INTEGRATION OF LXD SYSTEM CONTAINERS WITH OPENSTACK, CHEF AND ITS APPLICATION ON A 3-TIER IOT ARCHITECTURE

A Thesis

by

SIDDHANT RATH

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Rabi Mahapatra
Committee Members,	Dilma Da Silva
	Srinivas Shakkottai
Head of Department,	Dilma Da Silva

August 2017

Major Subject: Computer Science

ABSTRACT

Internet of Things has moved from being a 2-tier server-client into a 3-tier server-gateway-client architecture. The gateway plays a vital role in this 3-tier architecture with intelligence being built into it. With no proper standardization and with more vendors having proprietary apps, which are shared in this multi-tenant gateway, it demands sandboxing and isolation of apps at the gateway.

My thesis explores light weight LXD System containers and state of the art configuration management tools like Chef, to build an architecture, leveraging Infrastructure as a Code, creating an app delivery pipeline to deploy apps in jailed environments at an IoT Gateway while maintaining a minimal overhead. The framework also provides ways to automate tests for deployment validation.

.

To **M**y **P**arents, **D**r. **P**urna **C**handra **R**ath **A**nd **B**ijayalaxmi **M**ishra

For **A**ll **T**he **L**ove **A**nd **S**upport **T**hey **H**ave **G**iven **M**e.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Rabi Mahapatra, and my committee members, Dr. Dilma Da Silva, and Dr. Srinivas Shakkottai, for their knowledge, patience, motivation, and support throughout the course of this research.

I would like to express my gratitude to Blake Dworaczyk, Brad Goodman, Steve Herring and the College of Engineering IT Linux Team for their guidance, ideas and for sharing their immense knowledge in this subject.

Thanks also goes to my friends Shantanu Bansal, Ghanshyam Bhutra, my colleagues and the department faculty and staff for their contributions and making my time at Texas A&M University a great experience.

Finally, thanks to my mother and father for their love and encouragement.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Dr. Rabi Mahapatra and Dr. Dilma Da Silva of the Department of Computer Science and Engineering, and Dr. Srinivas Shakkottai of the Department of Electrical and Computer Engineering.

All work for the dissertation was completed independently by the student.

Funding Sources

There are no funding contributions related to the research and compilation of this document.

NOMENCLATURE

CD	Continuous Delivery
CGROUPS	Control Groups
CI	Continuous Integration
DevOps	Development Operations
HWRP	Heavy Weight Resource Provider
LSM	Linux Security Module
LWRP	Light Weight Resource Provider
VM	Virtual Machine
IAAC	Infrastructure as a Code
IAAS	Infrastructure as a Service
IoT	Internet of Things

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Over the years the IoT stack has transformed itself into a 3 tier based architecture, where many end devices (things) connect to a gateway which communicates with an enterprise back end IoT Server. The gateway plays a similar role to a wireless router placed in one's home but with more roles and responsibilities than routing packets. In state-of-the-art IoT systems, most data are processed, and decisions are taken at the gateway level. With multiple vendors developing IoT solutions, apps share the gateway pushing it towards multi-tenancy. However, these vendor specific apps usually prefer having elevated access and acquire control of the entire system. With no proper standardization and with more vendors having proprietary solutions, sharing of these proprietary apps in the gateway inflict on each other affecting their functionality and performance.
It calls solutions for both app isolation and formation of an app delivery pipeline.
Though app isolation is provided by generic virtualization technologies, they are resource heavy to be used with IoT devices.

In this thesis, we would be exploring various virtualization technologies that use hypervisor at the center. Hypervisors is a software layer that enables a system to run one or more Virtual Machines on a physical server. Hypervisors achieve this by translating and tapping Instruction set architecture and sometimes completely simulating the hardware environment. This increases the overhead and reduces the performance of the system. We would be moving towards eliminating the hypervisor layer altogether by

bringing in the technology of containerization. Machine Containers use advanced Linux

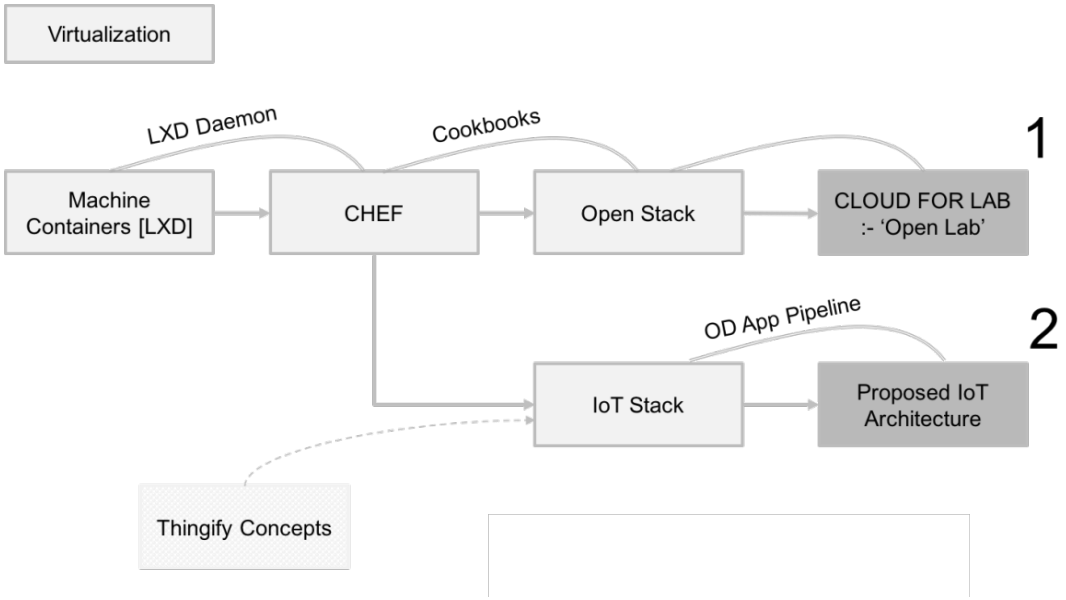

Fig 1.1 Thesis Overview.

concepts like cgroups, namespaces, chroot and LSM to provide a same level of isolation as of virtual machines but with a minimized overhead. We would then be using Chef's configuration management framework to build an app delivery pipeline with continuous delivery and continuous integration. To validate the pipeline a heavy weight resource provider called InSpec Handler will be developed that would perform automated tests.

To provide orchestration services, we would be using OpenStack's framework by integrating it with containers. We would also develop Chef modules to automate the deployment of OpenStack. We now have an end to end solution for bootstrapping, managing containers and an app delivery pipeline. This solution is termed as 'OpenLab,' and can be used to create lightweight automated software as a service cloud environment. Having a lightweight solution that provides system orchestration, configuration management, and device isolation, we have laid the foundation for a 3-tier multi-tenant

IoT system. However, to further optimize the solution we will replace OpenStack by developing a lightweight framework 'IoT Stack.' Together, Machine Containers, Chef, InSpec Handler and IoT Stack will form the core components of our proposed IoT architecture. We will be using various benchmarks to support our claims.

CHAPTER II

INTRODUCTION TO VIRTUALIZATION

2.1 Virtualization

Virtualization [1] is a software layer that is used to abstract out applications and their components from the hardware. The virtualized object is presented with a logical view of the underlying resource. This logical view is not always required to represent an exact replica of the hardware. Moreover, in most of the cases, this logical layer is very different from the physical layer and is designed in such a way that favors the application that is to be run on top of it. The main goal of virtualization is to provide isolation, scalability, reliability, availability and to create a unified process for IT protocols like maintaining security and management.

Virtualization was first implemented more than 30 years ago by IBM. Virtualization was used to partition the mainframe computers into separate virtual machines logically. These partitions allowed mainframes to multitask and run multiple application at the same time. Mainframes were expensive resources, and so they were partitioned in a way to fully leverage the investment.

However, during the 1980s and 1990s, the x86 architecture was brought into the market that made it possible to build inexpensive servers. This led to the formation of a new server-client model.

Instead of using one central computing center, an island of computers was clustered together to form a distributed computing environment. This model, being way

cheaper was widely adopted. Usage of mainframes fell drastically, and virtualization was effectively abandoned.

It was not until the 2000s when companies started realizing the problem of one machine - one OS - one app architecture, thus bringing virtualization back into the picture.

2.2 Goals of Virtualization

In the introduction, we stated that companies started realizing the problem of one machine - one OS - one app architecture. What was the problem exactly?

Most of the machines that were acquired to run the applications were built keeping in mind, the configuration required to run it at full capacity. However, in 85 to 90 percent of the time, there was relatively lesser load and about 15 percent of the total processing capacity was used. In a way, Moore's law became irrelevant to most of the companies as they were not able to take advantage of the increased power density. Given the powerful hardware resources, the software typically used only a fraction of the available processing power. In addition to that, the light loaded machine still took room space, consumed units of electricity and kept the running cost high making it nearly the same to full load running capacity. Taking Moore's law into account, next year, machines will have twice as much as hardware resources further increasing the gap of resource utilization to available resources.

The rise of internet caused companies to establish more communication with customers, partners and even led to collecting real-time data from objects through the IoT platform. To offer an example, Large Hadron Collider produces approximately 25

petabytes of data per year and is being continuously analyzed by the LHC computing grid. This means that more servers are being put together which is causing a real estate problem. Data centers need to be more efficient, with increasing the processing power per sq. ft. ratio.

Moreover, with the increase in the number of servers, there was an increasing demand for system administrators. Servers do not run by themselves. Many factors like installing an operating system, maintaining security patches, monitoring critical services, backing up server data etc. needed to be done. This is labor intensive and required more human resources to be put into it.

Considering all these, virtualization was brought back into the picture.

The blueprint goals [2] of virtualization are as follows: -

1. Isolation of an application from the operating system, allowing it to operate on a foreign operating system.

2. Isolation of workloads from other application to enforce security, thus enabling multiple application to reside in parallel without the risk of affecting other applications when one of them is compromised.

3. Optimizing the use of a single system by running it on near full capacity, decreasing the time the system remains idle.

4. Increasing the reliability or availability of a system through redundancy, if a virtual machine fails, another replica of the machine image should take its place.

5. Easier administration of the resources.

2.3 CPU Virtualization

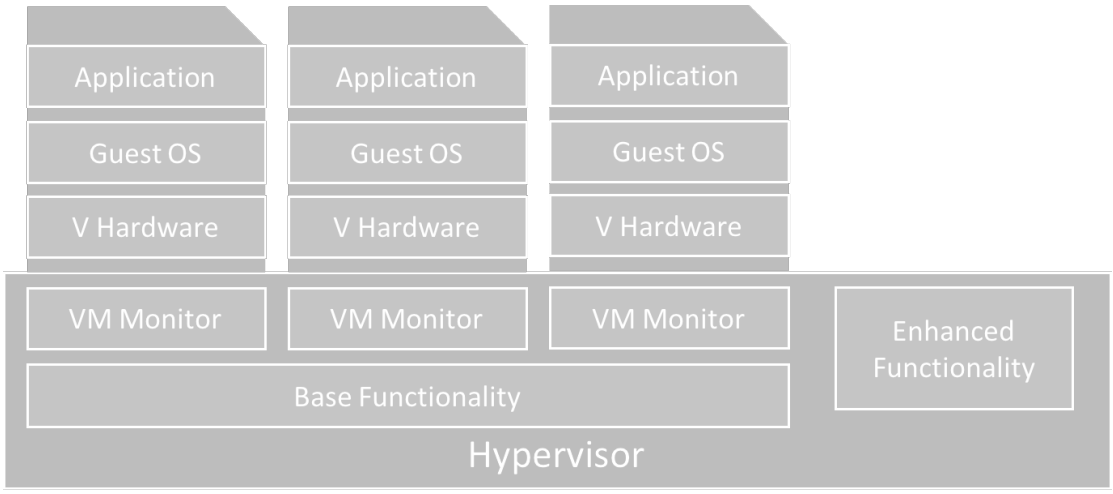

Figure 2.1 Hypervisor Managing Virtual Machines.

A hypervisor is a software layer that enables a system to run one or more Virtual Machines on a physical server. This foreign virtual machine is referred to as 'Guest OS / System' and the physical server in which this hypervisor runs is called the 'Host OS / System.' There are variants of virtualization techniques. Figure 1.1 depicts a virtualization layer in which the hypervisor is directly running on the hardware. The functionality of the hypervisor is greatly dependent on the underlying architecture it is implemented on. Each virtual machine monitor (VMM) running on the hypervisor has to partition and share the CPU, Memory, I/O devices, disks among other VMMs. This abstraction is responsible for running a guest OS.

Challenges of Hardware Virtualization

Operating Systems are designed to run on the bare-metal hardware directly, so they inherently assume the ownership of the complete hardware resource. This creates a problem for the hypervisor to be installed in the most privileged layer.

Considering an example of Intel's x86 architecture, the processor's segment protection mechanism recognizes four privilege levels. The lesser the number means the more privilege they have. Figure 1.2 Shows how these level of privileges can be interpreted as rings of protection. Level 0, which is at the center of the ring has the highest privilege and hosts segments of code containing the most critical software, usually the kernel of the operating system. Outer rings are for less critical software. The operating

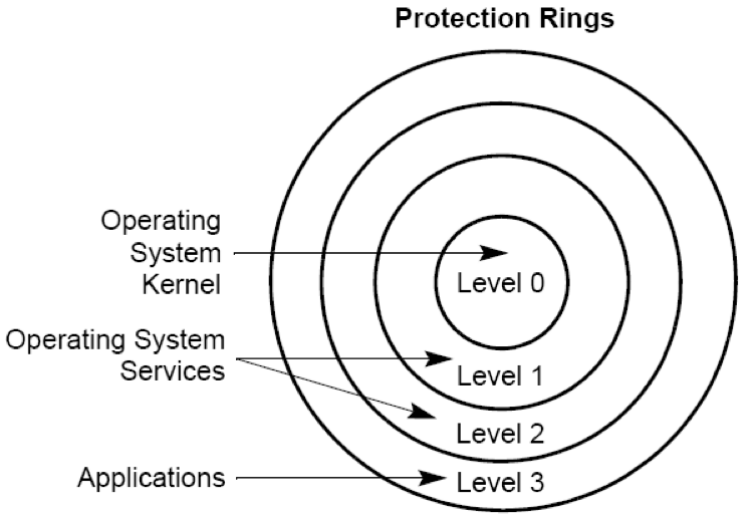

Figure 2.2a x86 Privilege Levels.

system services and APIs lie on layer 1 and layer 2. Applications reside on layer 3. The processor uses these privilege levels to prevent programs operating on the outer areas of the ring to access a segment of the inner ring, except under controlled situations. When a violation is detected, it generates a general-protection exception (#GP). To carry out privilege tests the processor defines the following privilege levels:

Current Privilege level (CPL)

This is the privilege level of the currently executing program. These are stores in the CS and SS segment registers

Descriptive Privilege Level (DPL)

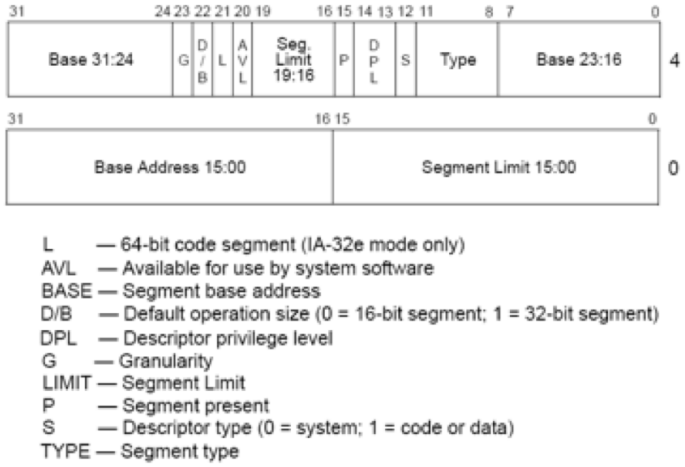

Figure 2.2b x86 Segment Descriptor [3].

