig Storage Service via industry- standard protocols, performs local read caching, and enables client-side deduplication.
- Hedvig Docker Volume Plugin – takes advantage of the Docker Volume API to integrate the Hedvig storage platform with Docker Datacenter and ensure persistent, portable access to storage resources. Installs on each host that requires access to Hedvig storage.

StorageOS

StorageOS delivers persistent storage for containers, delivering scalable, deterministic, low latency storage performance and simplifying provisioning and management of containerized storage. All while enabling secure data mobility for containers to move data between bare metal, virtual machines or cloud storage. StorageOS in the latest entrant into the SDN space and boasts of the following capabilities

- Designed to allow 3rd party data services as well as customer databases, analytics platforms and applications to run natively on the storage platform through use of containers.
- Deployable as containers and runs on bare metal servers, virtual machines or cloud instances in any combination.
- Storage services for any size customer from a small startup to a large enterprise.
- Monthly billing per Tetrabyte

4.7.3.2 Testing Software Defined Storage

We managed to test the principles of. SDN using two open source distributed databases. The scalability of this solutions is made possible by the the underlying cluster orchestration mechanism such as Docker Swarm, Kubernetes and Mesos.

4.7.3.2.1 Crate

Crate's architecture is based on the NoSQL architecture, but features Standard SQL. Crate is extremely simple to install and use, with auto-sharding, auto-partitioning and auto-replication. This enables realtime search & aggregations with the benefit of the ability to horizontally scale any crate deployment. Crate makes cluster setup as easy as possible, but there are things to note when building a new cluster.

Crate is designed in a shared nothing architecture, in which all nodes are equal and each node is self-sufficient. This means that nodes work on their own, and all nodes in a cluster are configured equally, the same as with a single-node instance. The crate

distributed Datastore architecture is based on Raft Consenus algorithm and its cluster leader election resembles that of Docker Swarm.

To setup Swarm cluster and run crate distributed database on it, we first create the Swarm cluster using the Bash script shown in figure 40. The script creates five nodes using the docker machine and using Docker Swarm initializes the cluster by generating a token. The token is a secret string that enables other nodes to join the cluster as manager nodes or as worker nodes.

```
# Define the number of managers/workers
MANAGER=2
WORKER=3
# Create the Docker hosts
for i in $(seq 1 $MANAGER); do docker-machine create --driver virtualbox manager$i; done
for i in $(seq 1 $WORKER); do docker-machine create --driver virtualbox worker$i; done
# start  manager  machine(s)
for i in $(seq 1 $MANAGER); do docker-machine start  manager$i;done
# start worker machine(s)
for i in $(seq 1 $WORKER); do docker-machine start  worker$i; done
#Regenearte certificates for the machine
for i in $(seq 1 $MANAGER); do docker-machine regenerate-certs  manager$i;done
for i in $(seq 1 $WORKER); do docker-machine regenerate-certs  worker$i; done
# remove worker(s) from cluster
for i in $(seq 1 $WORKER); do docker-machine ssh worker$i docker swarm leave --force ; done
# remove managers(s) from cluster
docker-machine ssh manager1 docker swarm leave --force
docker-machine ssh manager2 docker swarm leave --force
# Init the swarm
docker-machine env manager1 && eval $(docker-machine env manager1)
docker-machine ssh manager1 docker swarm init  --advertise-addr $(docker-machine ip manager1)  --listen-addr $(docker-machi
TOKENW=$(docker swarm join-token  -q worker)
TOKENM=$(docker swarm join-token -q manager)
# Add additional manager(s) to the cluster
for i in $(seq 2 $MANAGER); do docker-machine ssh manager$i docker swarm join --token $TOKENM  $(docker-machine ip manager1)
# Add workers to the cluster
for i in $(seq 1 $WORKER); do docker-machine ssh worker$i  docker swarm join --token $TOKENW  $(docker-machine ip manager1):
echo "Hi, I'm sleeping for 120 minutes..."
```

Figure 34: Docker Swarm cluster creation automation script.

To get a Crate cluster running on Swarm on the same Swarm cluster you can use docker compose whose yaml file is as shown below. From the directory where your docker-

compose yaml file is located you can use the **docker-copmpose bundle** command to create Distributed Application Bundle file (DAB) file.

This enables your application to be portable across various cloud environments without changing anything.

```
crate:
image: crate
ports:
    "4200:4200"
    "4300:4300"
    volumes:
    /mnt/data/crate:/data
     environment:
     CRATE_HEAP_SIZE: 16g
     command: crate -Des.config=/path/to/crate.yml -Des.cluster.name=cluster
    crate2:
     image: crate
     volumes:
      - /mnt/data/crate:/data
    node:
     build: .
     ports:
      - "8000:8000"
     links:
      - crate
```

4.8 Test Results for Validation of Scalable Architecture

The validation of the scalability model layer were performed using results generated by Jeff Nickoloff and available at https://github.com/allingeek/ecps-data . The raw data shows the number of containers in a cluster and their start delay time. This data is used obtain the number of container vs their throughput. Throughput is obtained by multiplying the number of container by the inverse of the average start-up time of a container. See appendix A for the CSV files. The test results were obtained as described in section 3.6 and are as tabulated below.

Table 3: start delay vs Number of containers measurements

No of containers	Mean delay time(ms)
10	262.78
20	178.140
30	190.40
40	164.70
50	289.30
60	416.64
70	446.34

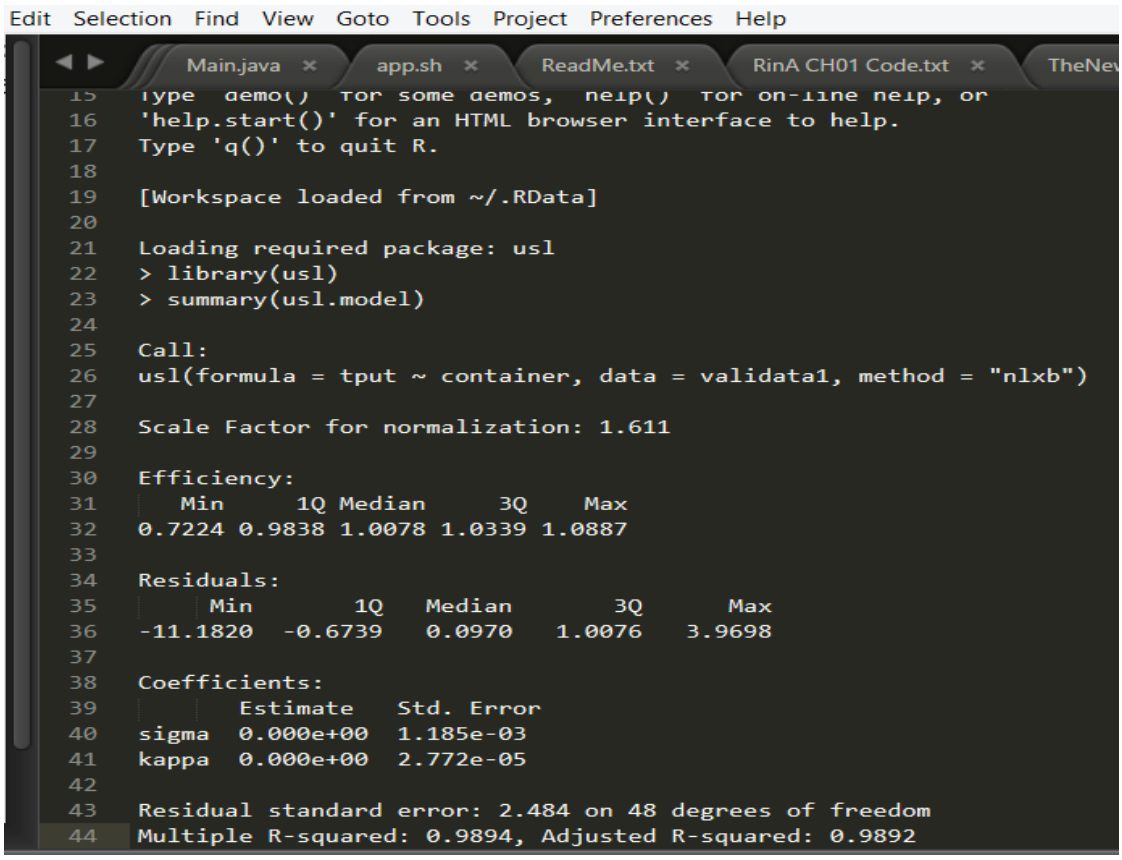


Figure 35: Validation of usability model using dataset1 (courtesy Nickoloff 2016)

