****

SOME PERFORMANCE ASPECTS CONSIDERATIONS OF A CLASS OF ARTIFICIAL NEURAL NETWORK

## A PROJECT REPORT

***Submitted by***

**V.KARTHEEYAYINI [REGISTER NO:211417104145]**

**S.MADHUMITHA [REGISTERNO:211417104137]**

**G.LALITHA [REGISTER NO:211417104129]**

***In partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

**IN**

## COMPUTER SCIENCE AND ENGINEERING

**PANIMALAR ENGINEERING COLLEGE, CHENNAI-600123.**

**ANNA UNIVERSITY: CHENNAI 600 025**

**APRIL 2021**

**BONAFIDE CERTIFICATE**

Certified that this project report **“AWS Cloud computing platforms deployment of Landing Zone - Infrastructure as a Code”**  is the bonafide work of “**V.KARTHEEYAYINI REG.NO:211417104109 ”**

who carried out the project work under my supervision.

## SIGNATURE SIGNATURE

**Dr.S.MURUGAVALLI,M.E.,Ph.D., C.JACKULIN**

**HEAD OF THE DEPARTMENT SUPERVISOR**

DEPARTMENT OF CSE, DEPARTMENT OF CSE,

PANIMALAR ENGINEERING COLLEGE, PANIMALAR ENGINEERING COLLEGE, NAZARATHPETTAI, NAZARATHPETTAI,

POONAMALLEE, POONAMALLEE,

CHENNAI-600 123. CHENNAI-600 123.

Certified that the above candidate(s) was/ were examined in the Anna University Project Viva-Voce Examination held on...........................

## INTERNAL EXAMINER EXTERNAL EXAMINER

**ACKNOWLEDGEMENT**

We express our deep gratitude to our respected Secretary and Correspondent **Dr.P.CHINNADURAI, M.A., Ph.D.** for his kind words and enthusiastic motivation, which inspired us a lot in completing this project.

We would like express our heartfelt and sincere thanks to our Directors

## Tmt.C.VIJAYARAJESWARI, Dr.C.SAKTHIKUMAR, M.E., Ph.D., and

**Tmt. SARANYASREE SAKTHIKUMAR B.E.,M.B.A.,** for providing us with the necessary facilities for completion of this project.

We also express our gratitude to our Principal **Dr.K.Mani, M.E., Ph.D.** for his timely concern and encouragement provided to us throughout the course.

We thank the HOD of CSE Department, **Dr. S.MURUGAVALLI , M.E.,Ph.D.,** for the support extended throughout the project.

We would like to thank my **Project Guide C.JACKULIN**  and all the faculty members of the Department of CSE for their advice and suggestions for the successful completion of the project.

## NAME OF THE STUDENTS

V.KARTHEEYAYIN

S.MADHUMITHA

G. LALITHA

**ABSTRACT**

The manual installation and configuration of IT systems has been a tedious and time consuming process that created several challenges to the engineers during the maintenance and management process of the IT systems. The introduction of cloud computing in combination with the rise of the virtualization technology have managed to address some of these challenges. However, these virtualized cloud systems are followed by a huge portfolio of new tools and platforms that are difficult to learn and maintain. As a result, organizations started investigating the software-defined technology as a new and effective way to meet these new standards and serve the constantly increasing demand of the industry. The software-defined technology describes every part of an IT system that can be performed entirely by software, ranging from the infrastructure to the deployment level of an IT system. The goal of this research project was to investigate the software-defined technology and suggest how it can be used in order to improve the static IT infrastructure of an organization. The literature study of this research focuses on the concepts and the available software-defined tools at each layer of an IT system. Based on the knowledge acquired from the literature study, a reference architecture of the infrastructure and the network layer of a generic software-defined system was proposed that describes the interconnections between the different software-defined concepts. The next step was the design of a software-defined system that uses specific tools and technologies, and is based on a specific list of requirements. The requirements were formed by studying the needs of an actual mission critical organization. The final step was the validation of the design, which was performed by conducting a series of semi-structured interviews with seven industry experts. The validation results showed that the software-defined technology can improve the scalability, upgradability and documentation of an IT system, but the proposed design involves high levels of complexity, which might affect the performance and the required learning curve of the system. Overall, the interviewees acknowledged the potential of this technology and mentioned that its current maturity is inadequate for mission critical systems. Based on these remarks, a list of recommendations and several aspects that require additional future research are included in this thesis.

# TABLE OF CONTENTS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CHAPTER NO.** | | **TITE** | | **PAGE NO.** |
|  | | **ABSTRACT** | | 4 |
|  | | **LIST OF FIGURES** | | 8 |
|  | | **LIST OF ABBREVIATIONS** | | 10 |
|  | |  | |  |
| **1.** | | **INTRODUCTION** | |  |
|  | | 1.1 Problem Statement | | 13 |
|  | | 1.2 Software-Defined Concept | | 15 |
|  | | 1.3 Scope | | 16 |
|  | | 1.4 Research Methodology | | 17 |
| **2.** | | **LITERATURE SURVEY** | | 19 |
| **3.** | | **SYSTEM ANALYSIS** | |  |
|  | | 3.1 Existing System | | 28 |
|  | | 3.2 Proposed system | | 29 |
|  | | 3.3 Requirement Analysis and Specification | | 30 |
|  | | 3.3.1 REQUIREMENTS | | 30 |
| **4.** | **SYSTEM DESIGN** | |  | |
|  | 4.1. Use case diagram | | 38 | |
|  | 4.2 Sequence diagram | | 40 | |
|  | 4.3 Activity diagram | | 40 | |
|  | 4.4 Class diagram | | 42 | |
| **5.** | **SYSTEM ARCHITECTURE** | |  | |
|  | 5.1 Architecture Overview | | 43 | |
|  | 5.2 System architecture  5.3 Network design | | 46    55 | |
| **6.** | **SYSTEM IMPLEMENTATION** | |  | |
|  | 6.1 Coding (Sample Code) | | 59 | |
|  |  | |  | |
| **7.** | **SYSTEM TESTING** | |  | |
|  | 7.1 Coding | | 71 | |
|  | 7.2 Coding standard | | 71 | |
|  | 7.3 Test procedure | | 72 | |
| **8.** | **CONCLUSION** | |  | |
|  | 8.1 Conclusion and Future Enhancements | | 76 | |
|  | 8.2 Screen shorts | | 78 | |
| **9.** | **PREFERENCE** | | 80 | |
|  |  | |  | |

**LIST OF FIGURES**

Figure 1: Overview of the DSRM

Figure 2: Main elements of the software-defined infrastructure.

Figure 3: Sample Terraform workflow

Figure 4: Basic structure of Chef

Figure 5: SDN vs traditional networks

Figure 6: Interfaces and SDN controllers

Figure 7: The architecture of Opendaylight

Figure 8: Comparison of hypervisor (a) and container-based (b) deployments

Figure 9: Design steps for the software-defined system

Figure 10: Realization of the infrastructure (a) and network (b) layer of a generic software-defined system

Figure 11: VM configuration model

Figure 12: Overview of the objects in Cobbler

Figure 13: Bare Metal Configuration Model

Figure 14: Container cluster configuration model

Figure 15: Architecture realization of the supported infrastructure layer

Figure 16: Architecture realization of the future possibly improved infrastructure layer

Figure 17: Configuration models of the network components Figure 18: Security group configuration model

Figure 19: FWaaS and security group protection

Figure 20: FWaaS configuration model and related concepts

Figure 21: Components of a Openflow & OVSDB physical switch

Figure 22: Components of an Open vSwitch

Figure 23: Architecture realization of the network layer

Figure 24: Traffic flows for scenario 1

Figure 25: Traffic flow for scenario 2

Figure 26: Traffic flows for scenario 3

Figure 27: Traffic flows for scenario 4

Figure 28: Physical Architecture of the current design

Figure 29: Physical architecture of the possibly improved design

Figure 30: Terraform file for creating Vms

Figure 31: Terraform file for provisioning bare metal machines with Cobbler

Figure 32: An example of a kickstart file

Figure 33: Terraform file for provisioning container clusters of containers

Figure 34: Terraform file for creating Openstack network

Figure 35: Terraform file for creating floating Ips

Figure 36: Terraform file for creating security groups

Figure 37: Terraform file for creating firewalls

Figure 38: Transport Zone example

**LIST OF ABBREVATION**

**AWS** Amazon Web Services

**BNSF** Base Network Service Function

**CapEx** Capital Expenditures

**COE** Container Orchestration Engine

**COTS** Commercial off the Shelf

**DDoS** Distributed Denial of Service

**DHCP** Dynamic Host Configuration Protocol

**DLUX** Opendaylight User Experience

**DNS** Domain Name System

**DSL** Domain Specific Language

**FWaaS** Firewall as a Service

**GPU** Graphical Processing Unit

**COE** Container Orchestration Engine

**COTS** Commercial off the Shelf

**DDoS** Distributed Denial of Service

**DHCP** Dynamic Host Configuration Protocol

**DLUX** Opendaylight User Experience

**DNS** Domain Name System

**DSL** Domain Specific Language

**FWaaS** Firewall as a Service

**GPU** Graphical Processing Unit

**GUI** Graphical User Interface

**HAL** Hardware Abstraction Layer

**HCL** Hashicorp Configuration Language

**IaaS** Infrastructure as Service

**IaC** Infrastructure as Code

**NaaS** Networking as a Service

**NFV** Network Function Virtualization

**NIC** Network Interface Card

**NOS** Network Operating System

**ODL** Opendaylight

**OpEx** Operating Expenditures

**OS** Operating System

**OVS** Open vSwitch

**OVSDB** Open vSwitch Database

**PAD** Programmable Abstraction of Data

**PXE** Preboot Execution Environment

**CHAPTER 1**

**INTRODUCTION**

**1.1 Problem Statement**

Setting up IT infrastructures has always been a long and challenging procedure, especially in the past, in which it used to be a tedious manual process. The servers were physically installed rack by rack and the hardware components were manually configured according to the requirements of the operating systems and the running applications. The manual installation, configuration and maintenance of the infrastructure is an expensive and time consuming process with a high chance of error. Specialized staff is required to perform the setup work (e.g., network engineers to set up the physical network, storage engineers to maintain the physical drivers, etc.) and real estate should be acquired to house all this hardware equipment. In addition, these huge data centers need maintenance, which adds up extra costs for security and operating costs, such as electricity and cooling. The servers are also prone to configuration errors and they tend to be inconsistent, as they are provisioned by many different engineers who are not in constant communication with each other and they do not share the same scope and goals. This often leads to undesired configuration abnormalities and errors, which can be crucial to the proper functionality of the entire system. Finally, in a traditional manual infrastructure, creating an isolated environment for testing and disaster recovery simulations is very costly and time consuming to be a feasible strategy, and the only way for testing and improving the system is to actually experience a disaster, which is highly risky and stressful. The introduction of cloud computing appeared as a promising solution to many of these previously mentioned problems. The rise of the cloud technology is highly related to the evolution of the virtualization technology, in which an application is abstracted away from the hardware which is emulated by a software layer called hypervisor. The combination of these technologies offers a more efficient way to set up and configure a relatively simple configuration, which would address the problems of scalability and agility of the infrastructure. However, many IT organizations still face problems with the configuration inconsistency of these systems. They tend to use processes and structures that they used to manage software before the introduction of the cloud technology, and most of the times the used tools are unable to keep up with the really short provisioning time (seconds or minutes) required by the new systems. 2 Furthermore, cloud computing promotes the usage of scripts. Writing scripts offers some benefits compared to the manual configuration, such as the automation and standardization of a company’s IT processes. Nevertheless they are not able to entirely solve the management and configuration problem. Scripts can highly vary in the way programmers write them and that means that multiple scripts performing the same task can coexist in an organization, causing troubles to the system administrators who have to spend a lot of time and effort on each script to understand it and potentially use it. Another important issue caused by scripting is their size and their complexity. The size of a script grows as the configuration gets more complex and demanding, which results in huge script files that are almost impossible to be understood by an engineer who is new to the organization, creating a feeling of uncertainty for the operation of the system. Finally, scripts are not suitable for long term configuration because they cannot provide idempotence to the system, since scripts cannot ensure the same results if they run several times. Idempotence is a key condition for long term configuration and management of the system, and even though it can be ensured by scripts, it is very hard to implement and most of the times it is not worth the effort.

**1.2 Software-Defined Concept**

In the last 5 years, the industry has been trying to make the next big step towards the improvement of the configuration and the deployment of the entire IT infrastructure. Their main goal is to completely move away from any hardware dependence and create a more dynamic and responsive platform of software functionality. The term that fully describes this movement is software defined everything (SDx), which can be defined as follows: “SDx is any physical item or function that can be performed as or automated by software”

The SDx is an umbrella term that can be encountered at the following levels:

1. IT infrastructure level (Software-Defined Infrastructure – SDI)

2.Network level (Software-Defined Networking – SDN)

3.Computing level (Software-Defined Computing - SDC)

4.Application deployment level (Software-Defined Deployment - SDD)

The new software-defined infrastructure should include and connect the technologies covered by all these levels and support applications on top of them that can connect to each other and ultimately support end users applications.

**1.3 Scope**

This thesis answers the following main research question: “How can the software-defined technology be used to improve a static IT infrastructure of an organization?” This research question cannot be directly used as a basis for a academic research because it is quite general. Therefore, it is divided into the following sub-questions:

Q-1 Which are the relevant SDx technologies/ tools at each level of an IT system?

Q-2 What does the SDx technology offer at each level of an IT system?

Q-3 How to build a reference architecture for a generic softwaredefined system?

Q-4 How to build a software-defined system for a specific case study?

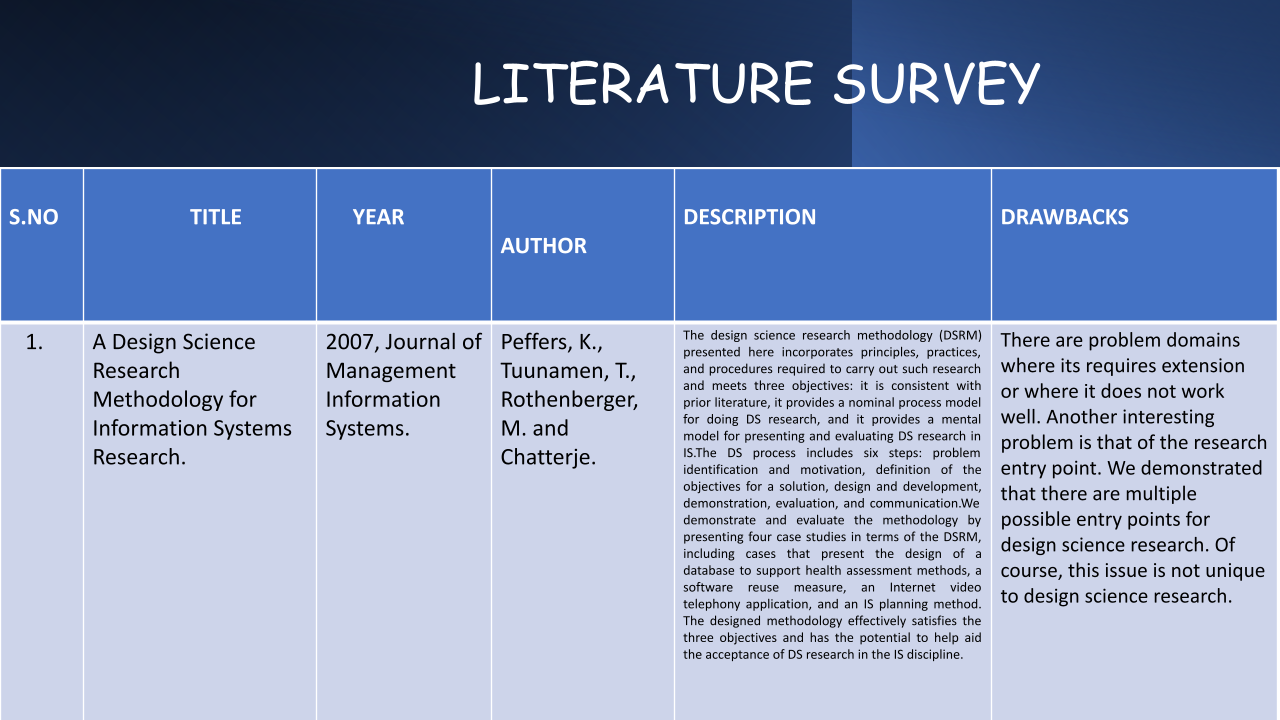
Q-5 How can a software-defined system be validated?

