Building Resource Aware Self-Healing Modern Cloud Native Environment on Commodity Hardware for Adoption of Microservice Architecture

S M Didarul Abedin
Student Id: 012202067

A Project

in

The Department

of

Computer Science and Engineering



Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Science in Computer Science and Engineering

United International University

Dhaka, Bangladesh

August 2022

© S M Didarul Abedin, 2022

Approval Certificate

This project titled "**Building Resource Aware Self-Healing Modern Cloud Native Environment on Commodity Hardware for Adoption of Microservice Architecture**" submitted by **S M Didarul Abedin, Student ID: 012202067**, has been accepted as Satisfactory in fulfillment of the requirement for the degree of Master of Science in Computer Science and Engineering on 14 August 2022.

Board of Examiners

1.

_____ Supervisor

Professor Dr. A.K.M. Muzahidul Islam
Professor
Department of Computer Science & Engineering (CSE)
United International University (UIU)
Dhaka – 1212, Bangladesh

2.

_____ Co-Supervisor

Professor Dr. Salekul Islam
Professor & Head of the Department
Department of Computer Science & Engineering (CSE)
United International University (UIU)
Dhaka – 1212, Bangladesh

3.

_____ Examiner

Professor Dr. Hasan Sarwar
Professor
Department of Computer Science & Engineering (CSE)
United International University (UIU)
Dhaka – 1212, Bangladesh

4.

_____ Ex-Officio

Professor Dr. Mohammad Nurul Huda
Professor & Director
Department of Computer Science & Engineering (CSE)
United International University (UIU)
Dhaka – 1212, Bangladesh

Declaration

This is to certify that the work entitled "**Building Resource Aware Self-Healing Modern Cloud Native Environment on Commodity Hardware for Adoption of Microservice Architecture**" is the outcome of the research carried out by me under the supervision of **Professor Dr. A.K.M. Muzahidul Islam** and co-supervision **Professor Dr. Salekul Islam.**



S M Didarul Abedin
Student ID: 012202067
MSCSE Program
Department of Computer Science & Engineering (CSE)
United International University (UIU)
Dhaka – 1212, Bangladesh

In my capacity as supervisor of the candidate's project, I certify that the above statements are true to the best of my knowledge.

Professor Dr. A.K.M. Muzahidul Islam
Professor
Department of Computer Science & Engineering (CSE)
United International University (UIU)
Dhaka – 1212, Bangladesh

Professor Dr. Salekul Islam
Professor & Head of the Department
Department of Computer Science & Engineering (CSE)
United International University (UIU)
Dhaka – 1212, Bangladesh

Abstract

Years ago, even today most of the applications were developed and architected in old school monolithic approach where each of the components of the software are tightly coupled; running on a single process or multiple number of processes spread across several servers, physical or virtual. This server-side enterprise application must support a variety of different clients including desktops browsers, mobile browsers, and native mobile application. The application might expose Application Program Interface (APIs) to consume, also for integrating with other application, accesses the database systems and exchanging messages. Due to the nature of this architecture, the release cycles are very slow and updated relatively infrequent. On the other hand, as the application is tightly coupled, it has much dependency on the operating systems. In case of failure, migrations of application are a daunting task; even also scaling. Though virtualization technology can mitigate those issues like availability on failure or scaling, at the same time it increases the operational responsibilities to maintain. Patching the operating systems, doing vulnerability scan at regular interval, mitigate them where necessary. The whole process makes life harder for developers and system administrator. Moreover, the application is fully dependent on this entire cycle because it is residing on top of operating system and these types of application are not resource aware. Thus, resource optimization is not possible in many cases.

Today, these legacy applications, most of them monolithic in architecture, are slowly being broken down into smaller, independent running components called microservices. As microservices are independent and decoupled from each other, they can be developed, deployed, updated, and scaled individually at any time. This enables business to change components quickly and as often as necessary to keep pace with today's rapidly changing customer requirements and business goals. With the help of few open-source technology, today, we can build a modern cloud native environment anywhere with traditional commodity hardware which allows us to develop, build, run, deploy and scale modern microservices application which are fully resource aware, where microservices scaled automatically based on resource requirement and descale whenever required. At the same time, application by itself can heal whenever required.

Acknowledgement

I would like to express my special thanks of gratitude to my supervisor as well as Co-supervisor who gave me the excellent opportunity to do this wonderful project on the topic (Building Resource Aware Self-Healing Modern Cloud Native Environment on Commodity Hardware for Adoption of Microservice Architecture), which also helped me in doing a lot of research. Cloud Native technologies are relatively new gears to adopt, and I came to learn about so many new things. I am really thankful to them.

Table of Contents

LIST OF TABLES vii

LIST OF FIGURES viii

1. Introduction 1

2. Background and Literature Review 4

2.1 Issues in Monolithic Architecture 6

2.1.1 Time Complexity 6

2.1.2 Code & Deployment Complexity 6

2.1.3 Large Development Cycle 7

2.1.4 Slow Time to Market 7

2.1.5 Expensive 7

2.1.6 Compliance 7

2.1.7 Complex Problem Identification 7

2.1.8 Vendor Lock-In 7

2.2 Advantages on Microservice Architecture 8

2.2.1 Flexibility 9

2.2.2 Lightness 9

2.2.3 Portability 10

2.2.4 Scalability & Agility 10

2.2.4 Open-Source Technology 11

2.2.5 Increased Productivity 11

3. Methodology 12

3.1 Kubernetes Architecture 14

3.1.1 Control Plane Components 14
3.1.2 Node Components 16
3.2 Kubernetes Features 17
3.3 Experimental Setup 18
4. Results 22
4.1 Auto-heal (pod unavailability) 22
4.2 Auto-heal (Service unavailability): 26
4.3 Resource Aware auto scale-up & scale-down 28
4.4 Performance Analysis between Monolithic & Microservice based application 34
4.4.1 Performance measurement of microservice application 34
4.4.2 Performance measurement of monolithic application 39
4.4.3 Application Response Time Comparison between Microservice vs Monolith 41
5. Conclusion 43
6. References 44

LIST OF TABLES

Table 1: Bare Metal/ Virtual Machine vs. Container 10

Table 2: Hardware and software stack 18

Table 3: Kubernetes cluster node details 19

Table 4: php-apache webserver deployment manifest 23

Table 5: Deployment status 34

Table 6: Replicaset status 34

Table 7: Container and Pod status 34

Table 8: Load balancer service status 35

Table 9: php-nginx container details 35

Table 10: Web service response time before and during load 38

Table 11: Web service response time summary with and without load 41

LIST OF FIGURES

Figure 1: A monolithic architecture e-commerce application 1

Figure 2: Evolution of infrastructure and application architecture 4

Figure 3: Microservice architecture of e-commerce application 5

Figure 4: Monolithic vs Microservice architecture component 6

Figure 5: Application layers in monolithic and microservice architecture 8

Figure 6: Prominent microservices features 9

Figure 7: Scalability option of Monolith vs. Microservice 11

Figure 8: Technological progress over the years 12

Figure 9: Traditional vs. Virtualization architecture 13

Figure 10: Kubernetes architecture 15

Figure 11: List of Kubernetes open API's 16

Figure 12: Few features of Kubernetes environment 18

Figure 13: Lab environment 18

Figure 14: Kubernetes cluster node usage 19

Figure 15: Kubernetes cluster control plane components and pods 19

Figure 16: sni-k8s-test-01 node details 20

Figure 17: sni-k8s-test-02 node details 20

Figure 18: sni-k8s-test-03 node details 20

Figure 19: Cluster details and status 21

Figure 20: Replication controller ensures self-healing of components 23

Figure 21: Deployed test container web application php-apache 24

Figure 22: Accessing container-based web application 24

Figure 23: Delete the pod and associated container 24

Figure 24: New container creation as old one terminated 25

Figure 25: Accessing container-based web application.....25
Figure 26: Pods CPU & memory utilization 25
Figure 27: Pods availability with liveness probe.....26
Figure 28: Stopping apache process 26
Figure 29: Container get restarted once detected by Liveness probe 26
Figure 30: Pods CPU & memory utilization 27
Figure 31: Configured Horizontal Pod Autoscaler (HPA) 28
Figure 32: Traffic load generation to web server 28
Figure 33: Load generator utilization (CPU & Memory).....29
Figure 34: Load generator utilization (Network).....29
Figure 35: Inspect HPA with targets and replicas 29
Figure 36: New container creation based on HPA targets.....30
Figure 37: Container CPU usage trend on load test 30
Figure 38: HPA target and replica decreases once the load generation stopped.....31
Figure 39: CPU usage pattern on load generation.....31
Figure 40: System CPU and load average status before load test 32
Figure 41: System CPU and load average usage during load test.....33
Figure 42: Deployment, Replicaset & Pod.....34
Figure 43: php-nginx web server service.....35
Figure 44: php-nginx container status 36
Figure 45: php-nginx web service response time & code 36
Figure 46: Load generation and Horizontal Pod Scaling (HPA) status.....36
Figure 47: CPU, memory & network status of load generator.....36
Figure 48: CPU usage of php-nginx pods.....37
Figure 49: Memory usage of php-nginx pods 37

Figure 50: Network usage of php-nginx pods 37
Figure 51: php-nginx web service response time & code 37
Figure 52: Microservice web service response time comparison 38
Figure 53: CNN RSS channel web service 39
Figure 54: TCP connection and network status in normal state 39
Figure 55: Web service response time & status without load 40
Figure 56: TCP connection and network status during load 40
Figure 57: Web service response time & status with load 40
Figure 58: Monolith web service response time comparison 41
Figure 59: Comparison between application without load 42
Figure 60: Comparison between application with load 42
Figure 61: Comparison between application in hourly average 42

Chapter 1

Introduction

Monolithic applications consist of components that are tightly coupled together and has designed, developed, deployed, and managed as one single entity, because they all run as a single OS process or a small number of processes running on servers. Figure 1 shows a sample monolithic architect e-commerce application where apache tomcat is used as a webserver, the application is developed in java and deployed as a java war file. A single webserver process runs in the server serving all services e.g., accounting, inventory, shipping etc. Changes to one part of the application require rearchitecting the entire application stack and redeployment of the whole application, and over time the lack of hard boundaries between the parts results in the increase of complexity and consequential deterioration of the quality of the whole system because of the unconstrained growth of inter-dependencies between these parts. (Villamizar, et al., 2015)

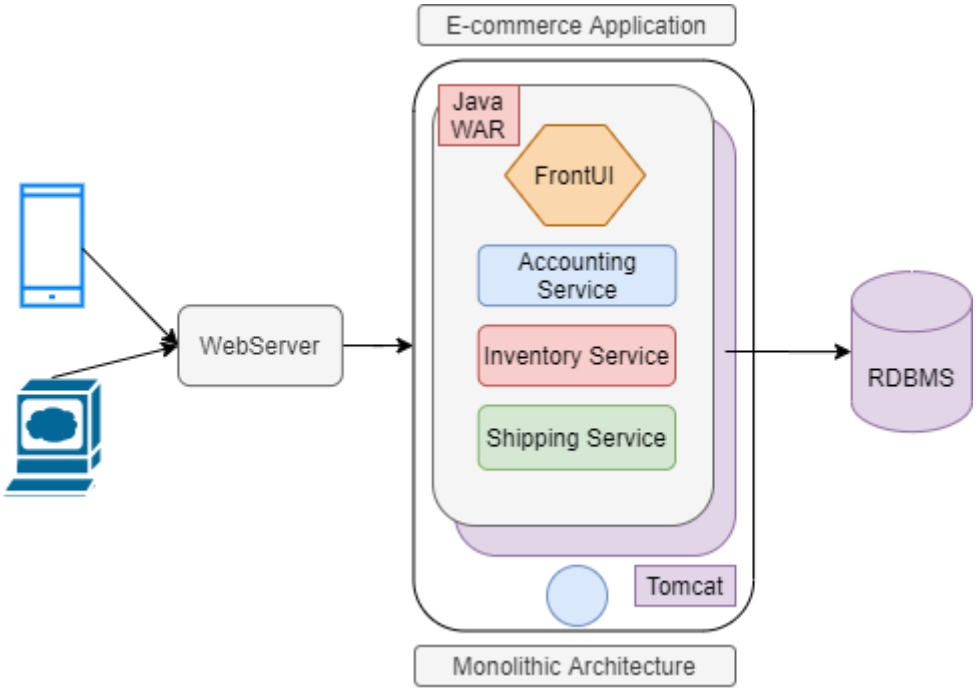


Figure 1: A monolithic architecture e-commerce application

Running a monolithic application usually requires several powerful servers that can provide enough resources for running the application and take anticipated loads. To deal with increasing loads on the system, resources can be scaled up vertically (also known as scaling up) by adding more CPUs, memory, and other server components, which requires a downtime of application/services or scale the whole system horizontally, by setting up additional servers and running multiple copies (or replicas) of an application (scaling out). However, in the later method, rearchitecting of application and additional component such as Global Server Load Balancing (GSLB) is required to load balance the application across globally distributed servers. While scaling up usually does not require any changes to the application it always has an upper limit. Scaling out, on the other hand, is relatively cheap hardware-wise, but may require big changes in the application code and isn't always possible—certain parts of an application are extremely hard or next to impossible to scale horizontally (relational databases, for example). If any part of a monolithic application isn't scalable, the whole application becomes unscalable as shown in Figure 1. "These and other problems have forced to start splitting complex monolithic applications into smaller independently deployable components called microservices. (Luksa, 2017) Each microservice runs as an independent process and communicates with other microservices through simple, well-defined application program interfaces (APIs)."

In traditional monolithic architecture, application suffers to furnish sudden additional requests with growing business need which comes without any prediction and applications cannot be built as resource aware. On other hand, applications failed to heal automatically if it encounters any problem.

Overall motivation of the project work is to build a modern cloud native environment on top of any commodity hardware with few opensource software's which helps us to achieve cloud computing features like elasticity, scalability, distributed and resilient infrastructure where small, independent, portable applications can be deployed in automated manner through a CI/CD (Continuous Integration/Continuous Delivery) pipeline.