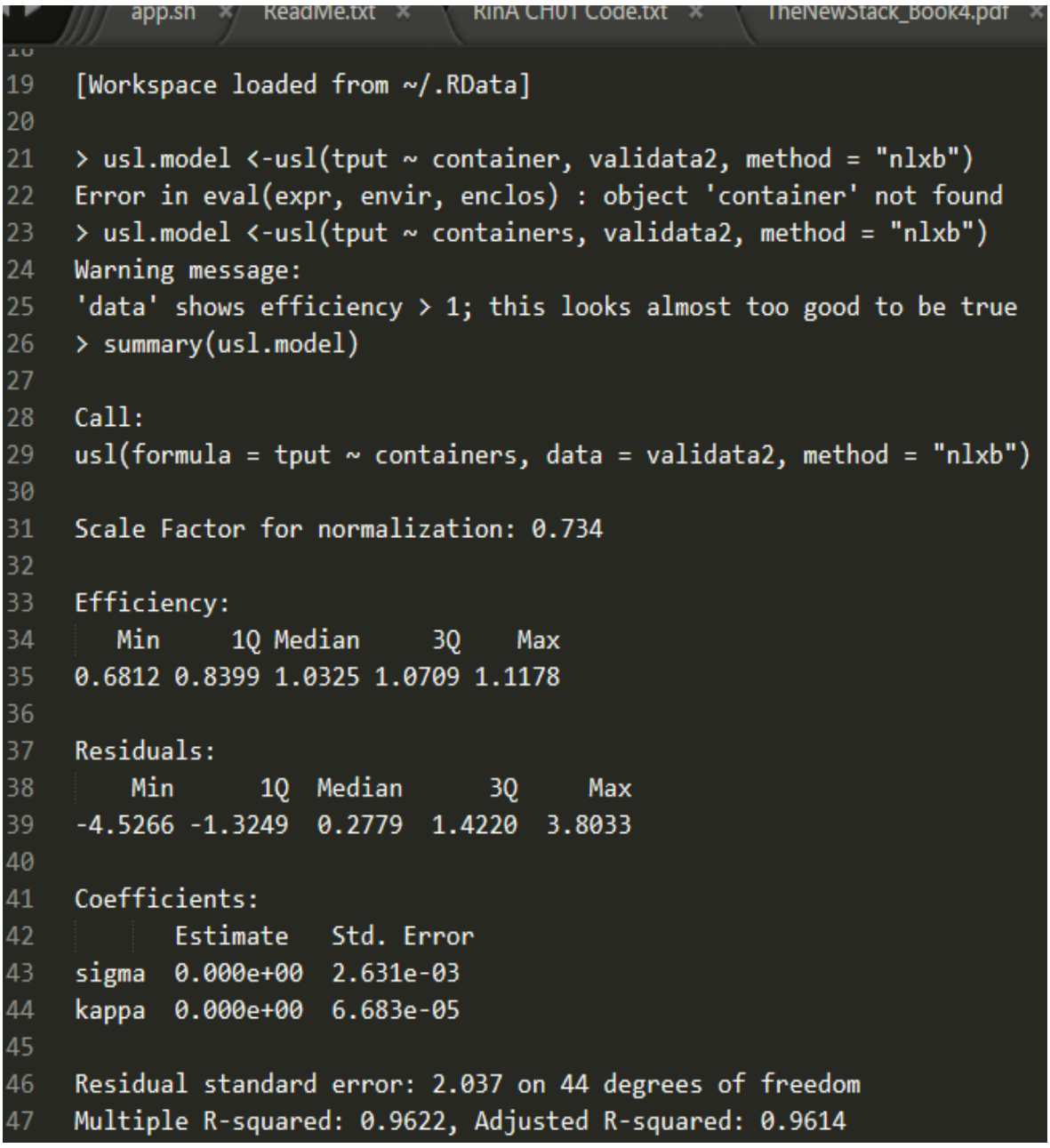


Figure 36 : Validation of usability model using dataset2(courtesy Nickoloff 2016)

```
Main.java × app.sh × ReadMe.txt × RinA CH01 Code.txt ×
> testdata1
  size   tput
1   10  38.17
2   20 112.36
3   30 157.89
4   40 243.90
5   50 173.01
6   60 144.23
7   70 156.95
> usl.model <-usl(tput ~ size, testdata1, method = "nlxb")
> summary(usl.model)

Call:
usl(formula = tput ~ size, data = testdata1, method = "nlxb")

Scale Factor for normalization: 7.454

Efficiency:
   Min     1Q Median     3Q    Max
0.3008 0.3933 0.5121 0.7298 0.8180

Residuals:
    Min      1Q  Median      3Q     Max
-33.487 -21.954  -8.105  -3.928  68.307

Coefficients:
       Estimate   Std. Error
sigma  0.0000000  0.0144191
kappa  0.0004475  0.0002843
```

Figure 37: **Determination of sigma and kamma coefficients for the scalability model based using regression analysis.**

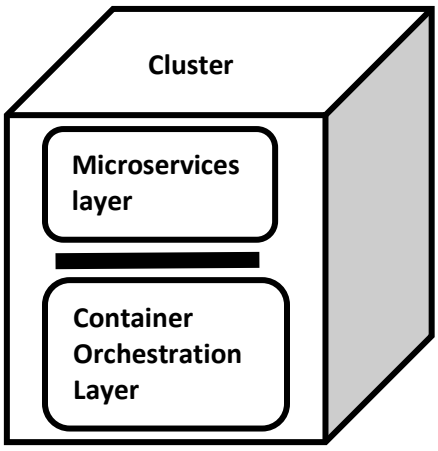


Figure 38: Layering of a cluster to abstract scalability away from the microservices to the orchestration layer

The Let's start by presuming (at least) two layers in each scalable application within a cluster of machines. These layers differ in their perception of scaling. The lower layer of the application understands the fact that more machines get added to make the system scale. In addition to other work, it manages through container orchestration the mapping of the upper layer of microservices to the physical machines and their locations. We are presuming that the lower layer i.e container orchestration layer provides a scale agnostic programming abstraction to the upper layer (microservices layer).

Using this scale-agnostic programming abstraction, the upper layer of Microservices is written without worrying about scaling issues. By sticking to the scale agnostic programming abstraction, we can write microservices logic without being worried about the changes happening at the orchestration layer.

4.9 Performance evaluation

The universal scalability Law was used to develop a model for evaluating the scalability of the Microservice Architecture. The start up time for a container was used as the response time and the throughput is determined by multiplying the reciprocal of the start up time x the no. of containers. The scalability was measured at the orchestration layer and at the application layer. To be able to measure the scalability of the orchestration layer we used data. The R software was used to validate the scalability based on the Universal Scalability Law.