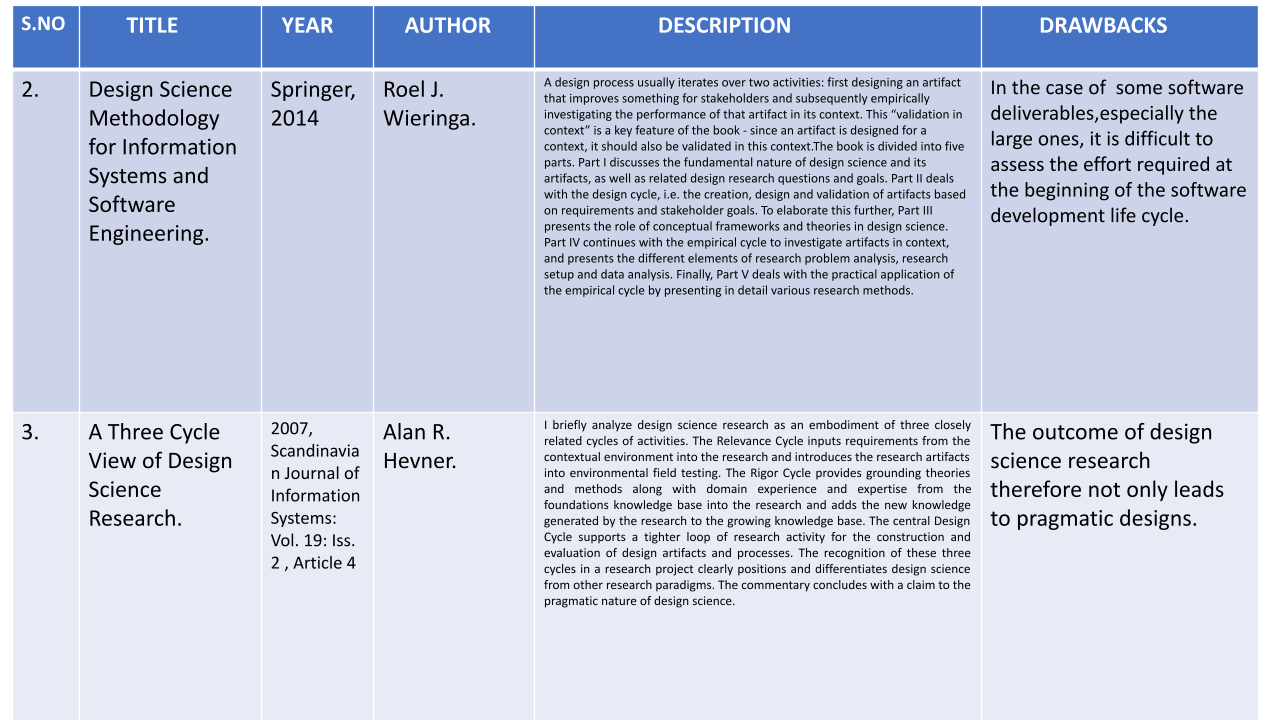
The primary goal of this research is to examine and evaluate the software-defined technology in practice by designing and describing an entirely software-defined IT architecture model that fulfills a specific list of requirements derived from a mission critical organization. Additionally, this software-defined system should be validated according to several aspects that define the quality of a software system such as functionality, performance and scalability. The validation process will determine whether the software-defined technology should be used in production-level deployments, or the maturity of this type of systems is inadequate for high-end systems, and improvement is required.

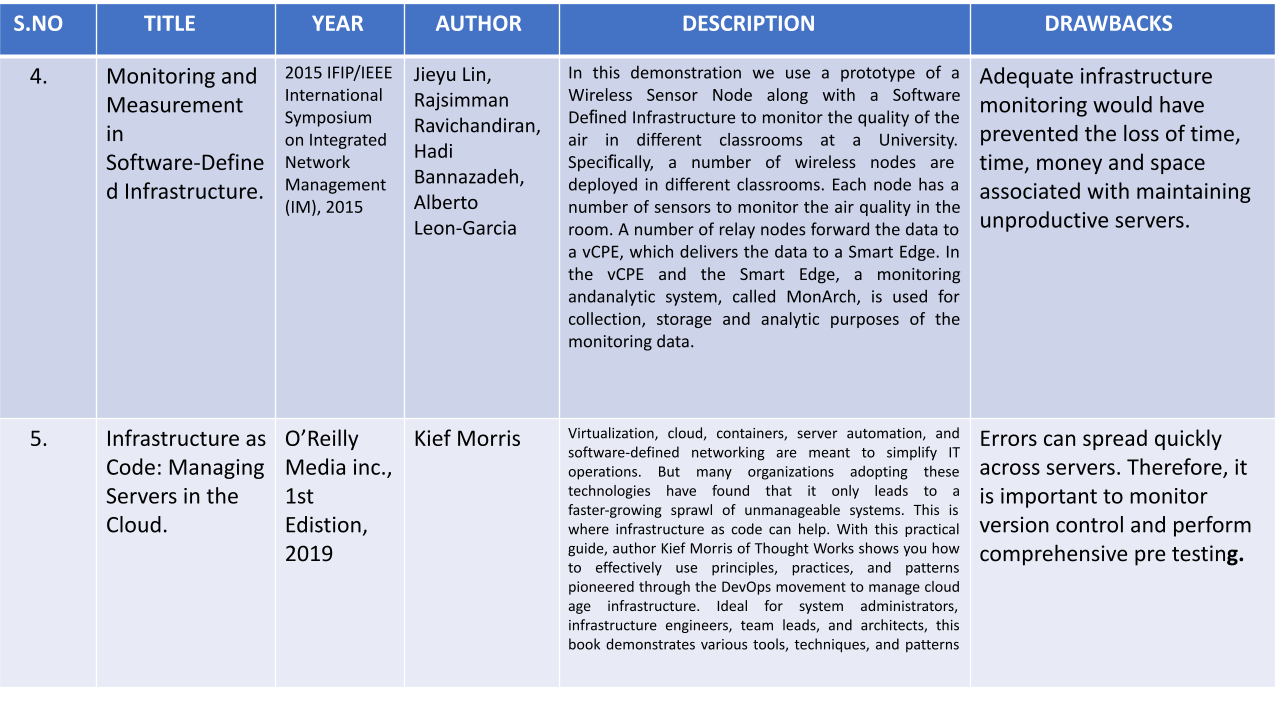
**1.4 Research Methodology**

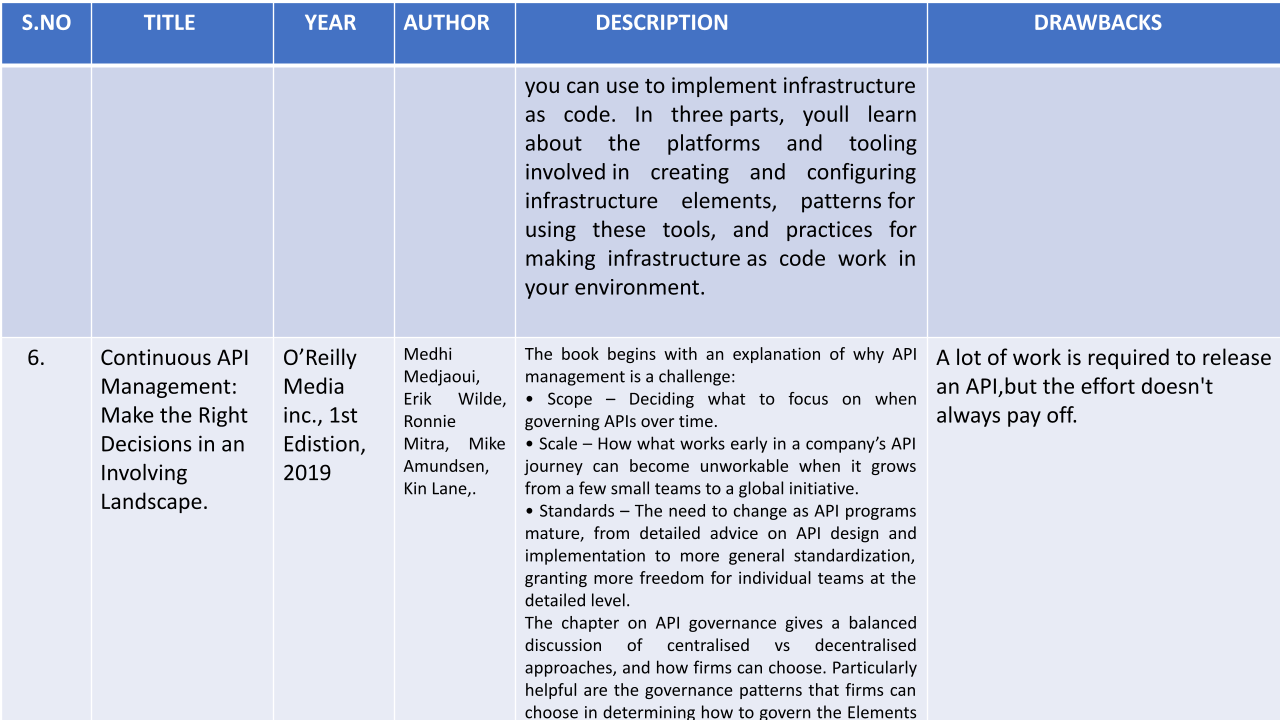
The structure of this research follows the Design Science Research Methodology (DSRM) defined by Peffers . The design science methodologies of Wieringa and Henver were also studied during the selection process; however they were not chosen, as they are not suitable for this specific research project. Wieringa’s methodology is a very thorough and strict method that provides a blueprint for performing design science research. This methodology converts the design science problems into a strict set of questions and steps that adds complexity and limitations during the research process. In contrast, the Henver’s design science framework gives more space and freedom to the researcher by proposing a cyclical process of development and evaluation. This framework is based on three design research cycles that are a combination of scientific literature and practical testing, which is not the appropriate approach for this research project. The DSRM by Peffers is the selected approach as it provides a less strict iterative model that provides guidelines to the researcher throughout the entire research process and is based on strong literature knowledge. Figure 1 depicts an overview of the DSRM.