Scope of work would be limited to build a modern cloud native computing environment with listed products and software's and test distributed and resiliency, resource awareness

and self-healing capability of applications. Contributions of this project work are listed as follows:

- Build a single master – multiple worker Kubernetes cluster
- Examined resource awareness feature of microservice workload
- Examined self-healing feature of microservice workload
- Examined distributed and resiliency of microservice workload
- Comparison between monolithic and microservice architecture
- Application performance measurement and evaluation between monolithic and microservice application

This chapter defines the brief introduction of the technology, problem statement and scope of work of the project work. The following chapter elaborately contains the entire infrastructure and application stack, application architecture and development strategy and application hosting methodologies. This chapter also describes advantages and disadvantages of monolithic and microservice architecture. The third chapter contains the procedure of preparing the experimental environment, stated in details of monolith and microservice architecture, describes Kubernetes architecture and all its components and functionalities. Later part of this chapter also contains experimental environment details. Results and comparisons are summarized in fourth chapter in this report. Finally, a conclusion was drawn in fifth chapter.

Chapter 2

Background and Literature Review

Technology has evolved a lot over the years. In early days, legacy applications were built in monolithic style, deployed in bare metal server, or hosted in mainframe systems. In early 2000s, virtualization or hypervisor technology evolves, and organizations were started adopting. With the help of virtualization technology, public cloud industry intensified a lot. Then applications were built on Service-oriented architecture (SOA) and deployed on virtual machines where multiple virtual machines can run on a single host making a good utilization of resources. Over last half of decade, application architecture changes from older apps to modern apps and instead of virtualization, a lightweight, robust, infrastructure independent container platform is used to deploy applications in public, private or hybrid cloud environment.

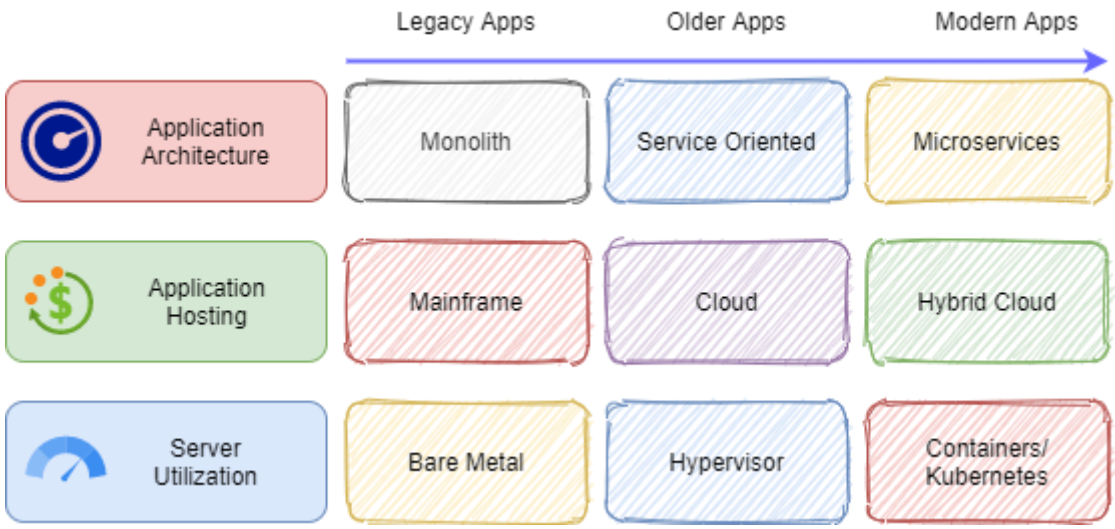


Figure 2: Evolution of infrastructure and application architecture

In old school architecture, the application is developed and deployed as a single monolithic application. For example, a Java web application consists of a single WAR file that runs on a web server such as Tomcat (Figure 1), multiple services are tightly coupled and bundled in a single process. Multiple instances may run on several servers

behind a load balancer to make sure the high-availability and scalability of the application. This e-commerce application can be broken down into small number of parts, decoupling the interdependent components to independent component, called microservices where each microservice is connected to other microservices as Figure 3. In this architecture, any microservice can be scaled at any time based on requirement, can be turned off due to maintenance purpose or even can introduce new service in between keeping the application alive. Additionally, each microservices are resource aware. Thus, based on user request and load those microservices can scale up horizontally and vertically. Once the additional requests have been served and system load become normal, microservices can scale down automatically without any intervention from administrator or DevOps team. Furthermore, microservices are built with self-healing capability. So, whenever a microservice become unhealthy and cannot serve requests, it gets remediated and healed automatically with an immutable image from private or public container image registry.

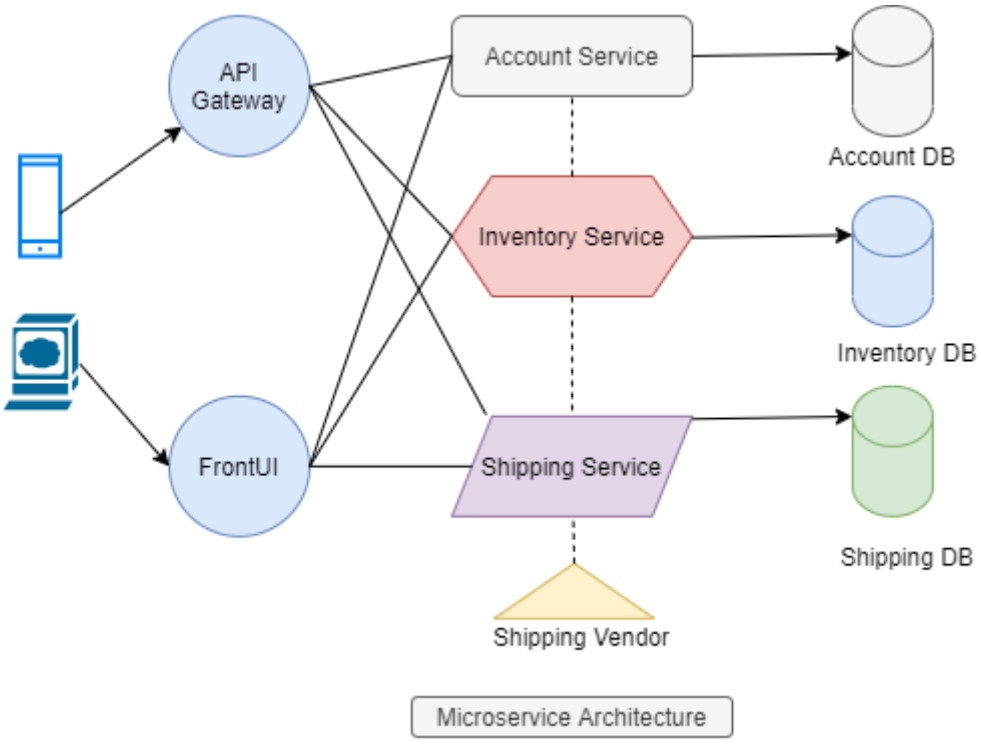


Figure 3: Microservice architecture of e-commerce application

2.1 Issues in Monolithic Architecture

Many research has been conducted on monolithic and microservices architecture to study the advantages and disadvantages of these two architectures. (Gos & Zabierowski, 2020) (Lauretis, 2019)

In the monolithic architecture, applications are created of different services that are tightly joined and dependent with one another in a single codebase. This can create plenty of difficulties for teams working in the same environment. Therefore, many companies are moving toward microservices architecture to enable their development teams to coordinate with each other easily. (Opara-Martins, Sahandi, & Tian, 2016)

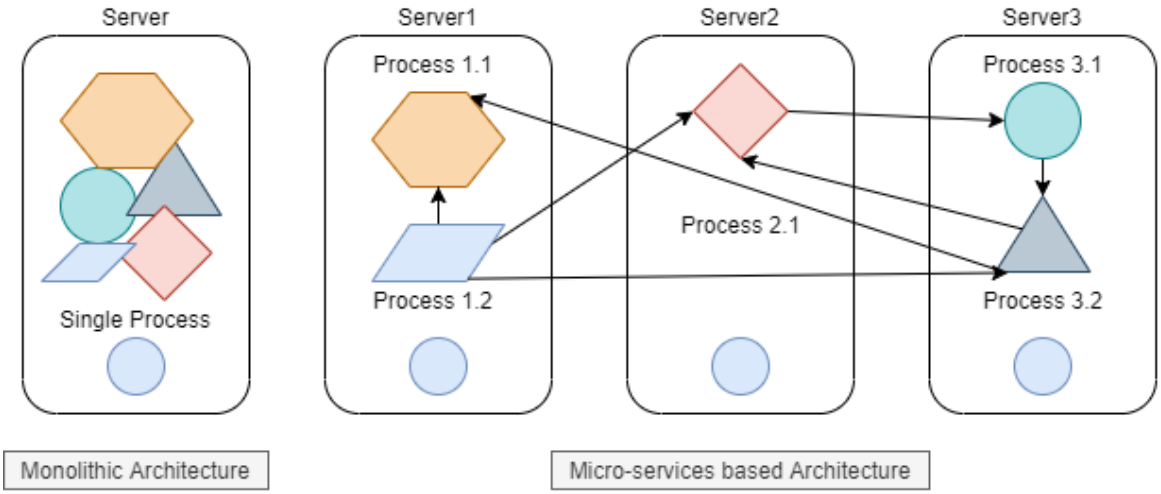


Figure 4: Monolithic vs Microservice architecture component

2.1.1 Time Complexity

By the nature of the architecture, as the components are interdependent and depend on each other, thus, to design and implement a business process is very complex and tedious task whether it is a new process or redefining an old one. (Villamizar, et al., 2015)

2.1.2 Code & Deployment Complexity

For inter dependency, it is hard to maintain and make changes in the code base of the application because the code base is very large and complex. (Gos & Zabierowski, 2020)

2.1.3 Large Development Cycle

As it requires much time to redefine an existing business process or introduce a new business functionality thus the development cycle is invariably infrequent and time to market is large compared to other products. (Villamizar, et al., 2015)

2.1.4 Slow Time to Market

Large development cycle slows down the time to market. Thus, in long run business cannot sustain because it is unable to meet customer's demand in right time. (Kalske, Mäkitalo, & Mikkonen, 2018)

2.1.5 Expensive

As the only way to scale the infrastructure is scale up or scale out thus managing infrastructure is expensive and multiple teams is required whether it is on-premises or cloud environment. (Kalske, Mäkitalo, & Mikkonen, 2018)

2.1.6 Compliance

Compliance limits flexibility of the application as components are tightly coupled. So, to ensure appropriate compliance, the whole stack of application should be compliant.

2.1.7 Complex Problem Identification

Due to the large code base and dependent business process and logic, problem identification is relatively complex compared to other architecture.

2.1.8 Vendor Lock-In

Another major concern in cloud computing environment is vendor lock-in. It is a situation in which a customer who is using a product or a service from a particular cloud service provider cannot easily transition to another. This is treated as one of the major barriers to the adoption of cloud computing mostly due to lack of standardization and interoperability to each other. (Opara-Martins, Sahandi, & Tian, 2016) Predominantly a significant cost and migration challenge are notable. During the migration time, to ensure availability of services is another challenge which should be dealt with priority.

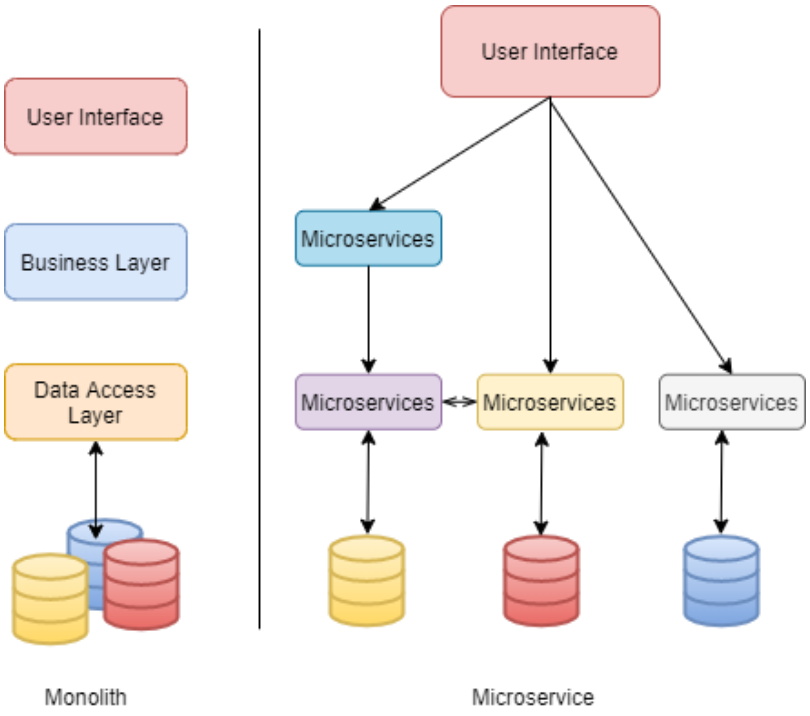


Figure 5: Application layers in monolithic and microservice architecture

2.2 Advantages on Microservice Architecture

Microservices are loosely coupled independent processes which ease the procedures of maintainability, ensure flexibility and re-usability, provide scalability, resiliency and availability, lightness, and automated deployment when it is utilized, and these are considered the advantages of microservices architecture. (Chen, Li, & Li, 2017)

First of all, microservices can rely on heterogeneous technology, which means each decoupled module can use different technology than the other services to achieve the desired goals and performance of the entire application. (Newman, 2015) (Carnell, 2017) Secondly, another advantage of microservices is if one module of the application fails, that particular service remains unavailable rather than affecting the whole application stack. This advantage is called as resilience in his book by the writer entitled Building Microservices. (Newman, 2015) (Carnell, 2017) The third advantage is that, the process of scaling can be more efficient compared to monolithic application because only the services that need actual scaling are scaled in microservices architecture, opposing to a

monolithic application requires scaling as a whole unit which may lead to additional hardware, compute resources, higher hardware usage, higher cost, and additional operational and maintenance effort. (Newman, 2015) (Carnell, 2017) Fourth, the functionality of CI/CD (Continuous Integration, Continuous Delivery) pipeline eases of deployment process where any individual module of an application can be deployed independently without affecting the performance of other services. (Amaral, et al., 2015) Few of the other advantages are discussed below:

2.2.1 Flexibility

As modules are loosely coupled and independent, it is very much flexible to alter an existing business logic and implement a new business process. (Gos & Zabierowski, 2020) (Bruce & Pereira, 2018) Also, the codebase is small and simple to understand for teams. Even problem identification is much easier because the developer or operations team does not understand the whole stack of application and process. Team can focus on every tiny microservices to identify the issue and resolution. (Tapia, et al., 2020) (Alshuqayran, Ali, & Evans, 2016)

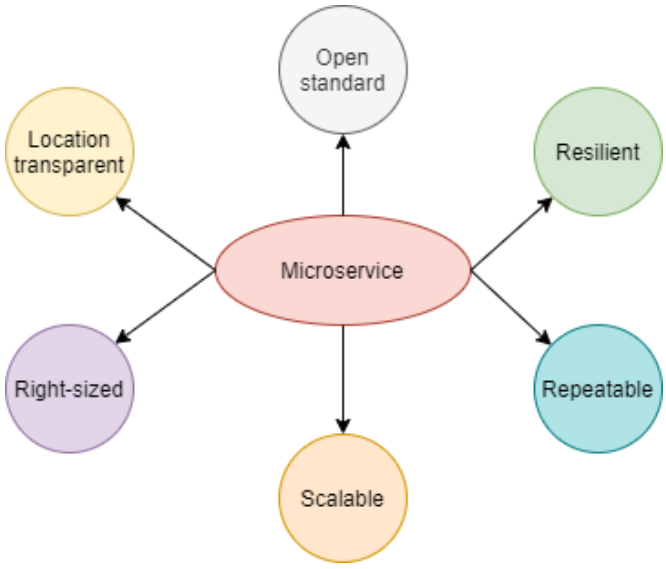


Figure 6: Prominent microservices features

2.2.2 Lightness

Microservices are deployed in containers, which is completely isolated a standard smallest unit of software, pre-configured and bundled with only necessary binaries,

libraries, and system files thus the size of the containers is very small compared to trivial operating system and application. Table 1 shows the comparative size and performance in three different categories of a nginx application. (Chen L. , 2018)

Item	Bare Metal Server/ Virtual Machine	Container
Image Size	>1024 MB	<250 MB
Deployment Time	10 Min ~ 2 Hour	2 Sec ~ 3 Sec
Restart/ Update Time	20 Sec ~ 5 Min	2 Sec ~ 3 Sec

Table 1: Bare Metal/ Virtual Machine vs. Container

2.2.3 Portability

It is treated as one of the finest features of container based microservices architecture as the application is built in the form of a container, a completely isolated and independent entity created with necessary libraries and binaries that can ship from one server to another, from on-prem to private or public cloud in a minute without interrupting the whole application stack. It is one of the premier reasons to gain much popularity in short period of time among developers. Once the code is being written, anyone from operations team can build and deploy the code without any interdependency of the application stack and complexity of the environment. (Chen L. , 2018) The term ‘DevOps’ comes into scene that, the same person can write the code, build and deploy it anywhere independently. (Balalaie, Heydarnoori, & Jamshidi, 2016)

2.2.4 Scalability & Agility

Another premier advantage microservice offers is the scalability where any of the modules, components or subcomponents can be scaled at any time based on actual resource utilization or on demand basis. (Hasselbring & Steinacker, 2017) However, monolith architecture supports only scale-up or scale-out methods increasing the cost and operational burden for team. The overall process of scale up and scale down is much agile, sometimes it’s about seconds or minutes whereas in monolithic architecture, scale up and scale down can take from hours to months.

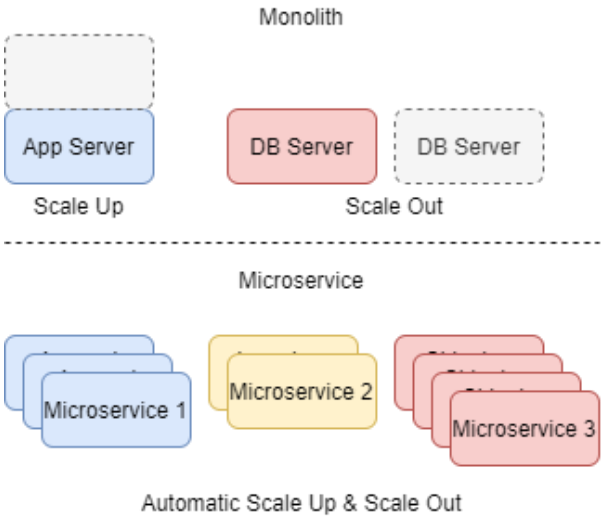


Figure 7: Scalability option of Monolith vs. Microservice

2.2.4 Open-Source Technology

To facilitate microservices architecture, the infrastructure stack is fully built with open-source technologies where everyone is free to use those resources, even can contribute. The whole idea is now governed by The Cloud Native Computing Foundation (CNCF) (https://www.cncf.io/).

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable.

2.2.5 Increased Productivity

Deployment with CD (Continuous Delivery) improves productivity of entire team. Products are deployed independently through an automated process without intervention thus have almost zero chance for error. As the deployments are independent, it takes much shorter time to deliver a product using DevOps culture. It also provides a zero-downtime product deployment or upgradation through this process. (Chen L. , 2018)

Chapter 3

Methodology

To compare the pros and cons of monolith and microservice architecture, first of all, there should be an application that can produce the results of microservices and monolithic applications, which can be compared and evaluated. At the same time, based on the findings, a comparative analysis can be done to identify the objectives of the project. To avail such, a cloud native serverless environment has been built in on-premises commodity hardware with the help of Kubernetes, an open-source container orchestration platform for automatic software deployment, scaling and management. Cloud Native architecture is a design methodology that utilizes cloud like features e.g., compute, storage, network etc. to design, architect, construct and operate workloads that are built on a highly scalable, distributed, elastic, resilient environment anywhere. Earlier waterfall development method was followed to architect monolith application, these days DevOps deployment process is followed to deploy application in the form of container.

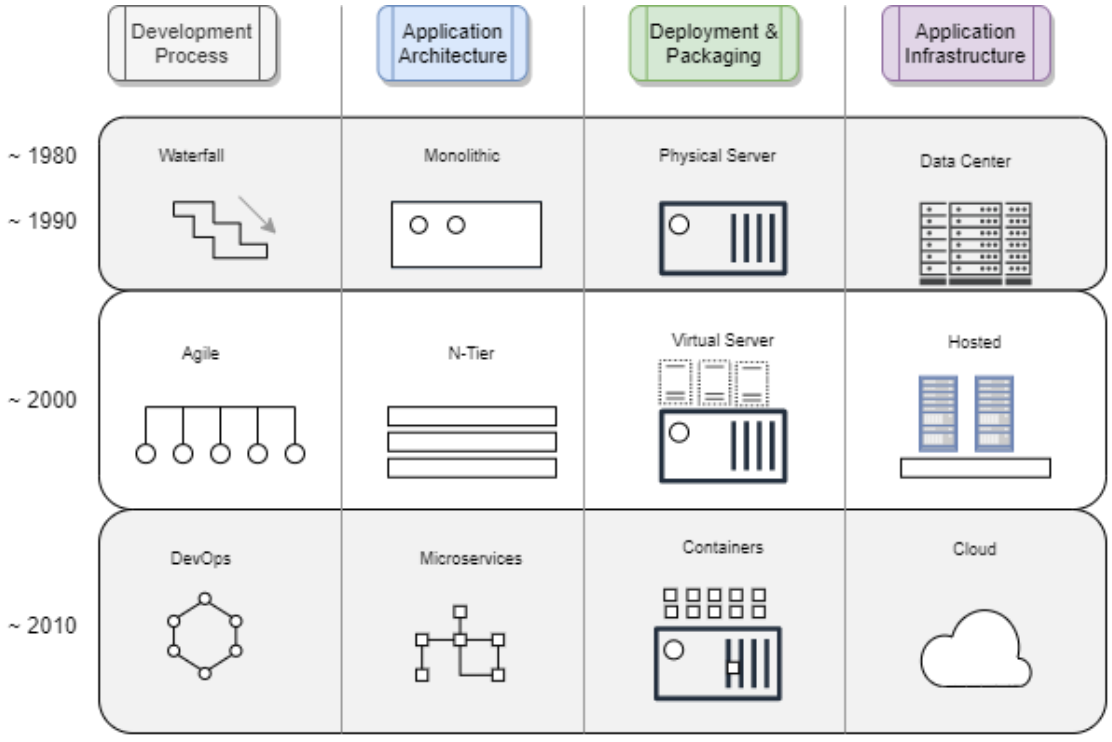


Figure 8: Technological progress over the years

Businesses are undergoing an extraordinary load on their IT infrastructure as they struggle to meet growing customer expectations for fast, reliable, and secure services. To meet increasing business requirement and customer expectations increase in compute and processing power and underlying storage capability are a day-to-day requirement which should adhere in regular basis. At the same time companies tries to find a robust, scalable, and secure IT infrastructure for continuous development and maintenance purpose. To facilitate extra processing, compute and storage, underlying network infrastructure is a mandatory requirement which should also be robust, scalable, and reliable. Space, power, and cooling requirement is the indispensable part of the architecture. (Taibi, Lenarduzzi, & Pahl, 2017)

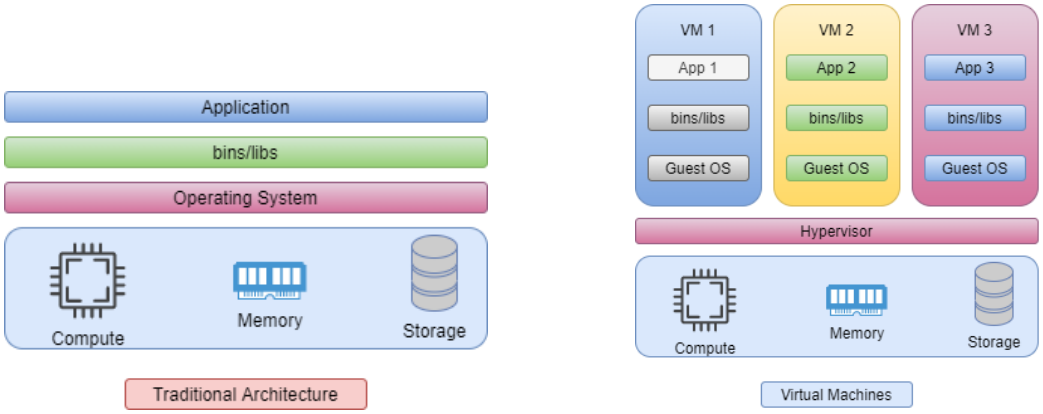


Figure 9: Traditional vs. Virtualization architecture

In traditional infrastructure Figure 9 applications are built directly on top of operating system accessing the hardware directly. For better utilization of resources and isolation, ensure redundancy and fault tolerance, virtualization technology comes into the picture. In this technology, a hypervisor in directly interacting with hardware. Applications are built on top of a guest operating system having their own set of binaries and libraries. With the added benefit, it also adds burdens of managing those environment and overall infrastructure.

Still the urge of achieving many additional features along with cloud like features in on-premises bring containerization and Kubernetes environment. During the last decade, container and Kubernetes technology evolves rapidly. Ride sharing giant Uber, financial data specialist Bloomberg, streaming service pioneer Netflix (Mauro, 2015) is using

containerization and Kubernetes for their application modernization ensuring continuous deliver and continuous integration, monitoring, testing and delivery.

3.1 Kubernetes Architecture

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. The worker node(s) host the Pods, the smallest components in Kubernetes architecture, that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers which is known as master nodes and a cluster usually runs multiple nodes, providing fault-tolerance, resiliency with distributed manner and high availability. (Luksa, 2017) (Rosso, Lander, Brand, & Harris, 2021)

3.1.1 Control Plane Components

A Kubernetes environment consists of a master and several worker nodes. The master node holds several critical components, sometimes referred to these as control plane components that plans, schedules, and monitors states of all components in the entire cluster. It continuously manages object states, responding to changes in the cluster, control the cluster, its state data, and its configuration. The Kubernetes control plane is responsible for ensuring that the Kubernetes cluster attains a desired state, defined by the user in a declarative manner. As Figure 10 shows, the control plane is made up of four major components: kube-apiserver, kubecontroller-manager, kube-scheduler and etcd. These control plane components can all run on a single master node or can be replicated across multiple master nodes on a high available architecture in a distributed manner. Worker nodes, on the other hand, holds the application in the form of containers.

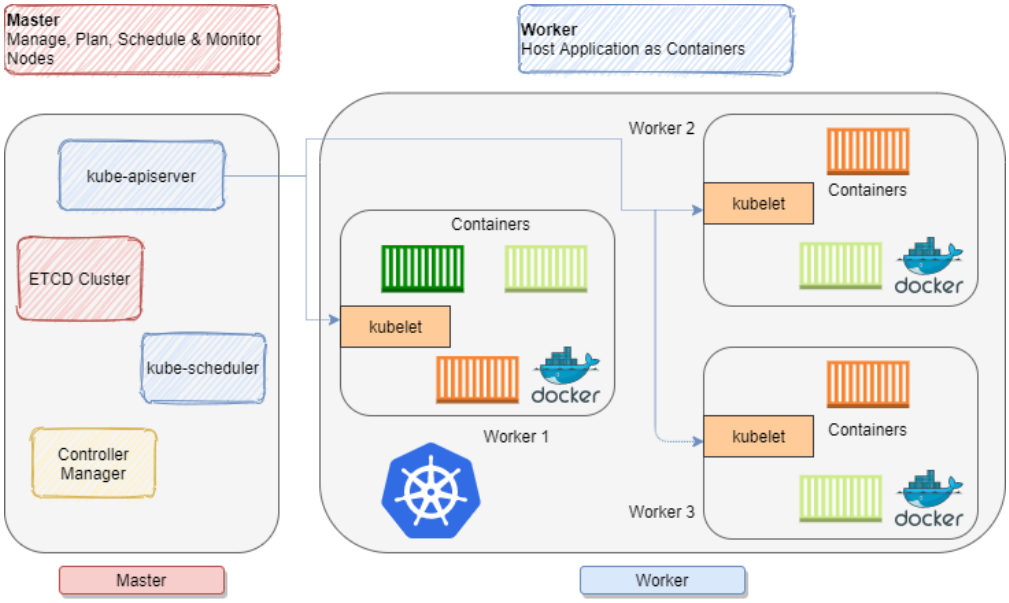


Figure 10: Kubernetes architecture

3.1.1.1 kube-apiserver

The Kubernetes API server is one of the major components which among control plane components which acts as a gateway of communication between master node and worker node. All instruction to and from master nodes to worker goes through kube-apiserver. It validates and configures data for the API objects which include pods, services, daemonsets, replicationcontrollers, and others. (Luksa, 2017) DevOps instructions either in the form of imperative approach or declarative, executes to the cluster through kube-apiserver invoking open API's (Application Programming Interface). Figure 11 shows few of the Kubernetes API's.