Table 4: Throughput vs Number of containers based container start delay time

No of containers	Mean delay time(ms)	Throughput = no of containers/delay time
10	262.78	38.16794
20	178.140	112.3596
30	190.40	157.8947
40	164.70	243.9024
50	289.30	173.0104
60	416.64	144.2308
70	446.34	156.9507

The measured data for maximum throughput was used to construct graphs of maximum throughput versus number of containers (functional scaling). Regression analysis determines which of the scalability models best describes the data. Details of the analysis are discussed in (Williams and Smith 2004). The analysis provides model parameters which are then used to extrapolate the behavior to higher numbers of containers.

Regression analysis indicates that using Amdahl's Law provides the best fit to the measured data. The functional scalability of this application is therefore described by Equation 1 (Williams and Smith 2004).

$$X_{\max}(p) = \frac{X_{\max}(1) \times p}{1 + \sigma(p-1)} \quad \text{..... Equation 6}$$

Where

p is the number of containers

Xmax(1) is the maximum throughput with 1 container

Xmax(p) is the maximum throughput with p container

σ is the fraction of the workload that is performed sequentially.

The value of σ obtained from the regression analysis is 0.0000

This means that the effect of contention on scalability is minimal

$$X_{\max}(p) = X_{\max}(1) \times p \text{.....6}$$

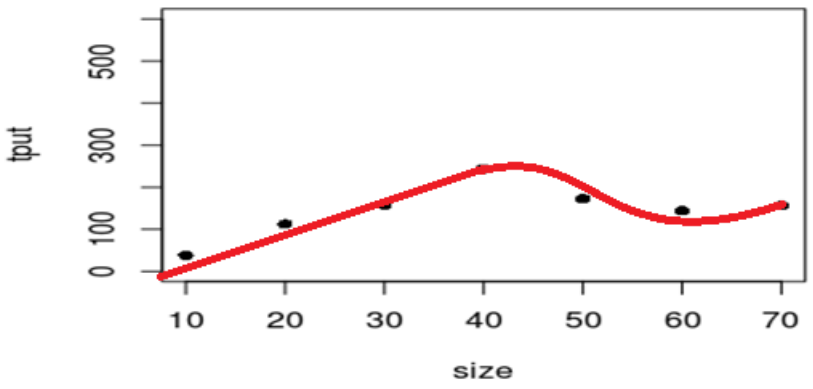


Figure 39: Obtained Results –Variation of throughput vs the number of containers

The Amdahl's Law extrapolation indicates that the maximum throughput with 100 containers would give throughput of 100/10(38)=380 and the maximum throughput with 1000 containers would be 3800 requests. Thus, the required throughput of 2000 can be achieved with 2000/380(100)= 526 containers.

The throughput decreases beyond 40 containers because the resources (RAM, DISK and PROCESSOR SPEED) becomes a limiting factor.

4.10 Automated Testing

The Microservice Architecture and DevOps advocates for Continuous Integration and Continuous

Delivery. The two processes requires automated Testing. In this thesis we implemented an automated testing involving Unit Testing and Integration Testing. The tests were carried out at the developers work station or laptop and at the Continuous Integration Server.

There was one JUnit test and one integration test to test how the CustomerRepository class saves the data to the in-memory data. As shown in the listing below, the test involved inserting one record into the database.

Listing 1: The Test Class using JUnit

```
package com.khakame.customerService;
import java.util.List;
import static org.junit.Assert.*;
        public class CustomerRepositoryTest {
        private CustomerRepository jdbcCustomerRepository;
        @Before
        public void setUp() {
                jdbcCustomerRepository = new CustomerRepository();
        }
        @Test
        public void insertToDoItem() {
                Customer newCustomer = new Customer();
                newCustomer.setNationalId("98765432");
                newCustomer.setFirstName("Khakame");
                newCustomer.setLastName("Peter");
                newCustomer.setTelephoneNumber("254721876005");
                newCustomer.setDateofBirth("29-3-1969");
                newCustomer.setEmailAddress("khakamewamboko@gmail.com");
                Long newId = jdbcCustomerRepository.insert(newCustomer);
                assertNull(newId);
                Customer persitedCustomer = jdbcCustomerReposito
                assertNotNull(persitedCustomer);
                assertEquals(newCustomer, persitedCustomer);
                }
        }
```

Methods marked with this annotation are always executed before every test method of class

Methods marked with this annotation will run as test case

Wrong assertion put there on purpose to provoke a failed test.

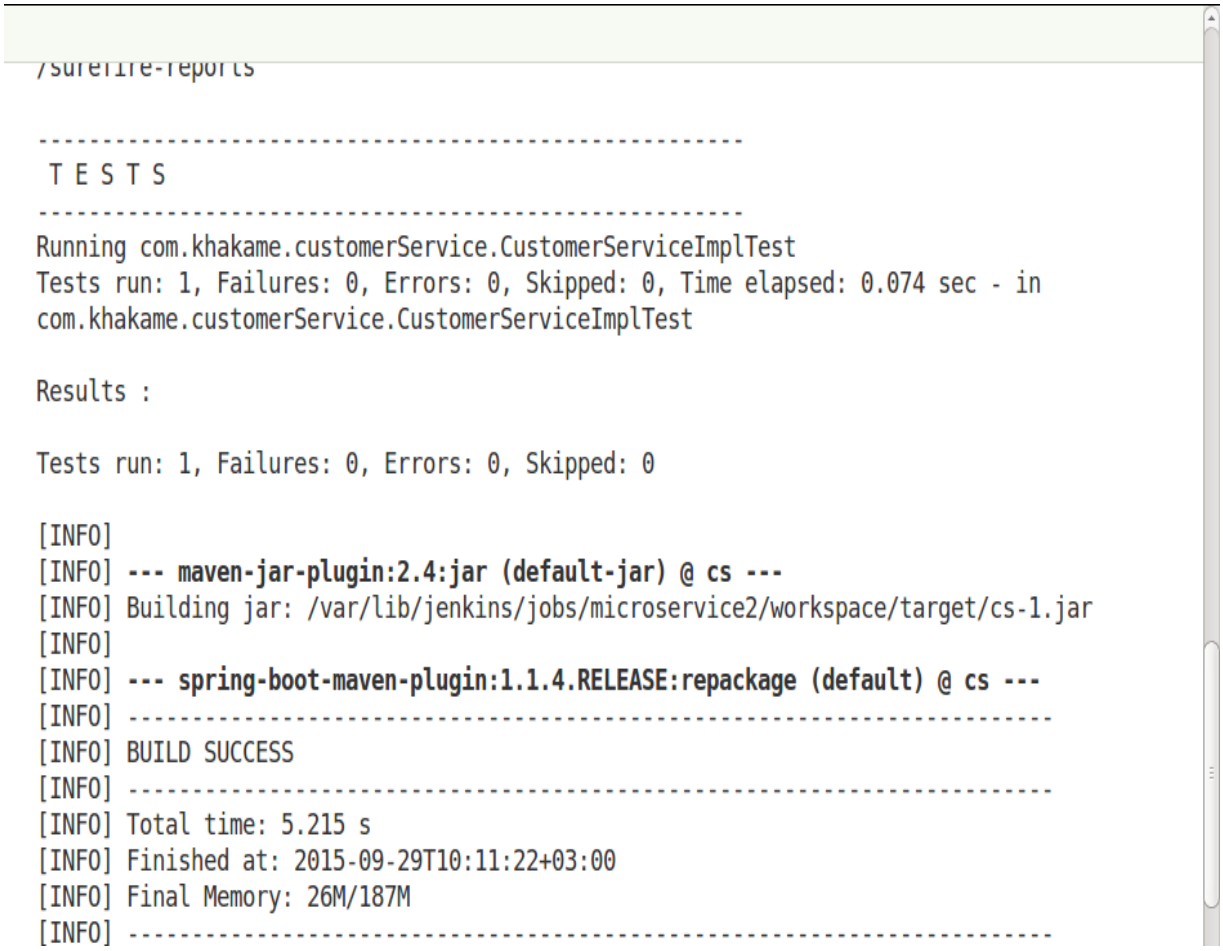


Figure 40: Test report integration tests conducted by the Continuous Integration server

CHAPTER FIVE : RESULTS ANALYSIS

5.1 Introduction

The aim of this thesis was developing a scalable Microservice Architecture for web services using OS-level virtualization. The monolithic architecture is prevalent in many organizations. The transition from monolithic architecture to microservices has been made necessary by the need by companies to re-engineer their systems to meet the ever changing business requirements and environment.