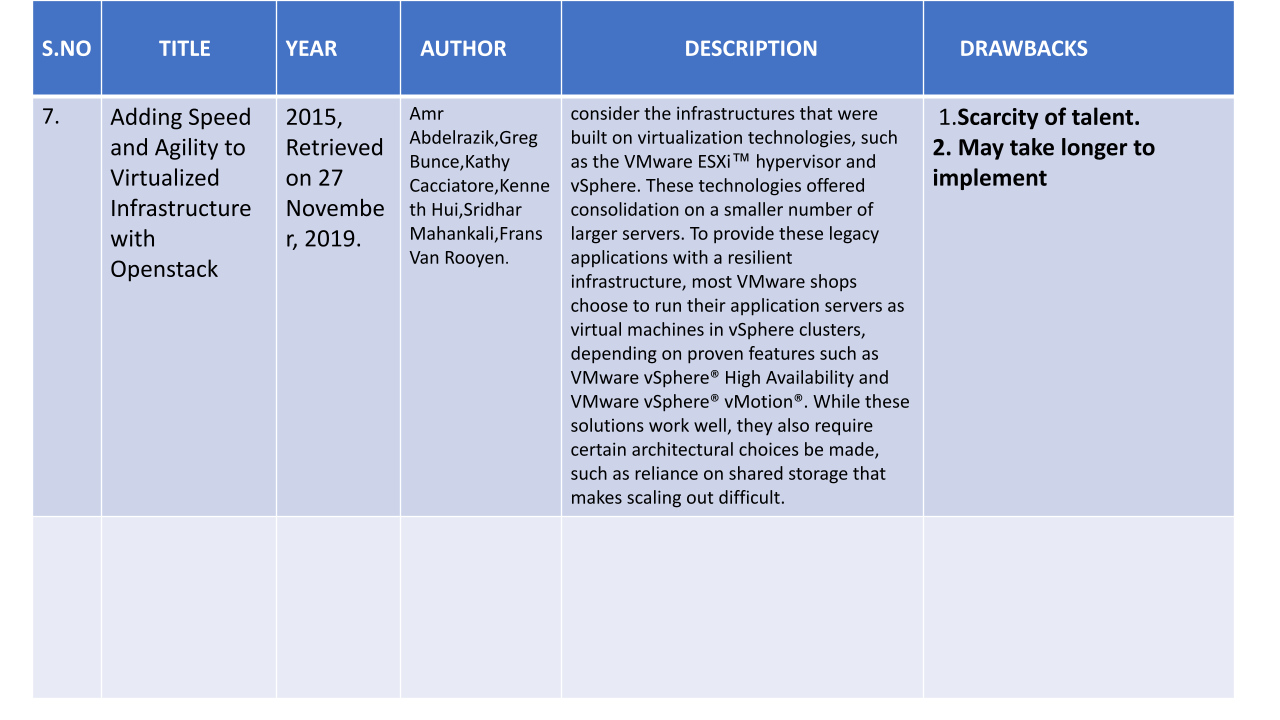
**CHAPTER 2**

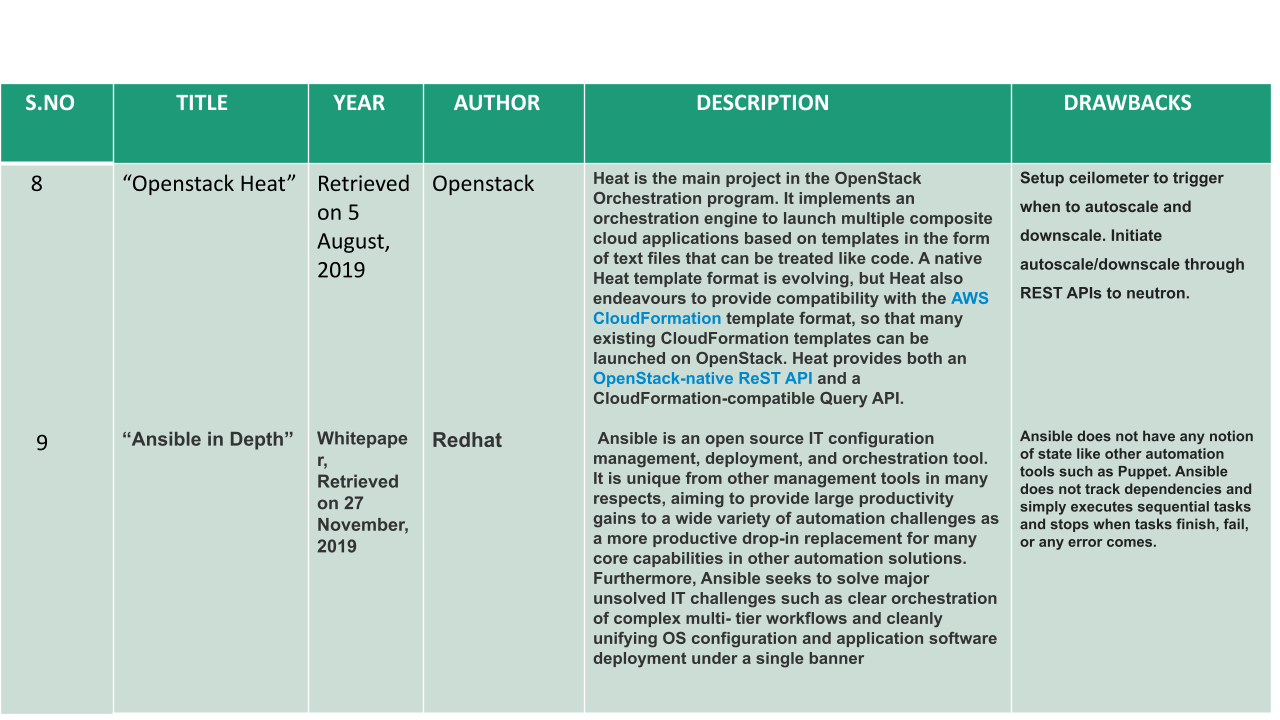


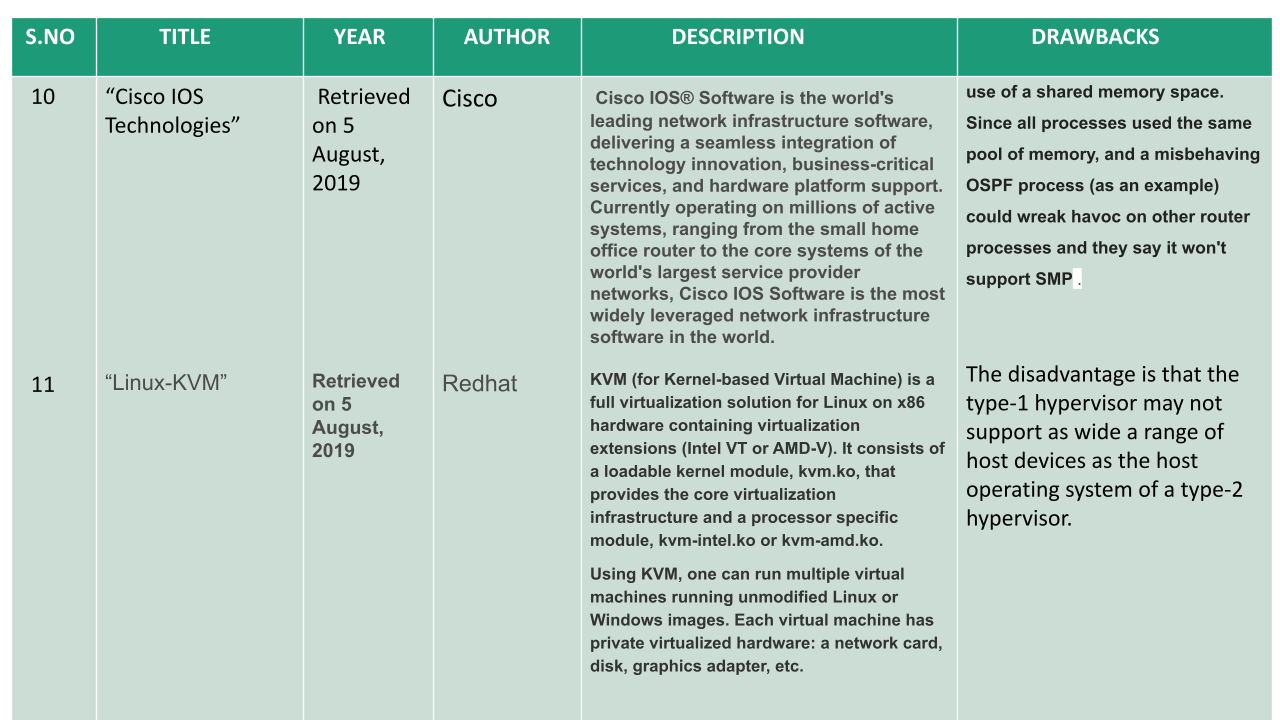


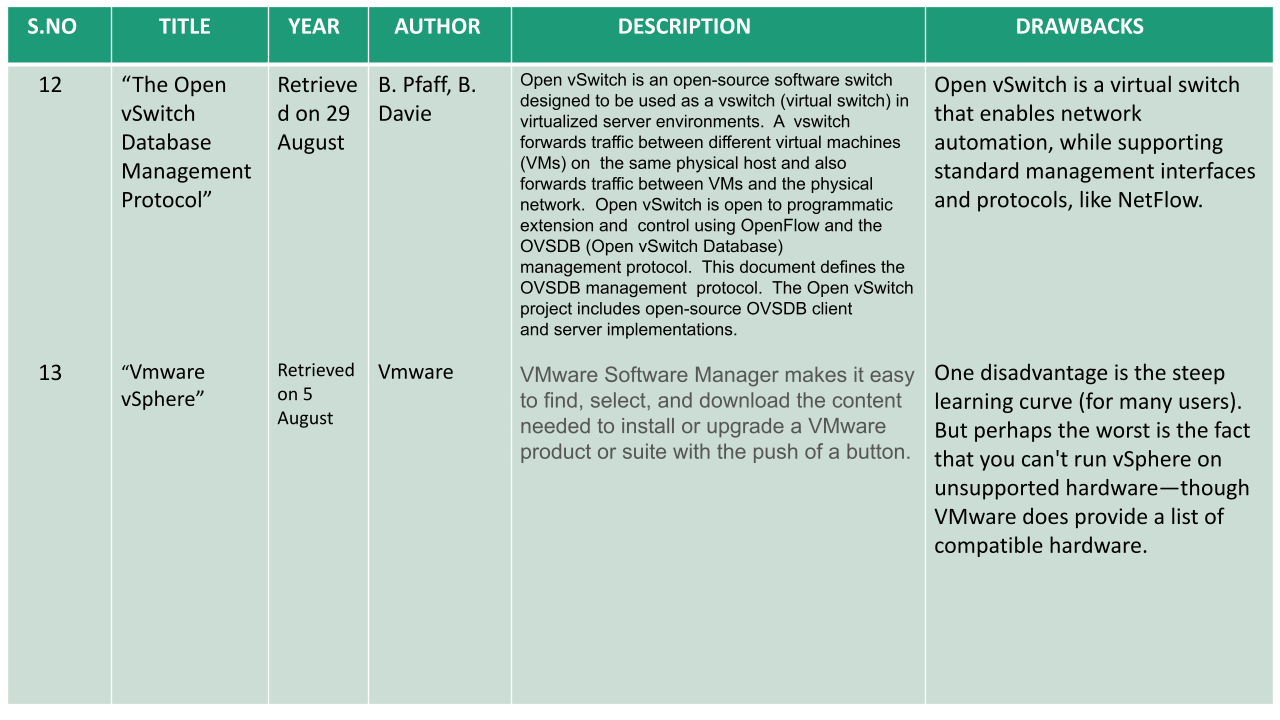


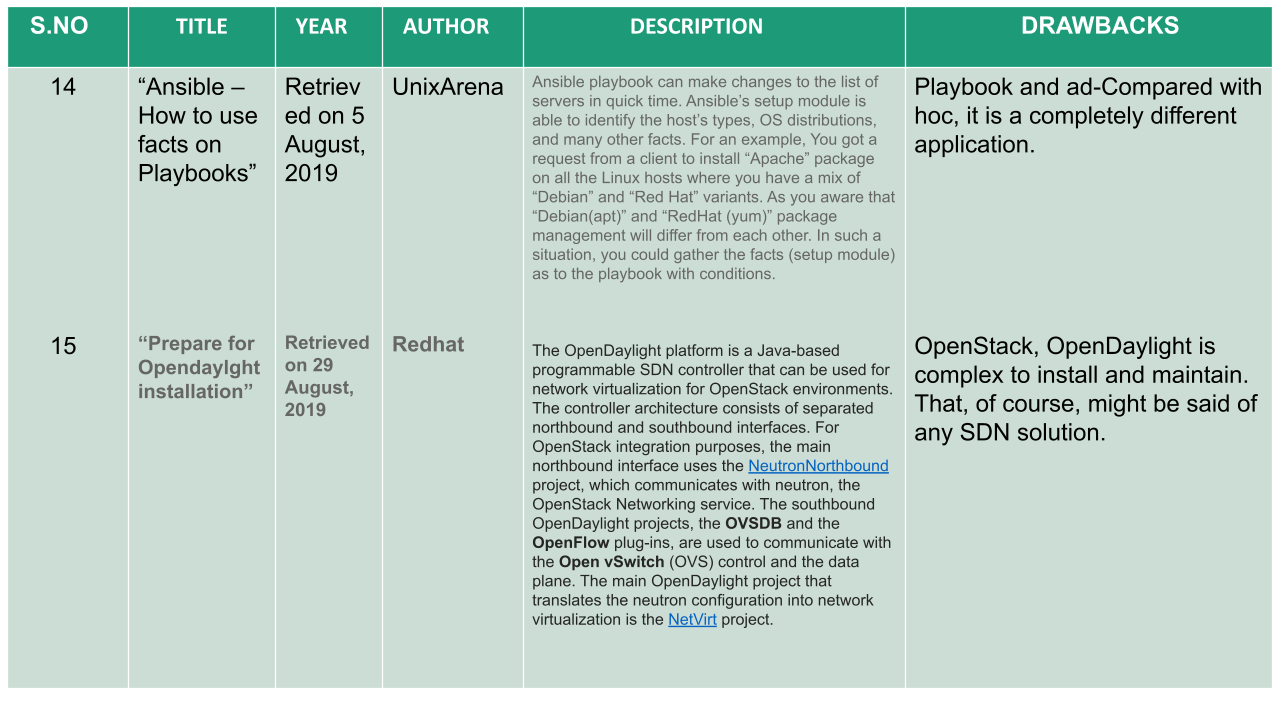


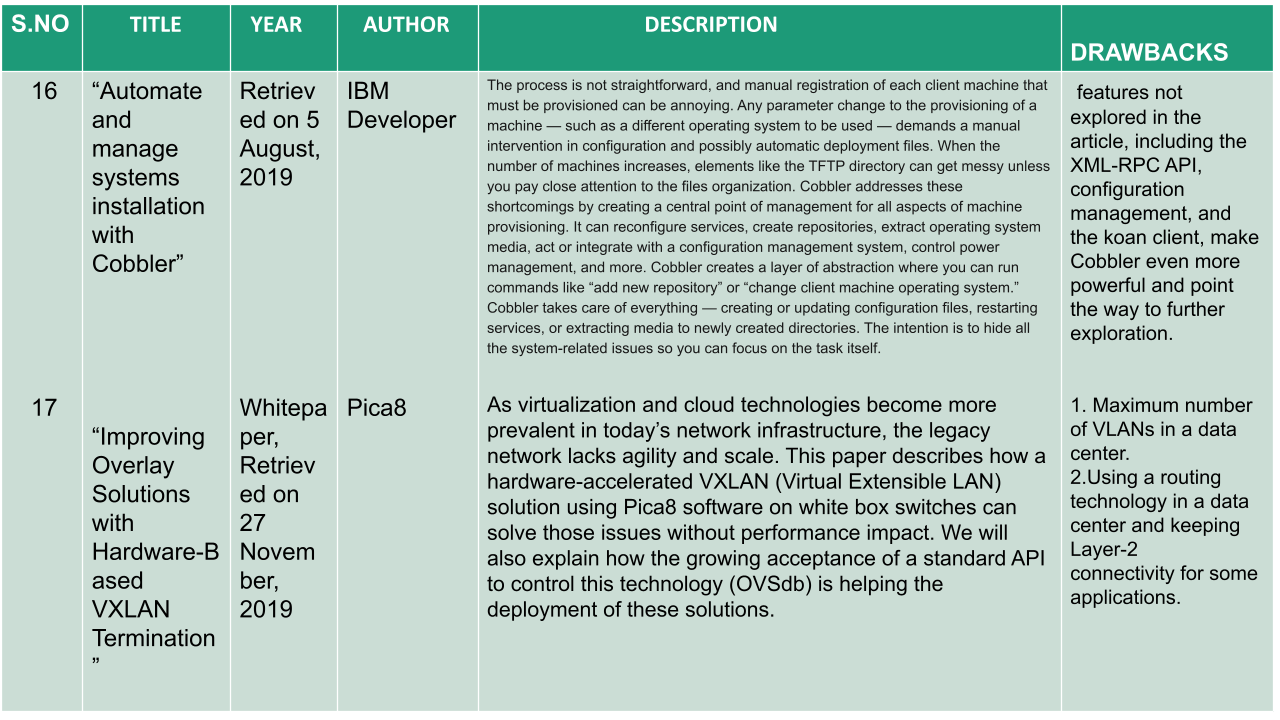


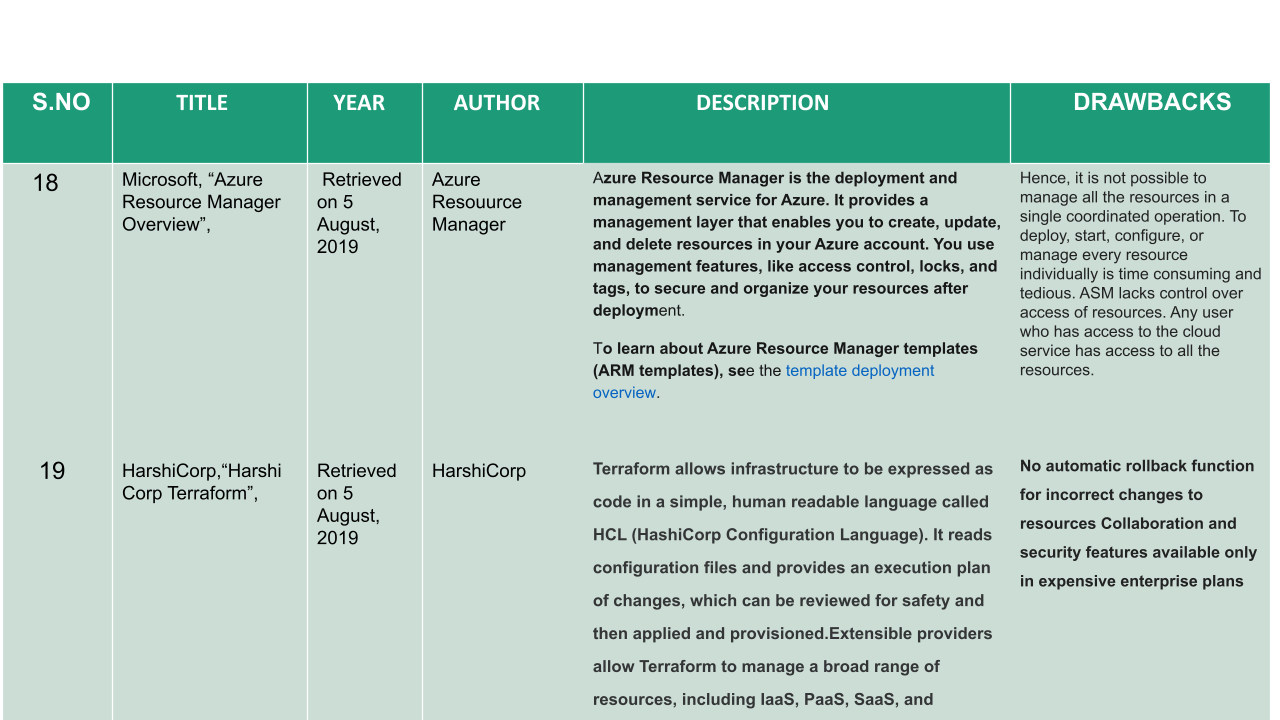


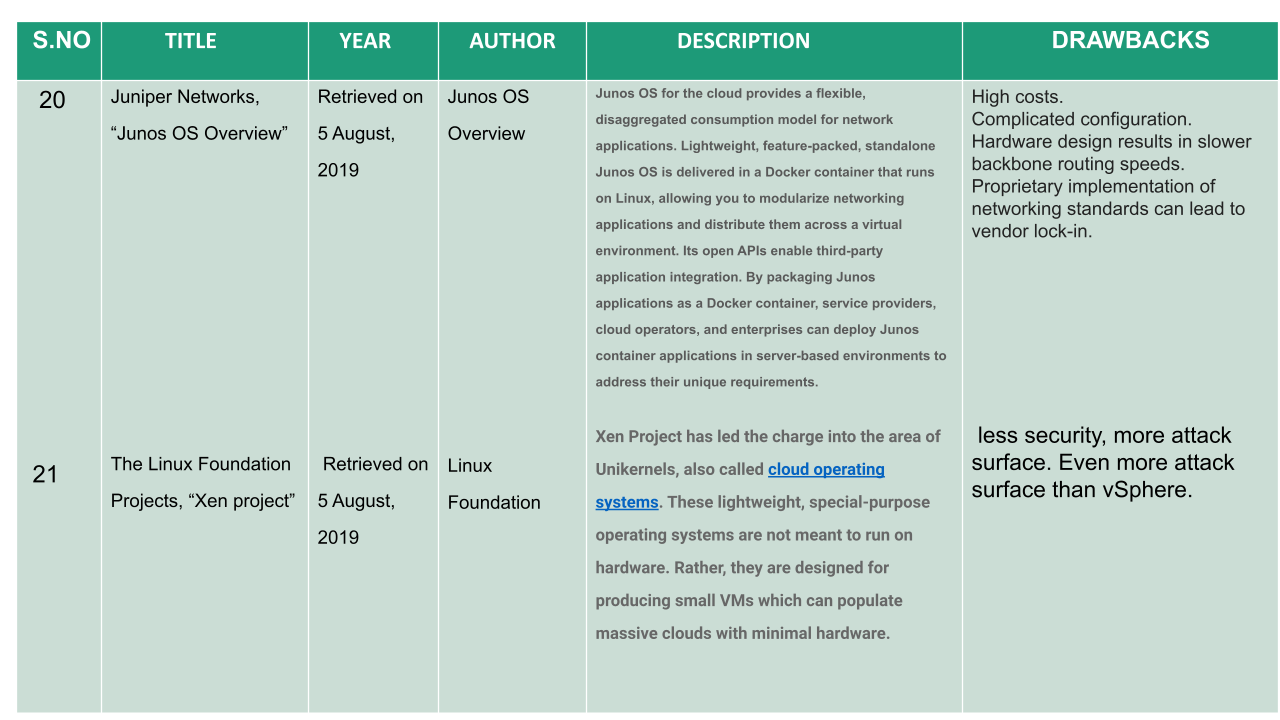


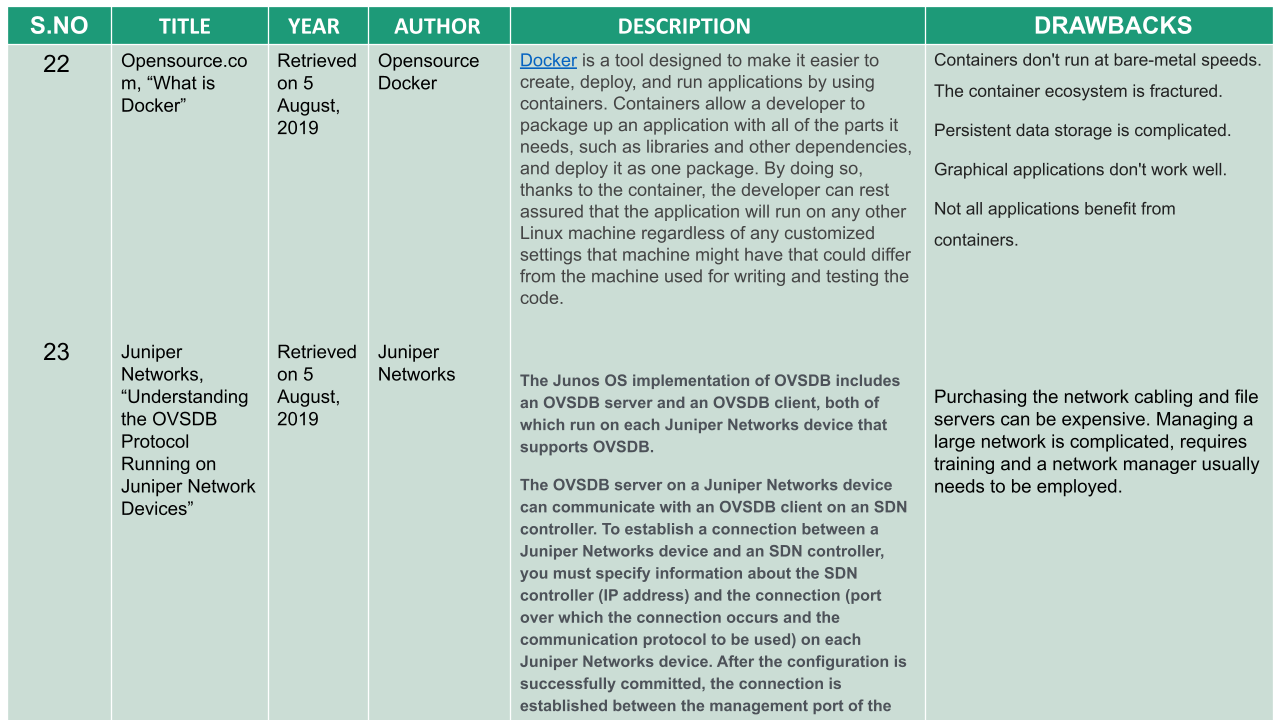


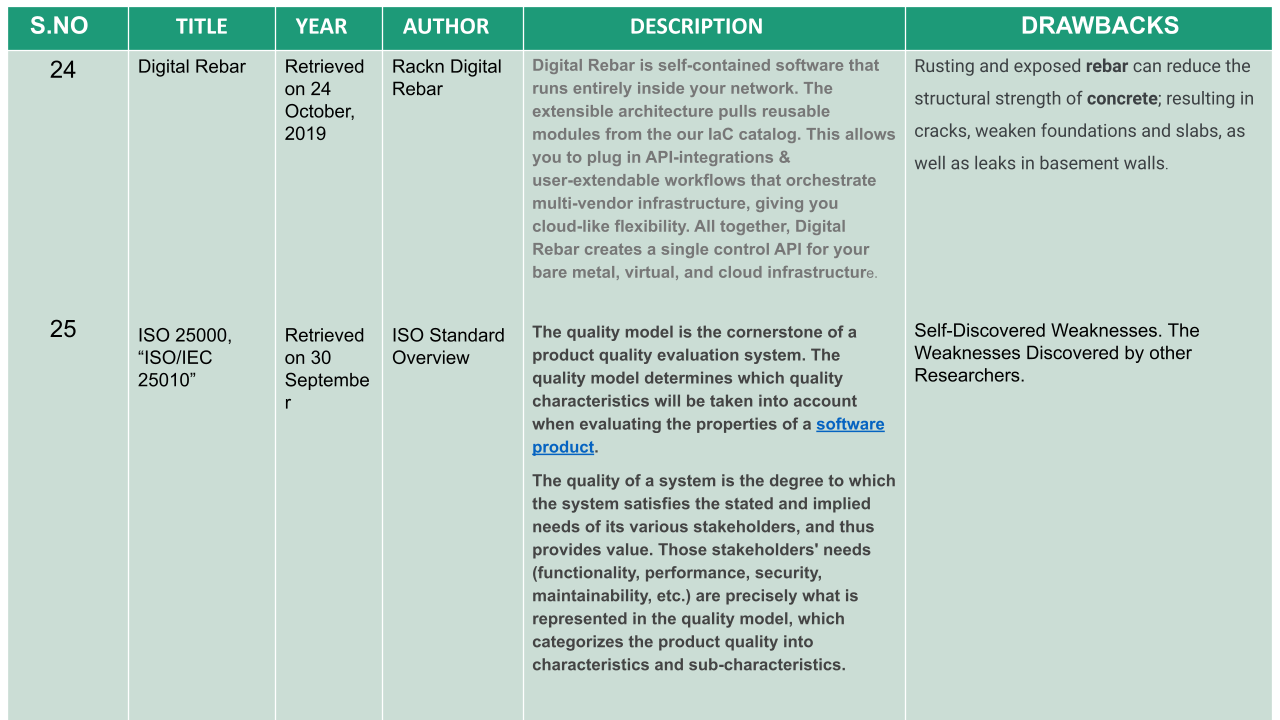


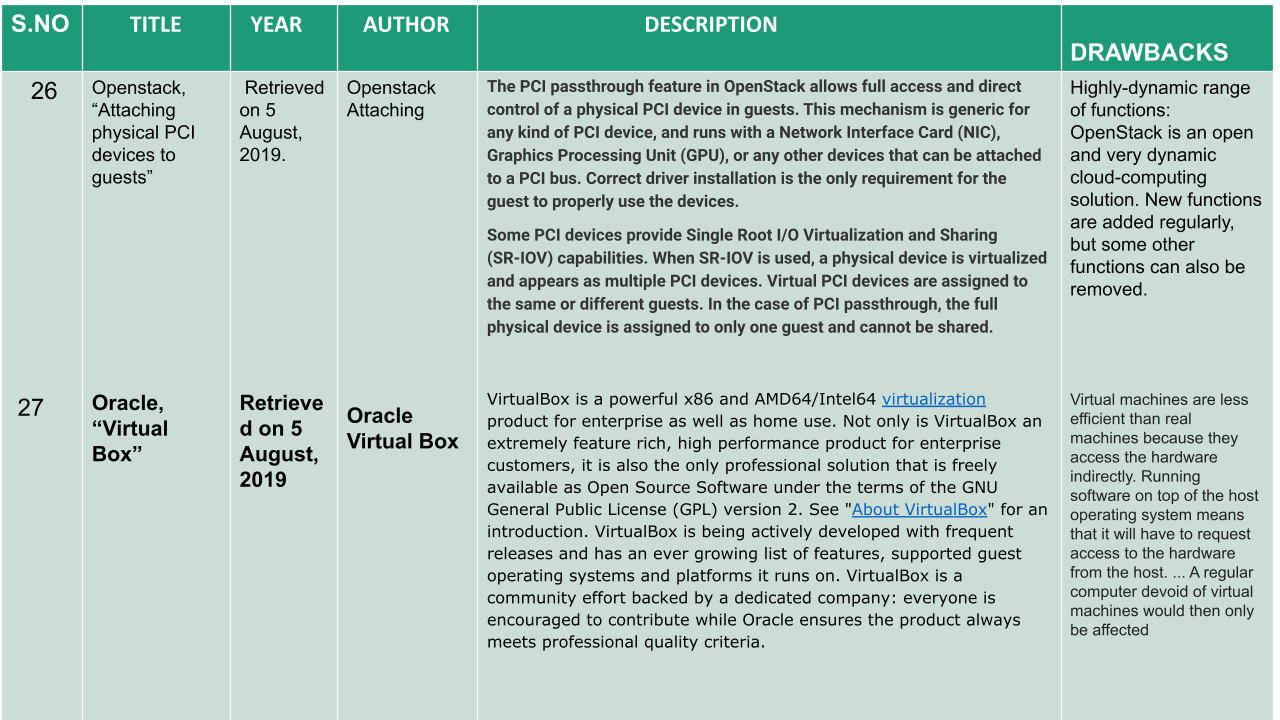












**CHAPTER 3**

**SYSTEM ANALYSIS**

**3.1 EXISTING SYSTEM**

As you manage your infrastructure with Terraform, you will create increasingly complex configurations. There is no intrinsic limit to the complexity of a single Terraform configuration file or directory, so it is possible to continue writing and updating your configuration files in a single directory. However, if you do, you may encounter one or more problems:Understanding and navigating the configuration files will become increasingly difficult.Updating the configuration will become more risky, as an update to one section may cause unintended consequences to other parts of your configuration.There will be an increasing amount of duplication of similar blocks of configuration, for instance when configuring separate dev/staging/production environments, which will cause an increasing burden when updating those parts of your configuration.You may wish to share parts of your configuration between projects and teams, and will quickly find that cutting and pasting blocks of configuration between projects is error prone and hard to maintain.In this tutorial, you will learn how modules can address these problems, the structure of a Terraform module, and best practices when using and creating modules.Then, over the course of these tutorials, you will use and create Terraform modules to simplify your current workflow .Managing cloud infrastructure and provisioning resources are often a headache that DevOps engineers are only too conversant in . Even the foremost capable cloud admins can get caught up with managing a bewildering number of interconnected cloud resources – including data streams, storage, compute power, and analytics tools.Take, for instance , the subsequent scenario: a customer has completed creating a Databricks workspace, and that they want to attach a Databricks cluster to a Redshift cluster in AWS. The diagram below demonstrates the resulting state if all of those steps are completed correctly, also as how data flows between each resource.Achieving this state can be a lengthy process, and each configuration step involves significant substeps.Scalability,Modularity,Consistency Lifecycle management .

**3.2PROPOSED SYSTEM**

This approach is suitable for giant projects, with clearly defined subdirectory structures containing multiple modules of varying levels of complexity, apart from the standard code. These modules can depend upon one another . including version control systems, these projects can make extensive use of workspaces. This approach is suitable for larger projects managing multiple apps, while reusing code the maximum amount as possible.Development, staging, quality assurance, and production infrastructure instances also can be housed under an equivalent project in several directories by counting on common modules, thus eliminating duplicate code and making the project the central source of truth. Here is that the file structure of an example project with a more complex structure, containing multiple deployment apps, Terraform modules, and target cloud environments.

**3.3 PREPARATION OF THE DESIGN:**

In this chapter, a specific case study is described that identifies the needs and requirements of an existing organization that is willing to improve their current systems with the use of the software-defined technology. Therefore, the purpose of this case study is to identify the required knowledge from a real-world organization and translate it into a list of requirements that express the desired functionality and behavior of a future fully software-defined system. This research was performed in cooperation with Thales Nederland, which is a mission critical organization that is in constant search of new technologies that can be used to improve their current systems in order to remain competitive in the market and provide high-end products to their customers. Therefore, mission critical organizations are highly interested in this recently emerged software-defined technology as a way to improve their current systems in regards to scalability, programmability and configurability.

**3.3.1 REQUIREMENTS:**

The first step towards the design and implementation of a fully software-defined system, is the definition of a list of requirements that the new system should fulfill in order to function properly and support all the mandatory use cases. Several meetings and discussions with the Thales engineers were required to acquire a better understanding of the currently used systems and compile a list of requirements that describe the desired future software-defined system. The resulting list of requirements (Table 3) mainly focuses on the infrastructure and the network layer of the system, with some additional general requirements. Based on the discussion with the Thales engineers and the personal understanding of the future system, the level of importance of each requirement was also identified and listed. These requirements are used in Chapter 5.3 as a basis for choosing the appropriate software-defined tools and technologies that have been used to design a prototype of the new software-defined system. The list of requirements includes some specific terminology that should be defined and explained in order to facilitate the interpretation of the requirements:

* System: The product of this research, which is a combination of software and hardware components that collaborate with each other in order to provide the functionality described by the requirements .
* Declarative approach : In this approach the user declares the desired state and structure of the model and a deployment engine enforces that state. The opposite is the imperative approach where the user has to explicitly describe in detail the required procedure to reach that state.

|  |  |  |  |
| --- | --- | --- | --- |
| **S.NO** | **Requirements** | **Level of Importance** | **Type** |
| 1. | The infrastructure layer of the system must follow a declarative approach . | High | Infrastructure |
| 2. | The network layer of the system must follow a declarative approach. | High | Network |
| 3. | The software components of the system must be implemented by using open source technologies. | High | General |
| 4. | The user of the system must be able to select between different kind of devices based on the capabilities of the available nodes (e.g., console nodes, server nodes, diskless nodes, switches, data diodes, firewalls, gateways). | High | Infrastructure |
| 5. | The user of the system must be able to determine for each node of the system at least the following capabilities: CPU architecture, amount of memory, number of cores, disk size, amount of network connections, GPU card and IO card. | High | Infrastructure |
| 6. | Each node in the system must have a unique identification, either MAC address, serial number or GUID. | High | Infrastructure |
| 7. | The user of the system should be able to select the OS that runs on the nodes of the system. | Medium | Infrastructure |
| 8. | The user of the system must be able to provision virtual machines, bare metal servers and containers. | High | Infrastructure |
| 9. | The system must support the network connection between the system nodes and the network devices. | High | Network |