It is the privilege level of a segment or a gate and is stored in the DPL field of the segment or gate descriptor for the segment or gate.

Requested Privilege Level (RPL)

The RPL is the override privilege level that is stored in bits 0 and 1 of segment selector. The processor checks the RPL along with the CPL to determine if access to a segment is granted.

Virtualizing the x86 architecture requires placing a virtualization layer under the operating system kernel, which resides in the innermost ring 0, to create and manage the virtual machines that deliver multiplexed resources. There are also some sensitive instructions which cannot be effectively virtualized as they have different semantics when they are executed outside of ring 0. The tapping of these instructions are required to be done in run time.

Full Virtualization Using Binary Translation

This approach involves in translating the non-virtualizable part of the kernel code into a newer set of instructions, which have a similar effect on the virtualized platform. Each virtual machine is supplied with all the services of a physical system including the BIOS, virtualized memory, virtualized block storages. The guest OS is completely abstracted from the physical hardware by the hypervisor. This means that the guest OS does not require any modifications to run. It is not even aware that it is being run in a virtualized environment. System level calls are translated on the fly by the hypervisor.

OS Assisted Virtualization or Paravirtualization

'Para' in Greek means along with. This refers to 'alongside' communication of the guest OS to the hardware. This type of virtualization involves in modifying the guest OS kernel to replace the non-virtualizable pieces of code with hyper calls that can directly interact with the virtualization layer without the need of binary level translation. These calls can include critical kernel operations like memory management, interrupts scheduling algorithms.

This is done to improve efficiency and performance. This results in lower virtualization overhead but drastically reduces compatibility. Modifying the kernel means operating systems like Win XP, cannot run off the shelf.

Hardware Assisted Virtualization

If the hardware is modified to incorporate virtualization, this can eliminate the use of binary translation and paravirtualization. The Intel Virtualization Technology (VT-x) and AMD's AMD-V does exactly that. The both target privileged instruction with a new

CPU execution mode that allows virtual machine monitors to run in a new root mode below ring 0.

Type II Virtualization

This includes emulating one operating system upon another. In this type of virtualization, which is mostly used in desktop virtualization, the hypervisor emulates a complete set of hardware on top of which a guest operating system runs. They can run without VT support by replacing sensitive instruction by emulation.

Memory Virtualization

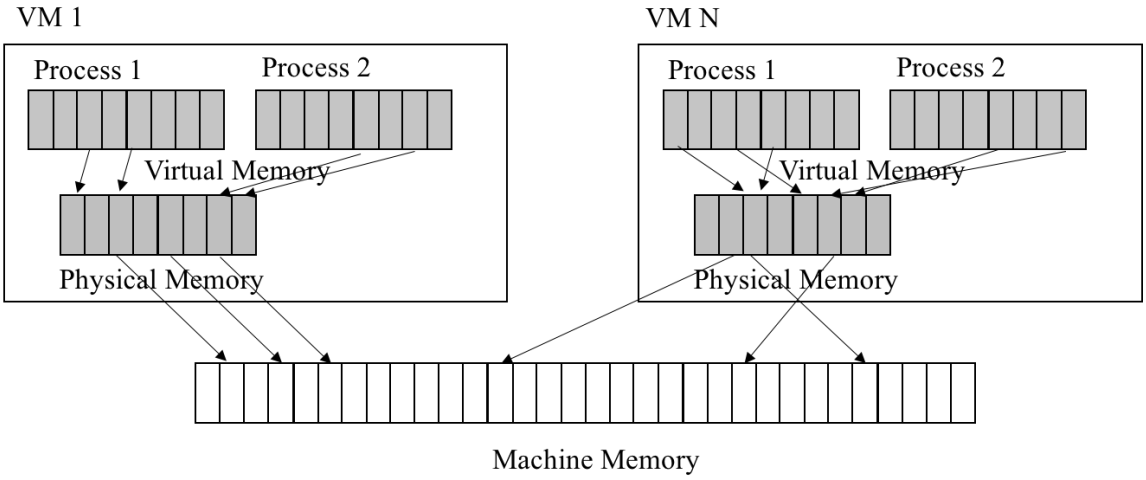

Figure 2.3 Memory Virtualization.

The next important component that is needed for a virtual system is memory virtualization. This involves in sharing the physical memory of the host system and making it dynamically available to the guest operating system. Applications see a contiguous address space that they can reserve for them. Apparently, these address space may not directly be tied to the underlying physical system. The operating system makes use of a Memory Management Unit called the MMU and a translation lookaside buffer (TLB) to optimize virtual memory performance.

To support multiple guest operating system, the MMU also needs to be virtualized. The guest OS controls the mapping of physical to virtual addresses, but in reality, the guest OS cannot have direct physical access to the host's memory.

This virtualization is done by the VMM which maintains a shadow page for the mapping.

CHAPTER III

CONTAINERIZATION

In chapter one we learned the advantages of using virtual machines; isolation and resource sharing among users. Virtual Machines, however, inflict overhead to the performance of the entire system. Each VM needs to run its OS. Each guest OS abstracts Compute, Storage, and Network Operations, and this is done by translating and tapping guest OS Instruction set architecture to host OS ISA and sometimes completely simulating the hardware environment. This increases the resource consumption and the overall efficiency of the system decreases.

The trivial question arises; can we do better? What if we eliminate the hypervisor altogether and somehow chunk out isolated spaces in the operating system that share the same kernel?

3.1 Containers

Containers are lightweight operating systems that run on the host system and execute instructions native to the core, eliminating the need of instruction set translation or emulating the hardware. Containers thus provide the same level of isolation and allow resource sharing without the overhead of expensive hypervisors. Linux Containers are implemented through system level virtualization.

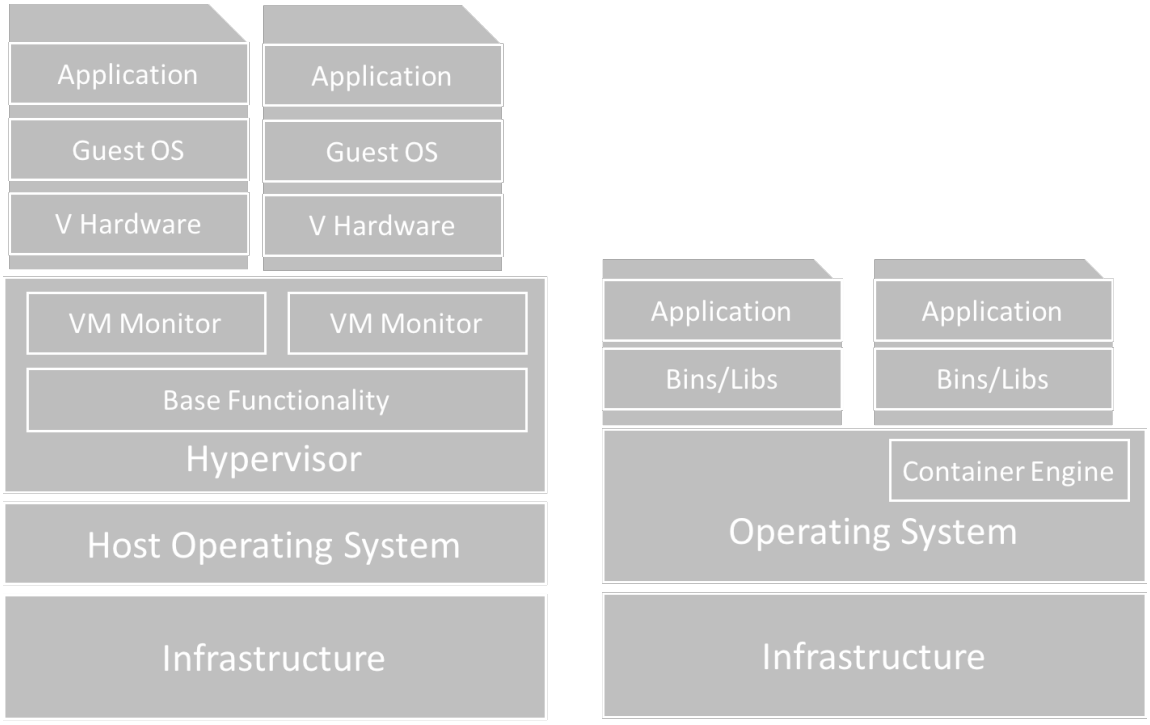

Figure 3.1 VM vs Container Virtualization.

Applications running inside the containers share the kernel of the host OS.

On a broader view, containers provide the following advantages [4] over VM.

1. Since containers do not make use of hypervisors and run directly on the host OS, they can provide near bare metal performances for the target application. On the other hand, VMs can never reach bare metal performance because of the translations, tapings and various other overheads of the hypervisor layer.

2. Containers boot up in seconds. My test results show almost 12 times speed up in booting a container with the same application in contrast to the Linux hypervisor KVM. This is because VMs first need to go through the processes of loading the entire operating system before running the application.

3. Containers require fewer running applications inside them. Memory management is efficient. A higher number of containers can be deployed on the same machine. Virtualization density is way higher.

4. Containers are highly portable. Live migration of containers is possible. A container running in one system can be transferred to another system without any downtime. High portability also makes it easy and readily available to continuous integration and continuous delivery pipelines.

5. They provide greater visibility to the behavior of individual application since the security of the kernel module is maintained at the host level.

3.2 Types of Containers

Containers by nature can be divided into two categories:

I. Application Containers

II. System Containers

Application containers are stripped down operating systems that has enough information to run a specific type of application. These are very light weight and are beneficial with microservices architecture. It allows creations of a container for each application component providing greater control over management, security and policy restrictions. They are very easy to ship, and the application placed inside the container has significantly fewer risks in terms of reliability, consistency, and compatibility.

Docker is one of the most widely used container service provider that has its own management APIs for dockerization.

The second type of containers and the one we will be looking in depth are system containers. They play a similar role to virtual machines. These are not application specific and allow to install different libraries, languages, and applications inside them. Unlike application containers these allow multiple processes to be run at the same time.

System Containers

System containers provide virtualization in a way such that the applications running inside them perceive the look and feel of a private operating system. They simulate the file systems, networking, and even the root user. This is achieved by using advanced Linux features like namespaces, control groups, chroot, and Linux security models (LSM).

For example, a container has its private users file at the standard location /etc/passwd, where that file's real path, at the host system is at /var/container-technology/$container1/etc/passwd. /var/container-technology/$container1 is mounted as / inside the container. Similarly container 2 will have its private users file at /etc/passwd with a real path at /var/container-technology/$container2/etc/passwd. This allows each container to have their own file systems. The root user inside the container has an UID of 0, which again is mapped to a non-privileged account at host OS, say UID XX.

Let us look in detail into each of the kernel features that allow containers to be isolated.

i. CGroups

Control Groups [5] or shortly known as cgroups allows to allocate resources like CPU time, system memory, network bandwidth to user-defined groups and their future children. These groups have specialized behavior and are associated with a set of parameters for

one or more subsystems. A subsystem is a resource controller that partitions resources and applies per cgroup limits. These cgroups can be hierarchical and are arranged in a tree-like structure. Each task in the system belongs to exactly one cgroup in the hierarchy with a set of subsystems, which are associated with some specific tasks. Each cgroup has its own virtual file system.

These cgroups are managed by root users who can create, destroy and query using the cgroup name about the processes attached to it and its hierarchy.

The main functions of cgroups are:

1. *Access*: - Attaching devices to cgroups.
2. *Resource Allocation*: - Limiting memory, CPU, device accessibility, block I/O.
3. *Prioritization*: - Setting priority of processes; deciding who gets more CPU
4. *Accounting*: - Resource usage per cgroup
5. *Control*: - Freezing and checkpoint
6. *Injection*: - Network Packet tagging

cgroup functionality are exposed as 'resource controllers' also known as subsystems, which are mounted on the filesystem. Top-level subsystem mount is the root cgroup and directories under this top level mount represent root mounts per cgroup.

Some of the cgroup subsystem include:

1. *blkio*: Limits IO access to and from block devices.
2. *CPU*: Scheduler for tasks in cgroups
3. *cpuacct*: Reporting subsystem for CPU resources.
4. *cpuset*: Assigns individual CPUs and memory.

5. *devices*: Manages device access per cgroup.
6. *freezer*: This subsystem suspends/resumes tasks.
7. *memory*: This is used to provide memory caps and restrictions.
8. *net_cls*: It tags network packets with class ids that allows Linux TC to identify packets.
9. *net_prio*: Deals with prioritizing network bandwidth per cgroup
10. *ns*: Namespace subsystem.
11. *perf_event*: It is used to evaluate performance.

Figure 3.2 Linux Cgroup FS Layout.

Control groups are created per container in each cgroup subsystem to limit access and provide priority. An example [6] of a container using cgroups is demonstrated in figure 3.3.

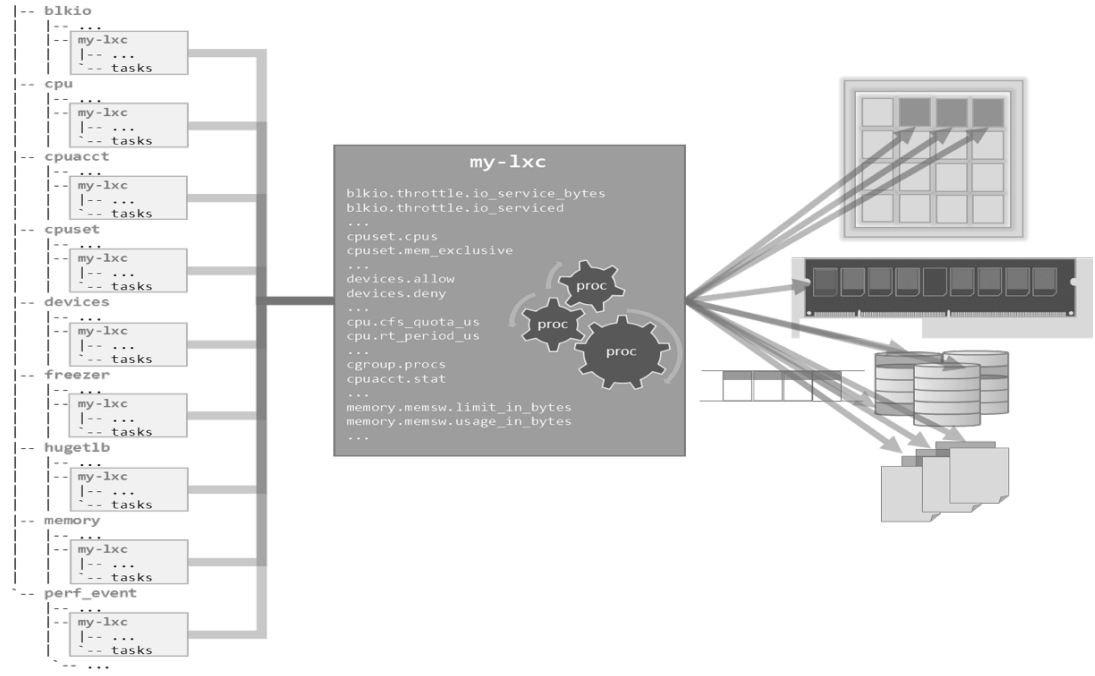

Figure 3.3 Container Cgroup Realization [6].