5.2 What Factors are Influencing the Adoption of Microservices

5.2.1 Virtualization

Virtualization is the new software order. Every business must enhance its IT in order to compete. This model has worked well for Amazon Web Services. Companies that will not transform themselves with the right IT systems will be driven out of business by the competition from outside their industry. Music stores, books, movies etc are being served from online stores. Netflix has succeeded by employing microservices based architecture to offer online services.

Every business should be aware of the competition from within the industry and from outside the industry. Cisco for example has been predominantly a hardware company dealing with the manufacture of switches and Routers for networking. However following recent trends , it is clear that software defined software defined networking, software defined storage is going to disrupt many established businesses. Cisco on sensing that its territory has been invaded from without decided to take the war to the competitors ground by declaring that it has decided to become a software company. The popular saying is that “software is eating the world” is true. Vendors of RDBMS are also facing disruption from new entrants who are introducing the new storage service called software defined storage. Hedvig and StorageOS are new kits in that space and established database vendors such Oracle have to wake up to the challenge. For this reason mergers and acquisitions of the disrupting companies has become the norm.

Therefore as software becomes the differentiating factor amongst many service providers, appropriate software architecture must be sought and embraced.

5.2.2 Containerization

Containerization as popularized by Docker is the just the beginning of the new era of cloud computing. Containerization is a continuum that spans from Virtual machines to Unikernels. The era of immutable infrastructure has just arrived. That means that one can use the cloud without changing anything on the hardware side. Containers are running processes that can only be replaced by other containers. Containers and microservices are the opposite side of the same coin. Containerization enables developers or DevOps teams to move software to the users much faster. Containerization is currently the best way Microservices can be packaged, deployed, and released on infrastructure. This leads to better infrastructure utilization, and simplifies the way a change is moved from a developer's machine to the production environment.

5.2.3 Internet of Things

According Top 10 Strategic Technology Trends for 2016 (Gartner Research, 2015), The age of Internet of Things or the Web of Things or the Web of Everything is around the corner and is likely to impact on the way we develop software. JavaScript for example is already enthroned as the language of choice for the internet. No single player can dominate the IoT landscape. The most likely scenario will be thousands of web applications with well defined API will rule the IoT. Hence the Web of Things will embrace the Microservice Architecture. Gartner Research states that "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".

5.3 Results Analysis

To what extend can containerization enhance design and implementation of Microservice Architecture?

Neil Gunther's Universal Scalability Law (USL) provides a formal definition of scalability, and a conceptual framework for understanding, evaluating, comparing, and improving scalability. It does this by modeling the effects of linear speedup, contention delay, and coherency delay due to crosstalk. It's been proven that blocking of any kind, anywhere in the system will measurably impact scale due to:

- **Contention**- waiting for queues or shared resources.
- **Coherency (Crosstalk)**- the delay for data to become consistent.

In thesis we investigated our scalability against this law and used this model to improve scalability of web services using OS-Level Virtualization. The no. of processors can be interpreted to mean the number of containers. Scalability is handled at the orchestration layer as shown in figure 14. The design of three cluster orchestration namely Docker Swarm, Kubernetes and Mesos was examined in sections 2.12.1 , 2.12.2 and 2.12.3 respectively. Basically 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 completely 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

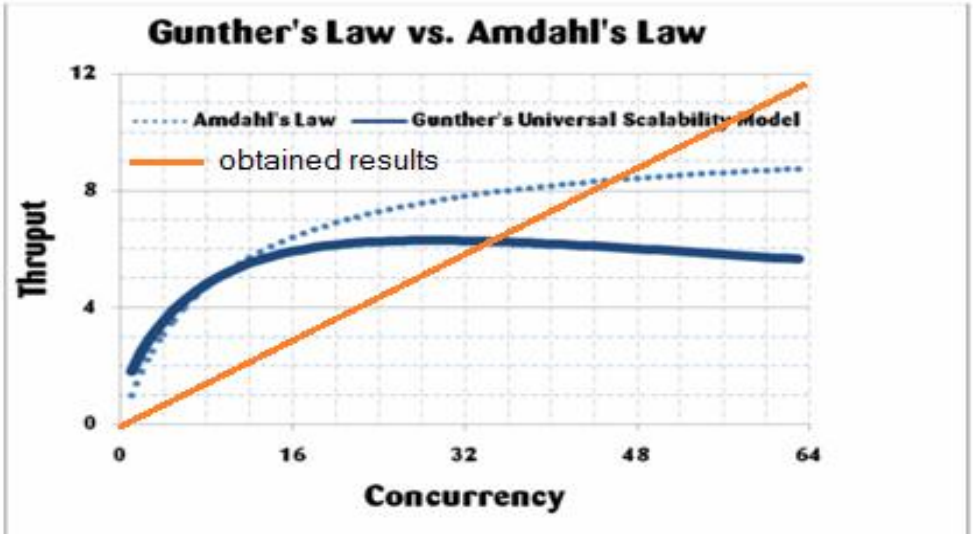


Figure 41: Expected behavior of throughput vs no of containers. (courtesy Gunther, 2007).

In section 2.12 we discussed the design of the orchestration layer for the three cluster orchestration software. Docker Swarm 1.12.0 has simpler architecture based on the Raft consensus algorithm to handle contention delay arising from the need to coordinate the state of the containers that is 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 centres. 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.

5.3.1 Model Validation

Using 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 container were increased from 1 to 30000. Regression analysis done using the R software found out that the sigma coefficient (σ) and kamma (κ) coefficient 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.

5.3.2 Scalability Testing

We measured the mean container start up times as we increased number of containers from 10 to 70. The throughput increased linearly with the number of containers up to 40 containers. Beyond 40 containers, the throughput decreases with the increase of number of containers. This was attributed due to limited RAM, Hard Disk and processor recourses on the 3 core 1.2 GHz 8GB Ram HP Laptop that was used to conduct the measurements. From regression analysis did using the R software, the sigma coefficient was found to be zero meaning effect contention shared resources was non-existent.

The kamma coefficient was found to be 0.0004475 as shown in figure 40. This means that there is minimal crosstalk as the system tries to achieve consistency across its data stores. From research findings it has been shown that strong consistency is not easy to attain in a distributed system. However one can settle for eventual or casual consistency. It was shown that casual consistency is more suitable to distributed data stores because the state of the system converges with time.

5.4 Automated Software Testing

Software testing is a daunting task and attempts to automate it will go a long way in improving software quality. While some organizations have invested a lot in automating testing, many financial institutions still rely heavily on manual testing for important areas like functional acceptance testing, integration testing, and security testing. A study conducted by PWC in 2014 found that only 15% of testing activities have been automated at major financial institutions. Because manual testing for large systems is so expensive, many firms outsource or offshore testing to take advantage of lower-cost

skills, handing the code off to test teams in India or somewhere else in a “follow the sun” approach to be tested overnight.