| 10. | The system should support redundant connection of nodes (e.g., bonding). Medium | Medium | Network |
| 11. | The user of the system should be able to create multiple separated networks by selecting the desired network technology (e.g., VLAN, VXLAN). | Medium | Network |
| 12. | .The user of the system should be able to manage the network topology based on the minimum required bandwidth and the burst behavior of traffic. | Low | Network |
| 13. | The user of the system should be able to manage the network topology based on the quality of service (e.g. latency and jitter). | Low | Network |
| 14. | The system must support and manage the connection between physical and virtual networks. | High | Network |

**3.3.2 TECHNOLOGY USED:**

* **AWS -** Amazon Web Services (AWS) is the world’s most comprehensive and broadly adopteD cloud platform, offering over 200 fully featured services from data centers globally. Millions of customers—including the fastest-growing startups, largest enterprises, and leading government agencies—are using AWS to lower costs, become more agile, and innovate faster.
* **TERRAFORM-** Terraform is an [open-source](https://en.wikipedia.org/wiki/Open-source_software)  infrastructure as a code  software tool created by  Hashi corp. Users define and provide data center infrastructure using a declarative configuration language known as HashiCorp Configuration Language (HCL), or optionally JSON.
* **SHELL-** In computing , a shell is a computer program which exposes an operating system’s services to a human user or other program. Command-line shells require the user to be familiar with commands and their calling syntax, and to understand concepts about the shell-specific scripting language .
* **MAKEFILE-**In software development, Make is a build automation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix and Unix-like operating systems.

**3.3.3 FUNCTIONAL REQUIREMENTS**

* 1. **S SELECTION OF TOOL:**
  2. The selection of tools that are used in the designed system is mainly based on the list of requirements . Requirements with high level of importance are the main factors in the selection process, while requirements with medium or low importance level have lower priority in this procedure. The order of selection is based on the design steps . A complete list of the selected tools can be found in below Table .

|  |  |  |
| --- | --- | --- |
| **Scope** | **Tool** | **Reasoning** |
| Provisioning | Terraform | Open source , Supports multiple dynamic platforms. The alternative options are vendor specific solutions. |
| Dynamic Platform | Openstack (VMs)      Cobbler (Bare Metal) | * Open source * Private cloud solution * Extensive support by Terraform * Provides SDN functionality (Neutron)   Open source.Supported by Terraform.Alternatives are proprietary solutions that do not offer private functionality. |
| SDN Controller | OpenDaylight | Open source .  Support of wide variety of southbound protocols, including Openflow, OVSDB, NETCONF , Openstack compatible |

**PROVISIONG TOOLS:**

Provisioning is the most important step for building a fully functional IT infrastructure. Therefore, the choice of the provisioning tool is critical for the entire software-defined system and should focus on offering scalability and maintainability to the system. A comparison of the available provisioning tools is available . Terraform is the most suitable solution from the list, as it is an open source (Requirement 3) and declarative solution (Requirement 1) that uses the HCL files to specify the desired state of the infrastructure and then uses APIs to reach that state. In addition, Terraform supports resources from multiple cloud providers simultaneously, and is capable for supporting several custom in-house solutions. As a result, the system has support for multiple dynamic platforms (e.g., Openstack, AWS, Azure, Google Cloud Platform) within the same HCL file which offers a wide range of infrastructure resources to build on, leading to more dynamic and scalable systems. Other provisioning tools are closed source (e.g., CloudFormation, Cloud Deployment Manager, Azure Manager) and vendor specific solutions (e.g., Openstack Heat), which automatically bonds the system to a specific platform and limits the number of available building resources.

**DYNAMIC TOOL:**

The dynamic platform offers the programmable resources that are used by the provisioning tool to set up the IT infrastructure. The most suitable platform for this specific use case is Openstack [49]. Public cloud platforms such as AWS, Google Cloud and Azure cannot serve the requirements of this particular use case, as the new software-defined system should be an open source private solution that could meet the privacy and security aspects of a mission critical organization such as Thales, and is located and managed locally by the engineers of the organization. The alternative cloud solutions can provide the usage and configuration of private servers that are specifically assigned to a customer by the vendor, however these servers are still located and managed at the central datacenters of the vendor.

Openstack is an open source software that can create private clouds that are managed and deployed locally. It is a free platform that is backed up by several large enterprises, and has strong community support that is constantly working on updates and new releases. Openstack can manage the pools of computing, storage and networking resources via the Openstack API or the Horizon dashboard . Openstack is highly modular, as it can run with a minimum set of core services or function with a variety of additional services. It also supports communication with a wide range of external cloud 42 providers such as AWS and Google Cloud Engine. In addition, it supports an extensive list of hypervisors, such as KVM, Xen and VMware ESXi, making it optimal for a heterogeneous IT infrastructure. Moreover, most of the Openstack resources are supported by Terraform, which is an important reason that led to the choice of Openstack. Another significant argument for selecting Openstack, is the support of the SDN technology. Neutron is the SDN Openstack project that provides networkingas-a-service (NaaS) in virtual compute environments. Furthermore, Openstack Magnum is an Openstack project that enables the use of container orchestrators such as Kubernetes in Openstack. With Magnum, the user can create entire clusters of containers that are managed by one of the supported orchestrators.

The new software-defined system, in addition to VMs and containers, should also be capable of provisioning bare metal machines (Requirement 8). Openstack supports bare metal provisioning with an integrated service, called Ironic . However, the Ironic features are not supported by the current release of Terraform. For this reason, another solution is required for provisioning bare metal machines. Cobbler was selected as the most suitable tool to fulfill that role. Cobbler is an open source software that automates the provisioning process of multiple bare metal servers with the use of services such as DHCP, TFTP and DNS. The PXE interface (Preboot Execution Environment) is used to provision the bare metal servers from a central administration point. Several main features of Cobbler, such as the creation and deletion of bare metal serves, are currently supported by Terraform, which is the distinctive factor compared to other bare metal provisioning tools such as Foreman.

**CONFIGURATION TOOLS:**

Configuration tools are used to configure and manage the already provisioned resources with the required dependencies and settings. A comparison of the most used configuration tools is available at Table 2 in Section 2.4.5, and Ansible is the preferred configuration software for this specific use case.