3.1.1.2 kube-scheduler

The Kubernetes scheduler is a control plane component or a process which assign pods to nodes. The scheduler determines which nodes are valid placements for each pod in the scheduling queue according to constraints, parameters, and available resources. The scheduler then ranks each valid node and binds the pod to a suitable node. (Rosso, Lander, Brand, & Harris, 2021) At any point of time if any of the worker node goes unavailable, kube-scheduler reschedules the pods automatically without any intervention from user or administrator.

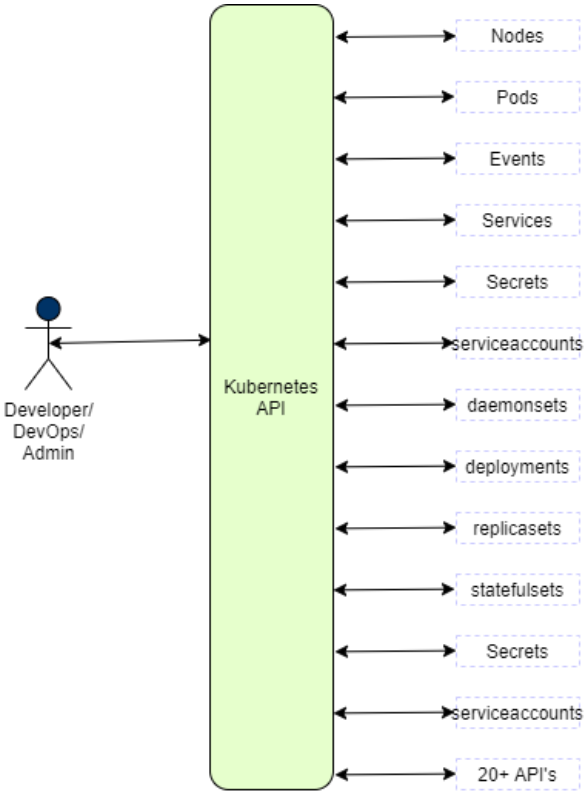


Figure 11: List of Kubernetes open API's

3.1.1.3 kube-controller-manager

The Kubernetes controller manager is one of the core components among the control plane components. In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. Among many controllers, few are replication controller, endpoints controller, deployment controller, namespace controller, and serviceaccounts controller. (Rosso, Lander, Brand, & Harris, 2021)

3.1.1.4 etcd

etcd is a strongly consistent and highly available distributed key value store with a focus on being simple, secure, fast and reliable which provides a consistent way to store data that needs to be accessed by a distributed system like Kubernetes. Kubernetes uses etcd to store all its configuration data, state, and metadata.

3.1.2 Node Components

Among the worker nodes in Kubernetes cluster, below are few major components:

3.1.2.1 kubelet

Kubelet is an agent installed and runs on each node in the cluster. Primary responsible of kubelet is to register the node with the apiserver. After a successful registration, other task of kubelet is to create pods and listen to the API server for instruction. (Rosso, Lander, Brand, & Harris, 2021)

3.1.2.2 kube-proxy

kube-proxy is a network proxy that runs on each node in the Kubernetes cluster, kube-proxy maintains a network rules on nodes which enables pod to pod communication across Kubernetes cluster. (Rosso, Lander, Brand, & Harris, 2021)

3.1.2.3 Container Runtime Interface

Container runtime is the software that is responsible for running containers. Kubernetes supports several container runtimes like Docker, containerd, CRI-O, rkt etc.

3.2 Kubernetes Features

As a leading container orchestration platform Kubernetes provides many features. Below are some of the most desirable features which are discussed in short.

- **Scheduling**: Automatically provision container instances across the cluster based on properties and requirement
- **Affinity/ Anti-Affinity**: Provision containers nearby or far apart from each other, helping availability and performance
- **Observability**: Continuous systems monitoring, collect states and metrices.
- **Failover**: Automatically detect and correct failures by re-provisioning from immutable container images taking from image registry
- **Auto Scaling**: Scale up or scale down to meet demand based on criteria and settings
- **Networking**: Manage an overlay networking for container communication with and across cluster
- **Service Discovery**: Enable containers to locate each other

- **Rolling Updates**: Coordinate incremental update with zero downtime deployment. Automatically roll back problematic changes



Figure 12: Few features of Kubernetes environment

3.3 Experimental Setup

Below hardware and software stack has been used and deployed in on-premises to build a cloud native environment with one master and two worker nodes to design and deploy microservice based application. For master and worker node, we used virtual machines with CentOS operating system which is a community edition and open to use. Other software's are listed in below table:

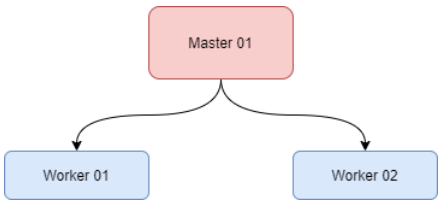


Figure 13: Lab environment

Table 2: Hardware and software stack

Hardware/Software	Purpose	When used
Hardware	To achieve computation	Any commodity bare-metal server
Docker/ Containerd	Container Runtime Environment	Open source
Kubernetes	Container Orchestration Platform	Open source
Weave/ Calico	Container Network Interface (CNI)	Open source
Storage Drivers	Container Storage Interface (CSI)	Open source
CoreDNS	Domain Naming System	Open source

Metallb/ Nginx	Ingress Load Balancer	Open source
Alpine Linux	Container Image Operating System	Open source.
Ubuntu Linux	Container Image Operating System	Open source
Grafana	Observability – Analytics & Visualization	Open source
Prometheus	Observability – Systems Monitoring	Open source
Alert Manager	Observability – Alerting	Open source

Name	Status	Roles	Version	Internal-Ip	OS-Image	Container-Runtime
sni-k8s-test-01	Ready	master	v1.19.16	10.96.38.51	CentOS Linux 7 (Core)	docker://19.3.8
sni-k8s-test-02	Ready	<none>	v1.19.16	10.96.38.52	CentOS Linux 7 (Core)	docker://19.3.8
sni-k8s-test-03	Ready	<none>	v1.19.16	10.96.38.53	CentOS Linux 7 (Core)	docker://19.3.8

Table 3: Kubernetes cluster node details

```
[root@sni-k8s-test-01 ~]# kubectl top nodes
NAME               CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
sni-k8s-test-01    215m         3%     3189Mi          54%
sni-k8s-test-02    142m         7%     2562Mi          66%
sni-k8s-test-03    201m         10%    2440Mi          63%
```

Figure 14: Kubernetes cluster node usage

```
[root@sni-k8s-test-01 ~]# kubectl get pods -n kube-system
NAME                                       READY   STATUS    RESTARTS   AGE
calico-kube-controllers-54cfbdd8db-fv9tt   1/1     Running   5          616d
calico-node-9kcqh                          1/1     Running   0          104d
calico-node-jhrpq                          1/1     Running   1          616d
calico-node-nzppq                          1/1     Running   1          339d
coredns-66bff467f8-29zxd                   1/1     Running   0          16d
coredns-66bff467f8-drkn5                   1/1     Running   1          16d
etcd-sni-k8s-test-01                       1/1     Running   1          16d
kube-apiserver-sni-k8s-test-01             1/1     Running   1          16d
kube-controller-manager-sni-k8s-test-01    1/1     Running   1          16d
kube-proxy-9hntq                           1/1     Running   1          16d
kube-proxy-dmbks                           1/1     Running   0          16d
kube-proxy-hjt42                           1/1     Running   1          16d
kube-scheduler-sni-k8s-test-01             1/1     Running   4          16d
kube-state-metrics-664f9fcc89-6jzgv        1/1     Running   0          25d
log-collector-8mgm5                        1/1     Running   7          97d
log-collector-s8xsw                        1/1     Running   6          80d
metrics-server-85655dc69d-272zs            1/1     Running   0          17d
[root@sni-k8s-test-01 ~]#
```

Figure 15: Kubernetes cluster control plane components and pods

Node: sni-k8s-test-01

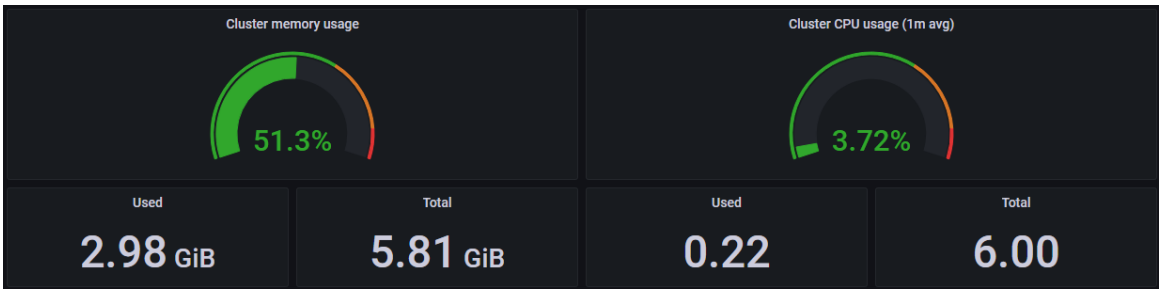


Figure 16: sni-k8s-test-01 node details

Node: sni-k8s-test-02

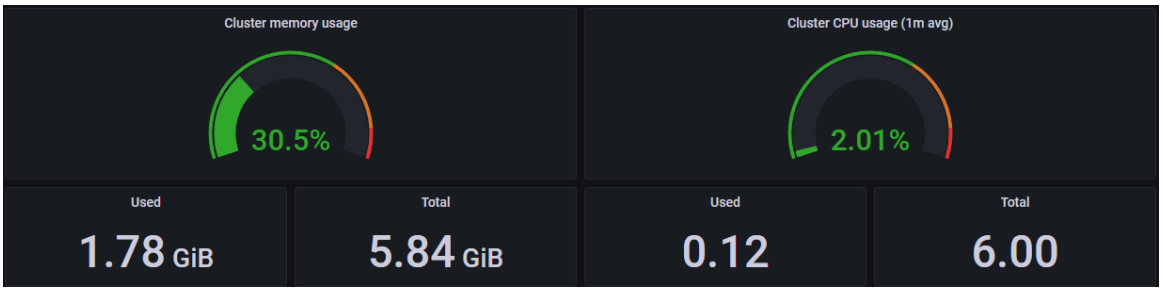


Figure 17: sni-k8s-test-02 node details

Node: sni-k8s-test-03

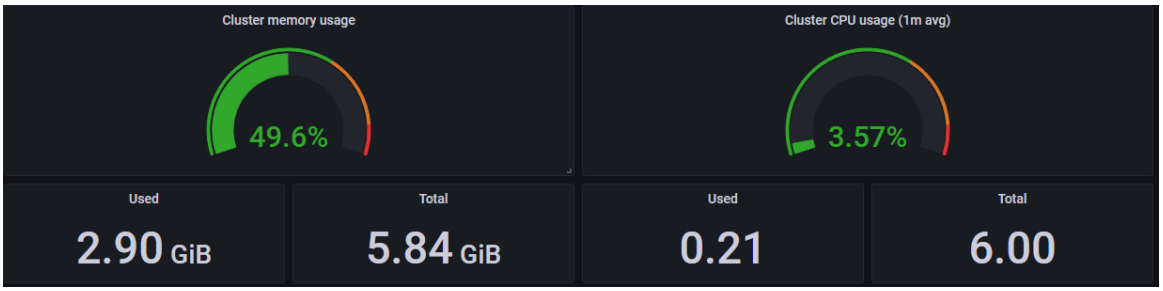


Figure 18: sni-k8s-test-03 node details

Kubernetes Cluster Status & Usage:



Figure 19: Cluster details and status

Chapter 4

Results

Among many features of cloud native environment, we tried to do examination on two features of modern application platform which Kubernetes provides. One of the premier features is self-resiliency of workloads which Kubernetes provides automatic recover of workload in distributed manner across geo location based on in build intelligence. Kubernetes always collect metrices, inspect environment and heal whenever necessary without any intervention of Devs or Ops team. Another feature we explored is resource awareness of workload. In traditional and monolithic architecture, lots of resources were unutilized with no scope of optimization. Furthermore, scale up or scale down is a very tedious and manual task to perform which costs additionally and sometimes unachievable. With Kubernetes, effortlessly we can achieve scale-up and scale-down feature based on resource utilization of the workloads at any time, optimizing the environment in distributed manner, which is another major need of modern application. In the last part of this chapter consists of a comparative analysis between a web application built on monolithic and microservice architecture.

4.1 Auto-heal (pod unavailability)

In this section, we inspect resiliency capability of containers in Kubernetes cluster. Initially, a container running a php-apache based web server is deployed. Then, we deleted the above-mentioned pod and associated container which was deployed. Kubernetes automatically detect the failure of the pod and creates the pod and container again automatically downloading the container image from local container image registry without any intervention from developer or operations team. At the backend, replication controller always checks what is the desired state and current state. If any deviation found between current state and desired state, replication controller ensures the current state be same as desired state, potentially ensured resiliency of containers and associated services.

