

# **Final Project Report**

## **Cloud Options for Containers as a Service**

**9<sup>th</sup> May, 2017**

### **Team 6**

Bijo Joseph : Lead

Abhishek Bandarupalle : Vice-Lead

Kshitija Murudi

Rakshit Ravishankar

Ruchika Verma

Computer Science Department  
NC State University

# **Cloud Options for Containers as a Service**

## **Abstract**

Visibility into performance characteristics of different cloud options for containers are required to select the platform suitable for client application deployment. Evaluation of Container as a Service option in the private and public clouds namely CoreOS and Amazon Web Service (AWS) is performed based on compute and memory requirements for the client applications. For evaluating compute intensive application parameters such as time to complete compute intensive applications, maximum number of simultaneous compute intensive applications supported and effect of these applications on server CPU were taken into consideration. CPU intensive applications were also evaluated by assigning dedicated cores in both the AWS and CoreOS platforms. To evaluate memory intensive applications, containers were deployed until there was contention for RAM resource. Effect of network utilization in deploying containers was evaluated as an additional performance metric on both AWS and CoreOS.

Customer's applications were analyzed and classified as compute intensive and memory intensive. Based on this classification and the analysis from the test results, we propose a better suited cloud option for the customer.

# Table of Contents:

Introduction.....	6
1.1 Problem statement.....	6
1.2 Motivation.....	6
1.3 Issues.....	7
1.4 Environment.....	7
Requirements .....	9
2.1 Functional Requirements .....	9
2.1.2 Memory Utilization.....	10
2.1.4 Initialization Time.....	10
2.1.6 Comparison of cloud options: Private Cloud and AWS .....	11
2.1.7 Customer SLA .....	11
2.2 Non-Functional Requirement.....	11
System Environment.....	12
3.1 Private cloud Setup .....	12
3.2 Setup on AWS.....	12
Design, Implementation and Results .....	14
4.1 Design Options.....	14
4.1.1 Comparison of CentOS and CoreOS.....	14
4.2 Private Cloud Setup .....	15
4.3 Public Cloud Setup.....	17
4.4 Design - Management Node Dashboard .....	17
4.5 Implementation .....	17
4.5.1 Implementation of Private Cloud setup.....	17
4.5.1.1 Installing CoreOS.....	18
4.5.1.3 Implementation of Management Node Dashboard .....	21
4.6 Workflow .....	21
4.7 Results.....	23
Verification and Validation.....	24
5.0 Mapping of requirements and verifications .....	24
5.1 Test cases and analysis of results of R1 .....	27
5.2 Test cases and analysis of results of R2.....	28
5.3 Test cases and analysis of results of R3 .....	29
5.4 Test cases and analysis of results of R4.....	30

5.5	Test cases and analysis of results of R5 .....	31
5.6	Test cases and analysis of results of R6 .....	33
5.7	Test cases and analysis of results of R7 .....	33
5.8	Test case and analysis of results of R8.....	34
5.10	Test case and analysis of results of R10.....	35
5.11	Test case and analysis of results of R12.....	36
5.13	Non-Functional Requirement.....	37
	Schedule and Personnel .....	38
	References.....	40
	Appendices.....	42

## List of Figures:

Figure 4.1:	Network topology for Private Lab set-up.....	16
Figure 4.2:	SSH from management node into CoreOS server .....	18
Figure 4.3:	Network setup for private cloud setup.....	20
Figure 4.4:	Workflow for user to use and deploy containers on private cloud setup.....	22
Figure 5.1.1:	Docker stats for CPU utilization of a container in a constrained environment .....	27
Figure 5.1.2:	Total CPU utilization vs Number of Containers.....	28
Figure 5.2.1:	CPU utilization per Core in a constrained environment on AWS.....	29
Figure 5.2.2:	Total CPU utilization vs Time as the number of containers were increased.....	29
Figure 5.3.1:	Docker stats for CPU utilization of a container in unconstrained environment.....	30
Figure 5.3.2:	Time taken to encode video versus number of containers .....	30
Figure 5.4.1:	Docker stats for CPU utilization of a container in unconstrained environment.....	31
Figure 5.4.2:	Total CPU utilization vs Time as the number of containers were increased in an Unconstrained environment on AWS.....	31
Figure 5.5.1:	Memory utilization of a memory-intensive application in a container on CoreOS.....	32
Figure 5.5.2:	Memory Utilization as requests for memory intensive containers increases.....	32
Figure 5.6.1:	Memory utilization of a memory-intensive application in a container on AWS.....	33
Figure 5.9:	Initialization time taken for containers on CoreOS platform.....	35

List of Tables:

Table 3.1: Physical Hardware technical specification.....12

Table 3.2: AWS technical specification.....13

Table 4.1: Comparison between CoreOS and CentOS .....15

Table 5.1: Mapping test-cases to requirements .....24,25,26

Table 5.2: Comparison between AWS & Private cloud set-up.....36,37

Table 6.1: Schedule.....38,39

# Chapter 1

## Introduction

The traditional approach of running each application on dedicated hardware resulted in underutilization of server's compute and memory resources. A 2012 New York Times article cited two sources which estimated the average server resource utilization rate to be 6 to 12 percent [1].

Server virtualization offered a way to run multiple applications on a single physical server. This was achieved by allocating a fixed share of the server's resources to a virtual machine. Virtualization increased resource utilization thereby reducing capital and operational expenditure. But each virtual machine (VM) incurred an overhead of emulating hardware and running an operating system. VMs also lack a standardized environment. An application developed in one VM might not run in another VM due to unavailability of necessary dependencies. The requirement to have a consistent environment for development, testing and deployment of applications ushered a move to deploy applications in containers.

Containers create a portable, consistent operating environment for development, testing and deployment. Containers run on the host operating system thus eliminating the overhead of guest operating systems. Containers are shareable and can be used on a variety of private and public cloud deployments. Evaluating the performance and cost associated with running an application in a container deployed on a public cloud and a private cloud is a necessary step in infrastructure design.

In our project, we analyze the performance and the cost effectiveness of cloud options for running applications on containers. Using this analysis, we recommend the best-fit platform to be used for a similar class of applications.

The problem statement, motivation, issues, and the environment in which our project is set up are described in this section.

### 1.1 Problem statement

It is required to evaluate the performance of applications deployed in containers on two cloud platforms – Private (CoreOS) and Public cloud (Amazon Web Services). This evaluation needs to be performed for compute and memory intensive applications. We also need to estimate the operational costs of running these client applications in containers on private and public cloud platforms. Based on the results of performance evaluation and cost analysis, we need to recommend the best suited cloud platform for these classes of application.

### 1.2 Motivation

Explore the feasibility of Containers as a Service option for application deployment. Containers increase developer efficiency by giving a standardized environment to develop and test applications. This has resulted in rapid increase in the number of applications developed and run on containers.

As students of Cloud Computing technology, this project motivated us to learn about the popular technologies that enable the deployment of containerized applications and the various platforms or cloud options to deploy containers.

## 1.3 Issues

### 1.3.1 AWS infrastructure visibility

**Description:** We are running applications on containers on bare-metal in private cloud setup. This private cloud setup has specific configurations defined. To do a performance analysis, it would be ideal to have the same specifications and level of virtualization in public cloud setup (AWS) also. But, given the limited visibility into the AWS infrastructure, it is difficult to perform the same base-level analysis.

**Action:** We reserved a dedicated host M4 which is a bare metal server image. The configurations of the image were as close to as we could get to our private cloud setup.

### 1.3.2 Docker Dashboard

**Description:** Management node for our private lab setup needed a Docker dashboard. We tried different readily available options, but could not use any one of them. The CoreOS based GUI dashboards that are available were either license-based or available only to premium users. We tried using mist.io for the dashboard as it is free. But as mist.io needed public IPs of the CoreOS machines and since CoreOS machines were behind a NAT box this option was also ruled out. Other GUI-based dashboards required us to reconfigure the cloud\_config.yaml file and do a complete re-installation. As we had almost completed our project, this option was also not feasible.

**Action:** We developed a CLI based dashboard using Python that allows us to get the statistics, spawn and delete containers.

### 1.3.3 Privilege Tree

**Description:** Each customer has the same privileges so they can choose any server they want to deploy their application. We could not limit the resource usage of a greedy customer thereby starving other customers.

**Action:** In future, we could set up appropriate privileges to restrict access to resources after a customer's threshold is exceeded.

## 1.4 Environment

We consider Amazon Web Services (AWS) as the public cloud, as it is readily available to the general public over the internet. We will be deploying applications in containers on AWS using Docker. The infrastructure is a general-purpose machine which has a balance of CPU, Memory, and I/O. Detailed specification of environment is given in Chapter 3.;

We consider the lab setup as a private cloud as only authorized clients can access the resources. It is not available for general public. The private cloud set-up is similar to the public cloud, where user accesses the server resources through a management node. Users deploy their applications in containers which is hosted on powerful servers with high compute, memory, and storage resources. Detailed specification of environment is given in Chapter 3

**Note:** We will be using the terms AWS and public cloud, lab setup and private cloud interchangeably throughout this document.



# Chapter 2

## Requirements

This section mentions the lists of tasks and the mandatory functional requirements as per the problem statement. The functional requirements cover the various metrics on which the container applications' performance will be evaluated, types of applications to be tested and set-up required for the private and public cloud platforms. The non-functional requirements are to provide the user meta data that mentions the current CPU and Memory utilization of each container.

### 2.1 Functional Requirements

#### 2.1.1 CPU Utilization

CPU utilization needs to be measured in two kinds of environment - constrained and unconstrained environment on both AWS and CoreOS platforms. In a constrained environment, we limit the number of cores dedicated to a container and in an unconstrained environment, the containers are free to utilize all the available cores. We run tests in constrained and unconstrained environment to decide the environment that gives higher CPU utilization of servers and better performance for the application.

**R1:** Evaluate the performance of CPU intensive applications running in containers in the Private Lab setup. Evaluation is performed in a constrained environment.  
Ensure that CPU utilization of the servers in the private lab setup increases with the number of containers spawned in a constrained environment.

**R2:** Evaluate the performance of CPU intensive applications running in containers on AWS. Evaluation is performed in a constrained environment.  
Ensure that CPU utilization of the servers on AWS increases with the number of containers spawned in a constrained environment.