The path toward automated testing is straightforward, but it’s not easy. It starts with the basics of Continuous Integration: automating unit testing and basic functional testing, and moving responsibility for regression testing onto developers. Using CI server we were able to carry out automated Unit Testing and Integration testing for one of the classes in the microservice.

Using Microservice Architecture, Containerization and DevOps makes Automated Testing possible.

Dedicated teams can be assigned to automate the tests particularly using Model driven Engineering.

Stateful containers where it is possible to migrate a microservice together with its associated databases will be an enabling technology for Automated Testing because no changes are necessary across the various testing platforms and environments.

CHAPTER SIX : CONCLUSIONS AND RECOMMENDATIONS

6.1 Introduction

The introduction of Microservice Architecture has reinforced the resolve for agility in software industry. Containerization is promising to transform the IT in 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 important 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 of software components whose functionality has varying demand from the users, it becomes wasteful to equally assign more resources to all software components. The Microservice Architecture addresses this problem by enabling scalability on a functional dimension. By splitting the application into smaller units the manpower per unit can be resized accordingly to enhance productivity. The number of technologies supported may be scaled accordingly as need arises.

By employing distributed data stores in Microservice Architecture scalability is enhanced through eliminating coherency delay that is prevalent in relational databases. The developer has the flexibility to choose the right database technology.

6.2 What Factors are Influencing the Adoption of Microservice Architecture

Factors influencing adoption of Microservice Architecture include the virtualization, containerization, and Internet of Things as discussed in section 5.2.

6.3 To what Extend 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 in to microservices such load balancing, health checks etc are handled at the orchestration layer. Similarly the functions such as service discovery, scheduling, inter-container communication are hidden from the developer and handled at the orchestration layer. The Docker Architecture is extensible through use of plugins. Volume plugins allow third-party container data management solutions to provide data volumes for containers which 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 powerful cluster management tool that is capable of handling over 1000 host and scheduling up to 30000 containers.

6.4 To What Extend 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 a service by specifying the number of containers using Command Line Interface. By varying the number of container 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 centres 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 proper mix of containers and virtual machines you are can achieve high isolation and security making multi-tenancy in data centres. With orchestration layer forming an abstraction layer the complexity arising from microservices is hidden from the developer. Based on the results analysis 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 host in 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 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.

6.5 To What Extend Can Microservices Testing be Automated?

Automated Testing is a prerequisite for Continuous Deployment and Continuous Delivery. According to PWC report 15% of financial sector have embraced automated testing. However it is important to note that automated testing for legacy monolithic systems is not easy. Introduction of Microservice Architecture and DevOps will hasten implementation of Automated Testing.

6.6 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 dictates that DevOps as the best development methodology there is 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 centres to supplement Virtual machines should be investigated further to device mechanisms of integrating all Orchestration technologies to improve the isolation of processes running on a single host. This will enhance security in data centres and hasten the adoption of containers in the cloud. There is need to develop 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 need to enhance the Docker Compose files to be capable having enough information for creating microservices without 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 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.

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

Edward ClusterHQ, “Flocker Documentation” Accessed on (17/07/2015) URL:

https://docs.clusterhq.com/en/1.0.3/introduction/index.html

Gartner Cool Vendor DevOps Report (Accessed 4/07/2015), URL: https://www.gartner.com/doc/3034319/cool-vendors-DevOps/

Humble Jez, et al.(2010), " Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation", Addison-Wesley.

IBM (2015), "Reducing Development Time, Risk and Cost through Microservice"

IBM (2015) "DevOps for Dummies", John Wiley & Sons

Iron.io (2015) White Paper

Ivan Dwyer, (2016), "Serverless Computing: Developer empowerment reaches new heights" Iron.io

J. Lakshmi, (2010), "System Virtualization in the Multi-core Era - a QoS Perspective", Phd Thesis Supercomputer Education and Research Center Indian Institute of Science Banglore –India .

Jeff Nickoloff, (2016) " Docker in action", Manning.

Jeremy Cloud. (2013), "Decomposing Twitter: Adventures in Service-Oriented Architecture". In: QCon New York . URL: http://www.infoq.com/presentations/twitter-soa

Joab Jackson, (2016), " Docker Swarm Wins Scaling Benchmark but Don't Take That as Gospel" The New Stack March.

Jon Loeliger, (2009), " Version Control with Git" , First Edition O'Reilly Media.

Jonas Bonér, (2016), "Reactive Microservice Architecture: Design Principles for Distributed Systems" O'Relly Media USA.

Jula, A., Sundararajan, E., & Othman, Z. (2014). Cloud computing service composition: A systematic literature review. *Expert Systems with Applications*, *41*(8), 3809–3824

Kavita Argarwal (2015). "A Study of Virtualization Overheads", Thesis Master of science in Computer Science, Stony Brook University.

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.

Pat Helland Life beyond Distributed Transactions: an Apostate's Opinion

Prince Mahajan et al, (2011), "Consistency, Availability, and Convergence", Department of Computer Science, The University of Texas at Austin Technical Report.

PwC, (2014), "An ounce of prevention: Why financial institutions need automated testing".

R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. (2011), "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems".

R. Buyya, J. Broberg, and A.M. Goscinski. (2011), Cloud Computing Principles and Paradigms. Wiley Publishing.

Rackspace, (2015), "Building Your DevOps Engine: A Guide To Tearing Down Organizational Silos to Create More Responsive Enterprise IT".

Rimal, B. P., Jukan, A., Katsaros, D., & Goeleven, Y. (2010). Architectural Requirements for Cloud Computing Systems: An Enterprise Cloud Approach. *Journal of Grid Computing*, *9*(1), 3–26.

Roy Thomas Fielding. (2000), "*Architectural styles and the design of network-based software architectures",* PhD thesis, University of California, Irvine.

Scott Chacon and Ben Straub , (2010), "Git Pro", First Edition Apress.

Tam Le Nhan. (2013), "Model-Driven Software Engineering for Virtual Machine Images Provisioning in Cloud Computing", Software Engineering. Universite Rennes.

Marko Luksu. (2016), Kubernetes in action Manning Publications

Verma, et al. (2015), "Large-scale cluster management at Google with Borg. Europeatn Conference on Computer Systems", Bordeaux, France.

Vladimír Jurenka (2015), "Virtualization using Docker Platform", Master of science Thesis.

Weaveworks, " Container Networking plugin" (accessed on 17-6-2015) URL: https://github.com/weaveworks/weave

Wilde, Erik ; Pautasso, Cesare, (2011), "REST: from research to practice", 2nd Ed., Springer

New York.
Yang Yu, (2007), "OS-Level Virtualization and its applications", PhD Thesis Stony Brook University.