For each container a cgroup subsystem is created that provides a virtual FS, IO Access, Network access, CPU time and memory to the applications that reside inside the cgroup.

ii. Namespaces

The most important technology for container isolation is namespaces. [7] Historically, the Linux kernel had maintained a single process tree which contained references to all the processes running on the system in a parent-child manner. However, with the addition of the namespace concept, it became possible to have nested process trees. Each process tree can have an entirely isolated set of process. The Linux kernel guarantees that processes belonging to one namespace cannot interfere with the process of another

namespace. This can be visualized as a variable declared in namespace x in a C++ program has no visibility in namespace y.

When a Linux system starts up, it has one parent process with PID 1. This process is the root of the tree, and then it further spawns other processes by starting up daemonized services. Down the line, further services have their parent as PID 1. With namespaces, the system makes a child subtree to be its own root. On this child subtree, the processes that are spawned have an illusion that they have been spawned by root(PID 1), but in reality, they are present in an isolated subtree system. This processes in this child namespace have no way of knowing of the grand parent's existence.

Nesting of namespaces is possible and associating more than one PID to a

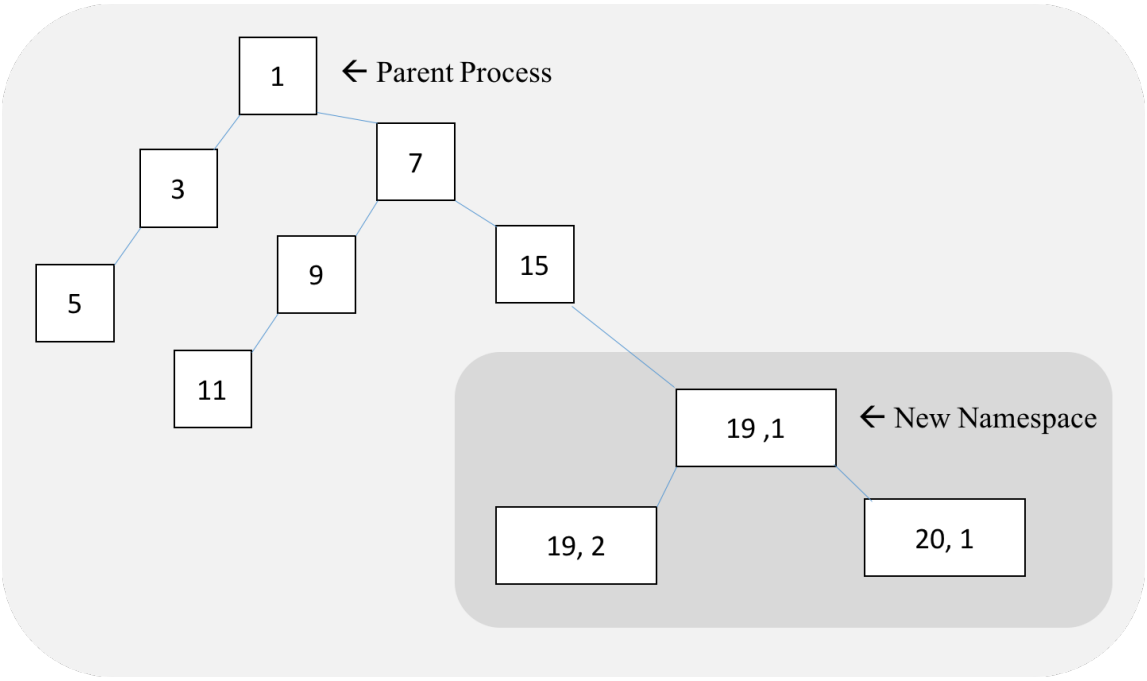

Figure 3.4 Container Namespace Tree.

namespace is also possible. Processes in namespaces have an illusion that they are the only process on the system.

The functionality [8] of namespaces is to provide process level isolation of global resources like: -

- PID (process)
- MNT (mount points, filesystems)
- IPC (System V IPC resources)
- NET (NICs, routing)
- USER (UID + GID)
- UTS (host and domain name)

The MNT namespace isolates the mount table. These include /, /proc, /mnt/ etc. This is typically used with chroot or pivot_root to effectively isolate root FS. The UTS namespace provides per namespace Hostname and NIS domain name. These allow containers to have their fully qualified domain name (FQDN).

The PID namespace is used for PID Mapping. As discussed earlier, the parent process can make a child process its own root, in other words the child process is assigned with a PID of 1. However, the parent process has a different PID associated with the child process (which is only visible to the parent process). This mapping of IDs is maintained by the PID namespace.

The IPC name space is used for interprocess communication and provides isolation for Semaphores, Shared Memory, Message Queues, Signals, Memory polling, Sockets, Pipes and file descriptors.

The NET namespace creates per name space network objects such as Network Devices, Bridges, Routing Tables, IP Addresses, Ports. The USER namespace is the most

security critical namespace and provides mapping of GID/UID outside the container to UID/GID inside the container. A root inside the container will never be root outside it.

An example of creating a namespace:

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
static char child_stack[1048576];
static int child_fn() {
  printf("PID: %ld\n", (long)getpid());
  return 0;
}
int main() {
  pid_t child_pid = clone(child_fn, child_stack+100011, CLONE_NEWPID | SIGCHLD,
NULL);
  printf("clone() = %ld\n", (long)child_pid);
  waitpid(child_pid, NULL, 0);
  return 0;
}
```

iii. LINUX CHROOT and L.S.M.

CHROOT changes the root directory of the calling process to that is specified in the path. The root directory is inherited by all the children of the parent process. This again provides isolation for containers. The process running in this environment cannot access files outside the jailed environment. In containers, this is used to mount the root FS system, launch and clone init process in a new MNT namespace.

Linux implements around 300 system calls, some of which should only be accessed by the host's privileged users. For example, say modifying the system clock and date time or changing the nice values of processes. With multiple containers running, this can be disastrous if not prevented. This is achieved by modules 'Linux Capabilities', 'seccomp', and 'Linux Security Module'. The LSM comprises of AppArmour or SELinux. They attach a label to each file and allow the process to access these file only if they are associated with the tagged label. If by any means a container "jail breaks" then the LSM denies escalated calls to the host system.

3.3 Performance Evaluation of Containers

This work will be contributing towards development of container management in cloud using OpenStack and Chef. We will be forking the LXD repository which is backed by canonical, the parent company of Ubuntu.

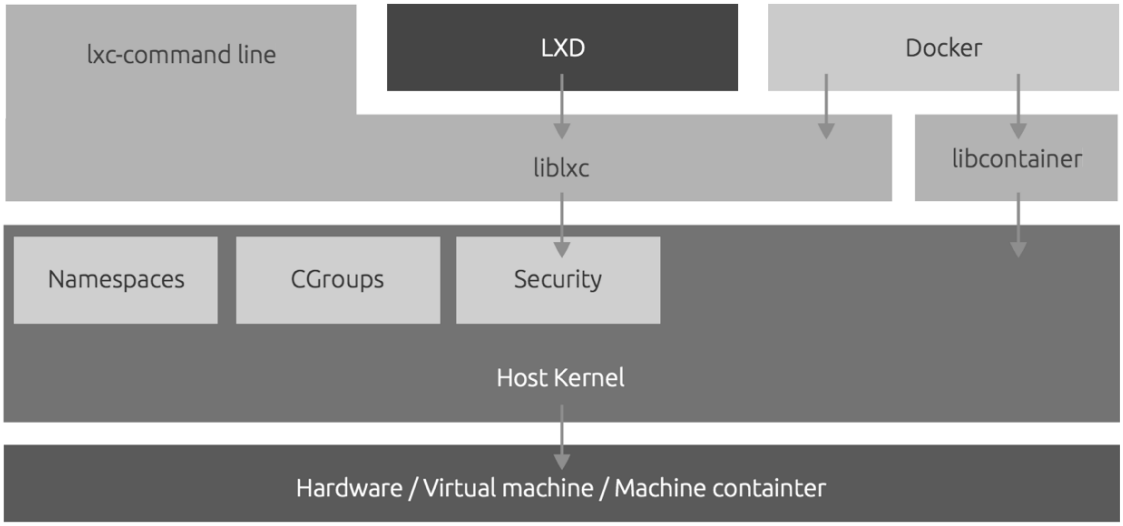

Figure 3.5 LXD Technology Stack [16].

LXD is an open source tool that provides a means to manage machine containers. These act more like virtual machines on traditional hypervisor than application container like Docker. Canonical terms this as 'pure container hypervisor'. LXD uses LXC at its backend and provides APIs to create, delete, move, clone and to perform administrative operations.

Here, in the scope of this chapter, we will examine the performance of one instance of LXD vs. VM.

Boot Time (VS Virtual Machine)

In this test [9] we create a container using LXD and a virtual machine using KVM. We measure the performance by recording the time taken to boot and reboot five times. The container takes around 46 seconds for the entire sequence, in contrast the virtual machine instance takes around 651 seconds. This behavior is expected. As we discussed earlier, virtual machines need to load the entire OS including the kernel before loading the desired applications.

Block IO (VS Bare Metal)

We claimed earlier that, container's performance is comparable to bare metal systems. In this test we perform IO operation for both the scenarios. The performance is almost equivalent.

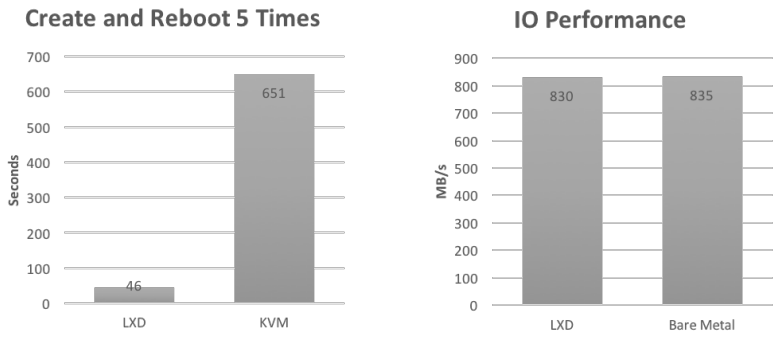

Figure 3.6 Performance Comparison.

CHAPTER IV

ORCHESTRATION MANAGEMENT USING OPENSTACK

4.1 Introduction

In this chapter, we will be dealing with methodologies for bootstrapping and managing an infrastructure of containers. In other words, it is called 'orchestration management.' We will be looking at the widely used framework 'OpenStack' for the management of virtual machines and will be reprogramming it to work with LXD Containers.

OpenStack [10] is an open source software platform for cloud computing, mostly deployed as infrastructure-as-a-service (IaaS). This framework has various components to control a diverse multivendor pool of resources like processing power, networking, block storage, image storage, etc. OpenStack also incorporates a web-based graphical user interface for easier management. This GUI dashboard, in the backend, uses RESTful APIs to call the core services.

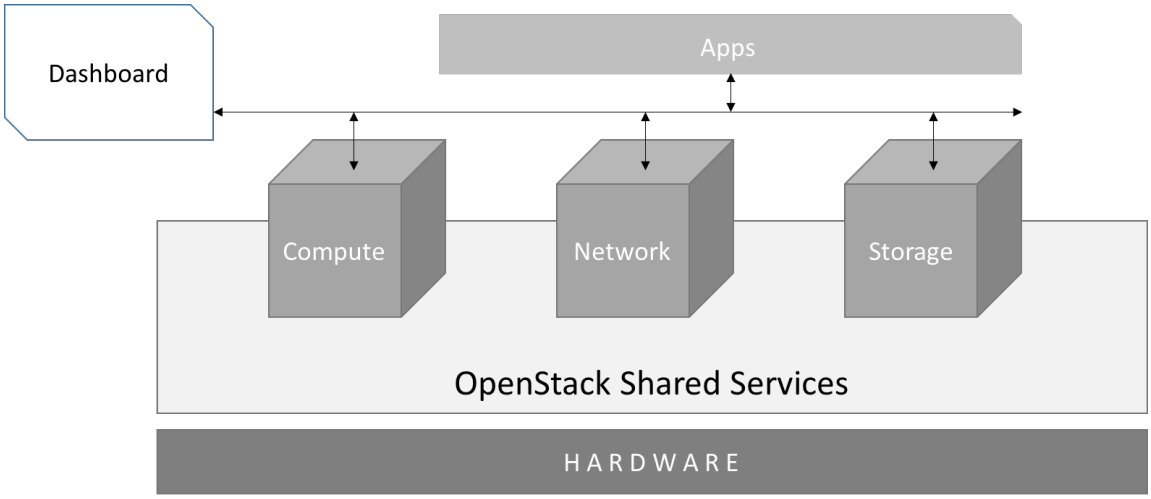

Figure 4.1 OpenStack Framework.

4.2 OpenStack Architecture

All components of OpenStack are modular that provide a set of core services that run independently. This facilitates elasticity and scalability.

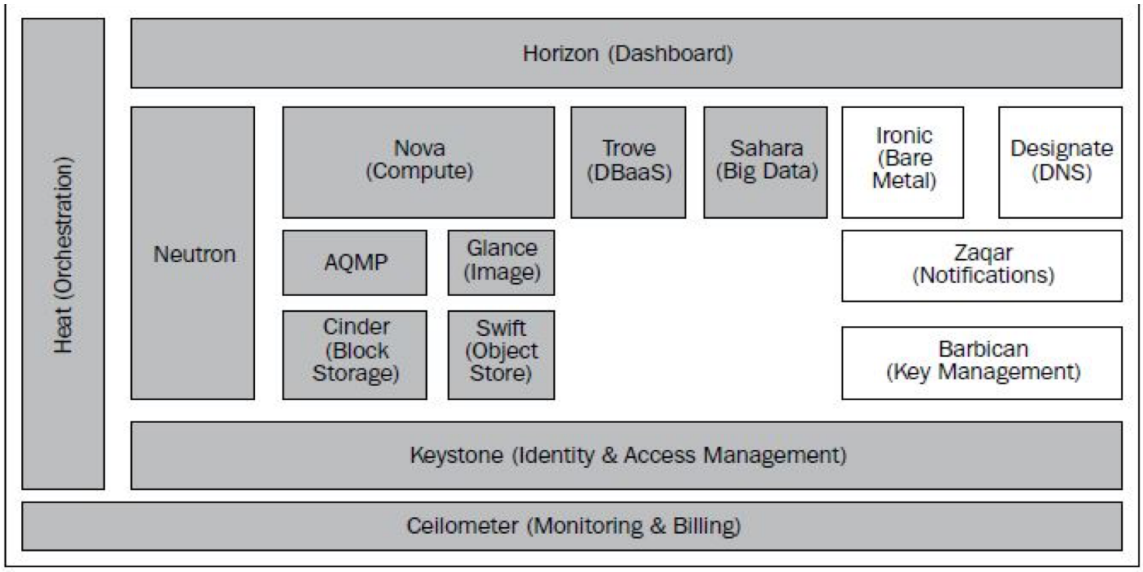

Figure 4.2 OpenStack Architecture [12].

The core modules can be categorized into: -

1. Compute Service (Nova)
2. Networking Service (Neutron)
3. Object Storage Service (Swift)
4. Dashboard (Horizon)
5. Identity Service (Keystone)
6. Block Storage Service (Cinder)
7. Image Service (Glance)
8. Orchestration Service (Heat)
9. Telemetry Service (Ceilometer)
10. Database Service (Strove)

Figure 4.3 expands on the preceding block diagram and depicts the different relationships amongst the different services.

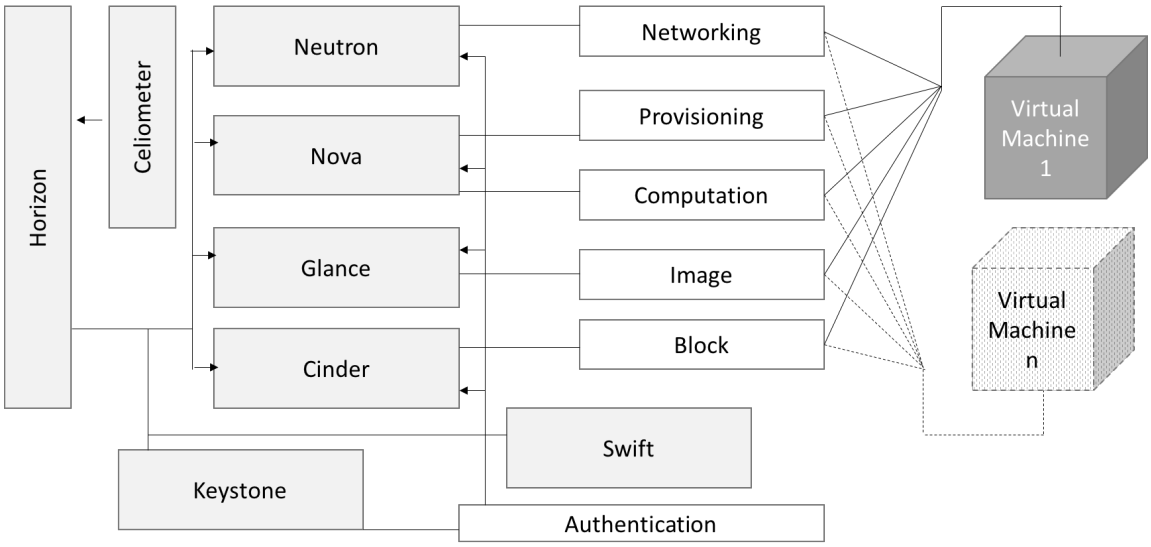

Figure 4.3 Service Relationship.

Keystone

OpenStack Identity (Keystone) provides an authentication system consisting of users mapped to the OpenStack services they are authorized to access. It also provides identity and access management for all the components of OpenStack. It has internal services such as identity, resource, assignment, token, catalog, and policy, which are exposed as an HTTP frontend. Keystone provides CAS for various operating system that can involve pluggable authentication module and active directory services. It supports a number of authentication mechanisms like clear text password, key, and tokens. Services that are available can be queried programmatically. This service is equivalent to the IAM service of the AWS public cloud.

Horizon

Horizon is the frontend of the stack and provides dashboard services to administrators and users. It has a graphical user interface hosted as a web page. It connects with the CLI APIs to perform the backend tasks. VM's, Networks, Subnets, Storage, can be managed through the Horizon interface. It also provides plugins for 3rd party apps. Most of the automation of building the infrastructure is done through the horizon. It can be branded as per the requirements of commercial vendors and service providers. This works on top of all other services. All functionality it provides can also be directly accessed by using the REST APIs. It can be compared to the AWS console used to create and configure the services.

Nova

Nova provides computing resources. It is programmed in a way to provide an automated pool of machines that can be used for computation. It is designed to work with most of the virtualization protocols using hypervisors like KVM, VMWare, Oracle, and Hyper-V. This can also be used to work with leading container technologies like docker, LXC and LXD. The main job of Nova is to load balance jobs that are assigned to it in the resource pool. It can be considered equivalent to the EC2 service of AWS.

Swift

Swift is an easily scalable and redundant storage system that is primarily used to store and retrieve Binary Large Object (BLOBs). Multiple disk drives that spread throughout servers in the data center are used to write objects and files. The OpenStack software is responsible for ensuring that data is not replicated and integrity is maintained

throughout the cluster. Storage clusters simply add new servers and scale horizontally. In the case of a server or hard drive failure, OpenStack duplicates the content from other active nodes to a new location in the cluster. Inexpensive commodity hard drives and servers can be utilized as OpenStack uses software logic to perform data replication and ensure distribution across different devices. It provides a variety of sub-services such as a ring, container server, updater, and auditors that have a proxy server as the frontend. The swift service is used predominantly to store Glance images. It is comparable to the EC2 AMIs that are stored in an S3 bucket where Swift is equivalent to AWS' S3 storage service.

Glance

Glance provides image services. It holds the OS images that are distributed across the stack. It interacts with REST APIs streaming images. It is aso used to generate backups. A number of OpenStack modules interact with Glance to exchange meta-information.

Cinder

This is one of the most important components of the stack. It provides block level storage and serves them as devices to different components in the system. These block level storage appear as standard disks when attached and can be formatted into any file system depending on the requirement of that particular system. Creating, attaching, detaching and destroying these devices are managed by Cinder. These actions are completely merged into OpenStack Compute and the dashboard Horizon, making it convenient for users to deal with storage and allowing them access to raw block level storage. It provides snapshots for data backup of block level storage. These snapshots can

be restored or utilized to create a new block storage volume. The subsystems include a volume manager, an SQL database, an authentication manager, etc. The client uses an AMQP such as Rabbitmq to provide its services to Nova. This service provides similar features to the EBS service of AWS.

Neutron

Neutron, previously known as Quantum, is a system for the management of networks and IP addresses. OpenStack Networking confirms that the network does not act as congestion or a limiting factor in cloud deployment. It also ensures that users receive self-service ability, even when it comes to network configurations.

Different applications or user groups are provided with networking models by OpenStack Networking based on their needs. Models can be used to separate the traffic flow from servers using VLANS. Neutron manages the IP's featuring both static and dynamic configuration. Maintenance mode is built into it so as it can redirect traffic.

It also supports modern networking protocols like SDNs (Software designed Networks) to provide high availability.

It provides several other functionalities such as Load Balancer as a Service and Firewall as a Service. As it is an optional service, we can choose not to use this, as basic networking is already built into Nova. Also, Nova networking is being phased out.

The system, when configured, can be used to create multi-tiered isolated networks. An example of this could be a full three-tiered network stack for an application that needs it. It is equivalent to multiple services in AWS such as ELB, Elastic IP, and VPC.

Heat

Heat defines standardized templates to manage cloud applications. It makes use of OpenStack-native REST API and a CloudFormation-compatible Query API. It means that you can script the different components that are being spun up in order. It is incredibly useful when deploying multicomponent stacks. The system integrates with a majority of the services and makes API calls to create and configure various components.

The template used in Heat, known as the Heat Orchestrator Template (HOT), is a single file which can script multiple actions. For example, we can write a template to create an instance, some floating IPs or security groups, and even create users in Keystone. The equivalent for Heat in AWS is the cloud formation service.

Celiometer

Ceilometer is a service that provides a single point of contact for billing systems. It contains all the counters needed to establish customer billing, over all present and future OpenStack units. The delivery of counters can be traced and audited, the counters must be easily extensible to support new projects and agents, and performing data collections should be independent of the complete system. A variety of subsystems is integrated into the Ceilometer such as polling agent, notification agent, collector, and API. It also facilitates the saving of alarms extracted by a storage abstraction layer to supported databases such as Mongo DB, SQL server or HBase.