**R3:** Evaluate the performance of CPU intensive applications running in containers in the Private Lab setup. Evaluation is performed in an unconstrained environment.  
Ensure that CPU utilization of the servers in the private lab setup increases with the number of containers spawned in an unconstrained environment.

**R4:** Evaluate the performance of CPU intensive applications running in containers on AWS. Evaluation is performed in an unconstrained environment.

Ensure that CPU utilization of the servers on AWS increases with the number of containers spawned in an unconstrained environment.

### **2.1.2 Memory Utilization**

RAM requirement for running customer's applications on AWS and CoreOS needs to be compared. Evaluation of platforms needs to be performed based on the number of containers that can be deployed until there is a contention for RAM resource.

**R5:** Evaluate the memory utilization of applications running in containers in the private lab setup. Ensure that the server's RAM is completely utilized. Also predict the maximum number of simultaneous memory intensive applications that can be run.

**R6:** Evaluate the memory utilization of applications running in containers on AWS. Ensure that the server's RAM is completely utilized. Also predict the maximum number of simultaneous memory intensive applications that can be run.

### **2.1.3 Network Overhead**

Docker containers use a 'docker0' bridge. All containers that are spawned automatically connect to the 'docker0' bridge. This bridge is used to communicate between the containers and to connect the container to the outside world. We need to evaluate the overhead on network created by Docker bridge.

**R7:** Ensure that the Docker bridge created for container networking does not cause additional network overhead in the private lab setup.

**R8:** Ensure that the Docker bridge created for container networking does not cause additional network overhead on AWS.

### **2.1.4 Initialization Time**

Image has the application and its dependencies. A container is a running instance of an image. We need to measure the time taken to spawn a container from an image on our private cloud setup and AWS.

**R9:** Measure the initialization time of applications running on containers. Measure and compare the time taken to spawn different number of containers in the private cloud setup.

**R10:** Measure the initialization time of applications running on containers. Measure and compare the time taken to spawn different number of containers in AWS.

### **2.1.5 Comparison of Operating system options – CentOS and CoreOS for private lab setup**

As we deploy the customer's application in a container, it was required to choose an operating system optimized for containers. We need to evaluate and analyze the operating system best suited for our project.

**R11:** Compare operating system options CoreOS and CentOS. Analyze the advantages of running CoreOS for Docker applications.

Since this is part of our private cloud design, detailed analysis is provided in the design section.

### **2.1.6 Comparison of cloud options: Private Cloud and AWS**

Containers as a Service can be provided on a private cloud or a public cloud. We evaluate the benefits and drawbacks of each cloud setup.

**R12:** Compare and evaluate the cloud options based on ease of initial setup, cost involved for initial setup, maintenance, management overhead to keep the infrastructure running.

### **2.1.7 Customer SLA**

**R13:** Study the feasibility of accommodating customers with different applications.

## **2.2 Non-Functional Requirement**

**NFR1:** Provide a CLI based dashboard which displays the utilization statistics to the customers. Provide a CLI based dashboard option to the users to deploy and delete the containers which are running applications.

# Chapter 3

## System Environment

### 3.1 Private cloud Setup

The private cloud setup consists of two Dell T610 servers and is used to host containers. The management node for this design is hosted on a Dell T3400 machine. Client requests for application deployment through the management node. The two CoreOS machines and the management node are part of the same broadcast domain. We use Netgear FS524 to connect all the machines. Dell 390 is a test node used during the testing phase to deploy containers on servers by manually logging into the servers.

Table below provides the technical specifications of each of the machines used in Private cloud setup.

	Dell T610(2x)	Dell T3400	Dell 390
RAM	24GB	8GB	4GB
Storage	1 TB HDD	500GB HDD	300GB HDD
Processor	Dual Quad Core 2.4GHz	Quad Core 2.4Ghz	Dual Core 2.4GHz
Operating System	CoreOS	Ubuntu	Ubuntu

Table 3.1: Physical Hardware technical specifications

**Switch:** Netgear FS524

**Container Provisioning Tool:** Docker for provisioning containers on CoreOS machines.

### 3.2 Setup on AWS

We need to make sure that the environment used in AWS has the same capabilities as our private cloud setup. This is to ensure an appropriate comparison between the performance of applications running on private and the public cloud environments. The public cloud setup on AWS is based on AWS's Elastic Compute Cloud(EC2) instances. We host Docker containers on EC2 Container Service (ECS) and run applications on them.

Table given below summarizes the technical specifications of an Amazon Instance

	Test Instance	Dedicated Host
Instance	Amazon Linux AMI 2017.03.0 (HVM), SSD Volume Type	Amazon Linux AMI 2017.03.0 (HVM), SSD Volume Type
Flavor	t2.micro	m4.large (Dedicated Host) m4.2xlarge (Instances)

CPU	1 core @ 2.4 Ghz	24 Physical core CPU @ 2.4 Ghz (dedicated host) 8 Core vCPU @ 2.4 Ghz (Instance)
RAM	1 GB	32 GB

Table 3.2: AWS technical specification

**Container Provisioning Tool:** Docker for provisioning containers on AWS, using Amazon ECS (EC2 container service)

# Chapter 4

## Design, Implementation and Results

We were given the task to come up with cloud options for Containers as a Service. There are 4 types of clouds - Public, Private, Hybrid and Community. As Hybrid and Community clouds are a mix-match of the Public and Private clouds, we chose to further explore only the public and the private cloud as platforms to deploy our containers and check their performance. We have explained the design of Private Lab setup, ways to use existing instances on the AWS, to compare the performance results of the containers.

### 4.1 Design Options

For our private lab set-up, we had to choose the operating system to be installed on the bare metal machines. CentOS and CoreOS were the two operating system options that we had. We did a comparative study on the feasibility of running container applications for the two operating system options. The comparison results are mentioned in the following section.

#### 4.1.1 Comparison of CentOS and CoreOS

We used CentOS while learning about Docker containers. We ran containers of different applications, configured networking between containers and used ‘Docker stats’ to pull stats of containers in both CoreOS and CentOS. Since one of the requirements of our project was to run container applications on CoreOS, we have used CoreOS as the operating system for the private cloud setup. Table showing the comparison results between CentOS and CoreOS for running containers is given below.

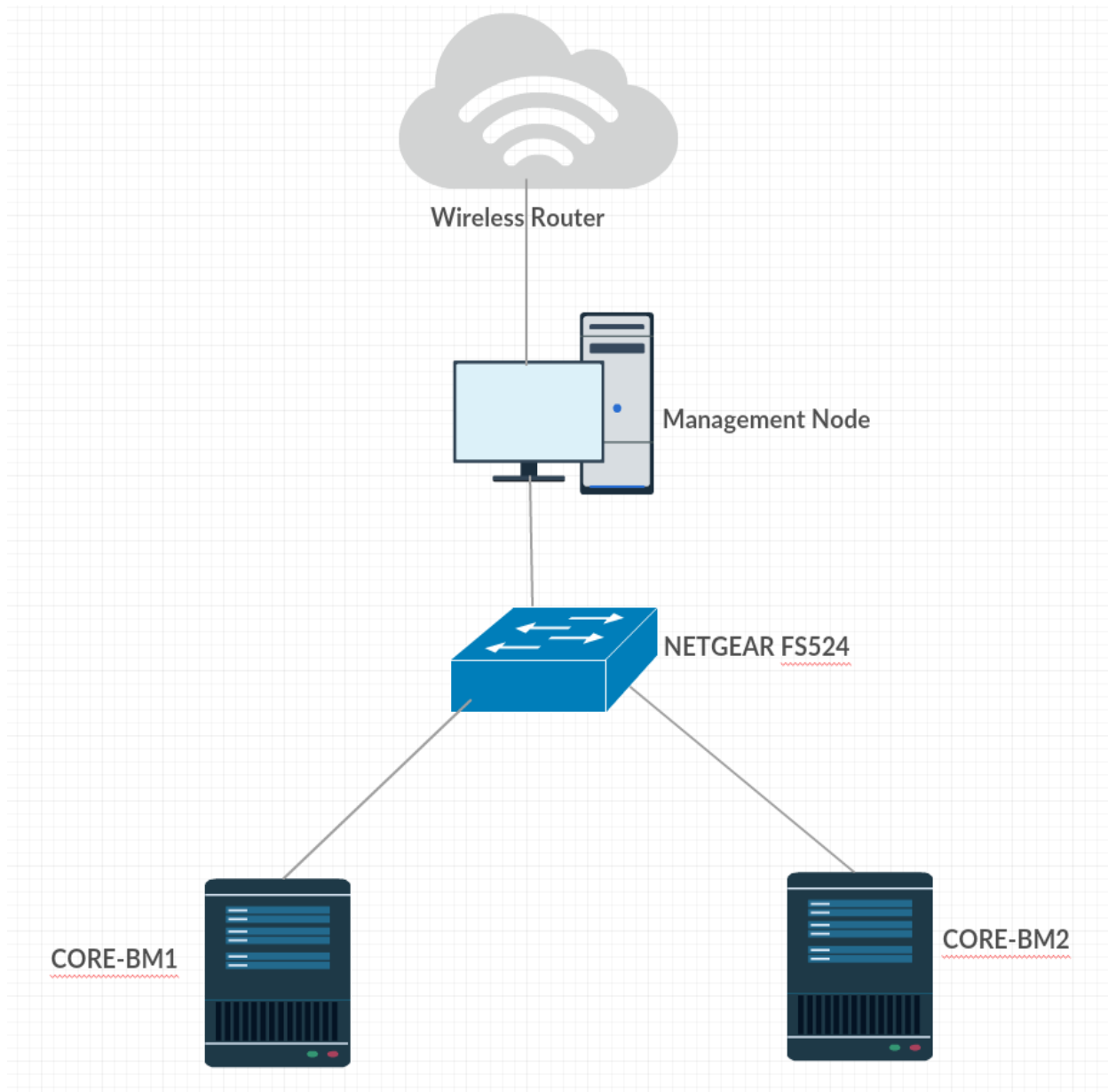
Feature	CentOS	CoreOS
<b>Uptime and availability</b>	Can use “Docker Swarm” to provide high availability. Configuring “Docker Swarm” can be hard.	CoreOS has a built-in feature called fleet that helps set up high availability. Configuration is simpler
<b>Ease of management</b>	Panamax, Cockpit and DockerUI are some of the Docker management tools.	CoreOS only supports Panamax Docker management tool.
<b>Size of OS image</b>	CentOS image is larger in size. Images are at least ~1GB.	CoreOS image is lighter and is limited to 260MB
<b>Package Manager</b>		CoreOS doesn’t have any default package manager.