APPENDIX A1

KUBERNETES ORCHESTRATOR :

Table 5: Measurements of throughput(tput) as the number of containers is increased from one to 414

container	Tput	container	tput	container	Tput	container	tput	container	tput
1	0.5	47	37.3	93	71.5	139	109.4	185	121.7
2	1	48	36.6	94	58	140	108.5	186	147.6
3	2.3	49	38.6	95	72	141	110.2	187	147.2
4	3.1	50	33.1	96	76.2	142	112.7	188	145.7
5	2.7	51	41.1	97	59.5	143	94.1	189	119.6
6	3.3	52	39.4	98	77.8	144	91.7	190	146.2
7	5.6	53	41.7	99	76.7	145	113.3	191	152.8
8	4	54	35.8	100	64.9	146	90.1	192	148.8
9	5.4	55	43.7	101	82.8	147	115.7	193	131.3
10	5.2	56	44.4	102	84.3	148	102.1	194	155.2
11	8.8	57	41	103	64.4	149	94.9	195	158.5
12	9.2	58	47.2	104	63.4	150	121	196	121.7
13	8.3	59	47.2	105	80.8	151	117.1	197	156.3
14	10.8	60	48	106	82.2	152	114.3	198	161
15	12	61	48.4	107	69.5	153	122.4	199	159.2
16	12.6	62	42.2	108	86.4	154	122.2	200	125.8
17	10.4	63	42	109	87.2	155	104.7	201	155.8
18	12.3	64	47.8	110	88.7	156	120.9	202	156.6
19	14.8	65	50	111	72.5	157	120.8	203	123.8
20	16.3	66	52.8	112	72.7	158	122.5	204	104.6
21	11.7	67	53.2	113	82.5	159	90.3	205	132.3
22	15.5	68	53.5	114	87.7	160	127	206	167.5
23	16.8	69	54.8	115	94.3	161	96.4	207	161.7
24	14.4	70	44.6	116	94.3	162	126.6	208	162.5
25	15.4	71	55	117	94.4	163	124.4	209	158.3
26	21	72	45.9	118	91.5	164	130.2	210	166.7
27	21.3	73	58.9	119	98.3	165	132	211	167.5
28	19.2	74	60.7	120	94.5	166	107.8	212	168.3
29	19	75	49	121	76.6	167	130.5	213	142
30	24.4	76	59.4	122	89.1	168	133.3	214	167.2

31	23.7	77	57.5	123	98.4	169	138.5	215	168
32	19.6	78	60.9	124	78.5	170	108.3	216	167.4
33	26	79	61.7	125	100.8	171	130.5	217	172.2
34	26.2	80	48.5	126	97.7	172	133.3	218	171.7
35	27.8	81	64.8	127	100.8	173	143	219	176.6
36	29	82	64.6	128	99.2	174	138.1	220	174.6
37	28.9	83	64.8	129	77.7	175	138.9	221	172.7
38	29.7	84	51.9	130	103.2	176	102.9	222	170.8
39	24.1	85	68	131	87.9	177	143.9	223	167.7
40	25.6	86	70.5	132	105.6	178	138	224	141.8
41	32	87	68	133	103.9	179	134.6	225	170.5
42	32.6	88	69.8	134	106.3	180	118.4	226	178
43	32.3	89	67.9	135	106.3	181	143.7	227	176
44	36.1	90	55.6	136	104.6	182	145.6	228	181
45	33.8	91	54.2	137	88.4	183	145.2	229	187.7
46	30.5	92	73	138	108.7	184	142.6	230	174.2

containers	Tput	containers	tput	containers	tput	containers	tput
231	179.1	277	216.4	323	256.3	369	251
232	179.8	278	217.2	324	207.7	370	280.3
233	183.5	279	218	325	244.4	371	229
234	150	280	224	326	252.7	372	224.1
235	180.8	281	199.3	327	251.5	373	298.4
236	195	282	183.1	328	260.3	374	299.2
237	194.3	283	216	329	231.7	375	293
238	184.5	284	180.9	330	230.8	376	239.5
239	191.2	285	214.3	331	245.2	377	238.6
240	164.4	286	173.3	332	261.4	378	290.8
241	191.3	287	167.8	333	268.5	379	300.8
242	192.1	288	221.5	334	271.5	380	301.6
243	153.8	289	227.6	335	257.7	381	307.3
244	159.5	290	185.9	336	206.1	382	243.3
245	194.4	291	179.6	337	257.3	383	304
246	196.8	292	186	338	260	384	252.6
247	160.4	293	220.3	339	269	385	313
248	196.8	294	237.1	340	265.6	386	253.9
249	188.6	295	232.3	341	196	387	312.1
250	159.2	296	187.3	342	271.4	388	245.6
251	200.8	297	189.2	343	243.3	389	311.2
252	204.9	298	231	344	264.6	390	307.1
253	164.3	299	239.2	345	175.1	391	305.5
254	171.6	300	238.1	346	266.2	392	316.1
255	205.6	301	238.9	347	279.8	393	319.5

256	209.8	302	232.3	348	220.3	394	255.8
257	187.6	303	233.1	349	192.8	395	292.6
258	170.9	304	233.8	350	246.5	396	242.9
259	156	305	189.4	351	200.6	397	315.1
260	206.3	306	239.1	352	275	398	256.8
261	205.5	307	234.4	353	232.2	399	302.3
262	181.9	308	240.6	354	274.4	400	317.5
263	205.5	309	245.2	355	284	401	318.3
264	203.1	310	200	356	289.4	402	245.1
265	199.2	311	246.8	357	266.4	403	276
266	211.1	312	177.3	358	271.2	404	331.1
267	213.6	313	211.5	359	221.6	405	263
268	212.7	314	249.2	360	290.3	406	324.8
269	211.8	315	252	361	261.6	407	228.7
270	209.3	316	235.8	362	287.3	408	300
271	176	317	194.5	363	228.3	409	324.6
272	175.5	318	244.6	364	293.5	410	256.3
273	213.3	319	247.3	365	292	411	318.6
274	222.8	320	248.1	366	234.6	412	327
275	206.8	321	248.8	367	291.3	413	317.7
276	167.3	322	192.8	368	227.2	414	339.3

APPENDIX A2

SWARM ORCHESTRATOR:

Table 6: measurements of throughput(tput) as the number of containers is increased from one to 500

Container	Tput	container	tput	container	tput	container	tput	container	tput
1	1.7	51	81.0	101	162.9	151	260.3	201	340.7
2	2.8	52	88.1	102	150.0	152	257.6	202	315.6
3	4.9	53	88.3	103	166.1	153	186.6	203	338.3
4	6.6	54	70.1	104	115.6	154	265.5	204	313.8
5	8.6	55	91.7	105	178.0	155	258.3	205	347.5
6	10.0	56	98.2	106	186.0	156	255.7	206	355.2
7	11.5	57	95.0	107	178.3	157	296.2	207	323.4
8	13.3	58	71.6	108	177.0	158	263.3	208	352.5
9	15.3	59	98.3	109	181.7	159	274.1	209	342.6
10	16.9	60	100.0	110	117.0	160	266.7	210	344.3
11	18.3	61	100.0	111	188.1	161	268.3	211	376.8
12	20.7	62	92.5	112	189.8	162	265.6	212	353.3
13	19.7	63	105.0	113	179.4	163	281.0	213	355.0
14	19.7	64	108.5	114	183.9	164	215.8	214	339.7
15	23.8	65	110.2	115	194.9	165	305.6	215	370.7
16	27.6	66	111.9	116	193.3	166	276.7	216	366.1
17	27.0	67	121.8	117	191.8	167	253.0	217	361.7
18	26.5	68	107.9	118	178.8	168	284.7	218	275.9
19	28.8	69	119.0	119	205.2	169	281.7	219	353.2
20	31.7	70	107.7	120	166.7	170	293.1	220	333.3
21	30.0	71	126.8	121	189.1	171	275.8	221	368.3
22	34.9	72	120.0	122	203.3	172	307.1	222	376.3
23	38.3	73	109.0	123	195.2	173	279.0	223	371.7
24	42.1	74	125.4	124	210.2	174	271.9	224	386.2
25	29.1	75	127.1	125	211.9	175	286.9	225	368.9
26	40.6	76	135.7	126	217.2	176	247.9	226	269.0
27	43.5	77	106.9	127	219.0	177	310.5	227	366.1
28	44.4	78	121.9	128	216.9	178	287.1	228	386.4
29	47.5	79	136.2	129	230.4	179	293.4	229	394.8
30	46.2	80	135.6	130	228.1	180	305.1	230	396.6
31	50.8	81	124.6	131	222.0	181	296.7	231	339.7
32	53.3	82	134.4	132	231.6	182	313.8	232	386.7
33	47.8	83	140.7	133	211.1	183	315.5	233	394.9
34	52.3	84	133.3	134	223.3	184	311.9	234	403.4
35	56.5	85	149.1	135	225.0	185	220.2	235	385.2