Ansible was selected over Chef, Puppet and Saltstack as it offers agent-less deployment which adds speed and reliability to the system, compared to the master-agent deployment model of Chef and Puppet. In addition, the agent-less model eliminates points of failure and several performance problems in the system, and the use of SSH for the communication between the nodes improves the overall security of the infrastructure. Moreover, Ansible offers a fast and easy installation and overall is a fairly easy solution for new users. Ansible can also support the dynamic nature of Cobbler and Openstack, where new hosts are constantly spinning up and shutting down in response to the demands of the systems. Ansible can serve that demand with the use of dynamic inventories , which come into the form of scripts that can be added to Cobbler and Openstack directories to keep track of the infrastructure. Thales has already been using Ansible for configuring servers and fulfills the requirements of their current infrastructure adequately. The choice of Ansible makes it unnecessary for Thales to migrate to another configuration software, saving the required additional time and effort for learning a new technology.

**CHAPTER 4**

**SYSTEM DESIGN**

**4.1 Use Case Diagram:**

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. The standard is managed and was created by the Object Management Group. UML includes a set of graphic notation techniques to create visual models of software intensive systems. This language is used to specify, visualize, modify, construct and document the artifacts of an object oriented software intensive system under development.

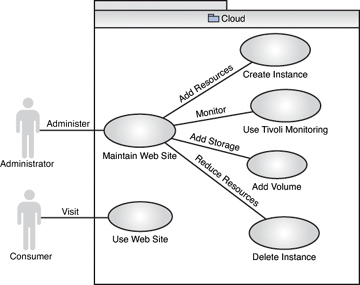
**4.1.1. USECASE DIAGRAM**

A Use case Diagram is used to present a graphical overview of the functionality provided by a system in terms of actors, their goals and any dependencies between those use cases.

Use case diagram consists of two parts:

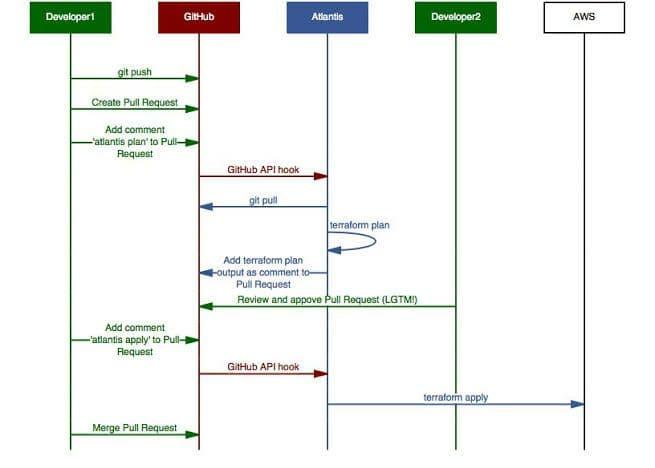
**Use case:** A use case describes a sequence of actions that provided something of measurable value to an actor and is drawn as a horizontal ellipse.

**Actor:** An actor is a person, organization or external system that plays a role in one or more interaction with the system.



**4.2 Sequence Diagram:**

A Sequence diagram is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of Message Sequence diagrams are sometimes called event diagrams, event sceneries and timing diagram.

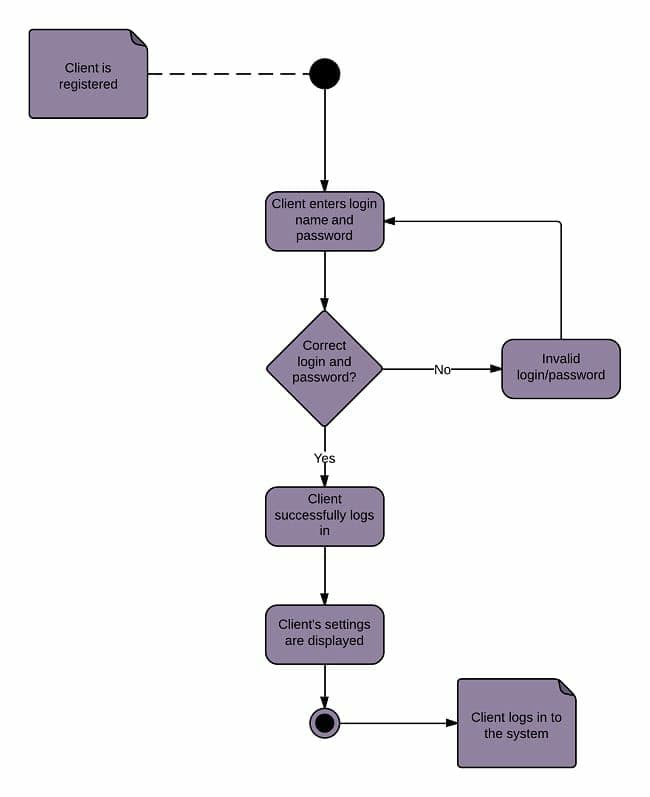
****

**4.3 Activity Diagram:**

Activity diagram is a graphical representation of workflows of stepwise activities and actions with support for choice, iteration and concurrency. An activity diagram shows the overall flow of control.

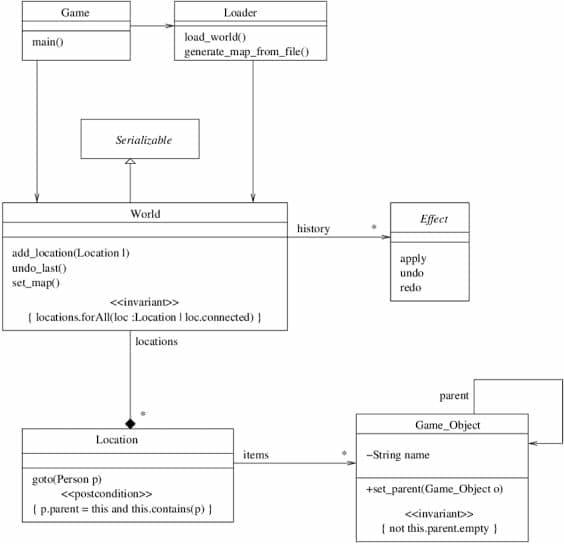
The most important shape types:

* Rounded rectangles represent activities.
* Diamonds represent decisions.
* Bars represent the start or end of concurrent activities.
* A black circle represents the start of the workflow.
* An encircled circle represents the end of the workflow.



**4.4 Class Diagram**

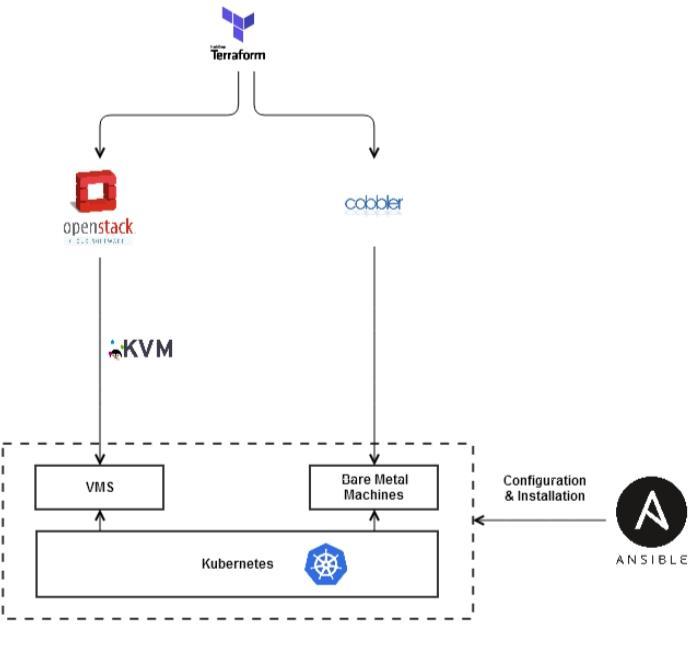
A Class diagram in the Unified Modeling Language is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

****

**CHAPTER 5**

**SYSTEM ARCHITECTURE**

**5.1 Architecture overview**

****

The purpose of this section is to map the described selected tools from the previous chapters to the generic components described and explain the interconnections between them in order to realize the architecture of the infrastructure layer. The architecture realization of the supported infrastructure layer . This suggested architecture is supported by the current available features of the selected tools. Terraform is the main provisioning tool, which uses the Openstack provider to create VMs and the Cobbler provider to provision bare metal machines for high-end applications that require real hardware.

Openstack has support to a wide variety of hypervisors, and KVM is illustrated as default. Cobbler provisions the bare metal servers through the network using the PXE standard. Ansible is used for the configuration and the installation of software on the provisioned servers (VMs and bare metal).

The support of containerized applications is mandatory (Requirement 8), and Kubernetes should be installed on the computing machines. There are two possible ways to create a Kubernetes cluster, namely with Openstack Magnum and Ansible. With the use of Openstack Magnum, the user can create Kubernetes clusters that are deployed on both VMs and bare metal machines. However, the deployment on bare metal machines is not available, as Ironic is not supported by Terraform in the current release. The other solution for supporting Kubernetes is via Ansible, which should install Kubernetes on the VMs and the bare metal machines, and then Kubernetes can be instructed and configured via the Kubernetes provider in Terraform. The Ansible 52 approach is preferred, as it offers consistency on the configuration of the system while one tool is used for the configuration of VMs and bare metal machines. In addition, Ansible can install Kubernetes with a wide variety of network drivers, whereas Magnum can only install Kubernetes with Calico or flannel as network drivers, which is quite restrictive for a number of use cases.

The future possibly improved architecture for the infrastructure layer. In this architecture, Openstack is responsible for both the provisioning of VMs and bare metal machines. Cobbler is replaced by Ironic, which is the Openstack service for provisioning bare metal machines. The Ironic features are not supported by the current release of Terraform, however the Terraform community is working on improvements and releases a new version of the Openstack provider almost once per month. Based on the rate of releases, the integration of Ironic into the Openstack provider of Terraform is highly possible in the upcoming future. The creation of VMs follows the same steps as in Figure 15, while Ansible is still used for the configuration of the provisioned machines. However, in this case, the Kubernetes cluster can be deployed both on bare metal and VM servers by Openstack Magnum with the cooperation of Ironic for provisioning the bare metal clusters.

**5.2. SYSTEM ARCHITECTURE:**

The design of the software-defined system is a challenging process as it involves several new technologies that are highly dependent on each other in order to produce a fully functional system. The distinction of infrastructure layer and network layer facilitates and simplifies the design process, as the tool selection has been performed to fulfill the requirements at each layer. The design of the infrastructure layer is firstly described in this chapter, since it is the basis for designing a functional software-defined network, which is explained .

**5.2.1 MODULES**

* **Provisioning**
* **Creating Virtual Machines**
* **Provisioning Bare Metal Machines**
* **Provisioning Containers**

**MODULE EXPLANATION:**

**Provisioning :**

The initial step towards the development of an IT infrastructure is the provisioning of the required components and resources, such as VMs and bare metal machines. Terraform is the selected provisioning tool and can represent a large variety of infrastructure components with the form of Terraform resources. Terraform provisions the resources of a dynamic platform, and a Terraform provider is used to interact with the APIs and expose the resources from the corresponding dynamic platform. Terraform allows its users to declare resources from different providers in the same or in different HCL files. In this particular use case, the Openstack and Cobbler providers are used for provisioning VMs and the bare metal machines, respectively. Openstack should be installed on the entire data center of the organization, while Terraform and Cobbler should be installed on the central control nodes for building the infrastructure**.**

**Creating Virtual Machines :**

Terraform creates VMs [57] in Openstack by combining several resources from the Openstack provider. The user has a wide variety of options for the configuration of an VM instance in Openstack . Amongst others, the user is able to select:

 VM Name: Unique name that characterizes the VM instance.

 Image: Installed OS on the VM. The user can create several different images [59] using a separate Terraform resource. The names of these images are later used during the creation of the VMs. The configuration of an image includes:

 Name of the image

 OS URL: Url address of the preferred OS that Openstack downloads and uses.

 Image Format: E.g., iso, qcow2, vhd, etc.

 Hypervisor: Type of hypervisor that is used to create the VM. (e.g., qemu, vmware, xen, lxc)

 CPU Architecture of the VM : E.g., x86\_64, arm, ppc65

 Flavor: Capabilities of the VM. The user can create several flavors and based on the configuration of the flavor, the user can define several types of nodes for the system (e.g., central server nodes, console nodes, diskless nodes etc.). The configuration of a flavor includes:

 Name of the flavor

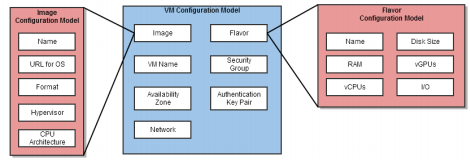
 RAM: Size of the available RAM on the VM

 CPU: Number of virtual cores on the VM

 Disk Size: Capacity of the VM

 GPU: Number of virtual GPUs on the VM

 I/O: Number of physical I/O that the VM can use.



 Network: Network where the VM is assigned to.

 Security Group: Group of network access rules that control the traffic types that interact with an instance.

 Availability Zone : Groups of compute hosts that are responsible for launching the VM instances.

 Authentication Key Pair: Public keys that are used to access a created VM.

**Provisioning Bare Metal Machines :**

Terraform uses the Cobbler provider to provision bare metal machines. Cobbler performs provisioning by using the PXE standard to boot the machines over the network. PXE works with the Network Interface Card (NIC) of the machine by transforming it to a boot device. The NIC of the client broadcasts a request to the DHCP server, which responds with the IP address of the client, the address of the TFTP server and the location of the boot files on the TFTP server. After receiving these data, the client connects with the TFTP server in order to receive the boot image. The TFTP server responds by sending the boot image and the client executes it. The boot image searches by default the PXE configuration directory on the TFTP server, seeking the boot configuration files. After finding the required files, the client downloads them and loads them in order to start the installation. The DHCP and the TFTP services are both managed by Cobbler, which is based on a specific set of objects for the provisioning process. The basic Cobbler objects are:

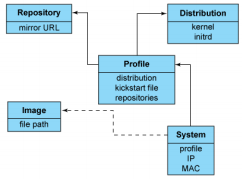
 Distribution: It basically describes the OS. It includes details about the kernel and the initrd and some other kernel related data.

 Profile: It indicates a distribution, a kickstart file and occasionally repositories.

 System: It indicates the machine for provisioning. It uses profiles for configuring the machine and holds information about IP and MAC addresses.

 Repository: It carries mirroring data for a yum or rsync repository.

 Image: It points to the file path where the OS is located. The image object is used to replace the distribution object in case specific files cannot be divided into the kernel and initrd categories.



Terraform uses the Cobbler provider to translate all the Cobbler objects to the corresponding Terraform resources for provisioning bare metal machines . The system object is the main element during the configuration, as it determines the desired machine for provisioning. The configuration of a system resource includes:

 Name of the system resource

 Profile: The name of the preferred profile for the provisioned machine. The user can create several profiles by using the respective Terraform resource. The configuration of a Terraform profile resource includes:

 Name for the profile resource

 Name of the preferred distribution: Distributions are a separate Terraform resource. The configuration of the distribution includes:

 Name for the distribution

 OS Breed: E.g., Redhat, Fedora, CentOS, Ubuntu

 OS Version: E.g., trusty

 OS Architecture: E.g., i385, x86\_64, ia64, etc.

 Kernel Path: Path in the filesystem that indicates the kernel files.

 Initrd Path: Path in the filesystem that indicates the initrd files.

 Path to the selected kickstart file

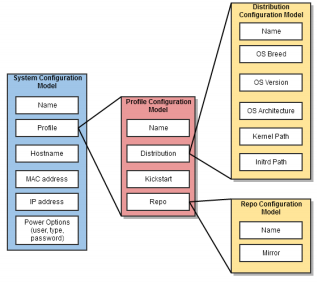
**** Name of the repos resource: The user creates repos with a separate Terraform resource and declares the name of the preferred repo resource during the configuration of the profile. The repo resource includes a name for the resource and a url address for the yum or rsync mirror.

 Hostname: The hostname of the machine after provisioning.

 MAC address: The MAC address of the machine that is selected for provisioning.

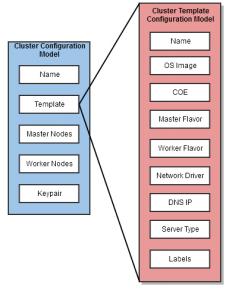
 IP address: The preferred IP address of the machine after provisioning.

 Power options: Aspects regarding power management, such as the type, the user and the password.



**Provisioning Containers :**

Magnum is the Openstack API service that enables the creation and management of container clusters in Openstack. Magnum offers container orchestration engines (COE) as first class resources in Openstack for the control and management of the clusters. Openstack Heat is used by Magnum for orchestrating the OS image that contains Docker and the selected COE, and runs that image on virtual machines or bare metal machines in a cluster configuration. The preferred COE for this case study is Kubernetes, as it has surged in popularity in the past several years and it is currently one of the biggest open source communities (more than 58.000 star in GitHub . The descripted Magnum features are integrated in Terraform. The configuration of a cluster in Terraform is shown in Figure 14 and it includes:



 Name of the cluster

 Template: The name of the used template for the cluster. A template describes the parameters of the cluster. The user can create several cluster templates using a separate resource. The template configuration consists:

 Name of the template

 COE: The name of the preferred orchestration engine for the cluster. The supported COEs are Kubernetes, Docker Swarm and Apache Mesos.

 OS Image: The OS that is installed on the nodes of the cluster. The supported images differ based on selected COE. Fedora-atomic and CoreOS are the only available for Kubernetes cluster, while Fedora-atomic and Ubuntu are the available images for Docker Swarm and Apache Mesos, respectively.

 Master flavor: The capabilities of the master nodes in the cluster.

 Worker flavor: The capabilities of the worker nodes in the cluster.

 Network driver: Driver that performs networking between the nodes of the cluster (e.g., flannel, calico).