	CentOS comes in with a default package manager 'yum'.	
<b>Community support</b>	This OS is used from a relatively long time and hence we can find solutions to our problems in a relatively easier way.	CoreOS is a recent technology and community support is limited. Solutions to many known issues are still being developed.

Table 4.1: Comparison between CoreOS and CentOS

## 4.2 Private Cloud Setup

Figure 4.1 shows the topology design for the private cloud setup in the lab. It consists of a management node (running Ubuntu OS) to manage the two CoreOS machines. The management node takes customer requests to deploy their applications in containers on CoreOS machines. The management node also gives internet connectivity to the CoreOS machines as it has NAT enabled on it. We use a switch to keep the servers within a single Local Area Network (LAN).



**Figure 4.1: Network topology for Private Lab set-up**

The design of the lab setup is such as to support the customers' requirements and be able to perform tests for verifying the satisfaction of all the requirements stated earlier. The setup consists of two servers with high processing capability and memory to be able to run memory and compute intensive applications for the customers. Management node manages the two CoreOS servers. The management node serves a dual purpose, it acts as a central point where customers can get server utilization statistics and it also prevents the customers from directly accessing the servers as we only allow the management node access to the servers.



The requirement was to have a lightweight operating system specially designed to deploy containers so we chose CoreOS. It is coupled with security in terms of who could gain access to the servers. Deploying containers directly on top of the bare-metal hardware helps us in avoiding virtualization overhead.

With the hardware and the OS in place, the last requirement was to network the devices so that the servers, management node and test management node can communicate with each other. Public internet access was also required so that the client could place requests to the management node and for servers to update images of containers and pull images from a repository for client application deployment.

### **4.3 Public Cloud Setup**

AWS is the public cloud setup we use, to deploy the client's application in containers. Docker hub repository is used for storing, pulling and pushing the Docker container images that we would be using to spin up containers and deploy customers' applications.

To have an environment similar to the private cloud, we reserve a dedicated host in AWS. The technical specifications of the dedicated hosts are mentioned in Chapter 3 – Section 3.2. For monitoring the CPU and memory utilization of the applications running in containers, we use Amazon's Cloud Watch monitoring service.

### **4.4 Design - Management Node Dashboard**

We have designed a CLI based management node dashboard for the clients to place requests. Dashboard is developed and written using Python. This dashboard design helped us address the issues mentioned in the Section 1.3.2.

### **4.5 Implementation**

#### **4.5.1 Implementation of Private Cloud setup**

To implement the design plan in the lab environment, the technical specifications of the hardware, the base operating systems and the functions of each device are mentioned in section 3.1.

The operating system running on the servers had to be lightweight, optimized for container deployment and secure. CoreOS was our operating system of choice as it is light weight, comes pre-installed with Docker engine and allows access only via key-value authentication. This security feature stands out as no operating system in the Debian, RHEL or OpenSUSE distribution have this capability. A user can only access a CoreOS installed machine via a separate remote host that has the appropriate private key.

The following sections elaborate on the installations of CoreOS on the servers, the network configuration, and the implementation of the management Node Dashboard to get our Lab setup running.

#### 4.5.1.1 Installing CoreOS

The steps for installing CoreOS on the Dell T610 machines are provided below

**Step 1:** Make a bootable pen-drive with a live image (we chose CentOS 7 live).

**Step 2:** Connect the pen-drive to the machine on which CoreOS needs be installed and boot the machine from the pen-drive. Make a `coreos_install.sh` and `cloud-config.yaml` file. Copy CoreOS installation script from gitHub to `coreos_install.sh` and make `coreos_install.sh` executable.

**Step 3:** Logging in to a CoreOS server authentication is done using keys, so generate RSA keys on the machine used to connect to a CoreOS machine. Make sure the RSA keys on the machine used to SSH into the CoreOS servers are in a location that does not get erased when the machine is shutdown (not in `/tmp`). Copy the public key into `cloud-config.yaml` file.

**Step 4:** After the `cloud-config.yaml` file has been populated validate it using the online validator.

**Step 5:** Install CoreOS using the following command -

```
$ sudo ./coreos_install.sh -d /dev/sdb -C stable -c cloud-config.yaml
```

-d is the storage device to install coreOS(chose the HardDisk)

-C is the channel, it can be stable, alpha or beta

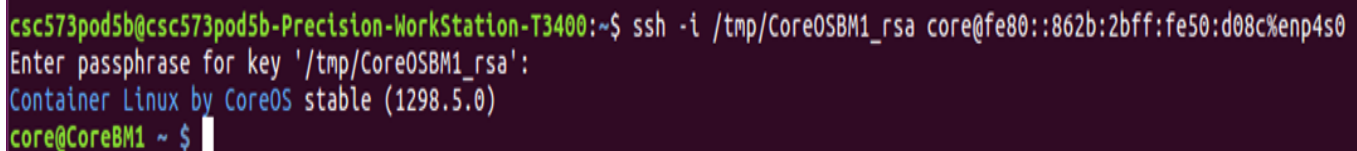
-c location of the cloud-config.yaml file

**Step 6:** Reboot the machine, and boot from hard disk as CoreOS is installed on the HardDisk. It will prompt for a login. Now go to the machine which has the private key and ssh into the CoreOS server.

```
$ ssh -i ./CoreBM1_rsa core@IPaddress
```

Here, `./CoreBM1_rsa` is the location of the private key used to authenticate.

**Step 7:** Use the pass phrase set for the private keys to log in. Figure 4.2 shows a successful login into a CoreOS machine from the management node.



```
csc573pod5b@csc573pod5b-Precision-WorkStation-T3400:~$ ssh -i /tmp/CoreOSBM1_rsa core@fe80::862b:2bff:fe50:d08c%enp4s0
Enter passphrase for key '/tmp/CoreOSBM1_rsa':
Container Linux by CoreOS stable (1298.5.0)
core@CoreBM1 ~ $
```

**Figure 4.2: SSH from management node into CoreOS server**

**Note:**

1. It is necessary to have an internet connection to download coreOS onto the machine, we have a wireless router for internet connectivity.
2. If you reuse the ssh keys when you reinstall CoreOS then you will have to remove known\_hosts and known\_hosts.old from .ssh directory. Commands we used for doing this were given below.

```
$rm /home/csc573pod5b/.ssh/known_hosts
```

```
$rm /home/csc573pod5b/.ssh/known_hosts.old
```

3. If the IP address is IPv6, specify the interface on the client machine -

```
$ ssh -i ./CoreBM1_rsa core@IPaddress%interface_name
```

#### 4.5.1.2 Network Configuration on CoreOS and Management Node.

Figure 4.3 shows the networking setup done for the private cloud platform. The steps followed to do the network configuration are given after the network diagram.

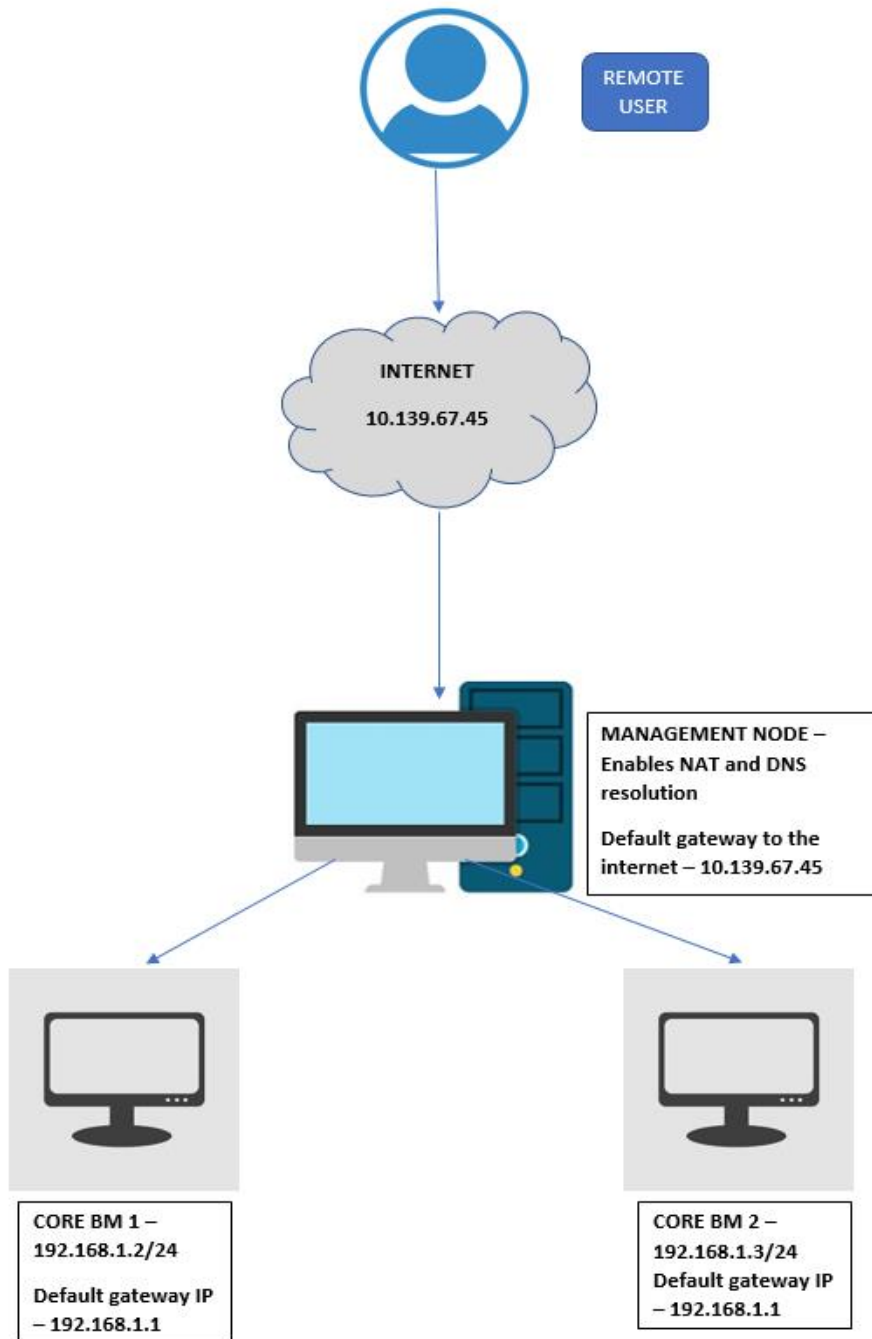


Figure 4.3: network setup for private cloud setup

For the network configurations to be persistent we create network files on CoreBM1 and CoreBM2 and assign private static IP's in the network 192.168.1.0/24. For DNS resolution, the /etc/resolv.conf file on CoreBM1 and CoreBM2 should have the management node private interface IP as the management node is tasked with DNS resolution.