36	57.1	86	138.7	136	238.6	186	295.2	236	421.4
37	60.7	87	150.0	137	214.1	187	316.9	237	388.5
38	64.4	88	151.7	138	184.0	188	324.1	238	403.4
39	63.9	89	103.5	139	231.7	189	245.5	239	419.3
40	65.6	90	155.2	140	241.4	190	322.0	240	369.2
41	66.1	91	146.8	141	243.1	191	303.2	241	422.8
42	67.7	92	161.4	142	189.3	192	304.8	242	410.2
43	69.4	93	157.6	143	234.4	193	306.3	243	379.7
44	66.7	94	164.9	144	244.1	194	298.5	244	406.7
45	75.0	95	163.8	145	241.7	195	319.7	245	422.4
46	74.2	96	160.0	146	208.6	196	225.3	246	396.8
47	79.7	97	142.6	147	245.0	197	317.7	247	380.0
48	77.4	98	158.1	148	246.7	198	319.4	248	381.5
49	81.7	99	159.7	149	248.3	199	343.1	249	401.6
50	83.3	100	163.9	150	254.2	200	322.6	250	431.0

Container	tput	container	tput	Container	tput	container	tput	container	tput
251	353.5	301	493.4	351	351.0	401	691.4	451	739.3
252	420.0	302	495.1	352	586.7	402	659.0	452	753.3
253	395.3	303	459.1	353	560.3	403	639.7	453	781.0
254	445.6	304	498.4	354	590.0	404	684.7	454	769.5
255	432.2	305	516.9	355	572.6	405	698.3	455	745.9
256	441.4	306	478.1	356	593.3	406	654.8	456	786.2
257	421.3	307	487.3	357	585.2	407	667.2	457	774.6
258	403.1	308	504.9	358	459.0	408	668.9	458	704.6
259	424.6	309	515.0	359	552.3	409	705.2	459	778.0
260	426.2	310	373.5	360	631.6	410	621.2	460	807.0
261	372.9	311	501.6	361	547.0	411	685.0	461	731.7
262	429.5	312	537.9	362	593.4	412	675.4	462	810.5
263	445.8	313	513.1	363	625.9	413	688.3	463	771.7
264	440.0	314	541.4	364	596.7	414	667.7	464	800.0
265	331.3	315	500.0	365	629.3	415	669.4	465	762.3
266	429.0	316	554.4	366	677.8	416	682.0	466	763.9
267	452.5	317	511.3	367	601.6	417	719.0	467	791.5
268	454.2	318	521.3	368	584.1	418	708.5	468	793.2
269	420.3	319	483.3	369	615.0	419	735.1	469	744.4
270	442.6	320	551.7	370	493.3	420	736.8	470	839.3
271	423.4	321	501.6	371	639.7	421	690.2	471	692.6
272	438.7	322	536.7	372	600.0	422	727.6	472	828.1
273	390.0	323	504.7	373	643.1	423	640.9	473	775.4
274	434.9	324	540.0	374	613.1	424	743.9	474	640.5
275	443.5	325	550.8	375	614.8	425	494.2	475	848.2

276	467.8	326	509.4	376	637.3	426	734.5	476	806.8
277	469.5	327	554.2	377	661.4	427	700.0	477	851.8
278	421.2	328	537.7	378	630.0	428	658.5	478	810.2
279	465.0	329	557.6	379	611.3	429	516.9	479	870.9
280	474.6	330	578.9	380	678.6	430	728.8	480	716.4
281	468.3	331	542.6	381	443.0	431	684.1	481	829.3
282	414.7	332	553.3	382	647.5	432	533.3	482	831.0
283	479.7	333	564.4	383	683.9	433	687.3	483	832.8
284	465.6	334	575.9	384	650.8	434	711.5	484	864.3
285	282.2	335	587.7	385	621.0	435	713.1	485	881.8
286	476.7	336	579.3	386	448.8	436	714.8	486	771.4
287	503.5	337	561.7	387	678.9	437	633.3	487	854.4
288	480.0	338	520.0	388	636.1	438	782.1	488	774.6
289	473.8	339	565.0	389	637.7	439	731.7	489	873.2
290	475.4	340	576.3	390	520.0	440	628.6	490	859.6
291	485.0	341	568.3	391	630.6	441	723.0	491	846.6
292	503.4	342	570.0	392	642.6	442	669.7	492	878.6
293	496.6	343	612.5	393	644.3	443	703.2	493	896.4
294	466.7	344	563.9	394	571.0	444	727.9	494	823.3
295	427.5	345	547.6	395	693.0	445	635.7	495	853.4
296	501.7	346	607.0	396	628.6	446	743.3	496	870.2
297	503.4	347	578.3	397	661.7	447	745.0	497	887.5
298	458.5	348	589.8	398	663.3	448	711.1	498	844.1
299	482.3	349	471.6	399	712.5	449	736.1	499	875.4
300	483.9	350	583.3	400	645.2	450	775.9	500	877.2

APPENDIX C

CONTAINER START DELAY MEASUREMENTS

```
khakame@default:~$ cd demo1/demo1
khakame@default:~/demo1/demo1$ eval $(docker-machine env manager1)
khakame@default:~/demo1/demo1$ swarm-bench -c 2 -n 10 -i nginx
[ 10%] 1/10 containers started
[ 20%] 2/10 containers started
[ 30%] 3/10 containers started
[ 40%] 4/10 containers started
[ 50%] 5/10 containers started
[ 60%] 6/10 containers started
[ 70%] 7/10 containers started
[ 80%] 8/10 containers started
[ 90%] 9/10 containers started
[100%] 10/10 containers started

Time taken for tests: 2.682s
Time per container: 262.784ms [mean] | 855.543ms [90th] | 981.566ms [99th]
```

Figure 42: Mean container start up measurements for 10 containers showing the 90th percentile and 99th percentile

```
khakame@default:~/demo1/demo1$ swarm-bench -c 2 -n 20 -i nginx
[  5%] 1/20 containers started
[ 10%] 2/20 containers started
[ 15%] 3/20 containers started
[ 20%] 4/20 containers started
[ 25%] 5/20 containers started
[ 30%] 6/20 containers started
[ 35%] 7/20 containers started
[ 40%] 8/20 containers started
[ 45%] 9/20 containers started
[ 50%] 10/20 containers started
[ 55%] 11/20 containers started
[ 60%] 12/20 containers started
[ 65%] 13/20 containers started
[ 70%] 14/20 containers started
[ 75%] 15/20 containers started
[ 80%] 16/20 containers started
[ 85%] 17/20 containers started
[ 90%] 18/20 containers started
[ 95%] 19/20 containers started
[100%] 20/20 containers started

Time taken for tests: 3.731s
Time per container: 178.140ms [mean] | 446.745ms [90th] | 599.071ms [99th]
```