 Server type: The type of the server of the cluster. The user can select between virtual and bare metal servers, however Ironic, which is the bare metal provisioning service of Openstack, is not supported by Terraform. As a result, Magnum cannot create bare metal clusters using Terraform.

 DNS IP: IP address of the DNS server for the cluster.

 Labels: Important external features based on the selected COE (e.g., monitoring features, such as Prometheus with Grafana and dashboard feature for Kubernetes).

 Master Nodes: Number of master nodes in the cluster

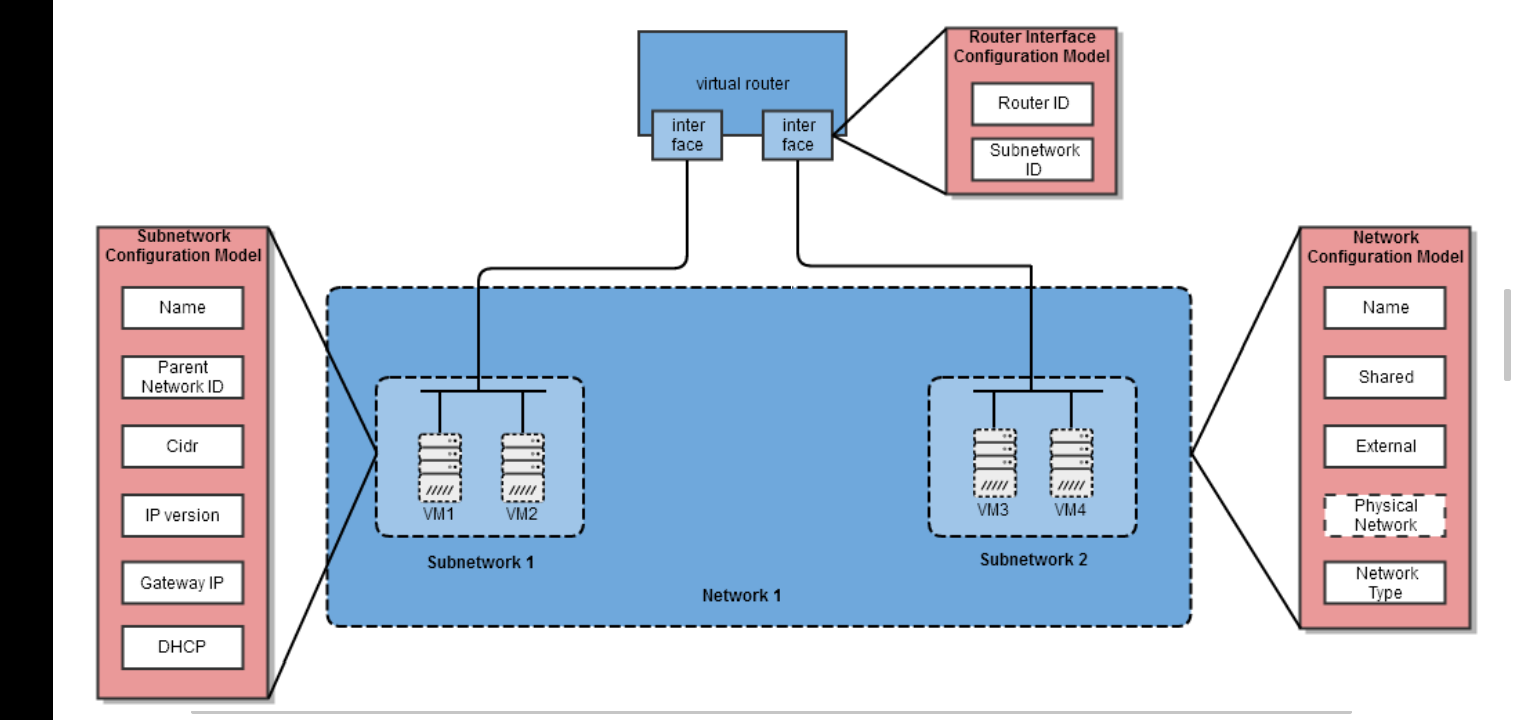
 Worker Nodes: Number of worker nodes in the cluster

 Keypair: Used for the secure communication between the cluster nodes (e.g., ssh key pair).

**5.3** **Network Design :**

The creation of a programmable and scalable network infrastructure that is centrally controlled, is essential for obtaining a complete software-defined system. The softwaredefined network layer should be able to create virtual networks, offer connectivity among them, create virtualized network functionalities (e.g., firewalls) and offer connectivity between the created virtual networks and the current physical ones. Neutron is the integrated Openstack service that offers networking as a service, and most of the networking features of Neutron are supported by Terraform.

**5.3.1 Creating virtual networks :**

The configuration models for the supported network components of the Openstack provider are displayed in Figure 17. The user should first create and configure a network in Openstack. The configuration of an Openstack network with Terraform includes:

 Name: A unique name for the created network.

 Shared: The administrator can define whether the network is accessible by any other users/ tenants.

 External: The user specifies whether the network has external routing facility.

 Physical network (Optional): The IP of the physical network that this network resource would be mapped to. These networks are called provider networks, while the default Openstack networks that are not mapped to a physical network are called tenant networks.

 Network type: E.g., VLAN, Flat, GRE Subnetworks can be created on top of a network with a different Terraform resource. The subnetworks host the compute instances (e.g., VMs) of the infrastructure. The configuration model of a subnetwork resource includes:

 Name: A unique name for the subnetwork.

 Parent network ID: The ID of the network that this subnetwork belongs to.

 Cidr: The IP range of this subnetwork based on the IP version.

 IP version: E.g., IPv4 or IPv6

 Gateway IP: The IP address of the default gateway used by the instances of this network.

 DHCP: The user can determine whether the DHCP name server is enabled on this subnetwork. Instances in the same network have by default layer 2 connectivity to each other. In case it is necessary to have layer 3 connectivity between instances connected to different networks, a virtual router should be created. The Openstack provider offers a separate resource for creating virtual routers . The subnetworks are connected to a virtual router by using the router interfaces , which are a separate Terraform resource. Two router interface resources should be created to connect two subnetworks with one router. The first interface resource would connect the router with the first subnetwork and the second interface would connect the other subnetwork with the same router. The subnetworks are connected to the same router, which connects them to each other. Neutron uses the L2 and the L3 agents to perform the required networking and agents are facilitated by the Open vSwitch technology. The configuration of a router interface includes the IDs of the router and the subnetwork that should be connected. The L3 agent, except the virtual router, also offers a service called floating IP . Floating IPs are assigned to instances, making them accessible from external public networks. As a result, an Openstack instance can have a private IP and a floating IP. The private IP is mostly used internally to grant access between the instances in a tenant network, while the floating IP is used to access the instance from public networks.

**5.3.2 Securing virtual networks :**

Openstack uses a collection of network access rules to limit the types of traffic that can interact with an instance. This collection of rules is called security group. An instance can have one or more assigned security groups. The rules in a security group control the traffic that is allowed to an instance and if an incoming traffic does not match with a rule from the group, is denied by default.

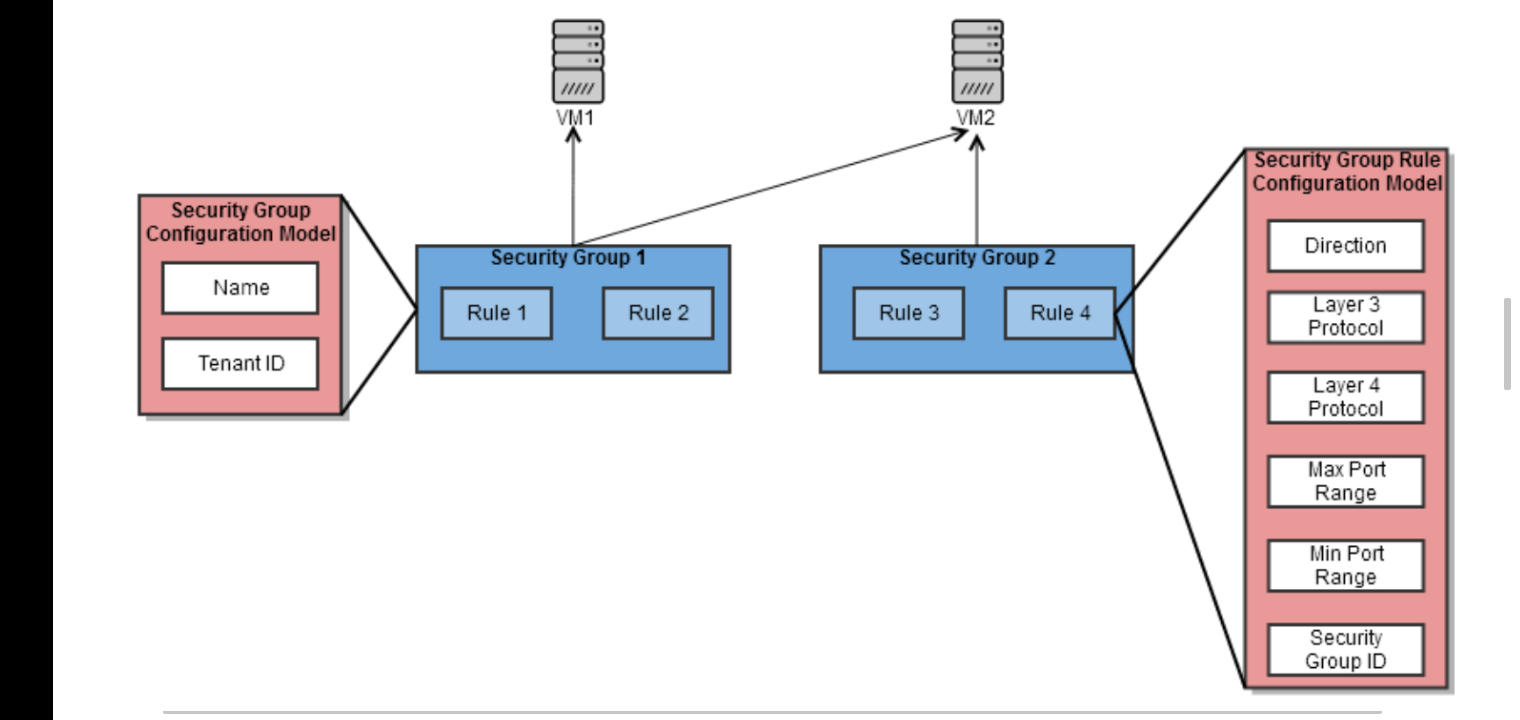


Figure 18 illustrates the configuration options during the creation of a security group with Terraform. Terraform uses two different resources in order to create security groups and rules. The configuration of a security groups [81] includes a unique name for the security group and the ID of the tenant/ user that can use this security group. The configuration of a security group rule [82] is more complex, and consists of:  Direction: The type of the traffic (e.g., ingress or egress)

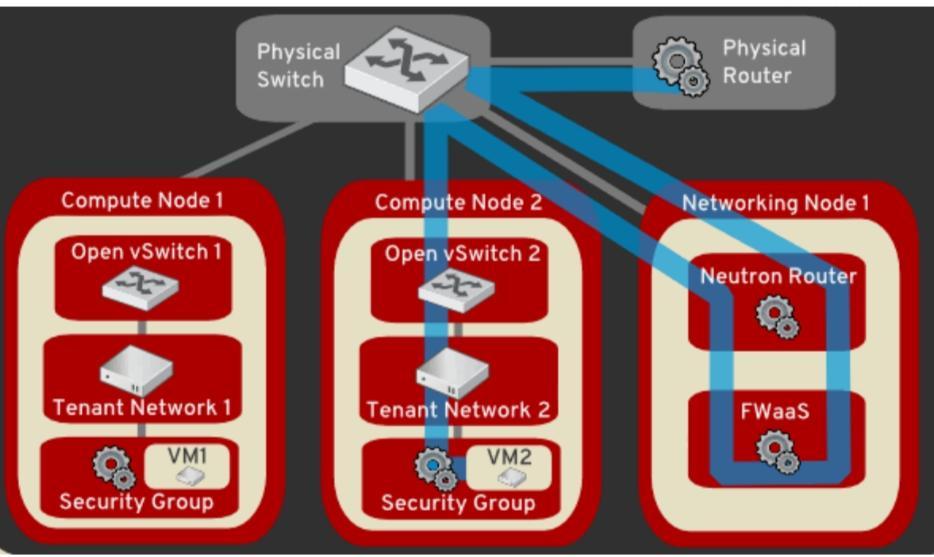
 Layer 3 protocol: E.g., IPv4 or IPv6

 Layer 4 protocol: E.g., TCP, UDP, ICMP

 Max port range: The allowed maximum port range.

 Min port range: The allowed minimum port range.

 Security group ID: The ID of the security group that this rule will be part of Openstack can also secure layer 3 networking by deploying Firewalls-as-a-Service (FWaaS). Firewalls are deployed on the virtual routers, while security groups operate on instance level (Figure 19). The main elements of an Openstack firewall are the rules and the policies, which are basically an ordered collection of rules. Firewalls operate differently based on the used driver. For instance, a firewall that uses IP tables as a driver, would use IP table rules, while an Open vSwitch driver implements firewalls using flow entries in the flow tables.



​

​

**CHAPTER 6**

**SYSTEM IMPLEMENTATION**

**6.1 Coding**

**BACKEND CONFIGURATIONS:**

**1. BACKEND TEMPLATE:**

region = "eu-west-1"

bucket = "@STATE\_BUCKET\_NAME@"

key = "backup\_state/terraform.tfstate"

dynamodb\_table = "@STATE\_LOCK\_DDB@"

encrypt = true

kms\_key\_id = "@KMS\_KEY\_ARN@"

**2.DEV-BACKEND:**

region = "eu-west-1"

bucket = "kat-infra-dev-infra-state"

key = "backup\_state/terraform.tfstate"

dynamodb\_table = "lock-kat-infra-dev-infra-state"

encrypt = true

kms\_key\_id = "arn:aws:kms:eu-west-1:191582817451:key/972c8c4d-1b04-4e48-a13e-cc1f5e2a9c3b"

**3.STAGING – BACKEND:**

region = "eu-west-1"

bucket = "kat-infra-staging-state"

key = "backup\_state/terraform.tfstate"

dynamodb\_table = "lock-kat-infra-staging-state"

encrypt = true

kms\_key\_id = "arn:aws:kms:eu-west-1:925919524744:key/4be8069e-d210-491a-beb4-de07eadfa6df"

**SCRIPTS:**

**BACKEND INIT SCRIPT:**

set -ie

project\_name=${1}

environment=${2}

region=${3}

kms\_key\_arn=${4}

if [ -z "${project\_name}" ];

then

echo "Project name not provided";

exit 1;

fi

if [ -z "${environment}" ];

then

echo "Environment not provided";

exit 1;

fi

name\_with\_env=${project\_name}-${environment}-state

# if region not set, default to ireland region

if [ -z "${region}" ];

then

region=eu-west-1

fi

if [ -z "${kms\_key\_arn}" ];

then

echo "KMS key not provided";

exit 1;

fi

# Create bucket

aws s3api create-bucket --bucket ${name\_with\_env} --region ${region} --create-bucket-configuration LocationConstraint=${region}

aws s3api put-bucket-versioning --bucket ${name\_with\_env} --versioning-configuration '{"Status": "Enabled"}'

aws s3api get-bucket-versioning --bucket ${name\_with\_env}

# Create dynamoDB

aws --region ${region} dynamodb create-table --table-name lock-${name\_with\_env} \

--attribute-definitions AttributeName=LockID,AttributeType=S \

--key-schema AttributeName=LockID,KeyType=HASH \

--provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5

# aws --region ${region} dynamodb update-continuous-backups --table-name lock-${name\_with\_env} \

# --point-in-time-recovery-specification PointInTimeRecoveryEnabled=true

echo "${name\_with\_env}";

echo "lock-${name\_with\_env}";

echo ${EIPALLOC}

echo "Bucket created named: ${name\_with\_env}"

echo "DynamoDB created named: lock-${name\_with\_env}"

if [ -e ../backend\_configs/${environment}\_backend ]

then

echo "File exists. Modify existing backend configuration for environment: ${environment}";

exit 1;

else

echo "File doesn't exist. Creating new backend configuration...";

sed "s|@STATE\_BUCKET\_NAME@|${name\_with\_env}|g;

s|@STATE\_LOCK\_DDB@|lock-${name\_with\_env}|g;

s|@KMS\_KEY\_ARN@|${kms\_key\_arn}|g" \

../backend\_configs/backend\_template > ../backend\_configs/${environment}\_backend fi

echo "New backend config file: backends/${environment}\_backend use it to initilise terraform backend. See README.md"

region = "eu-west-1"

bucket = "kat-infra-dev-infra-state"

key = "ec2-linux/terraform.tfstate"

dynamodb\_table = "lock-kat-infra-dev-linux-state"

encrypt = true

kms\_key\_id = "arn:aws:kms:eu-west-1:191582817451:key/972c8c4d-1b04-4e48-a13e-cc1f5e2a9c3b"

ssh\_key\_name = "kat-dev"

tags = {

Terraform = "true"

ServiceName = "kat-svc"

ServiceType = "kat"

Environment = "dev"

}

terraform {}

provider "aws" {

# assume\_role {

# role\_arn = var.assume\_role\_arn

# session\_name = "EKS\_deployment\_session\_${var.tags["Environment"]}"

# }

# version = "~> 2.69.0"

region = var.region

}

data "aws\_ami" "amazon\_linux" {

most\_recent = true

owners = ["amazon"]

filter {

name = "name"

values = [

"amzn2-ami-hvm-\*-x86\_64-gp2",

]