Trove

Trove, a database-as-a-service provisioning relational and non-relational database engine, uses Nova to create the computer resource to run DBaaS. It is introduced to the

system as a bundle of integration scripts that run with Nova. The service requires the creation of unique images that can be stored in Glance. It is similar to the RDS service from AWS

Designate

DNS is managed by this multi-tenant service called Designate. It opens up several REST APIs to interact with DNS. It abstracts out the complexity and generalizes commands over a diverse range of DNS providers like PowerDNS and BIND. It supports binding to custom made drivers as well. The main purpose of designated is not to provide a DNS service but rather to provide an interface with existing DNS servers to create zones per tenants. The service has various subsystems such as API, the Central/Core service, the Mini DNS service, and Pool Manager. The Designate service offers DNS services equivalent to Route 53 of the AWS.

4.3 OpenStack Networking

We would be using the OpenStack Networking service to setup and define network connectivity. The networking service consists of the neutron-server, a persistent database, and other plug-in agents. We will be using the Linux bridge mechanism driver and 'veth' pairs as interconnection devices. We will be managing Linux bridges at layer-2 on compute node, and in other nodes, we will use layer-3 protocols to provide routing, DHCP, and DNS. Figure 4.4 describes the layout of the networking using Linux bridge mechanism. It uses two networks, namely ‘Management Network’ and ‘Provider Network’. The compute node and the DHCP agent resides on the same network.

The modular network 2 plugin network allows layer 2 networking technologies that is found in all complex real world data centers. Each network type is managed by a driver that validates the specific information and is responsible for the allocation of free segment in the project networks.

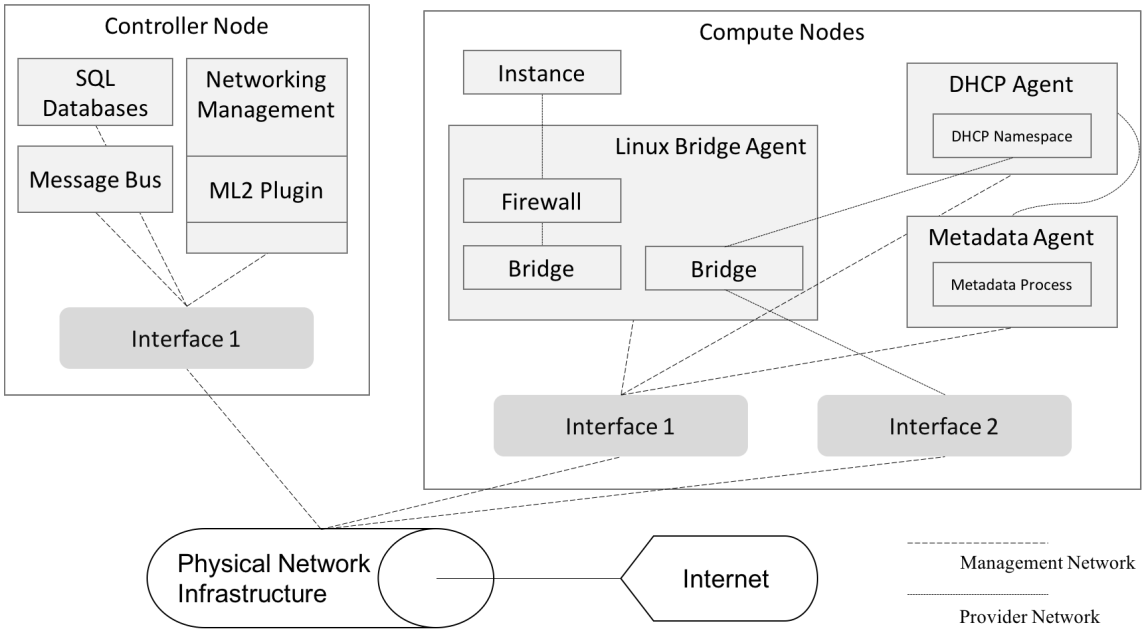

Figure 4.4 Network Layout.

CHAPTER V

CONFIGURATION MANAGEMENT WITH CHEF

5.1 Introduction

Today's IT infrastructure are a collection of complex integrated networks that include a variety of interdependent software and tools. With the agile workflow, starting from the development to deployment requires attention to configuration to keep the applications and hardware running smoothly. At the most intuitive level, configuration problems can lead to catastrophic system failures stopping critical services to work. This can also affect performance levels and degrade the productivity of business. With the systems not receiving proper upgrades and patches, it hides several security breaches keeping it vulnerable to hackers.

With DevOps coming into the picture, it is almost impossible to maintain a continuous integration and continuous delivery pipeline without having a proper configuration management.

On a not so intuitive level, if systems are kept running as is, at some point in time, an organization may not be able to upgrade them without a significant downtime and costing thousands of dollars. It might also cause deployment failures for software that worked in the development environment. As software developers keep their systems updated, critical libraries might be missing or may be incompatible with the older production systems.

5.2 Configuration Management

“Configuration management [11] is the process of creating a logical view and maintaining versions of the IT infrastructure keeping records of all configuration files and environments, so as, at any given time, the entire infrastructure can be spawned from scratch by running a set of codes. It maintains the relationship between the infrastructure's assets and the build”.

What Makes Configuration Management Difficult?

The main challenge arises with integrated systems. As systems get more integrated, more relationships and dependencies develop. A change in configuration in one module can break other modules. Without versioning or having a way to trace back, it creates difficulties in debugging. As the infrastructure gets bigger and systems are replicated, keeping the configuration files in sync is a challenge. With the use of containerized services felicitating continuous integration and delivery, it is vital to closely couple development, staging, and production environments. With configuration management tools, failures in app delivery are significantly less, mostly eliminating human errors.

5.3 IAAS and Chef Framework

The key concept of 'Infrastructure as a code' is that it should be possible to build our infrastructure as if it were code - that is to abstract, design, implement and deploy the IT infrastructure the very same way using tools as we do with any modern software projects. In other words, is to visualize infrastructure as a redeployable code repo that

works with the day to day software development methodologies of writing and developing software.

This approach gives us a number of benefits mainly, : -

1. **Repeatability**

Since we are now building our infrastructure with programming languages, committing our code, we can be confident that our infrastructure will be ordered and repeatable. Given the same input to our infrastructure system, it will generate the same infrastructure output. This means that we can rebuild our entire IT infrastructure executing these set of codes at any given point of time.

2. **Automation**

Since we are abstracting the infrastructure and capturing a meta code that can deploy the infrastructure, we can very well automate it.

3. **Agility**

The commitment to a code repo means we can move the infrastructure state forward or backward with time. In the event of problems, we can look at what has changed and who has changed it. This will bring down the downtime and will help expedient root cause analysis.

4. **Scalability**

With Repeatability and automation, we can bring up similar infrastructure by just executing a set of codes. This will enable us to set up an environment rapidly. For example, if we already have written codes to bring up a web server 1, we can bring up new 'n' web servers by executing the code n times in different hardware or containers.

5. Reassurance

Since the infrastructure is represented in codes written in a high-level programming language, anyone can look at it to get a glance of the system eliminating the dependency on an sys admin who can be a single point of failure.

6. Disaster Recovery

In the event of catastrophic disasters that might potentially wipe out the entire infrastructure, we can bring it back quickly as it has been broken down into modular components that are described as codes.

Chef is an open source framework that works with the concepts of IAAC. It provides a set of libraries and functions to build tools to managing infrastructure. It abstracts the underlying OS and tries to generalize the key concepts. It introduces the term 'convergence' which means with each Chef run, the infrastructure is converged to the targeted state.

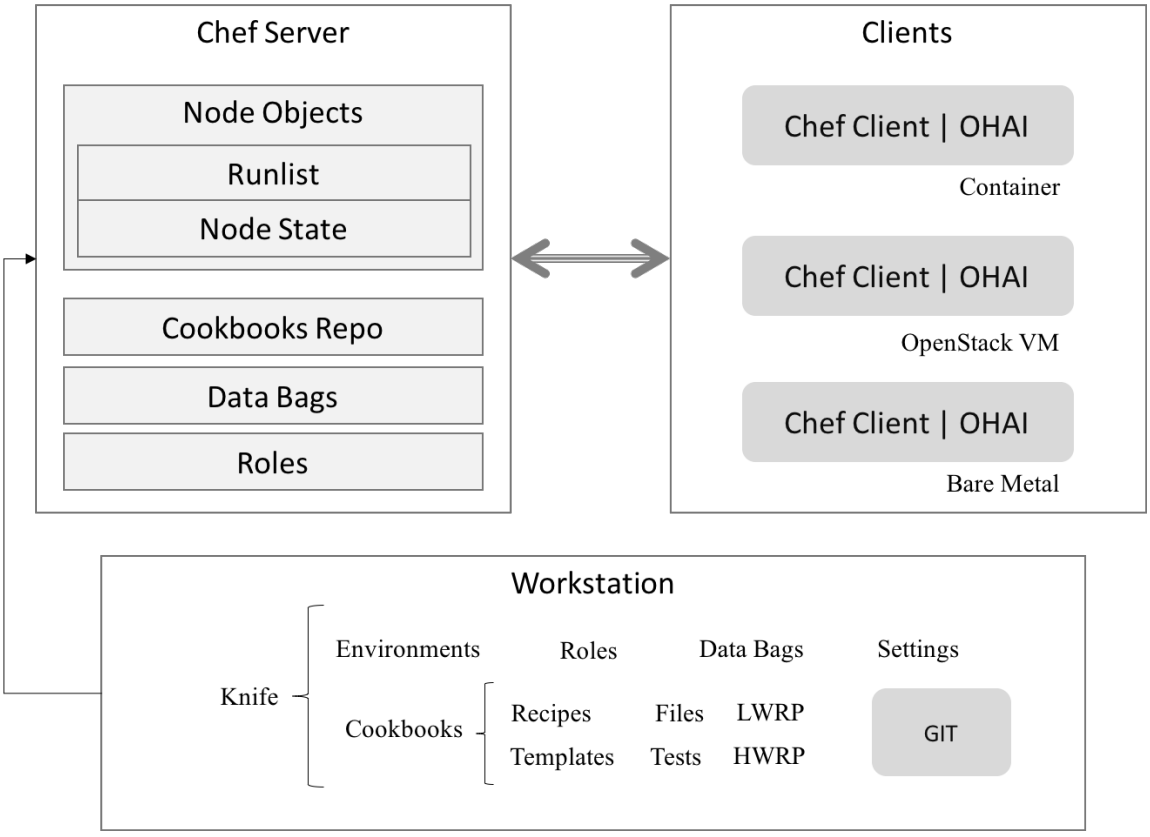

Figure 5.1 Chef Component Relationships.

Figure 5.1 shows the major components of the Chef [12] framework and the relationship that exists between them. Now we will look into each of the component with greater detail.

Chef Server

The Chef server the central element of the Chef framework. It serves as the central repository that holds the node objects, cookbooks, recipes, roles, data bags, and keys. All communications in the framework are routed via the Chef server. It has information of each node that is stored in the form of metadata which are connected to the Chef server. It also provides a web interface for managing the nodes and resources.

Cookbooks

Cookbooks are codes written in Ruby and in Chef DSL which defines the rules, regulations, and formation of an infrastructure. It contains recipes, templates, files, and necessary tests. These cookbooks are downloaded by the Chef-client depending on roles and are used to converge a system to the desired state. Broadly it contains: -

Recipes, that is the specification of a module of the IT infrastructure.

Resources, which are methods, or functions that are cross platform and are reused.

Attributes, these are variables that serve as inputs to the recipes. These are mainly the values of configuration files.

Files and Templates, which serves as configurations of various modules.

Resource providers, which are library functions to enhance the functionality of recipes.

Tests, Unit and smoke tests to check the state of convergence of a run.

Node Objects

Attributes and recipes are important aspects of the Chef client-run. However, the Chef system needs to keep track of data that relates to the node. These data are but not limited to the type of node, its operating system family, the list of users, software installed, etc. These data are stored in the node object. The node object is also responsible for resolving dependencies as more than one cookbook can be in the run-list, and these cookbooks can be dependent on some other cookbooks.

The node object has all these data stored in JSON format. Each time a client-run happens, the Chef-client reads data using the OHAI plugin and updates the node object in the server.

Run List

A run-list is an arranged way of keeping the recipes and roles. This is exactly how the node will converge. It is the specification of the node. The order is strictly maintained by the Chef-client and will never execute a recipe twice. The same run-list can exist for various nodes who share a similar role.

Role

A role defines a node. It is a collection of various paths for a node to achieve its final state. For example, a IOT server will have a role of IOT Server that comprises of cookbooks like, Apache Tomcat, RADIUS, LDAP, and some custom attributes. Whenever a role is made to run on a node, the node object of that node is matched against that role and the run-list is compiled.

Data Bags

Data bags contain variable data which are specific to the deployment machine. Data that are sensitive are generally kept in data bags as they can be encrypted.

Workstation

This is the developer side of the Chef infrastructure. This is where Chef cookbooks are written. It uses Chef development kit and knife to communicate with the Chef server.

The roles of a workstation are:

1. To maintain the cookbook repository and properly version it with version control tools like Git.
2. Develop recipes and cookbook.
3. Interact with nodes and bootstrap a node.

4. Set up roles and environment.

Knife is mainly used to communicate with the Chef server. It provides a command line interface to execute Chef commands.

5.4 Chef Client Run

A Chef-client run is a program that runs on each node to perform all the desired state that are necessary to take the node to the state of convergence. Each client node must have been registered with the Chef server for this to happen. The registration process is done by bootstrapping the client node from the server with proper keys.

RSA public key-pairs are used for the authentication process. Figure 5.2 shows the sequential steps that occur during a Chef-client run.

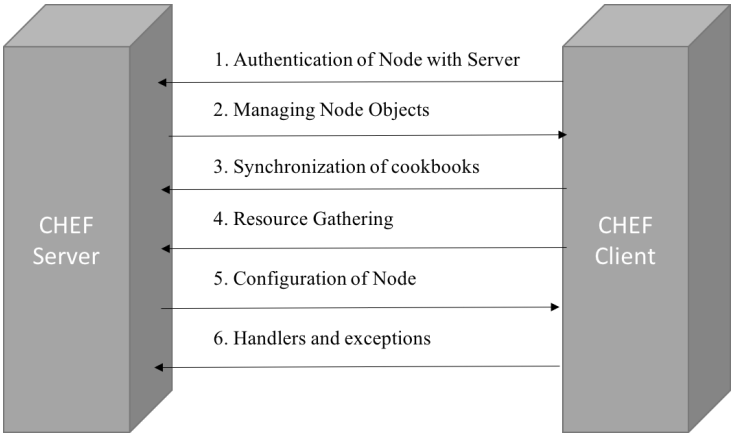

Figure 5.2 Chef Server Client Protocol.

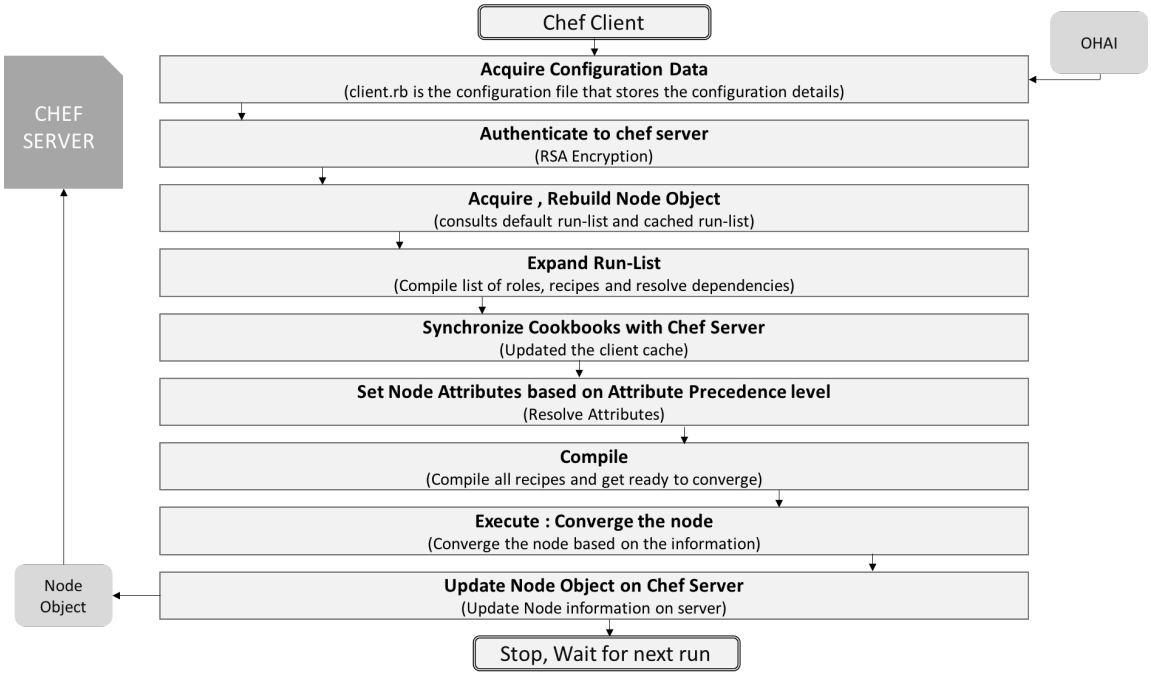

Figure 5.3 Chef Client Steps.

The following shows an example recipe to configure nova, the compute service of OpenStack. (Source is found at my GitHub https://github.com/sidxz/openlab-chef-cookbooks/)