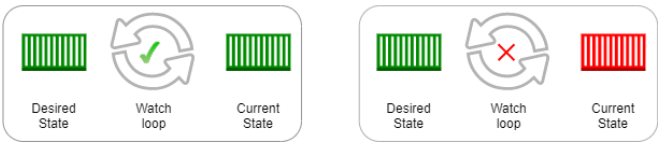


Figure 20: Replication controller ensures self-healing of components

Step - 1: Deploy php-apache container application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: 10.84.0.3/misc-public/php-apache-web
        ports:
        - containerPort: 80
          protocol: TCP
        livenessProbe:
          httpGet:
            scheme: HTTP
            path: /
             port: 80
          initialDelaySeconds: 10
          timeoutSeconds: 10
        readinessProbe:
          httpGet:
            scheme: HTTP
            path: /
             port: 80
          initialDelaySeconds: 5
          timeoutSeconds: 5
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
    type: loadBalancer
```

Table 4: php-apache webserver deployment manifest

Step - 2: Inspect the environment:

```
[root@sni-k8s-test-01 ~]# kubectl get pods
NAME                          READY   STATUS      RESTARTS   AGE
loadtest-ndtqg                0/1     Completed   0          22d
nginx-64b98ccd48-l7p88        1/1     Running     1          23d
php-apache-54c6566f94-nctgn   1/1     Running     0          29s
vote-app-65cb5f75c6-9rx9w     1/1     Running     0          22d
vote-app-65cb5f75c6-nwv6n     1/1     Running     1          22d
vote-app-65cb5f75c6-pvfj8     1/1     Running     1          22d
vote-app-65cb5f75c6-wr4v9     1/1     Running     0          22d
```

Figure 21: Deployed test container web application php-apache

Step - 3: Accessing the web server:

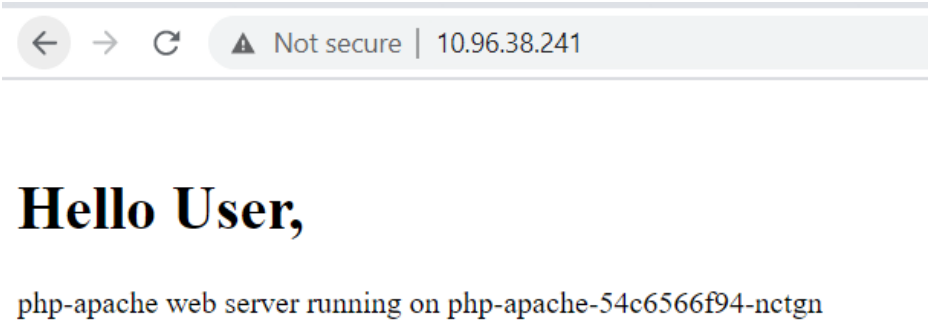


Figure 22: Accessing container-based web application

Step - 4: Deleting the web server pod & container:

```
[root@sni-k8s-test-01 ~]# kubectl delete pods php-apache-54c6566f94-nctgn
pod "php-apache-54c6566f94-nctgn" deleted
[root@sni-k8s-test-01 ~]#
```

Figure 23: Delete the pod and associated container

Step - 5: Inspecting the environment:

```
[root@sni-k8s-test-01 ~]# kubectl get pods -w
NAME                          READY   STATUS      RESTARTS   AGE
loadtest-ndtqg                0/1     Completed   0          23d
nginx-64b98ccd48-l7p88        1/1     Running     1          23d
php-apache-54c6566f94-nctgn   1/1     Running     0          12h
vote-app-65cb5f75c6-9rx9w     1/1     Running     0          23d
vote-app-65cb5f75c6-nwv6n     1/1     Running     1          23d
vote-app-65cb5f75c6-pvfj8     1/1     Running     1          23d
vote-app-65cb5f75c6-wr4v9     1/1     Running     0          23d


php-apache-54c6566f94-nctgn   1/1     Terminating   0          12h
php-apache-54c6566f94-cqdpq   0/1     Pending       0          0s
php-apache-54c6566f94-cqdpq   0/1     Pending       0          0s
php-apache-54c6566f94-cqdpq   0/1     ContainerCreating   0          0s
php-apache-54c6566f94-cqdpq   0/1     ContainerCreating   0          3s
php-apache-54c6566f94-cqdpq   1/1     Running             0          5s
php-apache-54c6566f94-nctgn   0/1     Terminating         0          12h
```

Figure 24: New container creation as old one terminated

Step - 6: Access the web server again:



Figure 25: Accessing container-based web application

Step - 7: Observability of the pod (CPU & memory usage):



Figure 26: Pods CPU & memory utilization

4.2 Auto-heal (Service unavailability):

In this step, from the container OS, the web service process has been stopped by executing "/usr/sbin/apachectl stop" command. Instantly, once the command is executed, the liveness probe come into action. Liveness probe is a mechanism provided by Kubernetes, helps to determine if applications are running within containers are operational. If liveness probe found containers are not operational, kubelet agent, a component of Kubernetes restarts the corresponding pod which ensures auto-heal of service.

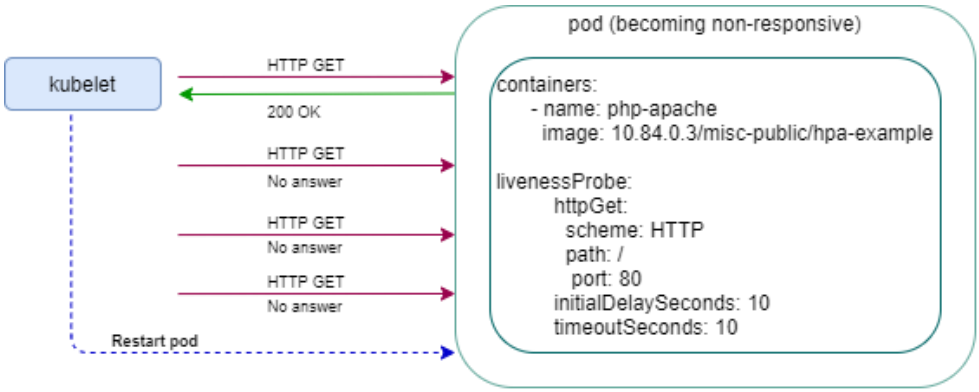


Figure 27: Pods availability with liveness probe

Step - 1: Stop apache web server process

```
[root@sni-k8s-test-01 ~]# kubectl exec -it php-apache-966f8647f-7tlsk /bin/bash
kubectl exec [POD] [COMMAND] is DEPRECATED and will be removed in a future version. Use kubectl kubectl exec [POD] -- [COMMAND] instead.
root@php-apache-966f8647f-7tlsk:/var/www/html#
root@php-apache-966f8647f-7tlsk:/var/www/html# /usr/sbin/apachectl stop
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 192.168.174.40. Set the 'ServerName' directive globally to suppress this message
Terminated
root@php-apache-966f8647f-7tlsk:/var/www/html# command terminated with exit code 137
```

Figure 28: Stopping apache process

Step - 2: Inspect the environment:

```
[root@sni-k8s-test-01 ~]# kubectl get pods -w
NAME                          READY   STATUS      RESTARTS   AGE
loadtest-ndtqg                0/1     Completed   0          25d
nginx-64b98ccd48-zc2dm        1/1     Running     0          2d16h
php-apache-966f8647f-7tlsk    1/1     Running     0          11h
vote-app-65cb5f75c6-9rx9w     1/1     Running     0          25d
vote-app-65cb5f75c6-frlt4     1/1     Running     0          2d17h
vote-app-65cb5f75c6-wr4v9     1/1     Running     0          25d
vote-app-65cb5f75c6-xxtb8     1/1     Running     0          46h


php-apache-966f8647f-7tlsk    0/1     Completed   0          11h
php-apache-966f8647f-7tlsk    1/1     Running     1          11h
```

Figure 29: Container get restarted once detected by Liveness probe

Step - 3: Observability of the pod (CPU & memory usage):

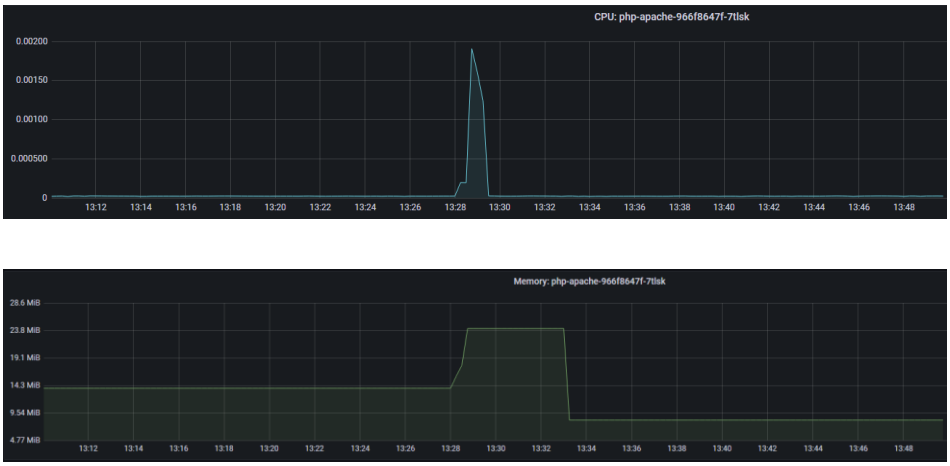


Figure 30: Pods CPU & memory utilization

4.3 Resource Aware auto scale-up & scale-down

It is one of the cloud features where workloads are auto scape-up and scale-down based on the needs of the system or application. With the help of Kubernetes, this cloud feature can be availed anywhere. In this test case, from a load generator, we perform a load test on the web server deployed earlier. A HPA (Horizontal Pod Autoscaler) autoscaler has been configured. Horizontal Pod Autoscaler (HPA) changes the shape of container workload automatically increasing or decreasing the number of pods in response to the workload's CPU or memory consumption, or in response to custom metrics reported from within cluster or external sources outside of the cluster. Based on the load on a particular container or service, Kubernetes automatically scale-up the containers horizontally and scale-down whenever resource requirement comes in a nominal range. The web server was created with certain limits of compute resources (e.g., CPU in this case), through the load generator, we generate extra load on the application and inspects the behavior.

resources:
  limits:
    cpu: 500m
  requests:
    cpu: 200m

Step - 1: HPA Configuration with 50% targets, a minimum pod of 1 with a maximum of 10 if it reaches the target.

```
[root@sni-k8s-test-01 ~]# kubectl get hpa
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache    0%/50%    1         10        1          25d
```

Figure 31: Configured Horizontal Pod Autoscaler (HPA)

Step - 2: Generate load on web server from load generator container

```
[root@sni-k8s-test-01 ~]# kubectl run -i --tty load-generator --rm --image=10.84.0.3/misc-public/busybox --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"
If you don't see a command prompt, try pressing enter.
<br/><h1>Hello User, </h1> <p> php-apache web server running on php-apache-966f8647f-7tlsk</p>
<br/><h1>Hello User, </h1> <p> php-apache web server running on php-apache-966f8647f-7tlsk</p>
<br/><h1>Hello User, </h1> <p> php-apache web server running on php-apache-966f8647f-7tlsk</p>
<br/><h1>Hello User, </h1> <p> php-apache web server running on php-apache-966f8647f-7tlsk</p>
<br/><h1>Hello User, </h1> <p> php-apache web server running on php-apache-966f8647f-7tlsk</p>
<br/><h1>Hello User, </h1> <p> php-apache web server running on php-apache-966f8647f-7tlsk</p>
<br/><h1>Hello User, </h1> <p> php-apache web server running on php-apache-966f8647f-7tlsk</p>
```

Figure 32: Traffic load generation to web server

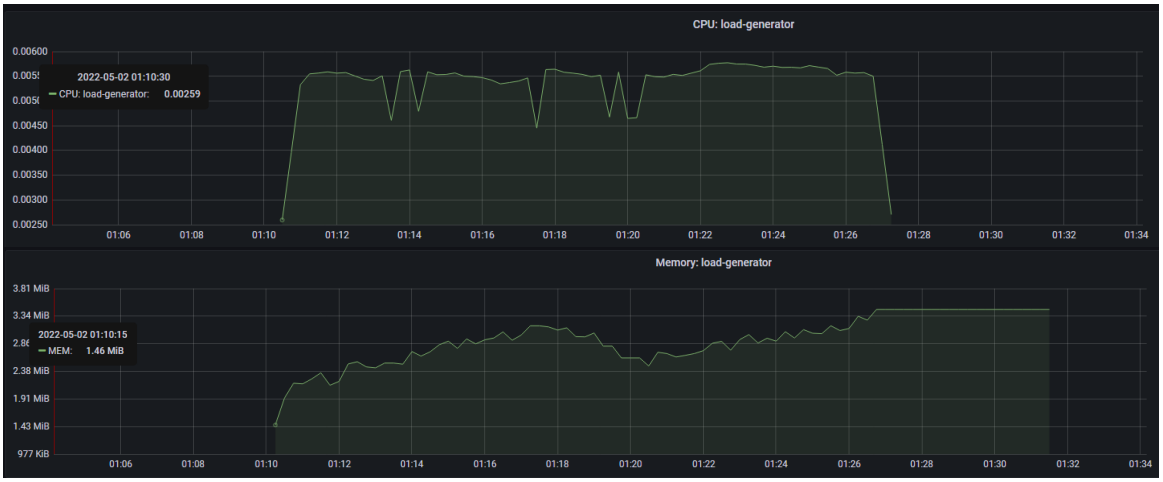


Figure 33: Load generator utilization (CPU & Memory)

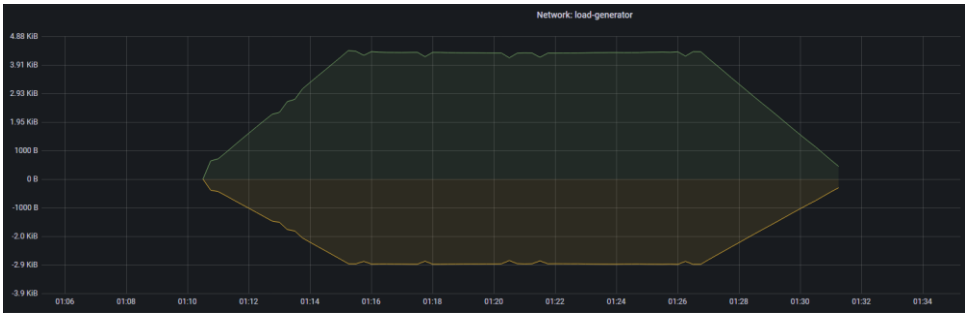


Figure 34: Load generator utilization (Network)

Step - 3: Inspect HPA, pods and container CPU usage