The contents of the '/etc/eno1.network' and the '/etc/resolv.conf' file for both CoreBM1 and CoreBM2 is given in the Appendix section.

### Management Node:

Steps to enable NAT on the management node are given below:

**Step 1:** Add a default gateway on both the CoreOS machines (BM1 & BM2) to the management node (Ubuntu machine) :-

```
# sudo route add default gw 192.168.1.1
```

**Step 2:** Enable post routing on the interface that connects to the public internet. We use wireless router's interface to connect to public internet.

```
#sudo iptables -t nat -A POSTROUTING -o <interface-name> -j MASQUERADE
```

**Step 3:** Restart the network manager service on the CoreOS machines:-

```
#sudo service network-manager restart
```

After this, CoreOS machines could access the internet.

#### 4.5.1.3 Implementation of Management Node Dashboard

To implement the CLI-based dashboard, we chose Python as our coding language as running shell scripts from Python is easy. We make use of the sub process module in python to call .sh files and execute them. These scripts have commands to get Docker statistics (CPU and Memory utilization) of the CoreOS servers. They also have commands that spawn new containers as per user requests. The bash scripts used are mentioned in the Appendix.

### 4.6 Workflow

Figure 4.4 illustrates the workflow when a user first logs in to our private cloud setup. We have shown two cases in the workflow, one showing the case where user successfully deploys the container and the other showing the case where the deployment fails.

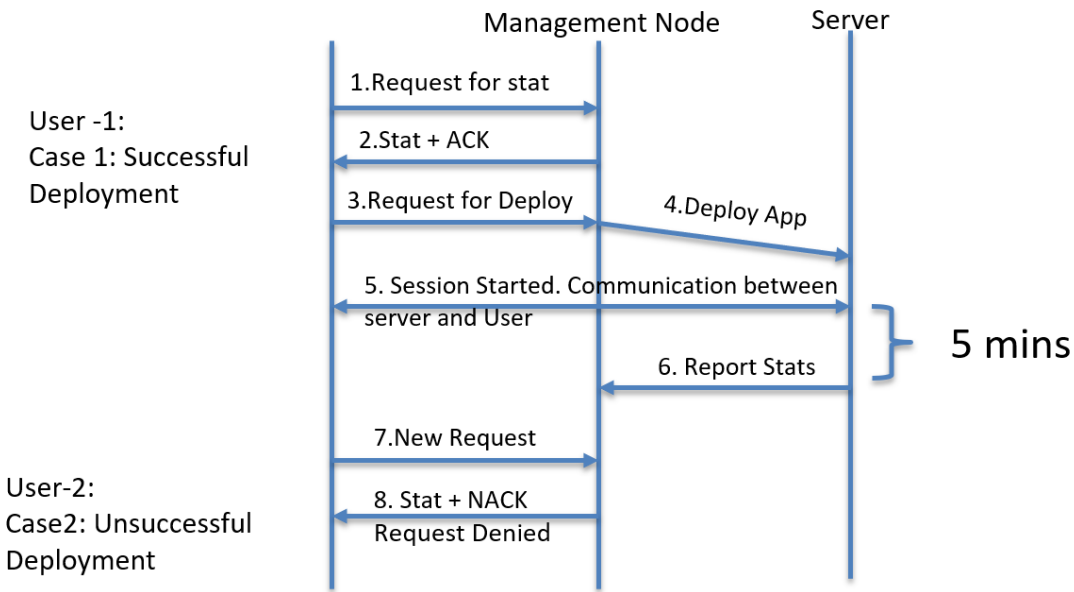


Figure 4.4: Workflow for user to use and deploy containers on private cloud setup

**Case1:** User successfully deploys his application container.

1. User logs in to the private cloud setup using the public IP address of the management node and requests for statistics.
2. The management node will respond back with the statistics and an acknowledgement as resources are available.
3. The user requests to deploy his/her application.
4. The management node will deploy the users desired application using the appropriate scripts.
5. After deployment is successful, a communication channel is established between the user and his containers on the CoreOS servers.
6. Statistics of the CoreOS servers are sent to the management node periodically.

**Case2:** Management node denies user's request to deploy containers.

7. A new user logs in to the private cloud setup using the public IP address of the management node and requests for statistics.
8. The management node will respond back with the statistics and a negative acknowledgement as resources are unavailable.

***Note: Statistics are memory and cpu utilization of the CoreOS servers***

## **4.7 Results**

This section highlights our findings based on the tests mentioned in section 5.1, 5.2, 5.3 and 5.4. We observed that the CPU utilization of servers are high in constrained environment for AWS as well as our Private cloud setup. The performance of the video encoding application (CPU intensive application) is better in a constrained environment as it has complete access to the assigned core, leading to constant completion time per scheduled task. But, in an unconstrained environment, the server CPU utilization of our private cloud is low and application performance degrades as time to complete a video encoding request increases with increase in simultaneously executing tasks. Whereas in AWS, the server utilization increases but application performance is degraded as time to complete a video encoding request increases with increase in simultaneously executing tasks. So, based on our results we suggest that a constrained environment is better for deploying a CPU intensive application. This is because the completion for the video encoding task was constant, irrespective of the number of simultaneously executing tasks. The better cloud platform for a CPU intensive application is AWS, the deciding factor was cost as the customer will only pay per use.

From the tests conducted for the memory intensive application on the private cloud setup and AWS, mentioned in section 5.5 and 5.6 we conclude our private cloud setup is better. Performance was better as with the increase in the number of requests on the private cloud, the server does not crash. Only the execution time for the requests were delayed. But on AWS, the dedicated host and the instance crashed as the number of requests increased beyond the dedicated hosts capacity.



# Chapter 5




## Verification and Validation

### 5.0 Mapping of requirements and verifications






In this section, we map the requirements mentioned in Chapter 2 to the test cases we perform to verify and validate our expected and observed results. There could be variance in estimated and observed results, for which we justify the cause of each variance.



Legend:

	<b>The expected results align with the actual results.</b>
	<b>The expected results do not align with the actual results.</b>

Req	Test Case	Expected Result	Actual Result	Match
R1	Spawn 1,5,10,16 containers with each container assigned a single core. Perform this in private cloud setup	CPU utilization should increase as the number of containers spawned increase	CPU utilization increase as the number of containers spawned increases	
R2	Spawn 1,5,10,16 containers – with each container assigned a single core. Perform this in AWS	CPU utilization should increase as the number of containers spawned increase	CPU utilization increases with the increase in the number of containers	
R3	Spawn 1,5,10 containers and the containers are allowed to use any	Total CPU utilization should rise with more containers being deployed.	Total CPU utilization rises initially but then settles at a level due to	



	core. Perform this in private lab setup		contention for CPU time between the containers	
<b>R4</b>	Spawn 1,5,10 containers and the containers are allowed to use any core. Perform this in AWS	Total CPU utilization should rise with more containers being deployed.	Total CPU utilization rises with the number of containers we run.	
<b>R5</b>	Run a memory intensive application in the private cloud setup.	CPU utilization should be low, Memory utilization should be high.	CPU utilization spikes initially and comes down. Memory utilization rises with the increase in the amount of read-write into RAM	
<b>R6</b>	Run a memory intensive application on AWS	CPU utilization should be low, Memory utilization should be high.	Memory utilization increased with the size of data written in to the RAM. As the number of containers increased, the server crashes because of the CPU scheduler of the Amazon Linux.	
<b>R7</b>	Iperf test for applications running on containers in private cloud setup.	Docker networking shouldn't cause significant overhead on TCP connections.	Network throughput decreases for the Iperf between containers. But the reduction is negligible	
<b>R8</b>	Iperf test on AWS container	Docker networking shouldn't cause significant overhead on the TCP connections.	Network throughput decreases for the Iperf between containers. But the reduction is negligible	

<b>R9</b>	Measured initialization time – private cloud setup	Initialization time of the Containers should increase with the number of containers spawned.	Initialization time of the containers increases with the increase number of containers.	
<b>R10</b>	Measured initialization time - AWS	Initialization time of the Containers should increase with the number of containers spawned.	Initialization time of the containers increases with the number of containers being spawned.	
<b>R11</b>	Comparison between CoreOS and CentOS is provided in Design Section	The comparison results are given in section 4.1.1		
<b>R12</b>	Comparison between AWS and private cloud setup for different scenarios is mentioned in one of the subsections of verification and validation.	The comparison is explained in detail in Table 5.2		
<b>R13</b>	Satisfying customer SLA in the private cloud setup	We create a hypothetical scenario and suggest how to support simultaneously requests for CPU and memory intensive applications. This is elaborated in section 5.12		
<b>NFR 1</b>	To have an end-user portal	Users should be able to use the dashboard at the Management Node for his/her ease of access and to control his/her own containers		

**Table 5.1: Mapping of test-cases to requirements**

A video encoding application was chosen to test the CPU utilization (Section 5.1, 5.2, 5.3 and 5.4). The number of computations required to encode a video increases as the requirement for video quality increases. This results in increased CPU utilization during the encoding period.

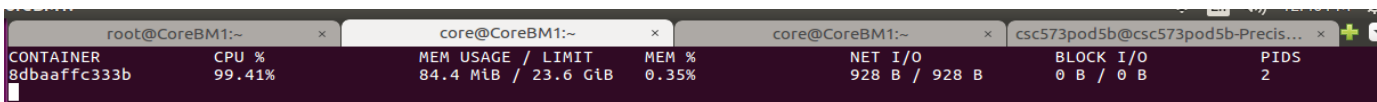
A MySQL database server is chosen to test Memory Utilization (Section 5.5 and 5.6). Reading from a large database Table ( $10^7$  entries) using a `SELECT *` query requires a large amount of data to be moved from disk to memory.