}

filter {

name = "owner-alias"

values = [

"amazon",

]

}

}

resource "aws\_launch\_template" "ec2-linux" {

name\_prefix = "launch-template-${var.tags["ServiceType"]}-${var.tags["Environment"]}-ec2-linux-"

image\_id = data.aws\_ami.amazon\_linux.id

key\_name = var.ssh\_key\_name

instance\_type = "t2.micro"

iam\_instance\_profile {

name = aws\_iam\_instance\_profile.ec2-linux.name

}

network\_interfaces {

associate\_public\_ip\_address = true

delete\_on\_termination = true

security\_groups = [aws\_security\_group.ec2-linux.id]

}

tags = var.tags

lifecycle {

create\_before\_destroy = true

}

}

resource "aws\_autoscaling\_group" "ec2-linux" {

name\_prefix = "${var.tags["ServiceType"]}-${var.tags["Environment"]}-ec2-linux-asg-"

desired\_capacity = 1

max\_size = 1

min\_size = 1

vpc\_zone\_identifier = ["subnet-0e99063b1d226437b","subnet-070e7a6ef132003ae"]

# vpc\_zone\_identifier = flatten([module.vpc.public\_subnets])

force\_delete = true

termination\_policies = ["OldestInstance"]

launch\_template {

id = aws\_launch\_template.ec2-linux.id

version = "$Latest"

}

tag {

key = "Name"

value = "${var.tags["ServiceType"]}-${var.tags["Environment"]}-ec2-linux"

propagate\_at\_launch = true

}

dynamic "tag" {

for\_each = var.tags

content {

key = tag.key

value = tag.value

propagate\_at\_launch = true

}

}

timeouts {

delete = "15m"

}

lifecycle {

create\_before\_destroy = true

}

depends\_on = [aws\_launch\_template.ec2-linux]

}

resource "aws\_security\_group" "ec2-linux" {

name = "Allow ssh access"

vpc\_id = "vpc-0e425a80e20fabfc2"

#vpc\_id = "vpc-09bca9f9fa35f52c6"

ingress {

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

tags = var.tags

lifecycle {

create\_before\_destroy = true

}

}

resource "aws\_iam\_instance\_profile" "ec2-linux" {

name = "${var.tags["ServiceType"]}-${var.tags["Environment"]}-ec2-linux-instance-profile"

role = aws\_iam\_role.ec2-linux.name

}

resource "aws\_iam\_role" "ec2-linux" {

name = "${var.tags["ServiceType"]}-${var.tags["Environment"]}-ec2-linux-role"

path = "/"

assume\_role\_policy = <<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Action": "sts:AssumeRole",

"Principal": {

"Service": "ec2.amazonaws.com"

},

"Effect": "Allow"

}

]

}

EOF

tags = var.tags

}

resource "aws\_iam\_policy" "ec2-linux" {

name = "${var.tags["ServiceType"]}-${var.tags["Environment"]}-ec2-linux-policy"

policy = <<EOF

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"ec2:DescribeAddresses",

"ec2:AllocateAddress",

"ec2:DescribeInstances",

"ec2:AssociateAddress",

"ec2:DisassociateAddress"

],

"Resource": "\*"

}

]

}

EOF

}

resource "aws\_iam\_role\_policy\_attachment" "ec2-linux" {

role = aws\_iam\_role.ec2-linux.name

policy\_arn = aws\_iam\_policy.ec2-linux.arn

}

resource "aws\_iam\_role\_policy\_attachment" "route53" {

role = aws\_iam\_role.ec2-linux.name

policy\_arn = "arn:aws:iam::aws:policy/AmazonRoute53FullAccess"

}

#####

# Outputs

#####

output "ec2-linux\_host\_instance\_profile\_arn" {

value = aws\_iam\_instance\_profile.ec2-linux.arn

}

output "ec2-linux\_host\_policy\_arn" {

value = aws\_iam\_policy.ec2-linux.arn

}

output "ec2-linux\_host\_autoscaling\_group\_arn" {

value = aws\_autoscaling\_group.ec2-linux.arn

}

output "ec2-linux\_host\_launch\_template\_arn" {

value = aws\_launch\_template.ec2-linux.arn

}

variable "ssh\_key\_name" {

default = "kat-dev"

}

variable "tags" {

type = map(string)

description = "Default tags attached to all resources."

default = {

ServiceType = "kat-eks"

}

}

variable "region" {

type = string

default = "us-east-1"

description = "AWS region in which resources will get deployed. Defaults to Ireland."

}

vpc\_cidr = "10.1.0.0/18"

private\_subnets\_cidrs = ["10.1.0.0/20", "10.1.16.0/20"]

public\_subnets\_cidrs = ["10.1.48.0/22", "10.1.52.0/22"]

eks\_enabled\_log\_types = ["api", "audit", "authenticator", "controllerManager", "scheduler"]

# log

vpc\_single\_nat\_gateway = true

vpc\_one\_nat\_gateway\_per\_az = false

ssh\_key\_name = "kats"

eks\_version = "1.16"

tags = {

Terraform = "true"

ServiceName = "kat-eks-dev"

ServiceType = "kat-eks"

Environment = "dev"

KubernetesCluster = "kat-eks-dev"

}

terraform {}

provider "aws" {

# assume\_role {

# role\_arn = var.assume\_role\_arn

# session\_name = "EKS\_deployment\_session\_${var.tags["Environment"]}"

# }

# version = "~> 2.69.0"

region = var.region

}

variable "region" {

type = string

default = "us-east-1"

description = "AWS region in which resources will get deployed. Defaults to Ireland."

}

variable "availability\_zones" {

type = list(string)

default = ["us-east-1a", "us-east-1b", "us-east-1c"]

description = "Availability zones for the default Ireland region."

}

variable "ec2\_instance\_types" {

type = list(string)

description = "Bastion instance types used for spot instances."

default = ["t3.nano", "t3.micro", "t3.small", "t3.medium", "t3.large", "t2.nano", "t2.micro", "t2.small", "t2.medium", "t2.large"]

}

variable "worker\_instance\_types" {

type = string

description = "Worker instance types used for spot instances."

default = "m5.4xlarge,m5d.4xlarge,m5a.4xlarge,m5ad.4xlarge,m4.4xlarge,m5n.4xlarge,m5dn.4xlarge"

}

variable "vpc\_cidr" {

description = "Amazon Virtual Private Cloud Classless Inter-Domain Routing range."

}

variable "private\_subnets\_cidrs" {

type = list(string)

description = "Classless Inter-Domain Routing ranges for private subnets."

}

variable "public\_subnets\_cidrs" {

type = list(string)

description = "Classless Inter-Domain Routing ranges for public subnets."

}

# variable "whitelist\_ips" {

# type = list(string)

# description = "Restrict access only to news IPs to restrict lb access"

# }

variable "eks\_enabled\_log\_types" {

type = list(string)

default = []

}

variable "tags" {

type = map(string)

description = "Default tags attached to all resources."

default = {

ServiceType = "kat-eks"

}

}

variable "ssh\_key\_name" {

default = "kat-dev"

}

# variable "hosted\_zone\_id" {

# description = "Hosted zone id used by bastion host."

# default = ""

# }

variable "worker\_instance\_type" {

default = ""

}

variable "vpc\_single\_nat\_gateway" {

type = bool

}

variable "vpc\_one\_nat\_gateway\_per\_az" {

type = bool

}

# variable "assume\_role\_arn" {

# type = string

# }

variable "oidc\_thumbprint\_list" {

type = list

default = []

}

variable "principal\_account\_ids\_ram" {

type = list(string)

description = "AWS account ID to share transit gateway with. By default it is empty."

default = []

}

data "aws\_security\_group" "default" {

name = "default"

vpc\_id = module.vpc.vpc\_id

}

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

# version = "2.44.0"

name = "${var.tags["ServiceType"]}-${var.tags["Environment"]}-vpc"

azs = var.availability\_zones

cidr = var.vpc\_cidr

private\_subnets = var.private\_subnets\_cidrs

public\_subnets = var.public\_subnets\_cidrs

enable\_dns\_hostnames = true

enable\_dns\_support = true

enable\_nat\_gateway = true

# enable\_vpn\_gateway = true # TODO - Policy review DC or outside of DC connection into VPC?

single\_nat\_gateway = var.vpc\_single\_nat\_gateway

one\_nat\_gateway\_per\_az = var.vpc\_one\_nat\_gateway\_per\_az

private\_subnet\_tags = {

"kubernetes.io/role/internal-elb" = "1"

}

public\_subnet\_tags = {

"kubernetes.io/role/elb" = "1"

}

tags = {

Terraform = "true"

Environment = "dev"

}

# VPC Endpoint for ECR API

# enable\_ecr\_api\_endpoint = true

# ecr\_api\_endpoint\_private\_dns\_enabled = true

# ecr\_api\_endpoint\_security\_group\_ids = [data.aws\_security\_group.default.id]

# VPC Endpoint for ECR DKR

# enable\_ecr\_dkr\_endpoint = true

# ecr\_dkr\_endpoint\_private\_dns\_enabled = true

# ecr\_dkr\_endpoint\_security\_group\_ids = [data.aws\_security\_group.default.id]

# VPC endpoint for S3

# enable\_s3\_endpoint = false

# VPC endpoint for DynamoDB

# enable\_dynamodb\_endpoint = false

}

output "vpc\_id" {

value = module.vpc.vpc\_id

}

output "public\_subnet\_ids" {

value = module.vpc.public\_subnets

}

output "private\_subnet\_ids" {

value = module.vpc.private\_subnets

}

output "private\_route\_table\_ids" {

value = module.vpc.private\_route\_table\_ids

}

**CHAPTER -7**

**CODING AND TESTING**

**7.1 CODING**

Once the design aspect of the system is finalizes the system enters into the coding and testing phase. The coding phase brings the actual system into action by converting the design of the system into the code in a given programming language. Therefore, a good coding style has to be taken whenever changes are required it easily screwed into the system.

**7.2 CODING STANDARDS**

Coding standards are guidelines to programming that focuses on the physical structure and appearance of the program. They make the code easier to read, understand and maintain. This phase of the system actually implements the blueprint developed during the design phase. The coding specification should be in such a way that any programmer must be able to understand the code and can bring about changes whenever felt necessary. Some of the standard needed to achieve the above-mentioned objectives are as follows:

Program should be simple, clear and easy to understand.

Naming conventions

Value conventions

Script and comment procedure

Message box format

Exception and error handling

**7.2.1 NAMING CONVENTIONS**

Naming conventions of classes, data member, member functions, procedures etc., should be **self-descriptive**. One should even get the meaning and scope of the variable by its name. The conventions are adopted for **easy understanding** of the intended message by the user. So it is customary to follow the conventions. These conventions are as follows:

**Class names**

Class names are problem domain equivalence and begin with capital letter and have mixed cases.

**Member Function and Data Member name**

Member function and data member name begins with a lowercase letter with each subsequent letters of the new words in uppercase and the rest of letters in lowercase.

7**.2.2 VALUE CONVENTIONS**

Value conventions ensure values for variable at any point of time. This involves the following:

* Proper default values for the variables.
* Proper validation of values in the field.
* Proper documentation of flag values.

**7.2.3 SCRIPT WRITING AND COMMENTING STANDARD**

Script writing is an art in which indentation is utmost important. Conditional and looping statements are to be properly aligned to facilitate easy understanding. Comments are included to minimize the number of surprises that could occur when going through the code.

**7.2.4 MESSAGE BOX FORMAT**

When something has to be prompted to the user, he must be able to understand it properly. To achieve this, a specific format has been adopted in displaying messages to the user. They are as follows:

* X – User has performed illegal operation.
* ! – Information to the user.

**7.3 TEST PROCEDURE**

SYSTEM TESTING

Testing is performed to identify errors. It is used for quality assurance. Testing is an integral part of the entire development and maintenance process. The goal of the testing during phase is to verify that the specification has been accurately and completely incorporated into the design, as well as to ensure the correctness of the design itself. For example the design must not have any logic faults in the design is detected before coding commences, otherwise the cost of fixing the faults will be considerably higher as reflected. Detection of design faults can be achieved by means of inspection as well as walkthrough.

Testing is one of the important steps in the software development phase. Testing checks for the errors, as a whole of the project testing involves the following test cases:

* Static analysis is used to investigate the structural properties of the Source code.
* Dynamic testing is used to investigate the behavior of the source code by executing the program on the test data.

**VALIDATION TESTING:**

The validation of the design follows a qualitative approach, which is the most suitable validation technique for this specific research project, as the rest of the validation techniques are not applicable for this project. Real-world testing of this design was impossible as real-world circumstances could not be reached due to the restrictions of the projects of a mission critical organization. For instance, Thales builds and tests the IT infrastructure on real-world environment such as ships, making real-world testing infeasible in the scope of a Master thesis project. In addition, using this design for solving a real-world problem for a mission critical organization requires additional requirements, which are derived by the needs of their customers and are strictly restricted to specific personnel within the organization.

**TESTING:**

Testing is a process of executing a program with the intent of finding an error. A good test case is one that has a high probability of finding an as-yet –undiscovered error. A successful test is one that uncovers an as-yet- undiscovered error. System testing is the stage of implementation, which is aimed at ensuring that the system works accurately and efficiently as expected before live operation commences. It verifies that the whole set of programs hang together. System testing requires a test consists of several key activities and steps for run program, string, system and is important in adopting a successful new system. This is the last chance to detect and correct errors before the system is installed for user acceptance testing.

**INTEGRATION TESTING:**

Regarding the current proposed design, some experts referred to Cobbler as a functional solution for provisioning bare metal machines, but they would rather have a more modern alternative solution. That is the main reason that convinced most of the participants to select the future design as an improved solution that offers one dynamic platform for provisioning and controlling all infrastructure resources. Using only one tool for provisioning reduces the complexity of the system by reducing the number of software interfaces between the components. Fewer interfaces lead to a system that requires less effort for development and debugging. However, the advantage of this solution is also its main downside according to one of the experts. Controlling everything with one tool (Openstack in this case) creates high dependence on this selected technology. S

**CHAPTER 8**

**CONCLUSION**

**8.1 Conclusion and future enhancement:**

The conclusions presented in this chapter have been drawn by examining and answering briefly the research questions that were defined in the first chapter. The limitations and the contributions of this research project are also described in this chapter. Based on the limitations, possible future work and recommendations are listed and analyzed.

Several points can be further researched by the academia in the future:

 Performance: Further research is required on the overall performance of the system. Thorough insights of the performance of the system can be gained by building a fully functional prototype of the proposed designs with all the necessary hardware and software components and testing the prototype with real world testing environments. Benchmark testing environments can be created that would focus on the performance of the virtual switches, especially on the failover time and the latency.

 Availability: Future research projects should focus on the redundancy of the software-defined technology. Redundancy is a crucial factor in mission critical systems, which cannot afford system outages and single points of failures should be eliminated.

 Security: Security and safety requirements are very important for every IT system and further research on this field seems necessary. The research could focus on how the software-defined technology handles different security domains that exists in one system. Additional research can focus on identifying the vulnerabilities of the available software-defined tools and finding methods to increase their security.

 Observability: Researchers should investigate the available practices that could increase the observability of a software-defined system. The first step towards observability would be the addition of a monitoring and tracing pipeline into the proposed designs. These tools would use logs, metrics and events to keep track of the state of the entire system.

 Storage: Further research is needed regarding the storage of data in a software defined system. The traditional ways of storing data are not efficient when they are used in a software-defined system, because they are monolithic solutions that are mainly based on proprietary hardware. As a result, future research is needed on storage techniques that are software-defined.