```
[root@sni-k8s-test-01 ~]# kubectl get hpa -w
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache    0%/50%     1         10        1          25d
vote         Deployment/vote-app      0%/40%     4         15        4          26d


php-apache   Deployment/php-apache    2%/50%     1         10        1          25d
php-apache   Deployment/php-apache    74%/50%    1         10        1          25d
php-apache   Deployment/php-apache    74%/50%    1         10        2          25d
php-apache   Deployment/php-apache    142%/50%   1         10        2          25d
php-apache   Deployment/php-apache    142%/50%   1         10        4          25d
php-apache   Deployment/php-apache    142%/50%   1         10        6          25d
php-apache   Deployment/php-apache    56%/50%    1         10        6          25d


php-apache   Deployment/php-apache    40%/50%    1         10        6          25d
```

Figure 35: Inspect HPA with targets and replicas

```
[root@sni-k8s-test-01 ~]# kubectl get pods -w
NAME                          READY   STATUS      RESTARTS   AGE
loadtest-ndtqg                0/1     Completed   0          26d
nginx-64b98ccd48-zc2dm        1/1     Running     0          3d4h
php-apache-966f8647f-7tlsk    1/1     Running     5          23h
vote-app-65cb5f75c6-9rx9w     1/1     Running     0          26d
vote-app-65cb5f75c6-frlt4     1/1     Running     0          3d5h
vote-app-65cb5f75c6-wr4v9     1/1     Running     0          26d
vote-app-65cb5f75c6-xxtb8     1/1     Running     0          2d10h


load-generator                0/1     Pending     0          0s
load-generator                0/1     Pending     0          0s
load-generator                0/1     ContainerCreating   0          0s
load-generator                0/1     ContainerCreating   0          0s
load-generator                1/1     Running             0          1s
php-apache-966f8647f-mfjgx    0/1     Pending             0          0s
php-apache-966f8647f-mfjgx    0/1     Pending             0          0s
php-apache-966f8647f-mfjgx    0/1     ContainerCreating   0          0s
php-apache-966f8647f-mfjgx    0/1     ContainerCreating   0          1s
php-apache-966f8647f-mfjgx    1/1     Running             0          1s
php-apache-966f8647f-knnl2    0/1     Pending             0          0s
php-apache-966f8647f-qxg4t    0/1     Pending             0          0s
php-apache-966f8647f-knnl2    0/1     Pending             0          0s
php-apache-966f8647f-qxg4t    0/1     Pending             0          0s
php-apache-966f8647f-knnl2    0/1     ContainerCreating   0          0s
php-apache-966f8647f-qxg4t    0/1     ContainerCreating   0          0s
php-apache-966f8647f-knnl2    0/1     ContainerCreating   0          1s
php-apache-966f8647f-qxg4t    0/1     ContainerCreating   0          1s
php-apache-966f8647f-knnl2    1/1     Running             0          1s
php-apache-966f8647f-qxg4t    1/1     Running             0          2s
php-apache-966f8647f-tzm46    0/1     Pending             0          0s
php-apache-966f8647f-8nw4r    0/1     Pending             0          0s
php-apache-966f8647f-tzm46    0/1     Pending             0          0s
php-apache-966f8647f-8nw4r    0/1     Pending             0          0s
php-apache-966f8647f-tzm46    0/1     ContainerCreating   0          0s
php-apache-966f8647f-8nw4r    0/1     ContainerCreating   0          0s
php-apache-966f8647f-tzm46    0/1     ContainerCreating   0          1s
php-apache-966f8647f-8nw4r    0/1     ContainerCreating   0          1s
php-apache-966f8647f-8nw4r    1/1     Running             0          1s
php-apache-966f8647f-tzm46    1/1     Running             0          1s
```

Figure 36: New container creation based on HPA targets

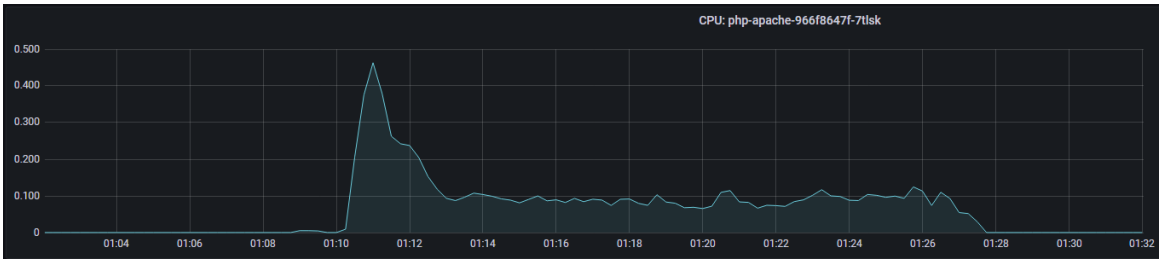


Figure 37: Container CPU usage trend on load test

Step - 4: Inspect HPA and pods

```
[root@sni-k8s-test-01 ~]# kubectl get hpa -w
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache    48%/50%   1         10        5          25d
vote         Deployment/vote-app      0%/40%    4         15        4          26d


php-apache   Deployment/php-apache    18%/50%   1         10        5          25d
php-apache   Deployment/php-apache    0%/50%    1         10        5          25d
php-apache   Deployment/php-apache    0%/50%    1         10        5          25d
php-apache   Deployment/php-apache    0%/50%    1         10        2          25d
php-apache   Deployment/php-apache    0%/50%    1         10        2          25d
php-apache   Deployment/php-apache    0%/50%    1         10        1          25d
```

```
[root@sni-k8s-test-01 ~]# kubectl get pods -w
NAME                      READY   STATUS        RESTARTS   AGE
loadtest-ndtqg            0/1     Completed     0          26d
nginx-64b98ccd48-zc2dm    1/1     Running       0          3d5h
php-apache-966f8647f-7tlsk   1/1     Running       5          24h
php-apache-966f8647f-8nw4r   1/1     Running       0          14m
php-apache-966f8647f-knnl2   1/1     Running       0          15m
php-apache-966f8647f-mfjgx   1/1     Running       0          16m
php-apache-966f8647f-qxg4t   1/1     Running       0          15m
vote-app-65cb5f75c6-9rx9w    1/1     Running       0          26d
vote-app-65cb5f75c6-frlt4    1/1     Running       0          3d5h
vote-app-65cb5f75c6-wr4v9    1/1     Running       0          26d
vote-app-65cb5f75c6-xxtb8    1/1     Running       0          2d10h


php-apache-966f8647f-7tlsk   1/1     Terminating   5          24h
php-apache-966f8647f-8nw4r   1/1     Terminating   0          20m
php-apache-966f8647f-qxg4t   1/1     Terminating   0          20m
php-apache-966f8647f-8nw4r   0/1     Terminating   0          20m
php-apache-966f8647f-7tlsk   0/1     Terminating   5          24h
php-apache-966f8647f-qxg4t   0/1     Terminating   0          20m
php-apache-966f8647f-7tlsk   0/1     Terminating   5          24h
php-apache-966f8647f-7tlsk   0/1     Terminating   5          24h
php-apache-966f8647f-qxg4t   0/1     Terminating   0          20m
php-apache-966f8647f-qxg4t   0/1     Terminating   0          20m
php-apache-966f8647f-8nw4r   0/1     Terminating   0          20m
php-apache-966f8647f-8nw4r   0/1     Terminating   0          20m
php-apache-966f8647f-knnl2   1/1     Terminating   0          21m
php-apache-966f8647f-knnl2   0/1     Terminating   0          21m
php-apache-966f8647f-knnl2   0/1     Terminating   0          21m
php-apache-966f8647f-knnl2   0/1     Terminating   0          21m
```

Figure 38: HPA target and replica decreases once the load generation stopped

Step - 5: Inspect workload CPU usage and trend

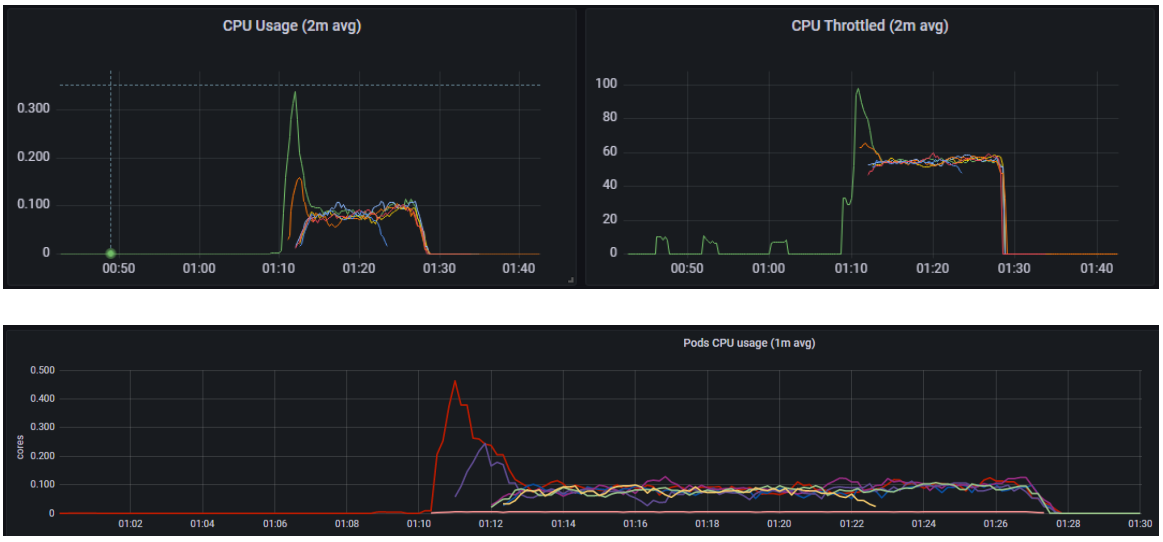


Figure 39: CPU usage pattern on load generation

Similar tests were performed on a Linux OS based simple monolithic web server where we observed that due to load generation many of the threshold crossed the limit, but resource utilization cannot be optimized, no scaling was happened automatically based on the system performance and utilization. Thus, overall performance of the application degrades due to the nature of the application architecture.



Figure 40: System CPU and load average status before load test

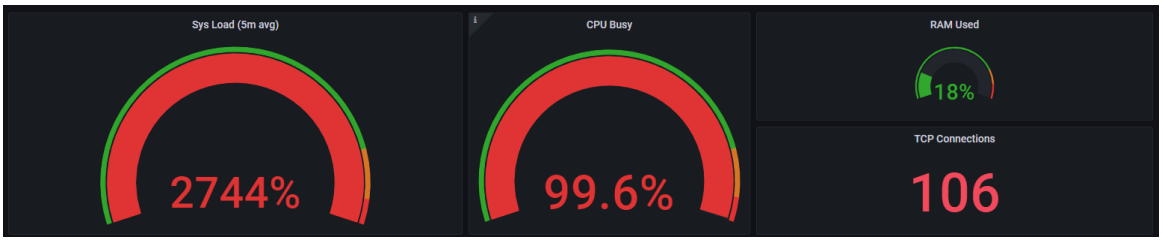




Figure 41: System CPU and load average usage during load test

4.4 Performance Analysis between Monolithic & Microservice based application

4.4.1 Performance measurement of microservice application

In this test case, a similar web server-based application is deployed in microservice based container and traditional monolithic stack application. By generating additional requests and loads on the webserver, performance is examined and reported.

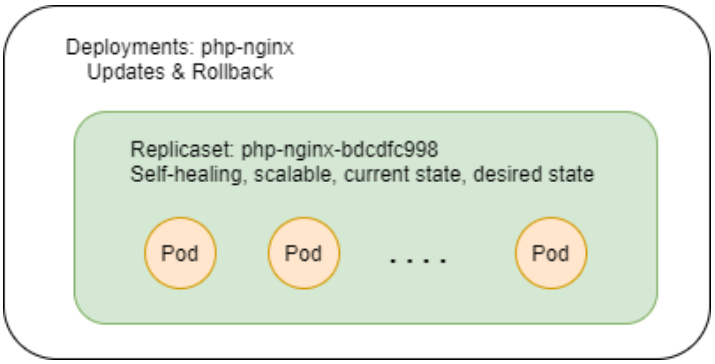


Figure 42: Deployment, Replicaset & Pod

Step - 1: Deploy the container and service

Below is the deployment, associated replicaset and pods. A php-nginx deployment was done with two containers running a webserver which pulls RSS news feed (Chase N., 2016) from cnn.com and this service was exposed with a load balancer IP. The docker image is published in dockerhub (didarsm/pub:rss-php-nginx) as a public image.

Name	Ready	Up-To-Date	Available	Age
php-nginx	2/2	2	2	113m

Table 5: Deployment status

Name	Desired	Current	Ready	Age
php-nginx-bdcdfc998	2	2	2	113m

Table 6: Replicaset status

Name	Ready	Status	Restarts	Age
php-nginx-bdcdfc998-7vlmz	1/1	Running	0	113m
php-nginx-bdcdfc998-qltqw	1/1	Running	0	113m

Table 7: Container and Pod status

Name	Type	Cluster-IP	External-IP	Port(s)	Age
php-nginx-1	LoadBalancer	10.102.231.194	10.96.38.230	80:31307/TCP	113m

Table 8: Load balancer service status

Step - 2: Browsing the service by accessing the load balancer IP

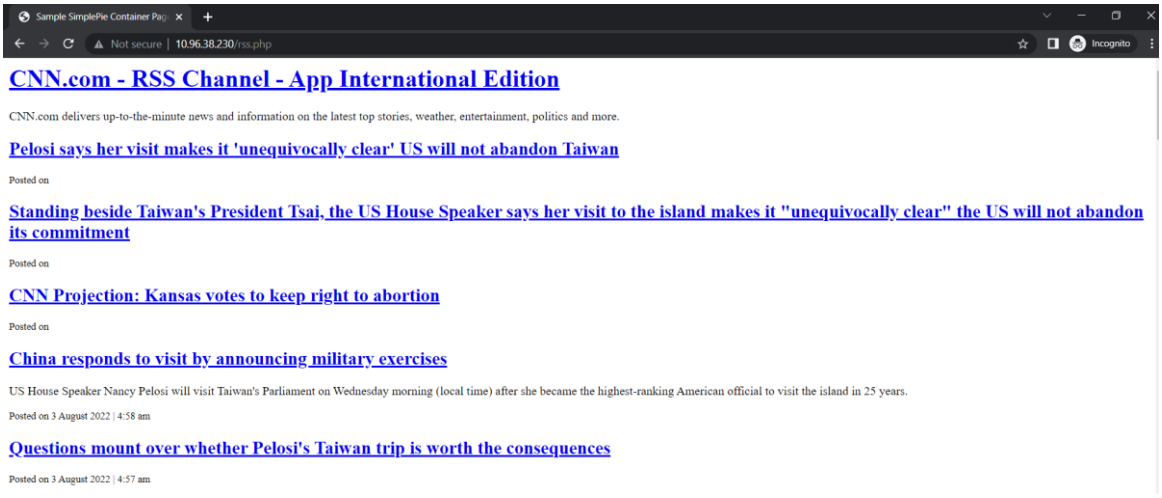


Figure 43: php-nginx web server service

Step - 3: Status of containers and associated web service

Pod Name	Container	Pod IP	CPU Request	CPU Limit
php-nginx-bdcdfc998-7vlmz	php-nginx	192.168.174.9	0.200	0.500
php-nginx-bdcdfc998-qltqw	php-nginx	192.168.59.214	0.200	0.500

Table 9: php-nginx container details





Figure 44: php-nginx container status

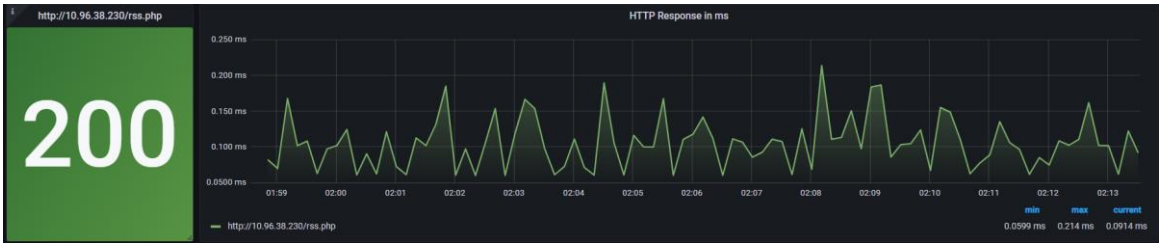


Figure 45: php-nginx web service response time & code

Step - 4: Generate load on php-nginx containers, observe status and web service