## 5.1 Test cases and analysis of results of R1

**R1 test case:** Measure the CPU utilization of a container when it is only allowed to use a single core.

**Test Case scenario:** Spawn one, five, ten and sixteen containers and assign only a core to each container. Verify that CPU utilization is always close to 100% for each core if the application is running. This helps us ensure that application is using the assigned core and is CPU intensive. Also verify that the total CPU utilization of the server should increase with each container deployed.

**Observation and Inference:** When an application is deployed in a container, it utilizes only the assigned core and its CPU utilization is near 100% as shown in Figure 5.1.1. As we increase the number of containers deployed, the server CPU utilization also increases which is illustrated in the Figure 5.1.2.



CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
8dbaaffc333b	99.41%	84.4 MiB / 23.6 GiB	0.35%	928 B / 928 B	0 B / 0 B	2

Figure 5.1.1: Docker stats for CPU utilization of a container in a constrained environment

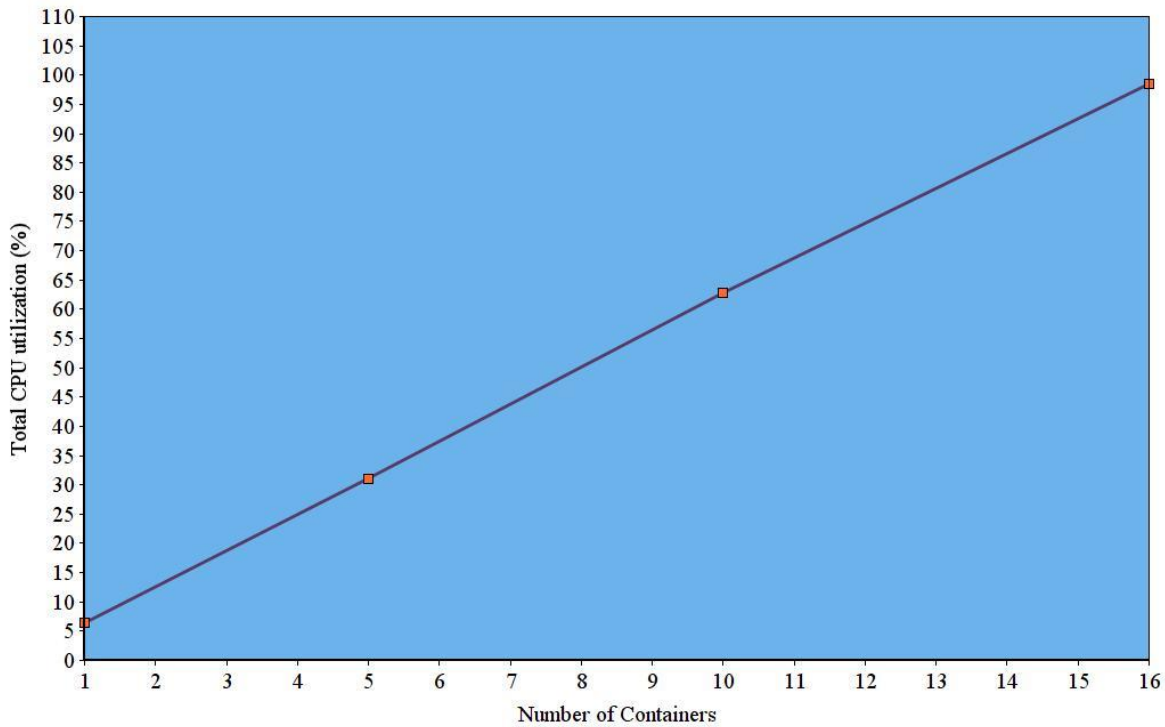


Figure 5.1.2: Total CPU utilization vs Number of Containers

## 5.2 Test cases and analysis of results of R2

**R2 test case:** Measure the CPU utilization of a container when it is only allowed to use a single core in AWS.

**Test Case scenario:** Spawn one, five, and eight containers and assign only one core to each container. Verify that CPU utilization is always close to 100% for each core if the application is running. This helps us ensure that application is using the assigned core and is CPU intensive. Also, the total CPU utilization of the server should increase with each container deployed.

**Observation and Inference:** When an application is deployed in a container, it utilizes only the assigned core and its CPU utilization is about 100% as shown in Figure 5.2.1. As we increase the number of containers, the CPU utilization increased gradually as illustrated in the Figure 5.2.2.

```
[ec2-user@ip-172-31-71-72 ~]$ sudo mpstat -P ALL 5
Linux 4.9.20-11.31.amzn1.x86_64 (ip-172-31-71-72)      05/05/2017      _x86_64_      (8 CPU)
```

		CPU	%usr	%nice	%sys	%iowait	%irq	%soft	%steal	%guest	%idle
10:04:48 AM	all		99.53	0.00	0.47	0.00	0.00	0.00	0.00	0.00	0.00
10:04:53 AM	0		99.20	0.00	0.80	0.00	0.00	0.00	0.00	0.00	0.00
10:04:53 AM	1		99.60	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
10:04:53 AM	2		99.60	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
10:04:53 AM	3		99.60	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
10:04:53 AM	4		99.80	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00
10:04:53 AM	5		99.20	0.00	0.80	0.00	0.00	0.00	0.00	0.00	0.00
10:04:53 AM	6		100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
10:04:53 AM	7		99.80	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00

		CPU	%usr	%nice	%sys	%iowait	%irq	%soft	%steal	%guest	%idle
10:04:58 AM	all		99.50	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00
10:04:58 AM	0		99.40	0.00	0.60	0.00	0.00	0.00	0.00	0.00	0.00
10:04:58 AM	1		99.20	0.00	0.80	0.00	0.00	0.00	0.00	0.00	0.00
10:04:58 AM	2		99.60	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
10:04:58 AM	3		99.60	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
10:04:58 AM	4		99.60	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.00
10:04:58 AM	5		99.80	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.00
10:04:58 AM	6		99.20	0.00	0.80	0.00	0.00	0.00	0.00	0.00	0.00
10:04:58 AM	7		99.20	0.00	0.80	0.00	0.00	0.00	0.00	0.00	0.00

Figure 5.2.1: CPU utilization per Core in a constrained environment on AWS

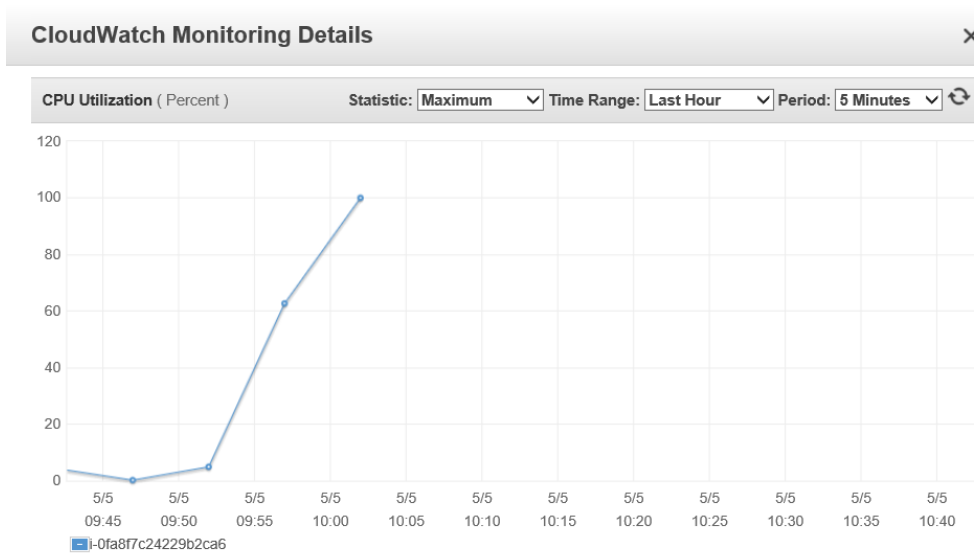


Figure 5.2.2: Total CPU utilization vs Time as the number of containers were increased

### 5.3 Test cases and analysis of results of R3

**R3 test case:** Measure the CPU utilization of a container when it can use any available core in private cloud setup.

**Test Case scenario:** Spawn one, five and ten containers. Verify that CPU utilization is more than 100% for each if the application is running. This helps us ensure that application is using all the required cores and is CPU intensive. Also verify that the total CPU utilization of the server increases with each container deployed.

**Observation and Inference:** When an application is deployed in a container, it utilizes any available core. Figure 5.3.1 shows that in an unconstrained environment, the container is utilizing all the available cores required for the application. In an unconstrained environment, as the number of containers increases the CPU load increases, then settles without increasing much, as number of containers spawned increases. The utilization does not increase further (Figure 5.3.2) as the containers contend for the same core, this is due to the scheduling algorithm of CoreOS.

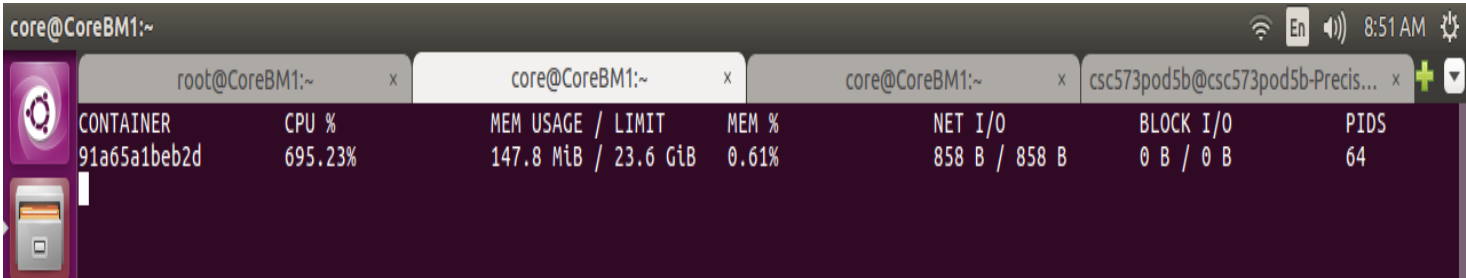


Figure 5.3.1: Docker stats for CPU utilization of a container in unconstrained environment