```
# SOURCE: www.github.com/sidxz
# Cookbook Name:: openlab_nova
# Recipe:: configure_compute
#
#Install nova
%w(nova-compute nova-compute-lxd).each do |pkg|
  package pkg do
    action [ :install ]
  end
```

```
end
#Configure nova
template "/etc/nova/nova.conf" do
  source "nova_compute.conf.erb"
  owner 'nova'
  group 'nova'
  mode 0711
  variables :nova_user_pass => node['openlab-compute']['install']['nova-user-pass'],
:rabbit_pass => node['com_rabbitmq']['rabbit_pass'], :neutron_user_pass =>
node['openlab-compute']['install']['neutron-user-pass']
  notifies :restart, 'service[nova-compute]', :delayed
end
#SERVICES
service "nova-compute" do
  action :nothing
end#
```

5.5 Continuous Integration and Continuous Delivery

The CI/CD [13] pipeline is one of the important aspects of the thesis, that in further chapters (chapter 6), we will see, how it sparks the idea of creating an app streaming service for Internet of Things. Knowing the concepts of Chef, let us now see how the Chef framework supports CI/CD at its core. But first, let us understand what CI/CD is.

Continuous Integration

Continuous integration is a practice of frequently integrating and testing newer codes as they are being developed. All developers in the team commit the changes to their branch of the Git code repo. As soon as the commit is pushed a set of CI tools builds the temporarily merged code base and runs an automated test suite. When a change in code causes a test to fail or breaks the system in the test environment, it is very clear which set of change caused it. Feedback is tremendously fast. The smaller the commits the faster problems are surfaced and fixed. To keep the CI pipeline effective, all failures are addressed immediately.

Continuous Delivery

CI addresses the challenges for work done in a single code base. Various developers work on the same code base simultaneously and push them to the source so that issues are addressed immediately. Continuous Delivery expands this principle to the entire infrastructure. The aim of CD is to ensure that all the pushed code are continuously validated against the entire system and can be termed production ready.

Without CD, people work independently with the approach that if the pieces work in isolation, there should be no problem in putting them together. This idea has been proven wrong and wrong again.

It is very critical that, the deployable isolated components are continuously pushed and validated with the entire system. The frequent tests build up confidence as we test them in different stages in the pipeline. If the changes are about to break production it will surely break the deployment pipeline to production first.

Chef CI/CD Pipeline

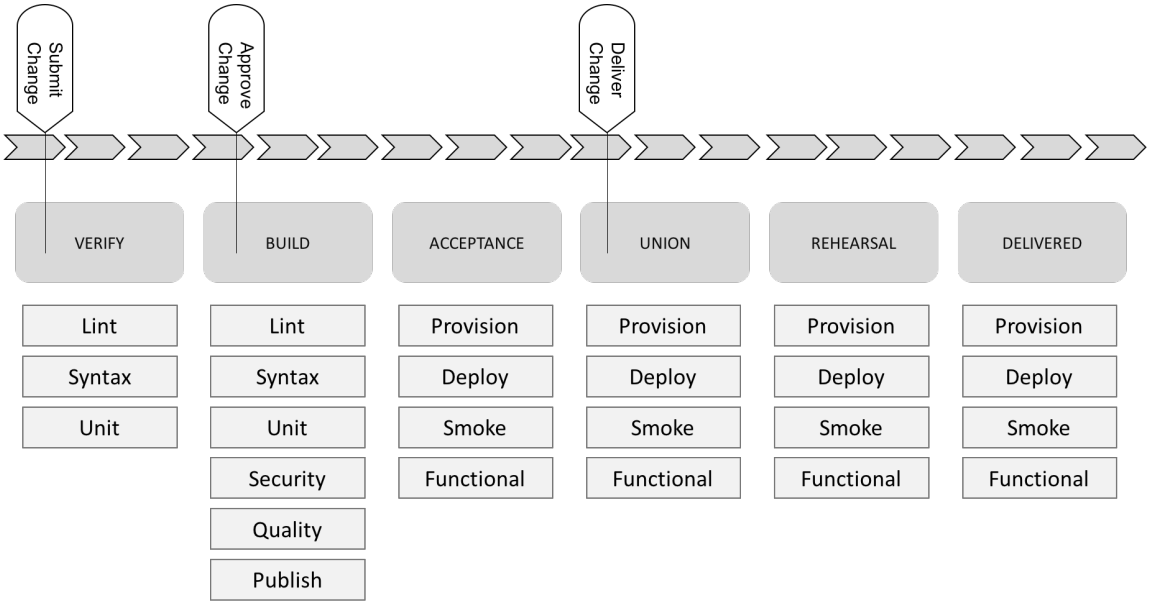

Figure 5.4 Chef CI/CD Pipeline.

The Chef's CI/CD pipeline gives the team a common platform to develop, build, test, and deploy cookbooks and applications to the infrastructure. It enables multiple teams to work simultaneously on systems made up of multiple modules which are interdependent on each other. The pipeline conducts a series of automated and manual test gates that flows the software changes from development to delivery. Pipelines have six stages: Verify, Build, Acceptance, Union, Rehearsal, and Delivered. Changes flow from one stage to another passing a set of validation tests. These tests include lint, syntax, unit, security, quality, smoke and functional tests. As the changed code flows along the pipeline passing these tests confidence is build up till it reached the final stage 'delivered,' where it is production ready.

CHAPTER VI

INSPEC HANDLER AND BUILDING THE OPEN LAB STACK

6.1 What Have We Seen So Far?

In the previous chapters, we had been looking into fragmented parts of relatively large systems and architectures. In the first and second chapter, we discussed virtualization and the benefits of containerization over virtualization. It formed the core components of our bigger picture. In the third chapter, we saw how to bootstrap these containers and manage the orchestration processes using OpenStack. This forms our bare infrastructure. Now we have a virtual data center with containers installed with operating system images ready to be configured. This configuration management is done with Chef that we read about in chapter 4. Now we will look into integrating all of them to form a lightweight data center. Then we will validate our infrastructure using InSpec handler.

6.2 Implementation

We use an Intel Core i7 processor with 16 gigabytes of memory. The host OS runs Ubuntu 14.04 LTS. Following is the complete hardware details:

Architecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 8

On-line CPU(s) list: 0-7

Thread(s) per core: 2

Core(s) per socket: 4

Socket(s): 1

NUMA node(s): 1

Vendor ID: GenuineIntel

CPU family: 6

Model: 60

Model name: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz

Stepping: 3

CPU MHz: 3509.171

CPU max MHz: 3900.0000

CPU min MHz: 800.0000

BogoMIPS: 6784.72

Virtualization: VT-x

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 8192K

NUMA node0 CPU(s): 0-7

The hard drive is partitioned into

sda

sda1

├─sda2 488M 0 part /boot

└─sda3 1.0T 0 part

├─ubuntu--vg-root 252:0 0 1.0T 0 lvm /

└─ubuntu--vg-swap_1 252:1 0 15.8G 0 lvm [SWAP]

sdb 1.0T

'sda' holds the necessary mounts for the host OS to work. 'sdb' is used to create a ZFS Pool.

ZFS provides integrated storage by combining the file system and the logical volume manager. File systems were earlier constructed on top of a single physical device. In a bid to address multiple devices and as to provide for data redundancy, the concept of a volume manager was introduced. It would provide a representation of a single device, making sure that file systems would not require alteration to take advantage of multiple devices. The design added another layer of complexity and ultimately prevented certain file system advances. It was because the file system could not control the physical placement of data on virtualized volumes. ZFS does away with volume management completely. Instead of creating virtualized volumes, ZFS allows one to combine devices into a storage pool. Storage space is allocated from a shared pool of physical storage devices. All file systems receive adequate space, and it can also be increased by only adding a new storage device to the pool.

We create the ZFS pool by using

zpool create container-pool /dev/sdb

The next step is installing LXD on top of the host OS. This is achieved by pulling it from the apt repo.

apt-get install lxd

We configure LXD to create a network bridge named 'lxdbr0' and use NAT forwarding via IPV4. To deploy OpenStack we will be using four containers namely

i) openlab-compute

ii) openlab-controller

iii) openlab-network

iv) openlab-storage

All of them use ubuntu 14.04 LTS as guest OS. They are created using the LXD Command

openlab-create {openlab-compute, openlab-controller, openlab-network, openlab-storage}

openlab-create is a shell script that is used to create containers. Source of this script can be found at https://github.com/sidxz/LinuxTools.

The /etc/hosts of each container is modified so that domain name of all the containers can be translated to respective IPs and be reachable from one another.

OpenStack can be installed by manually installing each of the required services, however our aim is to create an automated platform to deploy OpenStack. This will remove all the complexities and will install OpenStack in minutes. We create Chef Cookbooks for each of the OpenStack Services. The basic template of the Chef cookbook can be created using

chef generate cookbook <cookbook name>

where cookbook name = { *com-mariadb, com_rabitmq, openlab-chef-client, openlab-utils, openlab_compute, openlab_dashboard, openlab_global, openlab_identity, openlab_image, openlab_network* }

Each of these cookbook contains recipe to automate the installation of their respective service. The openlab_global cookbook stores attributes that are used in more than one cookbook. The source code for all these cookbooks can be found at https://github.com/sidxz/openlab-chef-cookbooks. Industry standard pratices have been followed while creating these cookbooks. All configuration files have been parsed into templates. Values of these configurations can be set using the Chef attributes file. An example recipe of cookbook com-mariadb that performs a secure installation of mariadb is as follows:

```
package "mariadb-server" do
  action [ :install ]
end
package "python-pymysql" do
  action [ :install ]
end
template "/etc/mysql/mariadb.conf.d/99-openstack.cnf" do
  source "99-openstack.cnf.erb"
  owner 'root'
  group 'root'
```

```
  mode 0600
  variables :bind_address => node['com-mariadb']['mysql-bind-address']
  notifies :restart, 'service[mysql]', :delayed
end
cookbook_file '/etc/mysql/mariadb.conf.d/50-server.cnf' do
  source '50-server.cnf'
  mode '0755'
  owner 'root'
  group 'root'
  notifies :restart, 'service[mysql]', :delayed
end
template "/tmp/mysql_secure_installation_silent.sh" do
  source "mysql_secure_installation_silent.sh.erb"
  owner 'root'
  group 'root'
  mode 0700
  variables :mysql_password => node['com-mariadb']['mysql-password']
end
directory '/root/.chefvars' do
  owner 'root'
  group 'root'
  mode '0700'
```

```
  action :create
end
execute 'mysql_secure_installation_silent' do
  command '/tmp/mysql_secure_installation_silent.sh && touch
/root/.chefvars/mysql_secure_installation_silent.bool'
  not_if {::File.exist?("/root/.chefvars/mysql_secure_installation_silent.bool")}
end
service "mysql" do
  action :nothing
end
```

The next step is to upload these cookbooks to the Chef Server.

berks install && berks upload

This is done by running the above command inside each cookbook.

The containers are ready to be bootstrapped with Chef. This would perform the automated installation. To bootstrap we again use a script

openlab-bootstrap <ip/fqdn> <node-name> <roles/cookbooks>

This takes a while. After the bootstrap process completes we can see these nodes listed in the Chef server.

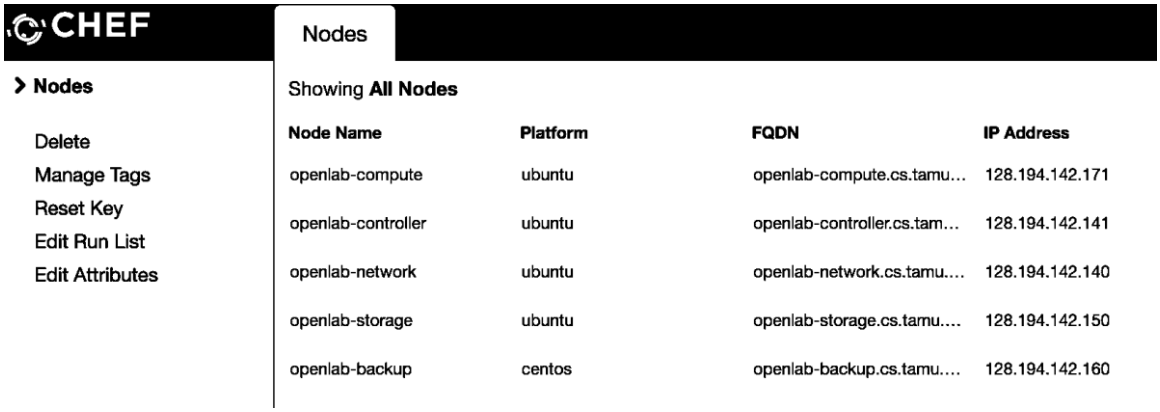

Figure 6.1 Chef Dashboard.

This completes the installation process. Since we have created the Chef Cookbooks, subsequent installations can reuse these cookbooks by changing the required attributes.

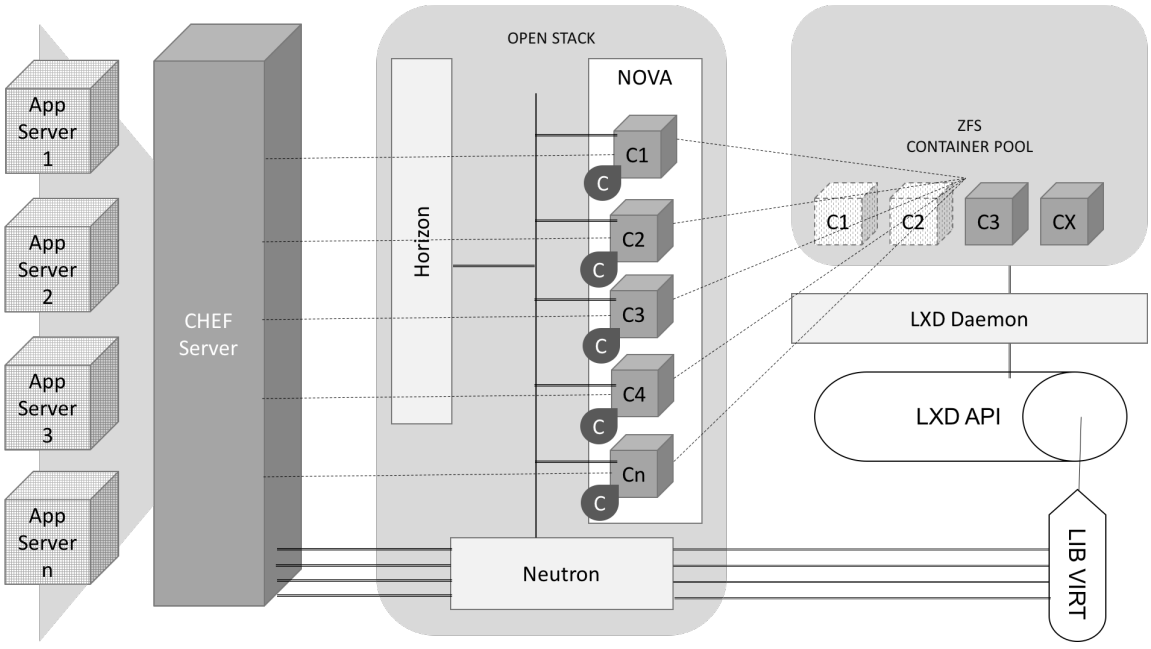

Figure 6.2 Platform as a Service.

The above figure shows the OpenLab framework in action. Chef further can be used for the compute nodes to maintain app specific delivery pipeline.

6.3 Validation of the Framework Using INSPEC Handler

InSpec is an open-source testing framework with a human-readable language for infrastructure testing as well as compliance testing which is optimized for DevOps. We develop a Heavy Weight Resource Provider(HWRP) called 'InSpec Handler' for Chef to perform automated inspec tests at the end of each Chef-client run. HWRPs in Chef are native ruby classes that are used to implement providers and handlers in Chef. They interact directly with Chef's native classes.

The InSpec handler [14] automatically detects the run list and runs their respective tests. It also takes care of the environment in which it is being run. In production environments, it runs only when there is a change in the state of the machine. It is implemented by wrapping it inside a cookbook. It takes the following parameters:

inspec_handler 'name' do

run_path	String
log_path	String
log_shift_age	String
enforced	TrueClass, FalseClass
abort_on_fail	TrueClass, FalseClass
whitelist	Array
blacklist	Array
test_environment	Array
production_environment	String
track_attributes	TrueClass, FalseClass

action Symbol, :hard_run if not specified

end

This provides an efficient way to validate the infrastructure that has been converged by Chef by firing up tests as soon as all the machine reaches the desired state.

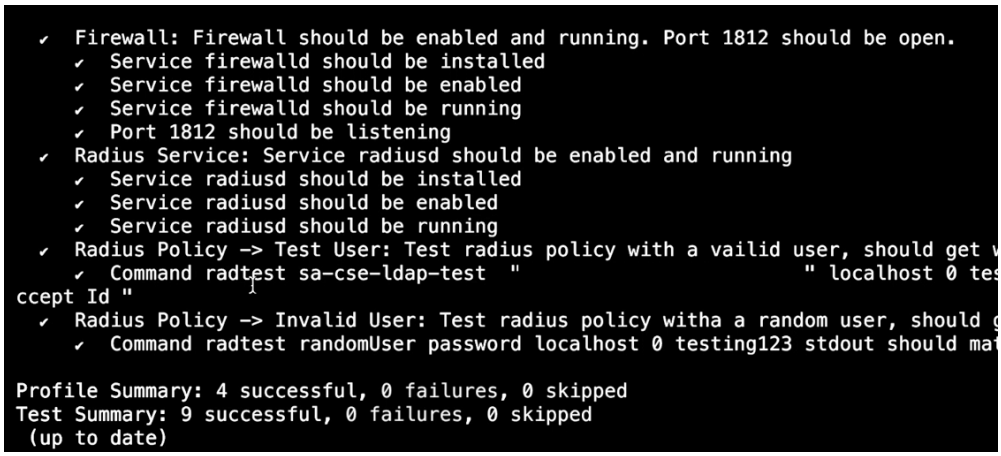

Figure 6.3 InSpec Handler.

The above figure shows a capture of InSpec Handler running inside a Chef client run for Radius Server that is running in the OpenLab framework. Source code for InSpec Handler can be found at https://github.com/sidxz/inspec-handler.

CHAPTER VII

APP ISOLATION AND ON DEMAND APP DELIVERY FOR A MULTI-TENANT 3-TIER IOT ARCHITECTURE

7.1 Introduction

The world has gone through a dramatic transformation, moving from isolated systems to Internet-enabled ‘things’ that connect to a common infrastructure and work ubiquitously. They generate data that are analyzed, extracting valuable information to take informed decisions. The 2-tier server-client model has evolved into a 3-tier model [15] that incorporates a middleware known as a ‘gateway’ that is placed in-between the server and the client. These gateways are embedded devices that have computational power. Activities like cleansing, processing, quick filtering, analyzing the processed data, and even on-site decisions, are computed at the gateway level. Information that needs more computation is sent over to the IoT Server. The ‘IoT Server’, the gateway (also known as the ‘IoT Gateway’), and the ‘End devices’ work together to form this edge intelligence that results in a much more efficient architecture. For this 3-tier architecture, there is a vast diversity of IoT applications that are being developed by a number of vendors. Each of these vendor desires to take more control of the gateway device. Gateway devices are analogous to the internet router placed in one's home (with more roles and responsibilities). Like the routers are shared by different devices that want internet access, the gateway needs to be shared among various end devices that need IoT services, pushing the gateway towards multi-tenancy. The majority of these gateways run Linux or UNIX-

like environment on which a diversity of IoT gateway applications reside. These apps usually prefer having elevated access and acquire control of the entire system. In day-to-days IoT scenario, with no proper standardization and with more vendors having proprietary solutions, sharing of these proprietary apps in the gateway may inflict on each other affecting their functionality and performance.

For example, think of two apps from two different vendors, both trying to bind over port 80 to host their APIs, or imagine a public IoT gateway of a smart city, in which multiple vendors are interested in publishing their presence. This can inadvertently result in a catastrophic failure for one or more apps. Isolation of these apps are of paramount importance.

Unlike proprietary gateways, in which the apps are burnt into the gateway's firmware, in a multi-tenant gateway, as the firmware is shared, installing and updating the apps is another challenge. Canonical's finding suggests that only 31% of the consumer update their apps as soon as it is available, 40% never update, and 40% believe that it is the responsibility of the vendor to perform these updates on their connected devices. In this shared scenario of apps in gateways, no well-defined app delivery mechanism exists.

Our main contribution in this work is to reuse our work in chapters 1 through 6, constructing bridges between them to develop an IoT architecture for on demand app delivery and isolation of apps at the IoT gateway with a minimum performance overhead.

The architecture is based on the spirit that whenever a new end device from a specific vendor is detected by the gateway, it's corresponding IoT server is contacted and necessary apps are downloaded on demand to an isolated environment in the gateway, forming the

3-tier architecture for the vendor. It establishes a dedicated pipeline for the vendor to install newer apps and push updates.

7.2 Proposed Architecture

The expectation from the proposed architecture are five folds:

1. Leverage existing production-ready tools to maximize code reusability.

2. Provide vendors with their personal isolated space in gateways to which they have full control.

3. Provide a platform through which vendors can configure their space and push their apps to it.

4. Rely less on the 'Honor System' for believing users to update their apps. Move control of updates from users to vendors.

5. Automate all of the above.

In chapter 2 we saw LXD containers could provide deep isolation yet yielding bare metal performance. The LXD Daemon makes creating and managing containers easy and provides REST APIs for it. In chapter 4 we walked over Chef that is used to configure and install packages to systems remotely.

Here we introduce two new services namely 'IoT App Store' and 'IoT Gateway Client,' functions of it are discussed later in the section.

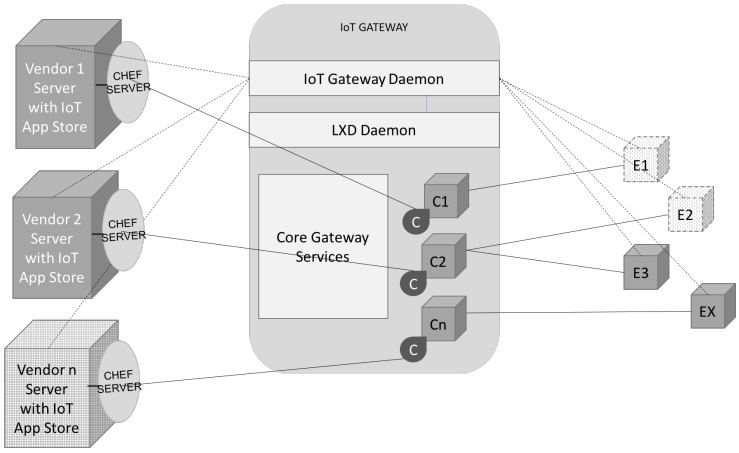

Figure 7.1 Proposed IoT Architecture.

The architecture consists of 3 layers, the Vendors 'IoT Server,' the 'IoT Gateway,' and the 'End Device.' Multiple end devices connect to an IoT gateway. The IoT gateway can interact with multiple IoT Servers from different vendors depending on the diversity of end devices that are connected to it. Each vendor's IoT Server can serve request to multiple IoT gateways. The 'IoT Gateway' comprises of three major modules.

1. The 'IoT gateway Client'
2. LXD Daemon
3. and a number of Chef Clients.

The 'IoT gateway Client' works as an administrator, collecting information from end devices; interacting with the IoT Server and considering the present configuration of the IoT gateway, provisions the personal isolated vendor's space by managing the LXD Daemon to either spawn new containers or make use of existing one. The LXD Daemon is a fresh install of well tested Canonical's LXD, that opens up API's for the IoT gateway client to use. The Chef clients run inside isolated containers and are responsible for creating the app delivery pipeline with the IoT Server. It should be noted that the vendor

specific app resides inside these containers; dedicating a container per vendor app, or a container per a group of apps from the same vendor. This distribution is decided by the vendor.

The Vendors IoT Server can be divided into four components.

1. The 'IoT Daemon'
2. Chef Server
3. IOT App Store
4. Server Side instances of IoT Apps

The ‘IoT Daemon’ is a service that runs on the vendor's IoT Server. This service is the one that interacts with the IoT gateway client present in the middleware IoT gateway. It receives meta information about the end device from the gateway, validates it and responds back the requirements that need to be provisioned in the gateway. The requirements can include asking the IoT gateway client to create a new container for it. The ‘Chef Server’ is used to bootstrap the newly formed container. Together with the instance of Chef-client running inside the container of the IoT gateway, the on-demand app delivery pipeline is formed.

The ‘IoT App Store’ is the repository of gateway apps that the vendor provides. Chef Streams these apps to the gateway as required.

The ‘Server Side instances of IoT apps’ forms the Server tier of the 3-tier architecture. These are completely vendor specific, interacting with the corresponding vendor specific app at the container in the gateway and further down to the end device.

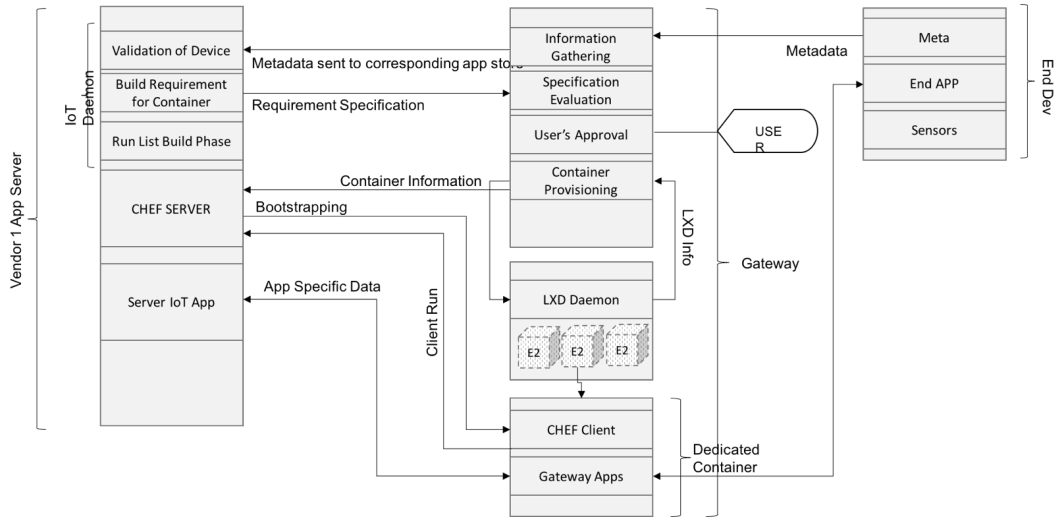

Figure 7.2 Proposed IoT Architecture Protocols.

The above figure describes the logical approach to the architecture.

7.3 Protocols

The following are the protocols that are followed when a new end device is detected.

1. The end device connects to the IoT gateway and sends it's meta information to the IoT gateway Client. The meta has information about its device type, signature and to which app store it should connect. If the device has been already provisioned, it is forwarded to its respective container. If it has not been provisioned, IoT gateway client connects it to the specified app store.