```
[root@sni-k8s-test-01 ~]# kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
load-generator                1/1     Running   0          8m12s
php-nginx-bdcdfc998-7vlmz     1/1     Running   0          39h
php-nginx-bdcdfc998-knl2x     1/1     Running   0          2m16s
php-nginx-bdcdfc998-knztx     1/1     Running   0          2m16s
php-nginx-bdcdfc998-qltqw     1/1     Running   0          39h
php-nginx-bdcdfc998-wfrb8     1/1     Running   0          2m1s
```

Figure 46: Load generation and Horizontal Pod Scaling (HPA) status



Figure 47: CPU, memory & network status of load generator

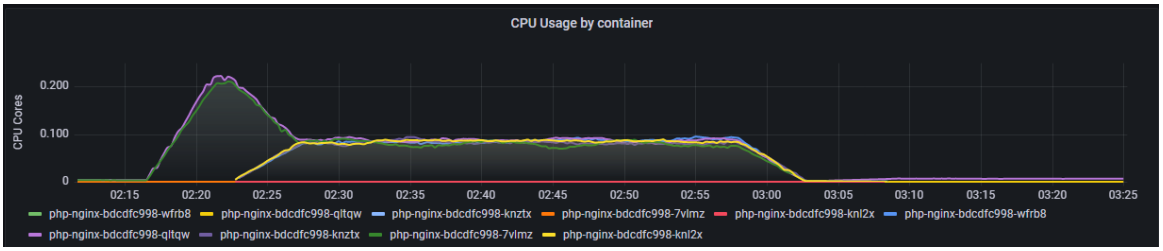


Figure 48: CPU usage of php-nginx pods

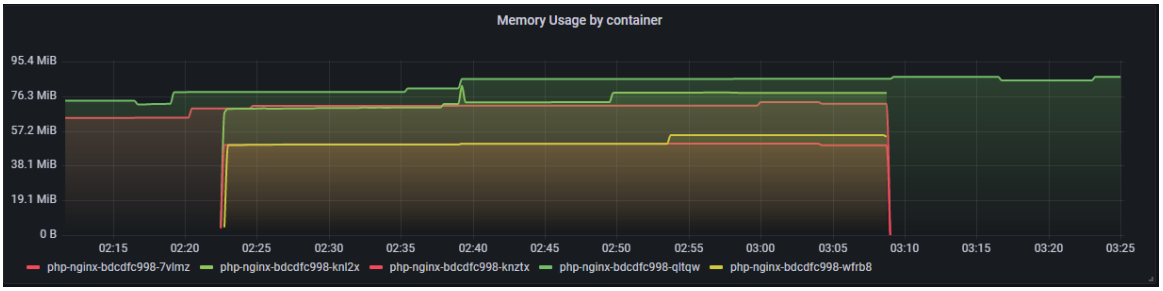


Figure 49: Memory usage of php-nginx pods

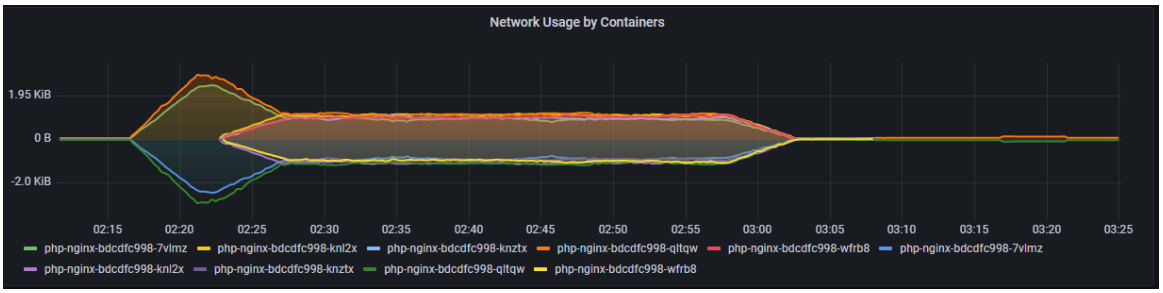


Figure 50: Network usage of php-nginx pods

http://10.96.38.230/rss.php

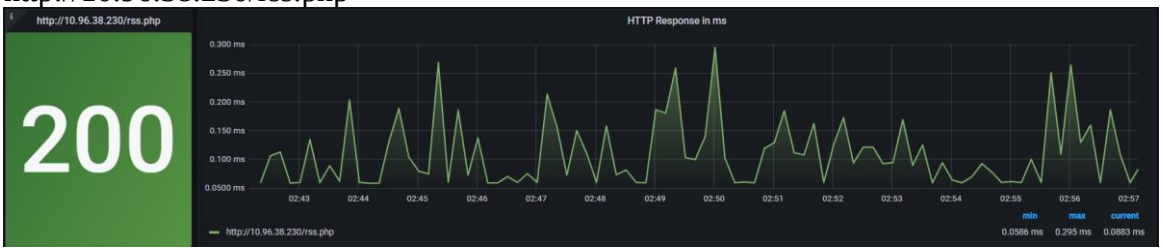


Figure 51: php-nginx web service response time & code

Summary:
Below table summarizes the results of this examination. Web server status, performance and response time was measure in millisecond before and during the load test which presented as follows:

Legend	Without Load	With Load	1 Hour Average
Min	0.0599ms	0.0586ms	0.0584ms
Max	0.214ms	0.295ms	0.310ms
Current	0.0914ms	0.0883ms	0.124ms
Number of pods	2	5	2~5

Table 10: Web service response time before and during load

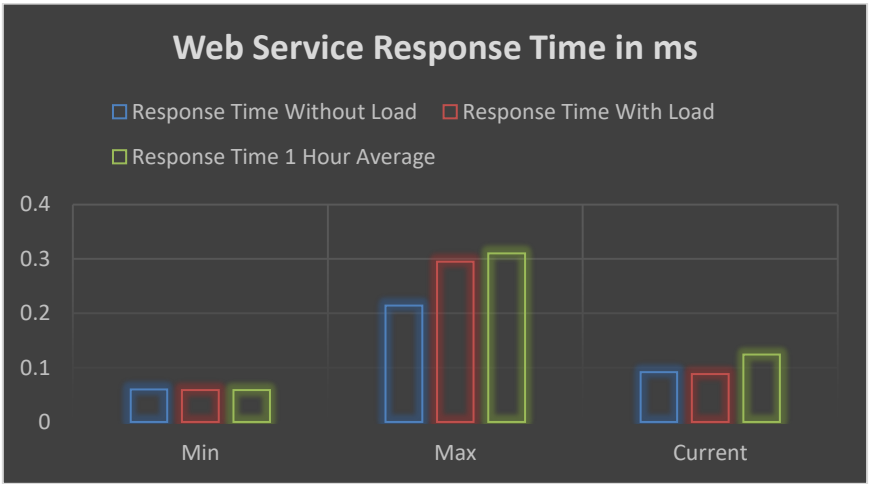


Figure 52: Microservice web service response time comparison

4.4.2 Performance measurement of monolithic application

At this stage, same web service application has been deployed on a virtual machine consisting of equivalent compute resources. By architecture, monolithic application doesn't comprise of auto-scaling and self-resiliency feature.

Step 1: Install nginx web server and deploy web application

http://10.96.65.240:8080/tss.php

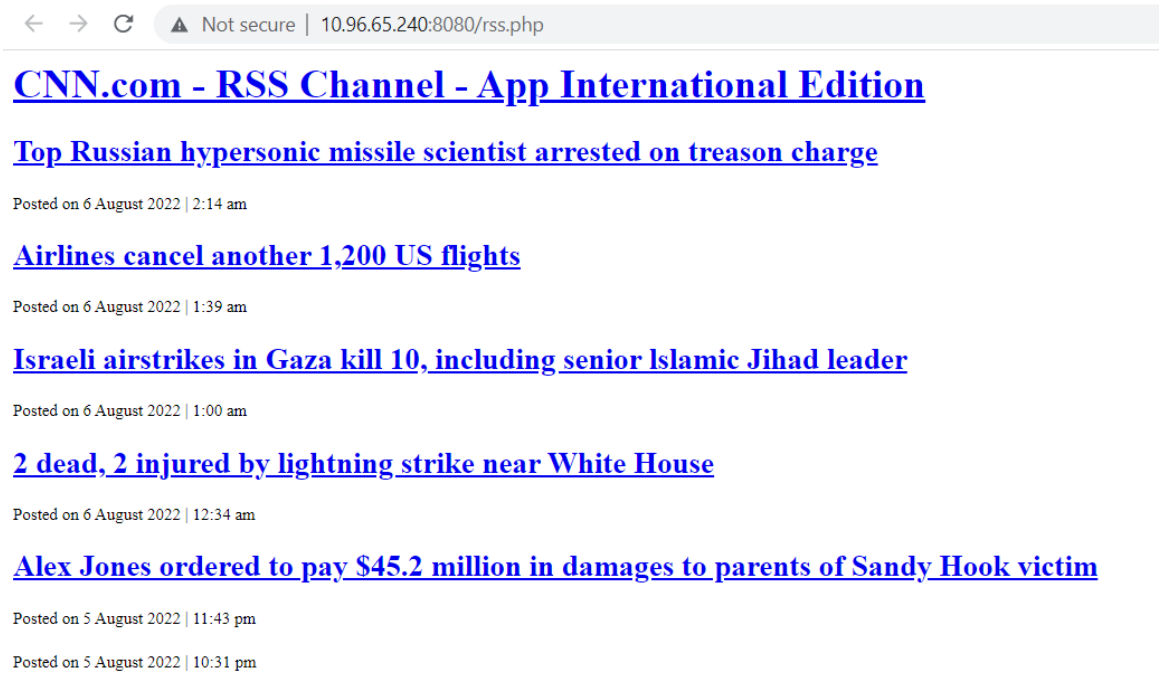


Figure 53: CNN RSS channel web service

Step 2: Measurement the status and performance of the web server and service

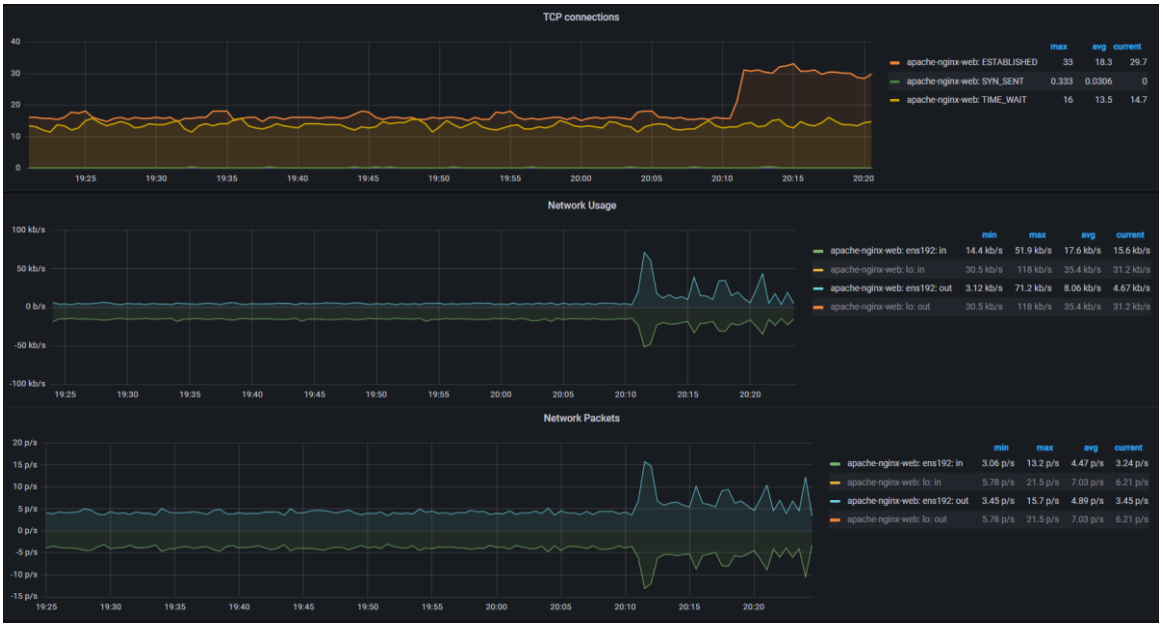


Figure 54: TCP connection and network status in normal state

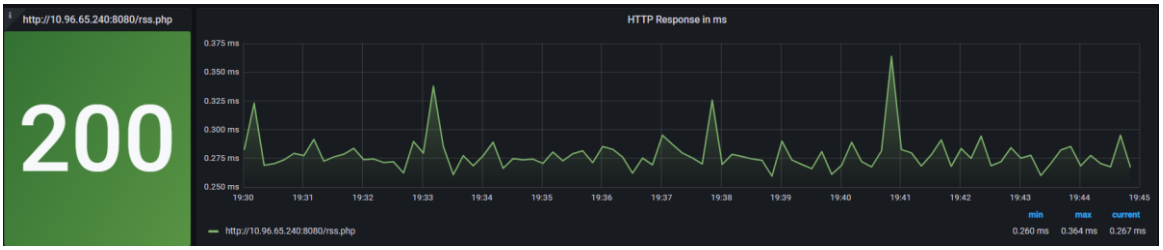


Figure 55: Web service response time & status without load

Step 3: Generate load and measure status, performance of the system & application

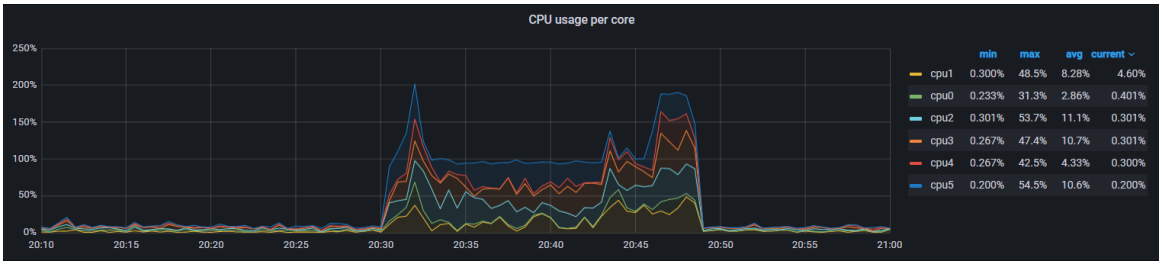




Figure 56: TCP connection and network status during load



Figure 57: Web service response time & status with load

Summary:
Below table consists of the value of the response time of the web service running on a virtual machine which is deployed as a monolithic application on top of a nginx server.

Legend	Without Load	With Load	1 Hour Average
Min	0.260ms	0.145ms	0.140ms
Max	0.364ms	0.316ms	1.33ms
Current	0.267ms	0.282ms	0.286ms
Web Server Instance	1	1	1

Table 11: Web service response time summary with and without load

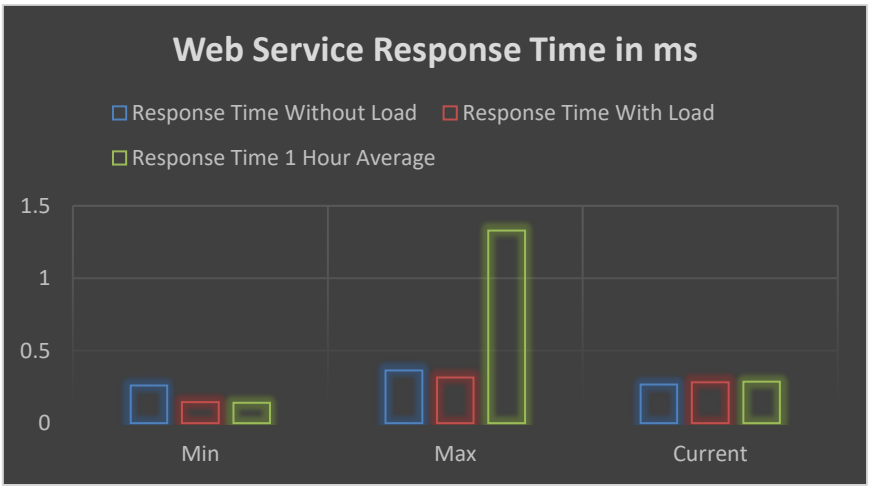


Figure 58: Monolith web service response time comparison

4.4.3 Application Response Time Comparison between Microservice vs Monolith

In this section, results obtained from earlier two sections were analyzed and compared. System resource utilizations were represented in earlier section which shows the container technology is far away from traditional stack. Response time of the web service application were carefully measured and evaluated. Three cases the obtained results were compared, and it has been observed and found that in all three cases microservice based application is significantly ahead of old-style application.

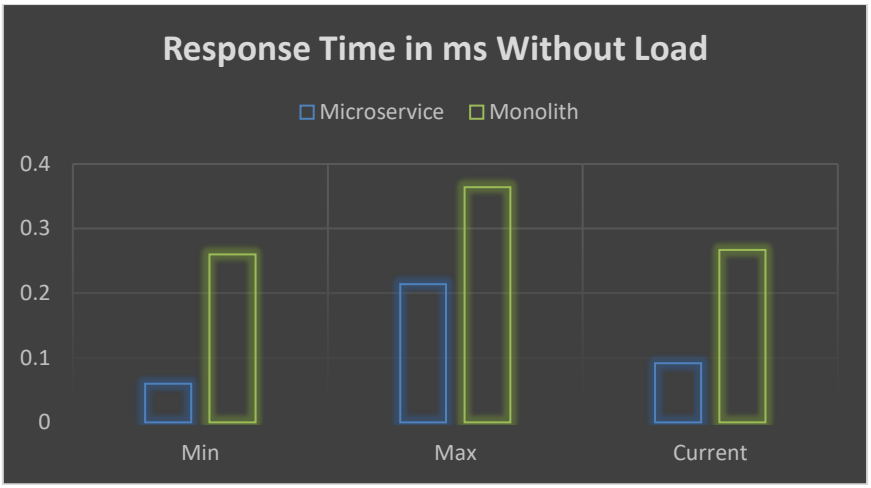


Figure 59: Comparison between application without load

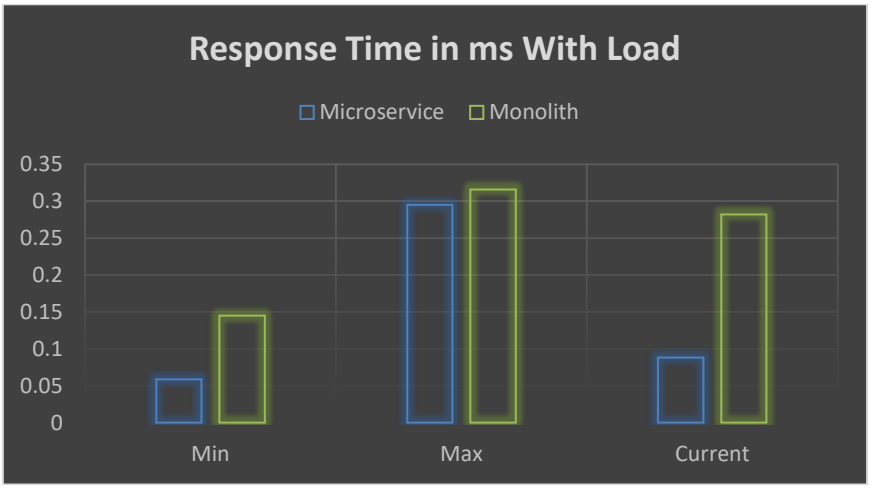


Figure 60: Comparison between application with load

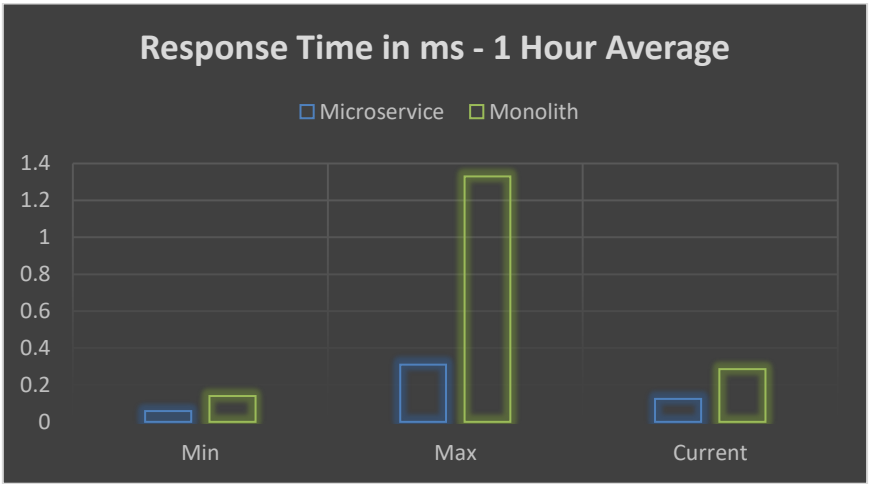


Figure 61: Comparison between application in hourly average

Chapter 5

Conclusion