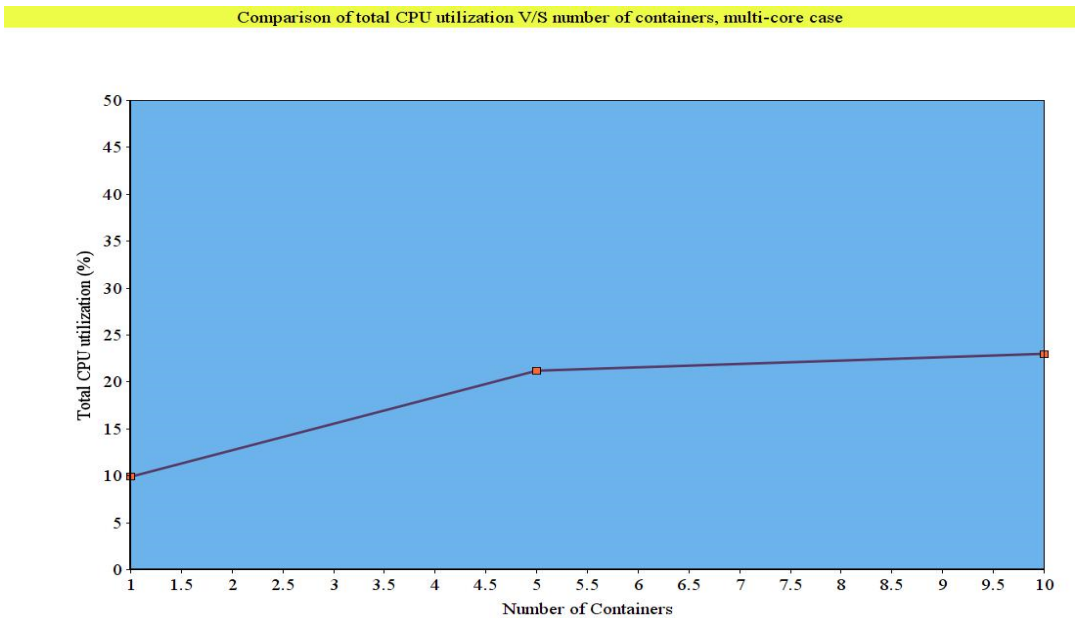


Figure 5.3.2: Time taken to encode video versus number of containers

5.4 Test cases and analysis of results of R4

**R4 test case:** Measure the CPU utilization of a container when it can use any available core on AWS.

**Test Case scenario:** Spawn one, five and ten containers. Verify that CPU utilization is more than 100% for each if the application is running. This helps us ensure that the application is using all the assigned core and is CPU intensive. Also verify that the total CPU utilization of the server, increases with each container deployed.

**Observation and Inference:** When a single Container is spawned, the CPU utilization went over 1000%, as it had access to 16 Cores as shown in Figure 5.4.1. As the number of Containers are increased, the server CPU utilization increased to a 100%, as shown in the Figure 5.4.2.

CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
4b8a4da17468	1143.65%	334.2 MiB / 31.42 GiB	1.04%	1.348 kB / 1.208 kB	151.9 MB / 0 B	0

Figure 5.4.1: Docker stats for CPU utilization of a container in unconstrained environment

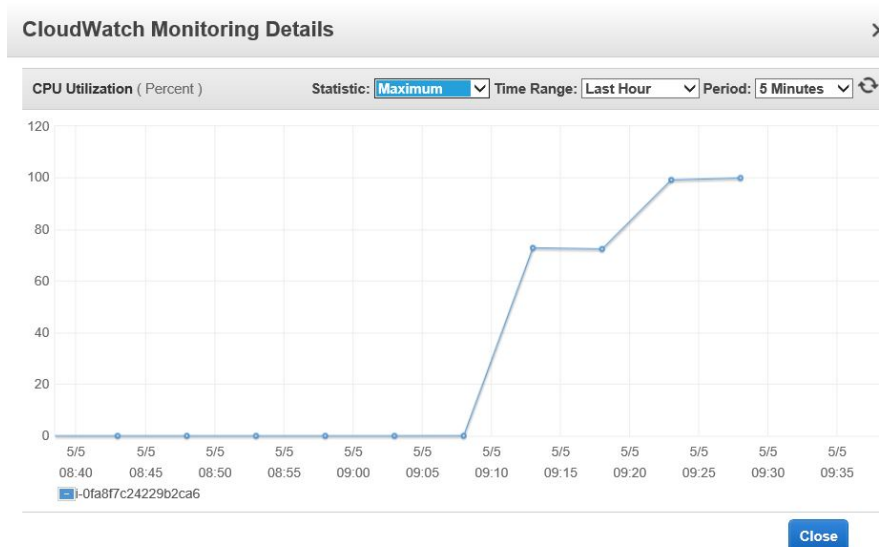


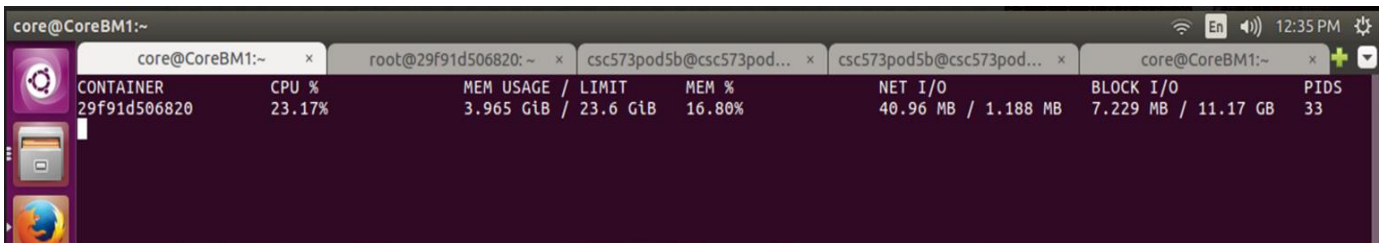
Figure 5.4.2: Total CPU utilization vs Time as the number of containers were increased in an Unconstrained environment on AWS

## 5.5 Test cases and analysis of results of R5

**R5 test case:** Measure the Memory utilization of a container when it is running in the Private cloud setup.

**Test Case scenario:** Spawn one container that has a very large database table. Perform a *SELECT \** operation on the table and verify that memory utilization increases.

**Observation and Inference:** It is observed that when we do a *SELECT \** operation, the memory utilization as shown in Figure 5.5.1 is significantly high when compared to the memory utilization in a CPU intensive application in Fig 5.2.1. From the high memory utilization, we can infer that contents of the database table are being written into the RAM.



The screenshot shows a terminal window with a table of container statistics. The table has columns for CONTAINER, CPU %, MEM USAGE / LIMIT, MEM %, NET I/O, BLOCK I/O, and PIDS. The data for container 29f91d506820 is as follows:

CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
29f91d506820	23.17%	3.965 GiB / 23.6 GiB	16.80%	40.96 MB / 1.188 MB	7.229 MB / 11.17 GB	33

Figure 5.5.1: Memory utilization of a memory-intensive application in a container on CoreOS

**Test Case Scenario:** Spawn 5, 10 and 15 containers that have a very large database table. Perform a *SELECT \** operation on the table in each container and estimate the number of containers that can simultaneously run.

**Observation and Inference:** When the number of containers increase, the memory utilization increases till all the memory is used. After the memory has been exhausted the new queries will be suspended until the containers have access to free memory. Figure 5.5.2 shows that memory utilization is 100% (we only have access to 23.6GB RAM) when the queries are being executed.

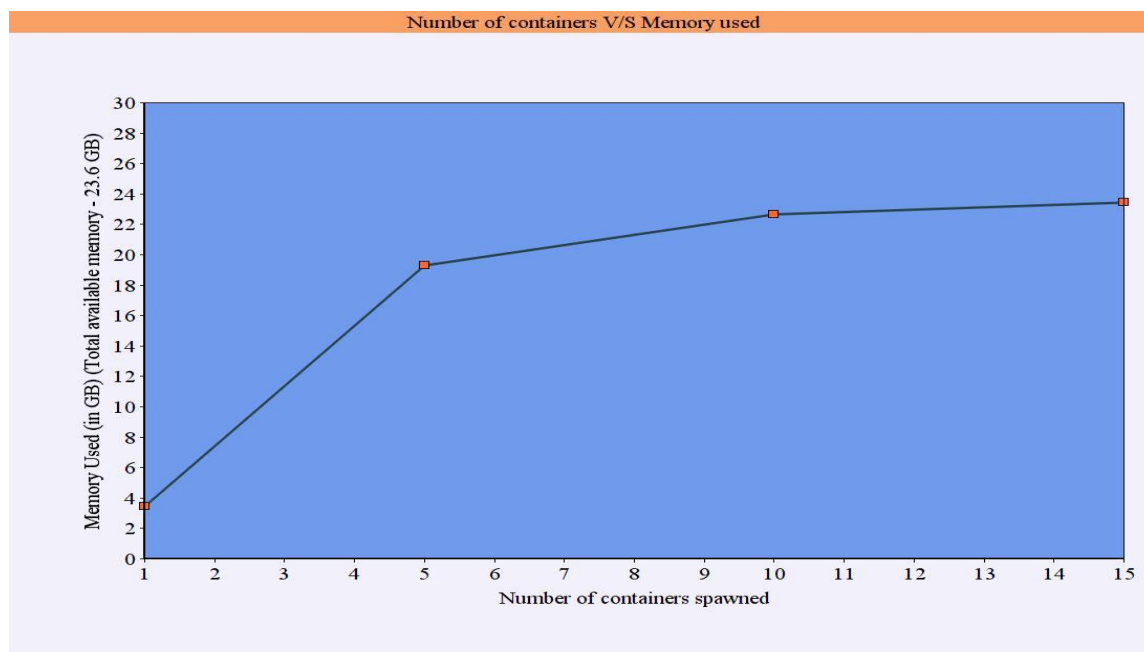


Figure 5.5.2: Memory Utilization as requests for memory intensive containers increases



## 5.6 Test cases and analysis of results of R6

**R6 test case:** Measure the Memory utilization of a container when it is running in AWS.

**Test Case scenario:** Spawn 1, 5, 10 and 15 containers container that has a very large database table. Perform a *SELECT \** operation on the table and verify that all the contents of the table are written into the RAM.

**Observation and Inference:** Memory utilization for the *SELECT \** operation increases with the size of the database table. And as the number of containers are increased, it directly effects the performance of the Server. The server crashes if the memory requirements exceed available RAM and for the server to recover to its normal state, it waits until all the tasks running within the containers are completed, which sometimes results in a *deadlock*.

The m4.2xlarge instance crashed at 15 containers running a memory-intensive application due to a deadlock.