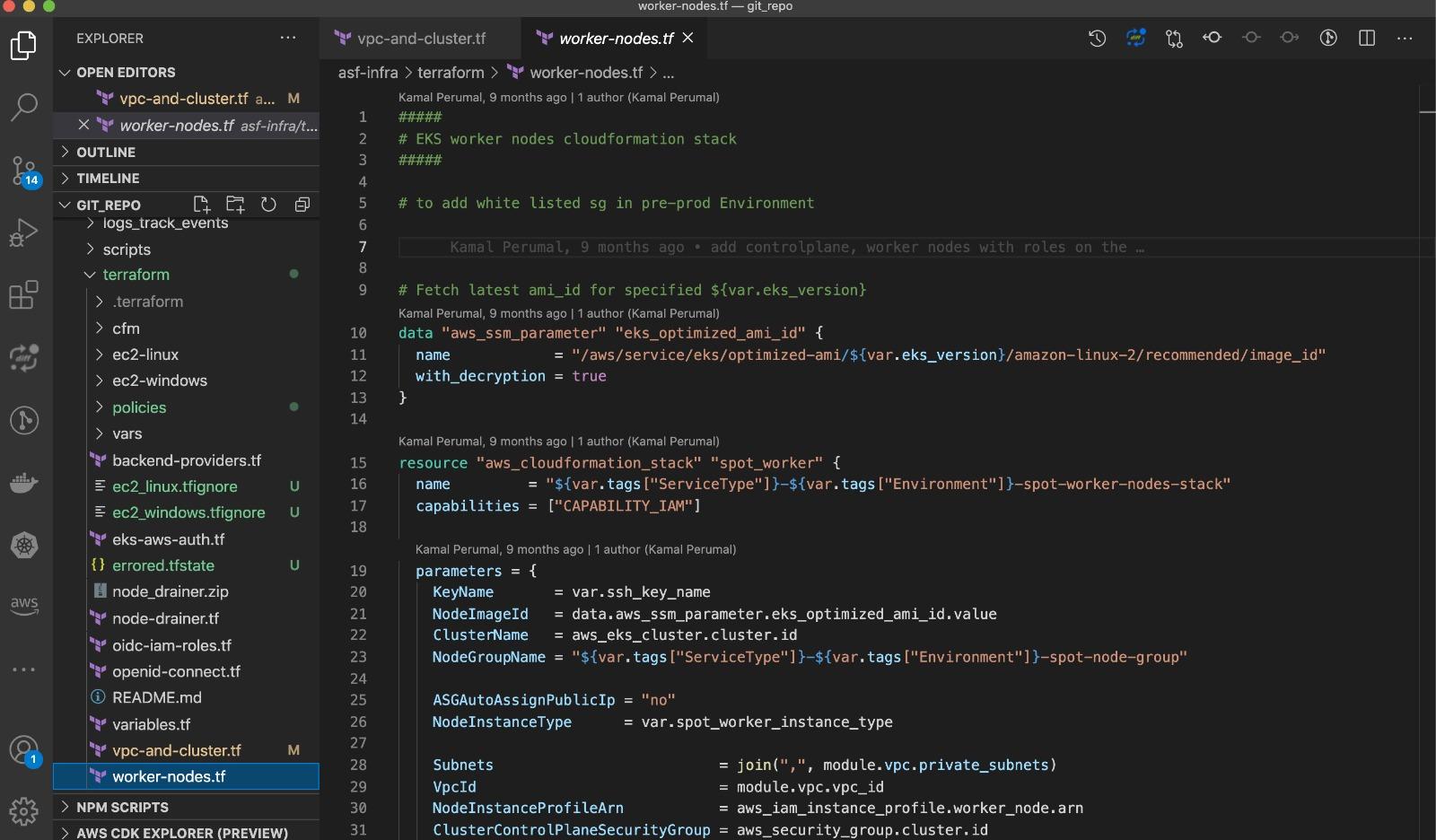
 Public solutions: This research was mainly focused on private cloud solutions for building and managing the infrastructure resources. However, public cloud technologies have showed great signs of improvement the last couple of years, and they can also offer private features that can be useful to mission critical organizations that require high security standards. Organizations can save money by outsourcing their infrastructure to public clouds, instead of setting up their own private cloud system, which requires expensive specialized hardware.

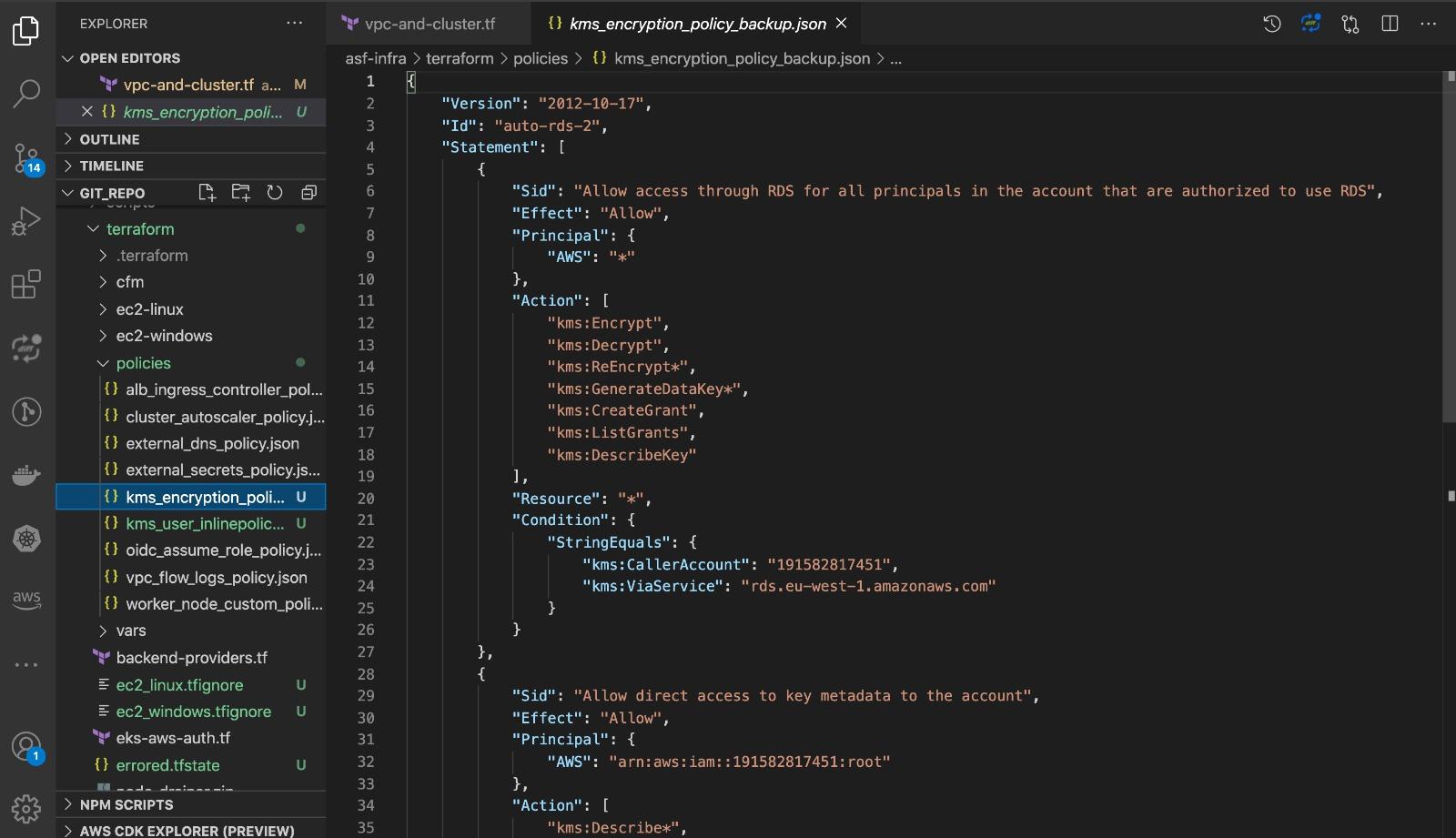
 Bare metal provisioning: Further research is required on the modern bare metal provisioning tools. Some examples of advanced bare metal provisioning tools are Metal3 [99], which is a Kubernetes API for managing bare metal hosts and Digital Rebar , which is a data center provisioning tool designed with a cloud native architecture.

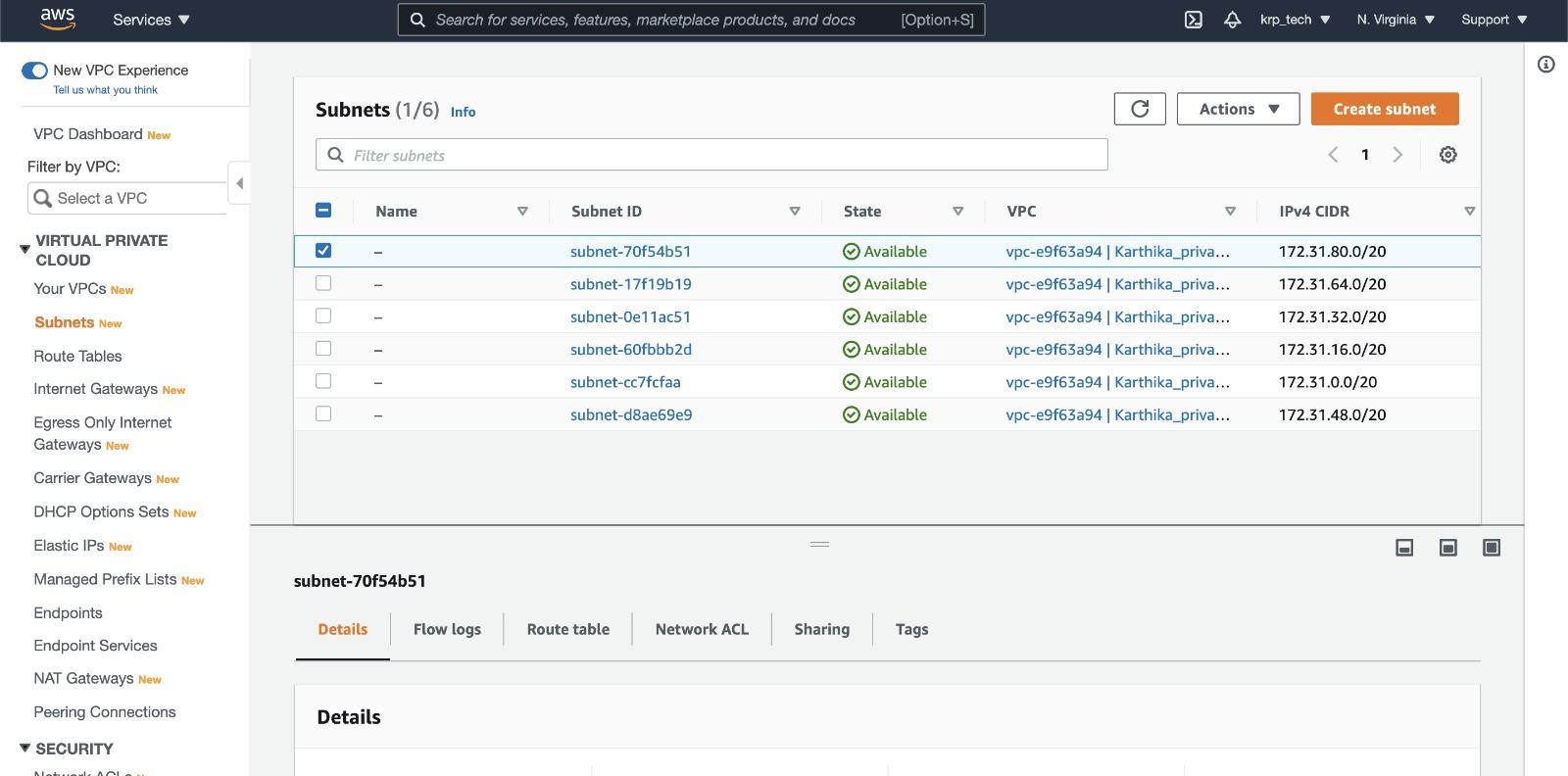
 Network layer: The SDN technology seems to be in its early stages and further research is necessary to fully leverage its capabilities. One field of future research is the combination of the SDN controllers with the container orchestration engines such as Kubernetes. For instance, there might be some available northbound APIs for the controllers that establish direct communication with the container orchestration engines or maybe the network capabilities of Kubernetes will significantly improve in the future, making a SDN controller unnecessary.

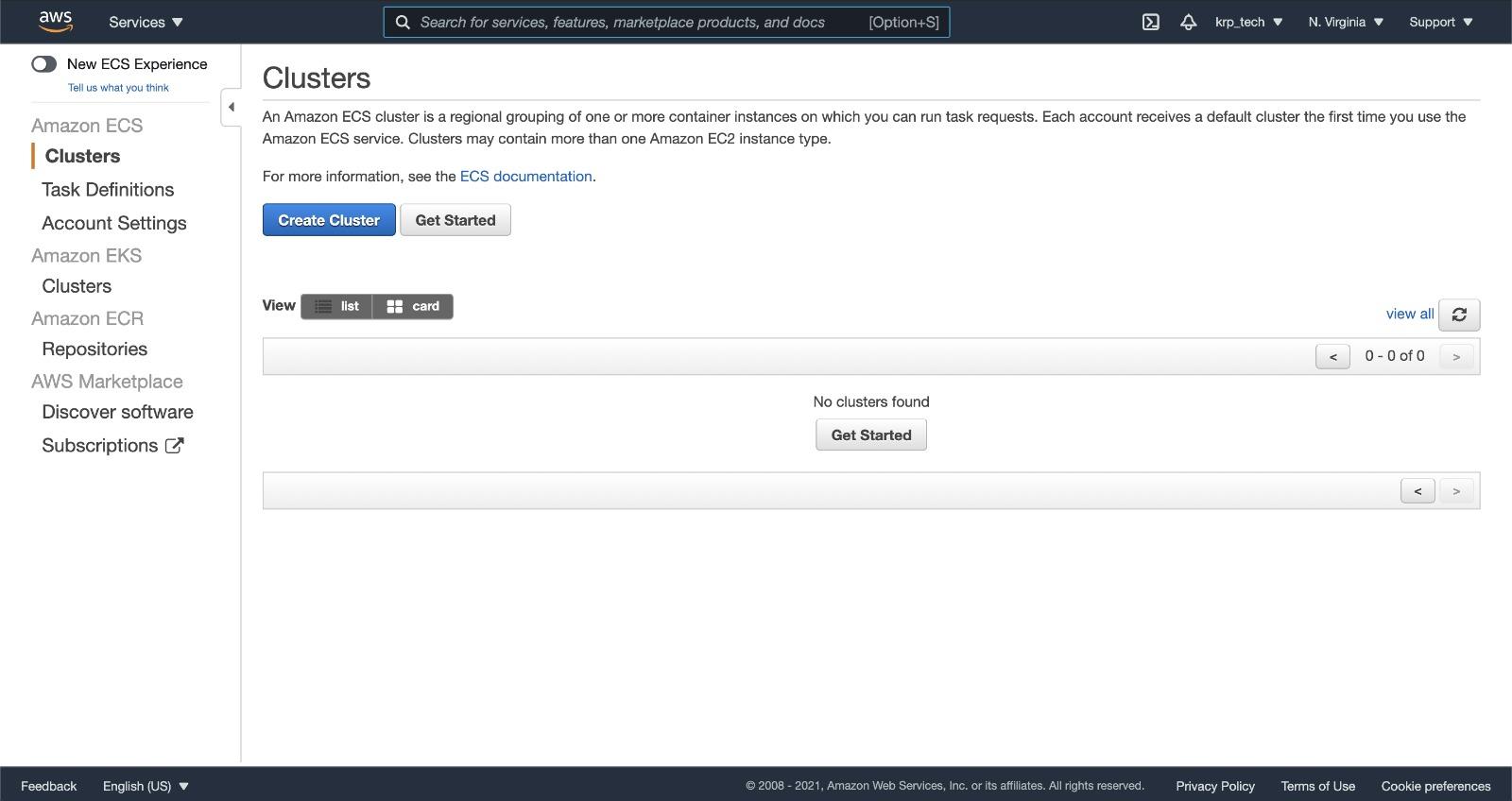
 Maturity model: Future research could focus on the definition of a maturity model that would help organizations acquire a clear overview of their current IT system. This model should assist the organizations in clarifying the level of automation and hardware abstraction of their existing IT systems, and based on that level, a list of suggestions should be available to help them convert their current system to a fully software-defined one

**8.2 Screen shorts:**









**CHAPTER 9**

**REFERANCES**

1. Peffers, K., Tuunamen, T., Rothenberger, M., & Chatterjee, S., “A Design Science Research Methodology for Information Systems Research”, 200.

2. Roel J. Wieringa, “Design Science Methodology for Information Systems and Software Engineering”, Springer, 2014.

3.Alan R. Hevner, "A Three Cycle View of Design Science Research," 2007, Scandinavian Journal of Information Systems.

4. Jieyu Lin, Rajsimman Ravichandiran, Hadi Bannazadeh, Alberto Leon-Garcia, “Monitoring and Measurement in Software-Defined Infrastructure”, 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)

5. Kief Morris, “Infrastructure as Code: Managing Servers in the Cloud”, O’Reilly Media Inc., 1st Edition, 2016.

6. . Medhi Medjaoui, Erik Wilde, Ronnie Mitra, Mike Amundsen, Kin Lane, “Continuous API Management: Make the Right Decisions in an Involving Landscape”, O’Reilly Media inc., 1st Edistion, 2019

7.Openstack, “Adding Speed and Agility to Virtualized Infrastructure with Openstack”, Whitepaper, 2015, Retrieved on 27 November, 2019.

8.Openstack, “Openstack Heat”, Retrieved on 5 August, 2019.

9..Redhat, “Ansible in Depth”, Whitepaper, Retrieved on 27 November, 2019.

10..Cisco, “Cisco IOS Technologies”, Retrieved on 5 August, 2019.

11.Redhat, “Linux-KVM”, Retrieved on 5 August, 2019.

12. B. Pfaff, B. Davie, “The Open vSwitch Database Management Protocol”, Retrieved on 29 August.

13..Vmware, “Vmware vSphere”, Retrieved on 5 August, 2019.

14.UnixArena, “Ansible – How to use facts on Playbooks”, Retrieved on 5 August, 2019.

15.Redhat, “Prepare for Opendaylght installation”, Retrieved on 29 August, 2019.

16.IBM Developer, “Automate and manage systems installation with Cobbler”, Retrieved on 5 August, 2019.

17.Pica8, “Improving Overlay Solutions with Hardware-Based VXLAN Termination”,Whitepaper, Retrieved on 27 November, 2019.

18.Microsoft, “Azure Resource Manager Overview”, Retrieved on 5 August, 2019

19.HarshiCorp, “HarshiCorp Terraform”, Retrieved on 5 August, 2019, .

20.Juniper Networks, “Junos OS Overview”, Retrieved on 5 August, 2019

21.The Linux Foundation Projects, “Xen project”, Retrieved on 5 August, 2019

22. Opensource.com, “What is Docker”, Retrieved on 5 August,

23.Juniper Networks, “Understanding the OVSDB Protocol Running on Juniper Network Devices”, Retrieved on 5 AuguST

24.Digital Rebar, Retrieved on 24 October, 2019

25.ISO 25000, “ISO/IEC 25010”, Retrieved on 30 September,

26.Openstack, “Attaching physical PCI devices to guests”, Retrieved on 5 August

27.Oracle, “Virtual Box”, Retrieved on 5 August, 2019.