2. Upon receiving the information, the IoT Daemon at the Vendor's IoT Server, verifies the authenticity of the device and generates the build requirement for the device. It might instruct the IoT Gateway Client, to either attach it to one of the vendor's container or create a new one for it.

3. After downloading the build information from the Vendor's IoT Daemon, the IoT Gateway Client, sends a notification to the user for approval. The notification can be sent to a paired mobile device or via a web portal. Approved requests are forwarded to the container provisioning sub module of the IoT Gateway Client.

4. The container provisioning sub module, talks with the LXD Daemon to spawn a new container or get the information of an existing one. Information such as IP address are captured. This is again sent to the IoT Daemon at the Vendors' IoT Server.

5. The IoT Daemon now builds up the run list, which has roles and cookbooks depending on the apps that are to be installed in the container for the end device to work.

6. Having the IP of the container, Chef bootstraps it with the run list and provisions the container with Chef-client and the listed apps.

7. After this bootstrap process, a pipeline is created between the Chef Server, the container at the gateway and the end device.

8. New apps and app updates can now be delivered by changing the run list at the Chef server. This also enables the vendor to automatically push updates.

One of the strengths of this system is, with Chef, there are several community cookbooks available and are developed by a large open source community. They include the best practices for configuring systems and are less vulnerable. With the quick updates being pushed, it makes the system secure. Also, since the apps are jailed inside their respective container, if one of those apps are compromised it will not have any effect on other apps.

7.4 Benchmarks and Results

Considering the potential benefits introduced by the proposed architecture, together with the significant increase in the diversity of applications, our further experiments show that the performance of IoT apps in the jailed environment of the architecture is as good as running them on bare metal boards. For the further experiments, we deployed IoT gateway in a System on Chip device, Raspberry Pi II having one gigabyte of memory and a dual-core ARM processor and on a HP Enterprise Edgeline EL10 which has an Intel E3826 dual-core Atom, operating at 1.46GHz with 4GB DDR3L 1333MHz memory.

We ran few standard benchmarks to support our claim.

1) CPU

We use sysbench [16] to compute prime numbers, comparing the performance between the proposed IoT Framework and running them in native raspberry pi. We performed the same experiment again using HPE Edgeline EL10.

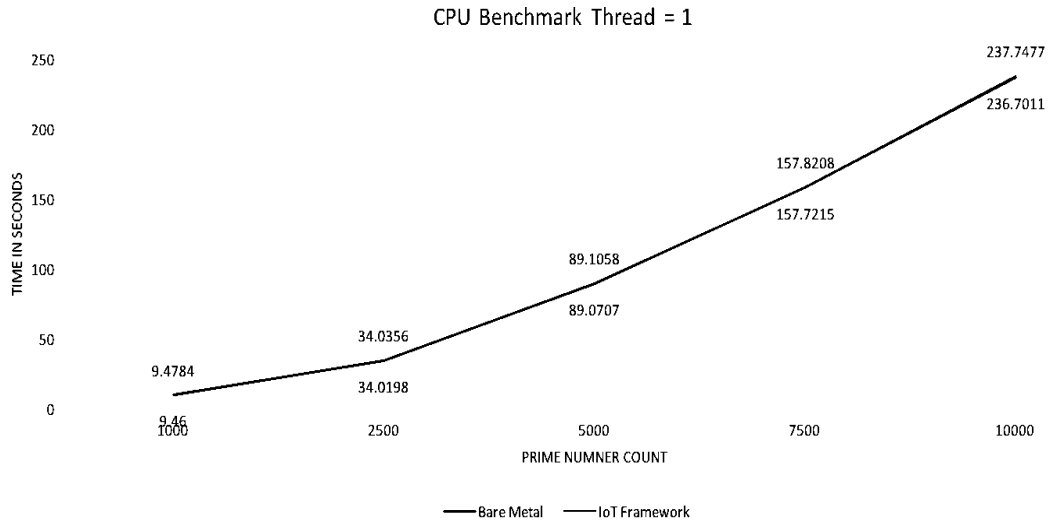

Figure 7.3a CPU Benchmark, R Pi Thread =1.

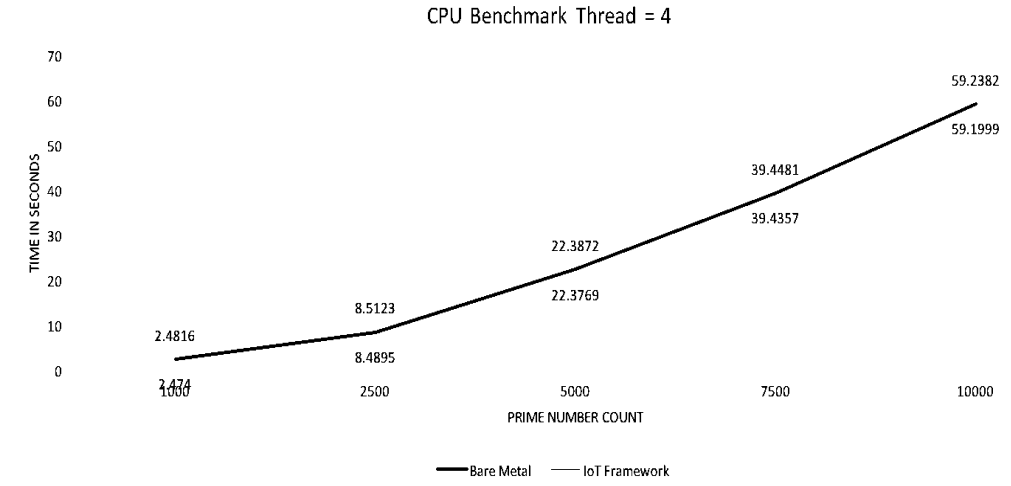

Figure 7.3b CPU Benchmark, R Pi Thread =4.

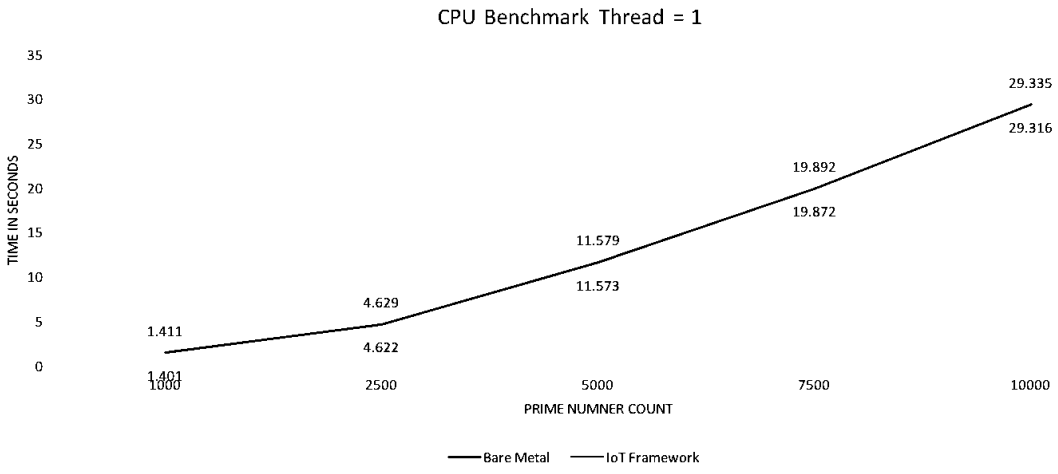

Figure 7.3c CPU Benchmark, HPE EL10 Thread =1.

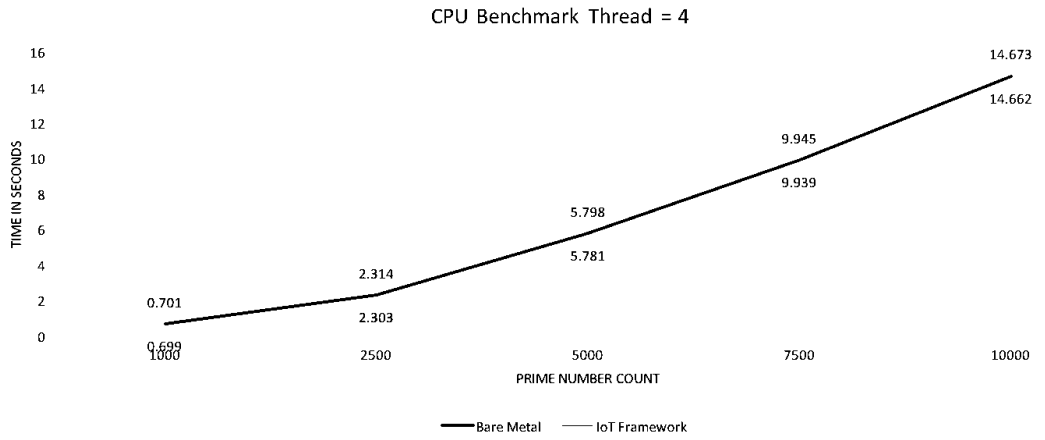

Figure 7.3d CPU Benchmark, HPE EL10 Thread =4.

The X axis represents the range up to which prime numbers are calculated. Y axis represents time in second that was required to do the computation. Figure 5.1a and 5.1c shows the run using one thread while 5.1b and 5.1d uses four threads. HP EL 10 having more computational power, runs the test faster. However, in both the devices, the lines for bare metal and IoT Architecture, run through the same trace, conveying that there was negligible loss in performance.

2) File IO

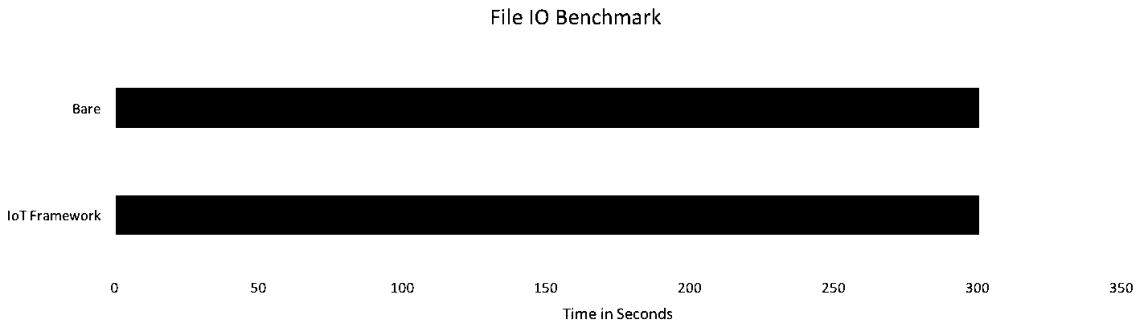

Figure 7.4 File IO Benchmark.

To evaluate file input output performance, we execute random read and write operations transacting one gigabyte of data. We record the time taken for both the cases, which are nearly similar. The IoT Architecture took 300.7 seconds and the bare metal took 300.5 seconds. This graph was similar both for R Pi and HP EL10, considering same type of IO devices used in both the cases.

3) Database Transaction

This was performed to test real database performance. It was done in a similar way in which IoT devices push data to the gateway. We transacted 100,000 records and plotted the performance by increasing the number of worker threads. From the graph, it can be observed how there is no relevant difference regarding performance between the bare metal and the proposed IoT Architecture.

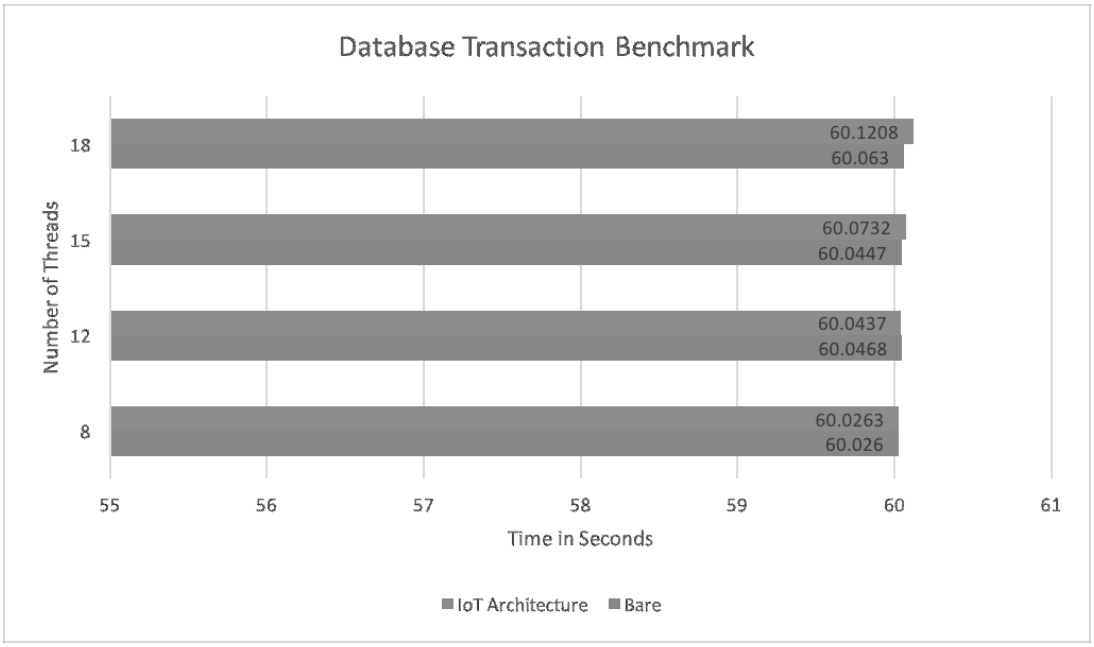

Figure 7.5 Database Transaction Benchmark.

4) App Delivery

In this, we test the time taken by the proposed IoT Architecture to receive an app from the IoT Server. This involves the continuous delivery pipeline following all the protocols to deliver an app on demand. Since this is a new feature of the architecture, benchmarking against bare metal is not in the scope. On one axis we keep increasing the size of the app, starting from 10 Megabytes and increasing it up to 750 Megabytes. On the other side we record the time taken for the end to end process to happen.

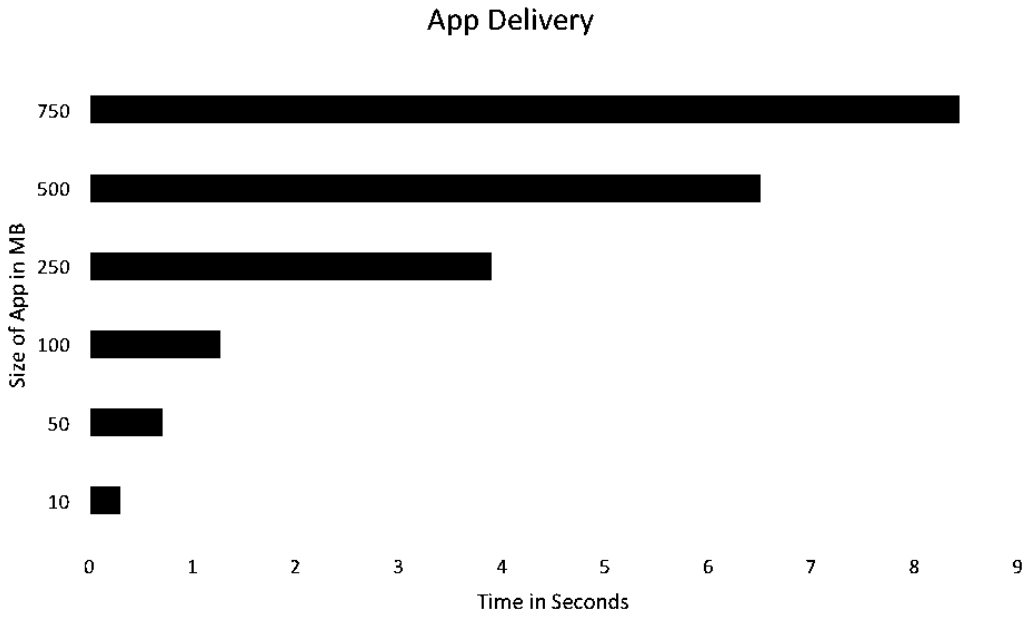

Figure 7.6 App Delivery Benchmark.

With the increase in app size, as expected, the time taken for the transfer also increase.

5) Container Density

In this experiment we study the effect in memory when the container density is increased. We record the memory used in the host machine as we keep increasing the number of container instances.

We see a consumption of around 16 MB of memory per container that is spawned, however this is the trade off to fully utilize the IoT architecture and its features. It should be noted that apps inside the container run in near bare metal performance, which we showed in the first three benchmarks.

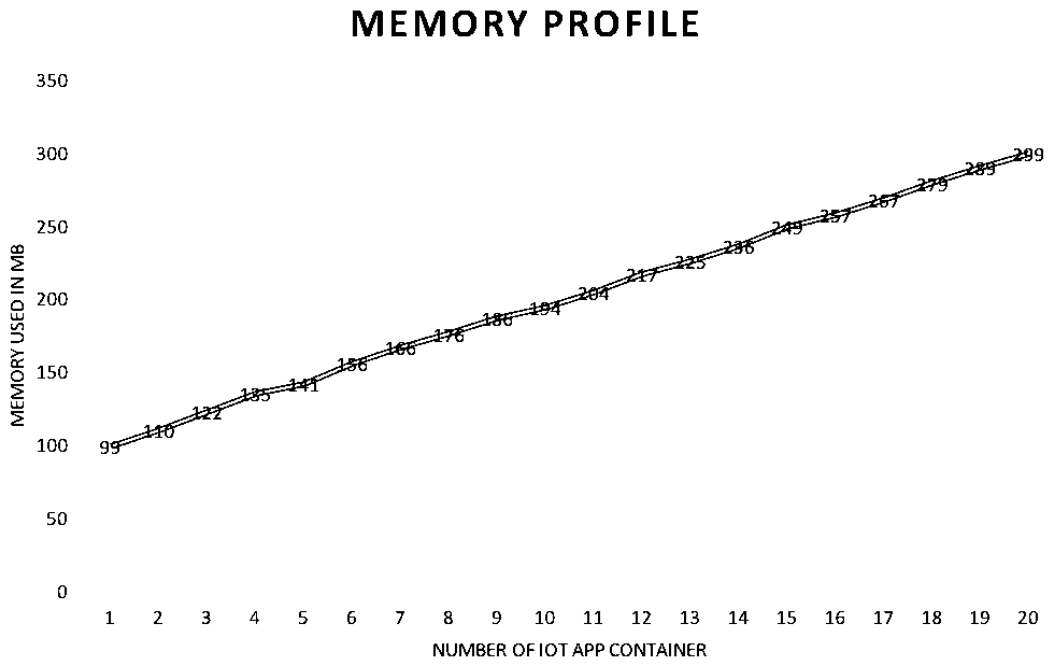

Figure 7.7a Memory Profile Benchmark for R Pi.

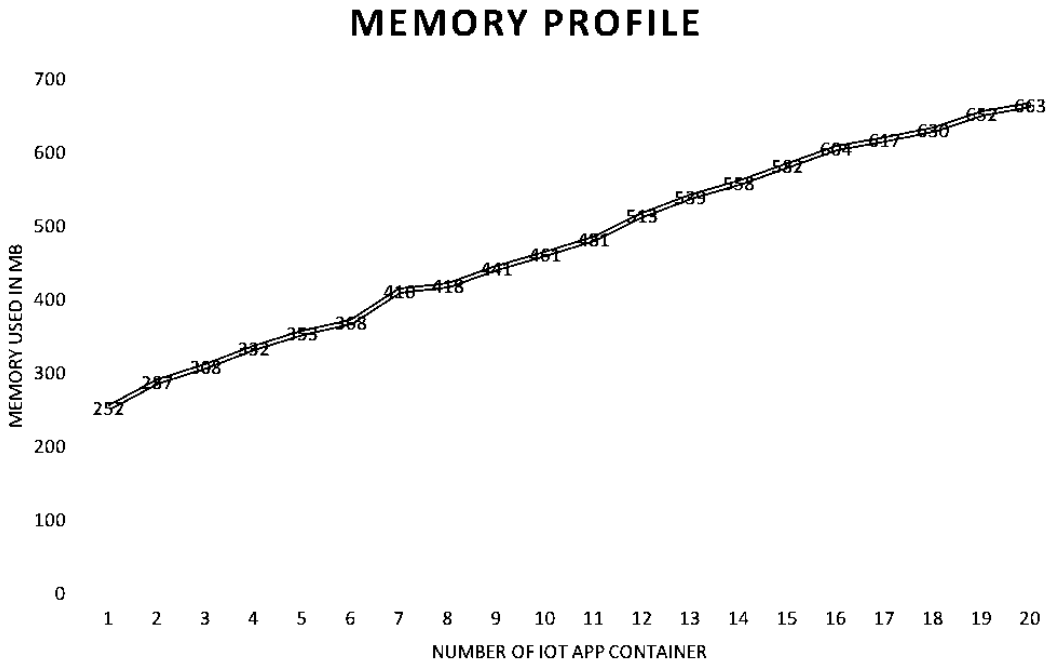

Figure 7.7b Memory Profile Benchmark for HPE.

6) Security

We run the architecture against the Lynis, which is an open source security auditing tool. It is used to detect security vulnerabilities of Linux and UNIX-like systems. We make use of some Chef cookbooks which are inbuilt in the framework and that provides basic hardening of the container image. Lynis reports a score of 74 out of 100. (Lynis score of a fresh copy of OS was 60). Things like minimum password age, maximum password age, default umask, configuration for DNS, Apache modsecurity configuration, SSH configuration, and others were hardened.

CHAPTER VIII

CONCLUSIONS

The architecture forms a logical partition of the gateway giving an illusion to the vendors as if they completely own the 3-tier architecture. This approach has many benefits. First, with machine containers, it enjoys all the privileges of virtualization and at the same time keeps the memory overhead low. The isolations make the overall system more secure. If one of the vendor's IoT architecture is compromised, tainted codes only flow to its corresponding container in the IoT gateway. Since the container is jailed, it won't affect apps of other vendors. A quick recovery can be made by disposing of the compromised container. Second, with Chef, an app delivery pipeline is maintained. Pushing new apps is as easy as publishing it to the Chef Server. Chef also maintains a supermarket that consists of cookbooks for the various commonly used application. These cookbooks are versioned in Git and leverage code sharing following the best practices in the industry. This makes a vendor more confident for their configurations of systems as they follow a certain industry standard. Third, it makes full use of the IoT gateway for proprietary apps. We believe, till standards are created for IoT (given the complexity, it might be a long way), this approach is the best way to address the present challenges.

One of the challenges that we did not address was the availability of domain names. In the current architecture, each container to work with Chef will require its own fully qualified domain name. It might not be a challenge for the larger organization but certainly is a

problem for home IoT users. However, the answer to this problem lies on dynamic DNS. More efforts are required to integrate it into the proposed framework.

REFERENCES

[1] VMWare Whitepaper, *Understanding Full Virtualization, Paravirtualization, and Hardware Assist* (online), 2008, March, Available: https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html

[2] Intel Whitepaper, *Virtualization Technology Introduction* (online), 2008, July, Available: http://slideplayer.com/slide/5055115/

[3] Zdnet Online Documentation, *Virtualization: Hardware and Software working in harmony* (online), 2012, Nov, Available: http://www.zdnet.com/article/virtualization-hardware-and-software-working-together-in-harmony/

[4] Salman Baset, Stefan Berger, James Bottomley, Canturk Isci, Nataraj Nagaratnam1, Dimitrios Pendarakis, and J. R. Rao, *"Docker and Container Security,"* IBM Research, 2016, July 15. Available:http://domino.watson.ibm.com/library/CyberDig.nsf/papers/040F7F7D5E62F0E58525804500433733/$File/rc25625.pdf

[5] Paul Menage, Paul Jackson, Christoper Lameter, and Kernel.org, *Linux Kernel Documentation on Cgroups* (online), 2006 Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[6] Boden Russel, IBM Conference, *Realizing Linux Containers* (online), 2014, Mar, Available: https://www.slideshare.net/BodenRussell/realizing-linux-containerslxc

[7] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia, Virtualization vs Containerization to support PaaS, *IEEE International Conference on Cloud Engineering*, 2014 Sep. DOI: 10.1109/IC2E.2014.41

[8] Mahmud Ridwan, Toptal Online Documentation, *Separation Anxiety: A tutorial for Isolating your system with Linux, Namespaces. (Online)* 2016 Available: https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

[9] R. Morabito, "A performance evaluation of container technologies on Internet of Things devices," 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), San Francisco, CA, 2016, pp. 999-1000.

[10] OpenStack Documentation. May 2017, San Antonio, Texas, Available: https://docs.openstack.org

[11] J. Hintsch, C. Görling and K. Turowski, "Modularization of Software as a Service Products: A Case Study of the Configuration Management Tool Puppet," 2015 International Conference on Enterprise Systems (ES), Basel, 2015, pp. 184-191.

[12] Chef Documentation. May 2017, Seattle, Washington, U.S, Available: https://github.com/chef/chef-web-docs

[13] Justin Ellingwood, DigitalOcean, *An Introduction to Continuous Integration, Delivery, and Deployment,* May 2017, Available: https://www.digitalocean.com/community/tutorials/an-introduction-to continuous-integration-delivery-and-deployment

[14] Siddhant Rath, *I am Inspec Handler (Online)*. May 2017, Available: http://sidx.me/chef/inspec/inspec-handler/

[15] Canonical White Paper, *Taking Charge of the IoT's security vulnerabilities, Whitepaper,* Jan 2017,

Available: https://pages.ubuntu.com/IoTSecurityWhitepaper-Fullreport.html.

[16] A. Krylovskiy, "Internet of Things gateways meet linux containers: Performance evaluation and discussion," 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), Milan, 2015, pp. 222-227.