Figure 43: Mean container start up measurements for 20 containers

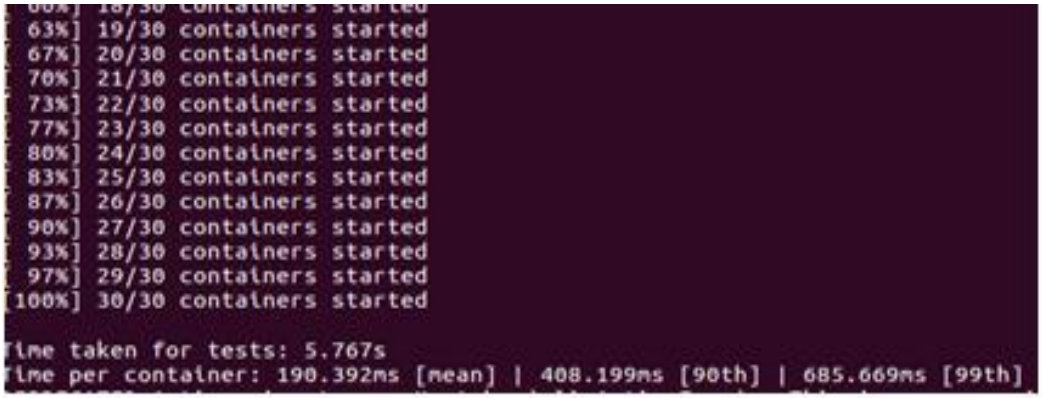


Figure 44: Mean container start up measurements for 30 containers

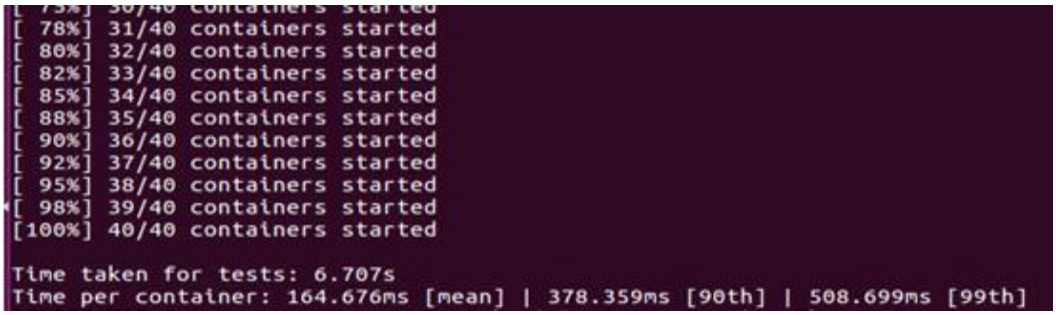
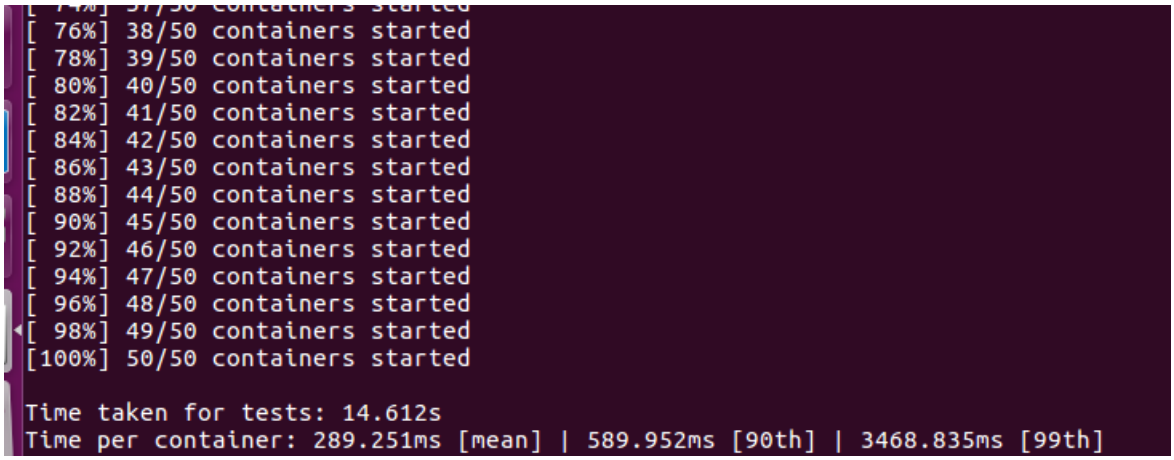


Figure 45: Mean container start up measurements for 50 containers



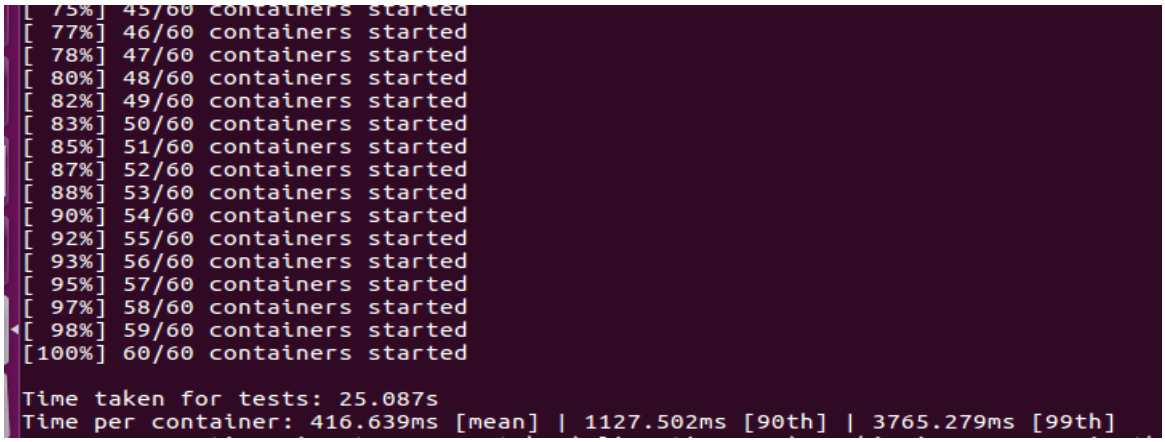


Figure 47: Mean container start up measurements for 60 container

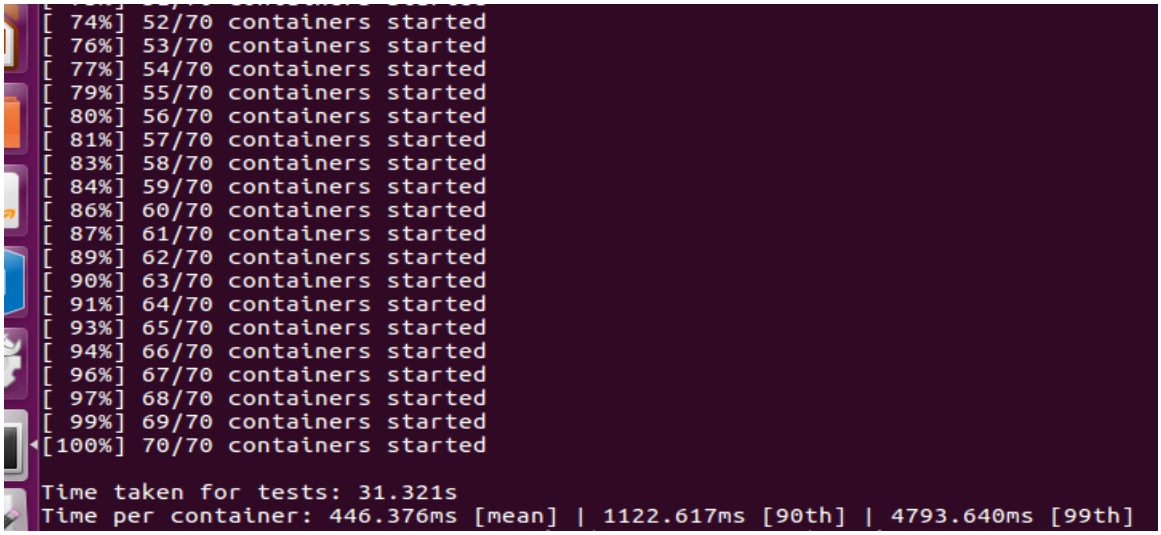


Figure 48: Mean container start up measurements for 10 containers