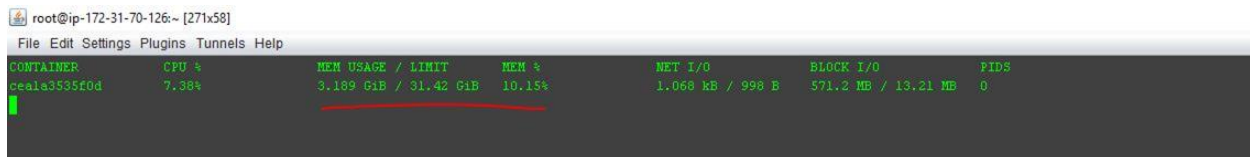


Figure 5.6.1: Memory utilization of a memory-intensive application in a container on AWS

## 5.7 Test cases and analysis of results of R7

**R7 test case:** Testing the effect of container networking on performance. Docker containers use a ‘docker0’ bridge by default. All containers that are spawned up automatically connects to the ‘docker0’ bridge. This bridge is used to communicate between the containers and to connect the container to the outside world.

**Test Case scenario:** Objective of this test case to analyze the impact of Docker containers networking. We use ‘iperf’ TCP tool to measure the results. Initially, we ran iperf between the CoreBM1 and CoreBM2 machines. We use a standard of 1G traffic as the base case for this test case. CoreBM1 is made as ‘Iperf Server’ and CoreBM2 is made as ‘Iperf Client’. The results of the throughput for the communication between the two machines is noted down. Later, we installed containers on both the Core machines. We initiated an Iperf session between the container in CoreBM1 and the container in CoreBM2. We captured the output of the ‘Iperf’ command and noted down the throughput.

**Observation and Inference:** We observe that there is not much variation in the throughput of the first test case and second test case. We observe only a small variation in terms of kilobits which can be tolerated for a gigabit

of traffic transfer. Also, we are ensuring that only the traffic generated by Iperf is running and there is no other network activity. Thus, we infer that container networking doesn't add much of an overhead in case of private lab setup.

## 5.8 Test case and analysis of results of R8

**R8 test case:** Testing the effect of container networking on performance in AWS set-up. Docker containers use a 'docker0' bridge by default. All containers that are spawned up automatically connects to the 'docker0' bridge. This bridge is used to communicate between the containers and to connect the container to the outside world.

**Test Case scenario:** Testing the effect of container networking in AWS set-up. We use 'Iperf' in this case. For this, we reserve two AWS images and start Iperf between the two images. We make one of the AWS images as 'Iperf server' and the other one as 'Iperf client'. The results of the throughput for the TCP communication between the two machines is noted down. Later, we started containers on both AWS machines. TCP connection was initiated between the two containers using Iperf. The throughput achieved is noted down.

**Observation and Inference:** We observe that there is not much variation in the throughput of the first test case and second test case. We observe only a small variation in terms of kilobits which can be tolerated for a gigabit of traffic transfer. Also, we are ensuring that only the traffic generated by Iperf is running and there is no other network activity. Thus, we infer that container networking doesn't add much of an overhead in case of private lab setup.

## 5.9 Test case and analysis of results of R9

**R9 test case:** Testing the initialization time for containers in Private cloud setup. Initialization time is the time from which the container is spawned till the application in it starts running. We check the initialization time for different number of containers and document the results. We expect the initialization time to increase, as the number of containers spawned increases.

**Test case scenario:** We first start with initializing 5 containers. We also start applications on those five containers. We measure the time taken to first initialize and start applications for 5 containers. We repeat the same for different number of containers. We have test cases where we spin 15, 50 and 100 containers and measure the time taken for the container and application to start.

**Observation and Inference:** Based on the observation from the test cases, we could infer that the initialization time increases almost linearly with the number of containers spawned. The graph of initialization time for containers(seconds) vs the number of containers is shown below in Figure5.9

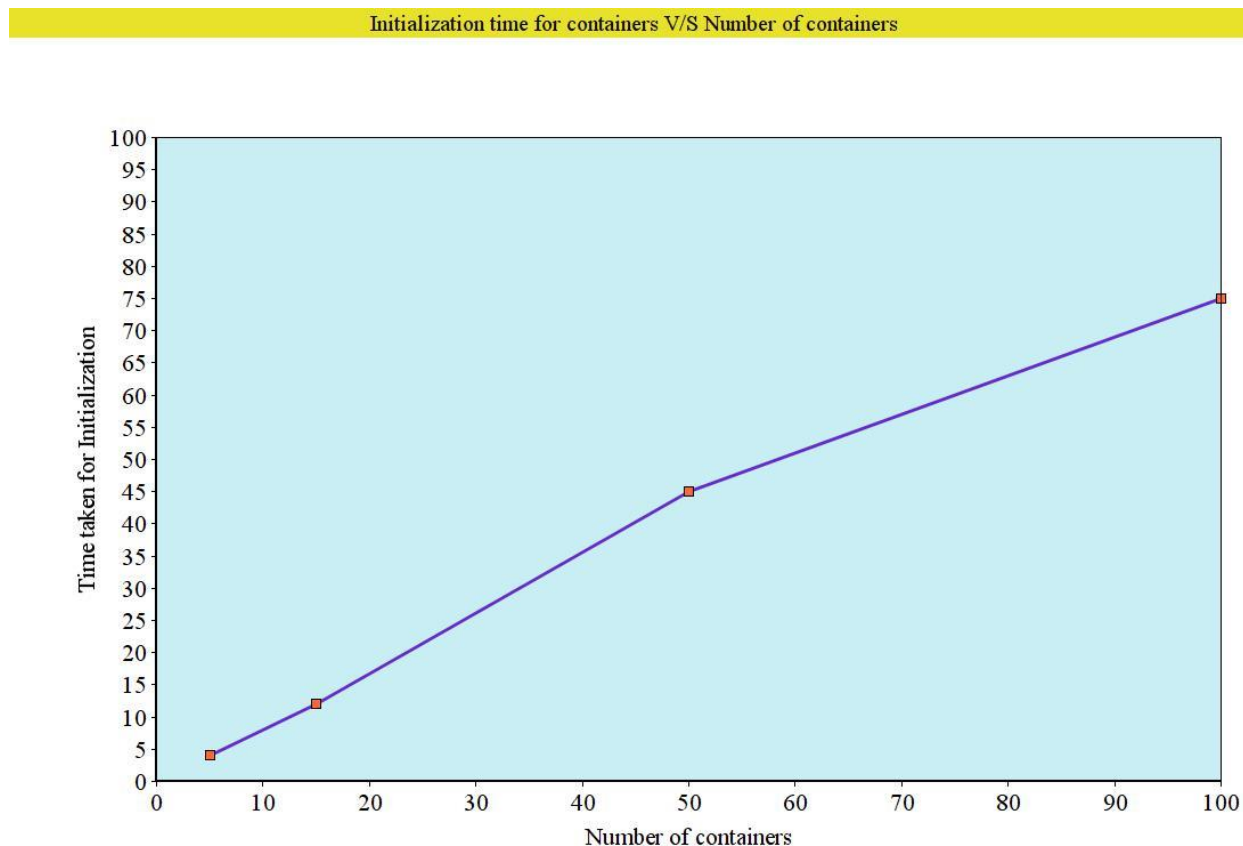


Figure 5.9: Initialization time taken for containers on CoreOS platform

## 5.10 Test case and analysis of results of R10

**R10 test case:** Testing the initialization time for containers in AWS. Initialization time is the time from which the container is spawned till the application in it starts running. We check the initialization time for different number of containers and document the results. We expect the initialization time to increase as the number of containers spawned increases.

**Test case scenario:** We first start with initializing 5 containers. We also start applications on those five containers. We measure the time taken to first initialize and start applications for 5 containers. We repeat the same for different number of containers. We have test cases where we spin 15, 50 and 100 containers and measure the time taken for the container and application to start.

**Observation and Inference:** Based on the observation from the test cases, we could infer that the initialization time increases almost linearly with the number of containers spawned.

### 5.11 Test case and analysis of results of R12

#### Comparison between Bare-metal and AWS setup

In AWS, we need to follow the process of creating an AWS account and then reserving and using the image that we want. We can choose our infrastructure based on the needs of the application. The task of procuring hardware, configuring operating system, network and storage is abstracted away from the user using the application. Management and maintenance of the underlying infrastructure is taken care by the cloud service provider.

In private cloud setup, first step is designing a cloud environment depending on the needs of the application. Since, the private cloud setup is set-up only once, the design must take many things into considerations. Some of them include scalability, reliability, fault-tolerance, and security. The next step is the hardware and software acquisition. We need to buy server, network and storage infrastructure components, licenses that comes along with that and other management related tools/software's to manage and maintain the infrastructure. Apart from this, we need a dedicated team with expertise in infrastructure and cloud to design, configure and maintain the infrastructure.

Table 5.2 summarizes the differences between AWS and the Private cloud.

Features	AWS	Private Cloud Setup
<b>Planning</b>	Only Application specific planning required.	Both Application and infrastructure planning required.
<b>Requirement Analysis</b>	Based on type of application and requirements of applications.	Based on type of application and on the type of infrastructure chosen.
<b>Design</b>	Design is relatively simpler as the infrastructure is already built.	Design is relatively complex as both application and infrastructure requirements needs to be met
<b>Infrastructure set-up</b>	Infrastructure set-up is easier. Required images are reserved and used on the go.	Comparatively complex. Hardware procurement Configure different components of the hardware.

<b>Integration and test</b>	All infrastructure components needed are reserved and used. Integration and testing of these components will be managed by the cloud provided.	All the components should be integrated after configuration. Network needs to be configured for the infrastructure. Need to test each infrastructure component. Working of the entire infrastructure also needs to be tested.
<b>Operations &amp; Maintenance</b>	Amazon completely manages operations and maintenance of AWS.	Operations and maintenance of the infrastructure to be done locally. Involves some cost.

Table 5.2: Comparison between AWS & Private cloud set-up

## 5.12 Test case and analysis of results of R13

### R13 test case: Customer SLA for the private cloud setup

Spawn containers running CPU and memory intensive applications simultaneously.

#### Observations and Inference:

There are certain limitations while running different types of containers simultaneously on one physical server due to the existing scheduling algorithm of CoreOS. As one application requires high CPU throughout its execution time and the other requires high CPU during its initialization period, running both CPU intensive and Memory intensive applications would result in a deadlock. Also, the CPU intensive application would be allocated lesser RAM than required when it is run simultaneously with a memory intensive application.