Through a comprehensive approach we accomplished the creation of a cloud-native environment with commodity servers and all open-source products which provides location independent IaaS (Infrastructure as a Service) and elasticity in a distributed manner. Among many features, resiliency and self-healing feature was surveyed with numerous approaches; where a traditional monolithic application built on a traditional stack failed to provide those features. However, microservices approaches helps to achieve those features with simple code and DevOps culture. At the same time, application performance was observed which is found with a significant improvement. As a future work, a complex two or three tier applications can be built with multiple microservice components, similar test cases shall be performed and examined.

References

Alshuqayran, N., Ali, N., & Evans, R. (2016, November). A Systematic Mapping Study in Microservice Architecture. *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA).* IEEE. doi:10.1109/soca.2016.15

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., & Steinder, M. (2015, September). Performance Evaluation of Microservices Architectures Using Containers. *2015 IEEE 14th International Symposium on Network Computing and Applications.* IEEE. doi:10.1109/nca.2015.49

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016, May). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software, 33*, 42–52. doi:10.1109/ms.2016.64

Bruce, M., & Pereira, P. A. (2018). *Microservices in Action.* Manning Publications.

Carnell, J. (2017). *Spring Microservices in Action.* Manning Publications.

Chen, L. (2018, April). Microservices: Architecting for Continuous Delivery and DevOps. *2018 IEEE International Conference on Software Architecture (ICSA).* IEEE. doi:10.1109/icsa.2018.00013

Chen, R., Li, S., & Li, Z. (2017, December). From Monolith to Microservices: A Dataflow-Driven Approach. *2017 24th Asia-Pacific Software Engineering Conference (APSEC).* IEEE. doi:10.1109/apsec.2017.53

Gos, K., & Zabierowski, W. (2020, April). The Comparison of Microservice and Monolithic Architecture. *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH).* IEEE. doi:10.1109/memstech49584.2020.9109514

Hasselbring, W., & Steinacker, G. (2017, April). Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* IEEE. doi:10.1109/icsaw.2017.11

Ibryam, B., & Huβ, R. (2019). *Kubernetes Patterns.* O'Reilly Media, Incorporated.

Kalske, M., Mäkitalo, N., & Mikkonen, T. (2018). Challenges When Moving from Monolith to Microservice Architecture. In *Current Trends in Web Engineering* (pp. 32–47). Springer International Publishing. doi:10.1007/978-3-319-74433-9_3

Lauretis, L. D. (2019, October). From Monolithic Architecture to Microservices Architecture. *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).* IEEE. doi:10.1109/issrew.2019.00050

Luksa, M. (2017). *Kubernetes in Action.* Manning Publications.

Mauro, T. (2015, February). *Adopting Microservices at Netflix: Lessons for Architectural Design*. Retrieved from https://www.nginx.com/: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media.

Opara-Martins, J., Sahandi, R., & Tian, F. (2016, April). Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing, 5*. doi:10.1186/s13677-016-0054-z

Rosso, J., Lander, R., Brand, A., & Harris, J. (2021). *Production Kubernetes.* O'Reilly Media, Incorporated.

Singh, V., & Peddoju, S. K. (2017, May). Container-based microservice architecture for cloud applications. *2017 International Conference on Computing, Communication and Automation (ICCCA).* IEEE. doi:10.1109/ccaa.2017.8229914

Taibi, D., Lenarduzzi, V., & Pahl, C. (2017, September). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing, 4*, 22–32. doi:10.1109/mcc.2017.4250931

Tapia, F., Mora, M. Á., Fuertes, W., Aules, H., Flores, E., & Toulkeridis, T. (2020, August). From Monolithic Systems to Microservices: A Comparative Study of Performance. *Applied Sciences, 10*, 5797. doi:10.3390/app10175797

Villamizar, M., Garces, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015, September). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *2015 10th Computing Colombian Conference (10CCC).* IEEE. doi:10.1109/columbiancc.2015.7333476

Chase, N. (2016, December). *Docker Run: How to create a Docker image for an application* : https://www.mirantis.com/blog/how-do-i-create-a-new-docker-image-for-my-application/