So, while writing an SLA with a customer who wants to run a CPU intensive application, the provider would have to allocate all the CPU intensive applications on one physical server and charge him accordingly. Similarly, with a customer running a memory intensive application, would be allocated another physical server.

So, with the current setup of 2 servers for hosting the containers, we could run two types of applications, one on each of the two physical servers.

## 5.13 Non-Functional Requirement

We made our own end-user portal which is basically a python program that calls the bash scripts to get the Docker stats, deploy containers and stop them as per the user's needs. The code has been added to the appendix.

## Chapter 6

### Schedule and Personnel

<b>Tasks</b>	<b>Current Status</b>	<b>Responsible person</b>	<b>Dates</b>
<b>CoreOS installation</b>	Completed	Bijo and Ruchika	24th Feb - 2nd March
<b>Topology design and networking</b>	Completed	Rakshit and Bijo	12 March -18th March
<b>Application deployment on AWS</b>	Completed	Kshitija and Abhishek	12th March -18th march
<b>Topology testing, network testing and verification of functionalities</b>	Completed	Bijo and Rakshit	19th March -26th March
<b>Shortlisting applications for testing CPU, Data, Memory and Network</b>	Completed	Network: Rakshit and Abhishek CPU: Bijo and Abhishek Memory: Kshitija and Ruchika	27th March- 2nd April
<b>Docker dashboard design</b>	Completed	Ruchika and Abhishek	3rd April - 9th April
<b>Dashboard deployment</b>	Completed	Ruchika and Abhishek	10th April - 14th April
<b>Testing of applications on COREOS</b>	Completed	Network: Rakshit and Abhishek CPU: Bijo and Abhishek Memory: Kshitija and Ruchika.	14th April – 21st April

<b>Testing of applications on AWS</b>	Completed	Network: Rakshit and Abhishek CPU: Bijo and Abhishek Memory: Kshitija and Ruchika.	28th April – 8th May
<b>Comparison between AWS and CoreOS</b>	Completed	Group will pull the reports of individual tests together and consolidate it into one.	5th May – 8th May
<b>Report</b>	Completed	Group	14 April - 9 May

**Table 6.1: Schedule and division of work**

## References

- [1] Power, Pollution and the Internet by James Glanzsept - The New York Times, September, 22, 2012  
  
<http://www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html>
  
- [2] CoreOS installation –  
Provides detailed information and a step by step guide to install CoreOS on our machines.  
  
<https://coreos.com/os/docs/latest/installing-to-disk.html>
  
- [3] Getting familiar to use Docker on CoreOS machines. This URL provides information on Docker CLI basics, launching, committing a container and also about the Docker registry.  
  
<https://coreos.com/os/docs/latest/getting-started-with-docker.html>
  
- [4] Reference to run Docker containers on AWS and how one can manage and schedule the Docker containers using AWS.  
  
<http://www.allthingsdistributed.com/2015/04/state-management-and-scheduling-with-ecs.html>
  
- [5] Amazon Web Services' official documentation which made us familiar with using Docker on the AWS platform. It helps in installing Docker on the AWS instances and to run a sample application by spawning a container.  
  
[http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html#install\\_doc](http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html#install_doc)
  
- [6] Using Docker CLI commands to know how to analyse applications. It helped us in pulling out statistics from the containers and test their performance.  
  
<https://docs.docker.com/engine/admin/runmetrics/>
  
- [7] Comparison information between CentOS and CoreOS.



<https://bobcares.com/blog/centos-vs-coreos-os-for-docker-web-hosting/>

[8] Guide and tips to monitor our AWS instances using CloudWatch tool –

<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-cloudwatch.html>

# Appendices

## Installation of CoreOS

The cloud-config.yaml file is used to install the CoreOS operating system. It is the file that CoreOS reads during boot. Ensure that the cloud-config file is valid and has no errors by verifying it with the help of an online cloud config validator. Links to our cloud-config.yaml files are given below.

CoreBM1

<https://github.ncsu.edu/bjoseph4/CSC-547-HW3/blob/master/CoreBM1/cloud-config.yaml>

CoreBM2

<https://github.ncsu.edu/bjoseph4/CSC-547-HW3/blob/master/CoreBM2/cloud-config.yaml>

## Network Configuration

While setting up the network configuration on our CoreOS machines, we had to make the network interface file /etc/eno1.network and make changes in the already existing file /etc/resolv.conf. Following are the contents of the files:

### On CoreBM1

The network interfaces file /etc/eno1.network giving the details of the interface, gateway IP and the DNS IP address is given below.

```
cat /etc/eno1.network
[Match]
Name=eno1
[Network]
Address=192.168.1.2/24
Gateway=192.168.1.1
DNS=192.168.1.1
```

The contents of /etc/resolv.conf file in CoreBM1 is given below. DNS resolution happens from the contents of this file. As explained in the design section, we have set the default gateway of CoreBM1 to the IP address of the management node.

```
cat /etc/resolv.conf

# This file is managed by systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients directly to
# all known DNS servers.
#
# Third party programs must not access this file directly, but only through the
# symlink at /etc/resolv.conf. To manage resolv.conf(5) in a different way,
# replace this symlink by a static file or a different symlink.
#
# See systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.
nameserver 192.168.1.1
```

## On CoreBM2

The network interfaces file /etc/eno1.network giving the details of the interface, gateway IP and the DNS IP address is given below.

```
cat /etc/eno1.network
[Match]
Name=eno1
[Network]
Address=192.168.1.3/24
Gateway=192.168.1.1
DNS=192.168.1.1
```

The contents of /etc/resolv.conf file in CoreBM1 is given below. DNS resolution happens from the contents of this file. As explained in the design section, we have set the default gateway of CoreBM1 to the IP address of the management node.

```
cat /etc/resolv.conf
# This file is managed by systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients directly to
# all known DNS servers.
#
# Third party programs must not access this file directly, but only through the
# symlink at /etc/resolv.conf. To manage resolv.conf(5) in a different way,
# replace this symlink by a static file or a different symlink.
#
# See systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.
nameserver 192.168.1.1
```

## Python Code for the CLI based docker dashboard

We have a python based Docker dashboard running on the management Node which provides the user with various options such as getting the stats from the coreOS machines, deploying memory intensive and CPU intensive containers. Our Python code makes use of several bash scripts, which it calls using the subprocess module of python.

The command to run the python code is: python ManagmentNode.py

### ManagementNode.py

```
import subprocess
import sys
```

```
def menu():
```

```

print '-----'
print 'Welcome to the Managment Node. Please select the option from the menu below to go ahead:\n'
print '1.Get Stats'
print '2.Get memory intensive containers'
print '3.Get CPU intensive containers'
print '4.Exit\n'
selection = raw_input('Enter the selection: ')
return selection

def Listen_always():
    while True:

        option = menu()
        if option == '4':
            sys.exit(0)
        machine_number = raw_input('Which machine do u want to go ahead
with:\n1.CoreOs1\n2.CoreOs2\n')
        #input_user = raw_input('Do you want to get stats(y/n): ')

        if option == '1':
            if machine_number == '1':
                #print 'hello'
                p=subprocess.call(['./tt.sh'])
            elif machine_number == '2':
                p=subprocess.call(['./tt1.sh'])
        elif option == '2':
            if machine_number == '1':
                ContainerNumbers = raw_input('Enter the number of containers to be spanned: ')
                p=subprocess.call(['./memory.sh',ContainerNumbers])
                q=subprocess.call(['./memory_start.sh', ContainerNumbers])

def main():
    Listen_always()

if __name__ == '__main__':
    main()

```

**Bash Scripts that are used to collect the statistics from the CoreOS machines are:**

- 1.) **tt.sh:** this script stores the last 4 lines of the top command into a file called test2 and the output of the docker stats command run on CoreOS1 machine into a file called test1 and sends it back to the management node for CoreOS2 machine.

```

#!/bin/bash
ssh -i Core1_rsa core@192.168.1.2 << EGG
top -b | head -n 4 > /tmp/script/test2 & >/dev/null
EGG

```

```
ssh -i Core1_rsa core@192.168.1.2 << EGGS
docker stats > /tmp/script/test1 & > /dev/null
EGGS
scp -i Core1_rsa core@192.168.1.2:/tmp/script/test2 .
scp -i Core1_rsa core@192.168.1.2:/tmp/script/test1 .
```

- 2.) **tt1.sh:** this script stores the last 4 lines of the top command into a file called test2 and the output of the docker stats command run on CoreOS2 machine into a file called test1 and sends it back to the management node for .

```
#!/bin/bash
ssh -i Core1_rsa score@192.168.1.3 << EGG
top -b | head -n 4 > /tmp/script/test2 & > /dev/null
EGG
ssh -i Core1_rsa core@192.168.1.3 << EGGS
docker stats > /tmp/script/test1 & > /dev/null
EGGS
scp -i Core1_rsa core@192.168.1.3:/tmp/script/test2 .
scp -i Core1_rsa core@192.168.1.3:/tmp/script/test1 .
```

## Bash Scripts to start the memory intensive docker containers on coreOS1

- 1.) **memory.sh:** This script creates a new memory\_exe.sh script that has the value of the number of containers that the user wants to deploy and then sends it to the concerning CoreOS machine where it it spawns memory intensive containers.

```
#!/bin/sh
cat <<EOL > memory_exe.sh
for i in {0..$1}; do docker run -itd --name=memory\${i} my_sql_image; done
for i in {0..$1}; do docker exec memory\${i} bash -c 'service mysql start ; service mysql status'; done
EOL
scp -i Core1_rsa memory_exe.sh core@192.168.1.2:~
ssh -i Core1_rsa core@192.168.1.2 << EGG
chmod +x memory_exe.sh
./memory_exe.sh
EGG
```

- 2.) **memory\_start.sh:** This script creates a new memory\_core\_start.sh script that the value of the number if containers that the user wants to deploy and sends it to the concerning CoreOS machine where it starts the applications in the containers spawned in 1.

```
#!/bin/bash
cat <<EOL >memory_start_core.sh
for i in {0..$1}; do docker exec -d memory\${i} mysql --user root --password=admin -e 'show databases;use testdb1; show tables; select * from animals' > bijo;done
EOL
scp -i Core1_rsa memory_start_core.sh core@192.168.1.2:~
ssh -i Core1_rsa core@192.168.1.2 << EGG
chmod +x memory_start_core.sh
./memory_start_core.sh
EGG
```

