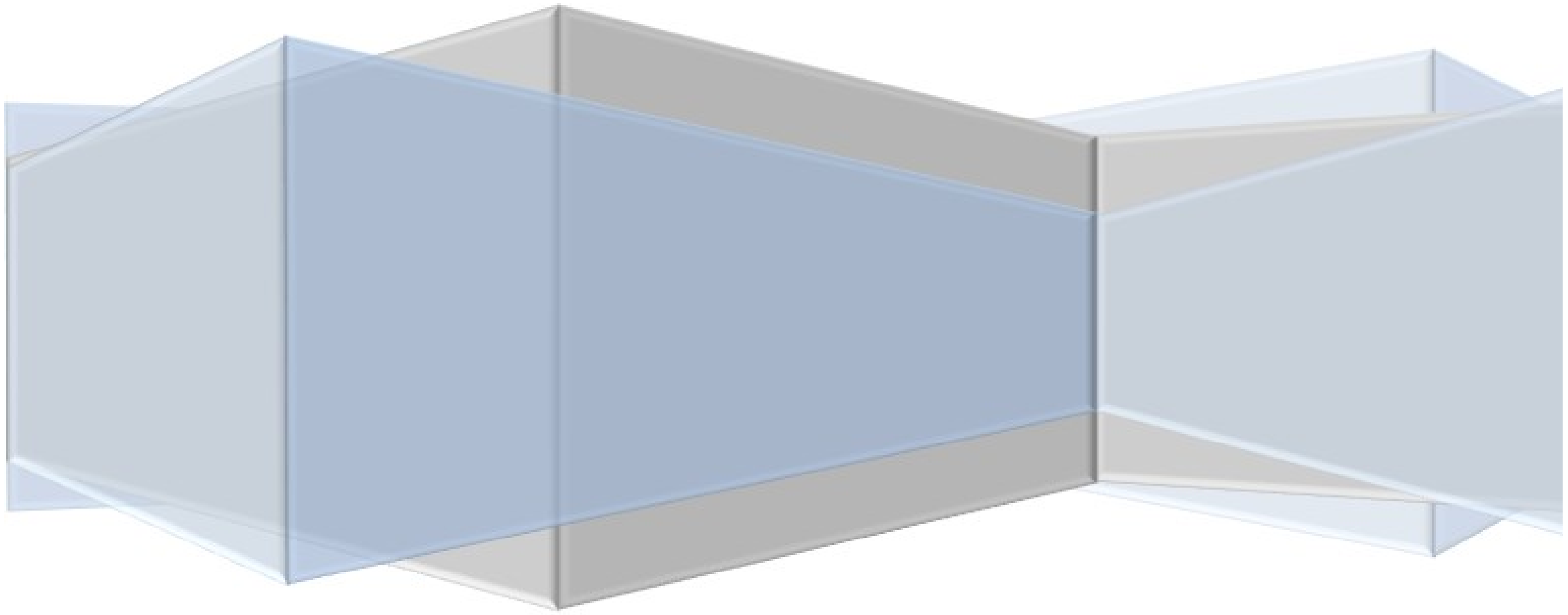
Infrastructure as Code

Towards Dynamic and Programmable IT systems

Sotirios Naziris

November, 2019



UNIVERSITEIT TWENTE. THALES

Infrastructure as Code

Towards Dynamic and Programmable IT systems

A thesis submitted in fulfillment of the requirements

for the Master of Science degree

in

Internet Science and Technology

Faculty of Electrical Engineering, Mathematics

and Computer Science

Author

Name:	Sotirios Naziris
Institute :	University of Twente 7500 AE Enschede The Netherlands

Graduation Committee

Chairman:	Dr. L. Ferreira Pires (UT)
Members:	Dr. M. J. van Sinderen (UT) ing. R. IJpelaar (Thales)

UNIVERSITEIT TWENTE.

THALES

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.

Preface

This thesis was written to fulfill the final step of my “Internet Science and Technology” master programme at the University of Twente. The report had been written within a seven months period, started on May 2019 and was officially finalized on November 2019.

The project was inspired and performed in cooperation with Thales Nederland, which is a company that specializes in naval systems, logistics and air defense systems and they are in constant search of new technologies that could help them improve their current systems.

At this point, I would like to express my sincere gratitude to all the people that helped me complete this master thesis project. First of all, I would like to thank my university supervisors, Luis and Marten for their guidance throughout this research procedure. Their feedback was really useful on improving the structure of the thesis and finding the correct research approach.

Furthermore, I would like to thank my supervisors from Thales for their useful feedback and support. I would like to thank Remco for his help especially on the technical part and for our great cooperation during our weekly meetings. I would also like to thank Gerrit Binnenmars for coming up with the idea of this project and for challenging me to think creatively and unconventionally. In addition, I would like to acknowledge the help from Andreas Frank who showed great interest into my project and provided fruitful feedback on the writing of this master thesis project. Additionally, I would like to express my gratitude to the rest of the Thales employees who participated in the validation interviews and offered their valuable insights on my designs.

Moreover, I would like to thank my parents for supporting me throughout my entire life and for giving me the opportunity to continue with my studies abroad. Last but not least, I would like to thank my girlfriend Virginia for her love and emotional support during the tough times that I faced during the completion of this project.

I hope that the reading of this thesis will be useful and interesting to you, and you can always contact me in case you have any questions or comments.

Sotiris

Enschede, Netherlands

Table of contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
BNSF	Base Network Service Function
CapEx	Capital Expenditures
COE	Container Orchestration Engine
COTS	Commercial off the Shelf
CPU	Central Processing Unit
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
QoS	Quality of Service
SAL	Service Abstraction Layer
SDC	Software Defined Computing
SDD	Software Defined Deployment
SDI	Software Defined Infrastructure
SDN	Software Defined Networking
SDx	Software Defined Everything
TEP	Tunnel Endpoint
TFTP	Trivial File Transfer Protocol
ToR	Top of the Rack
VLAN	Virtual Local Area Network
VTEP	VXLAN Tunnel Endpoint

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.

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"* [1].

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

- IT infrastructure level (Software-Defined Infrastructure - SDI)
- Network level (Software-Defined Networking – SDN)
- Computing level (Software-Defined Computing - SDC)
- 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 software-defined 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 [2]. The design science methodologies of Wieringa [3] and Henver [4] 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.

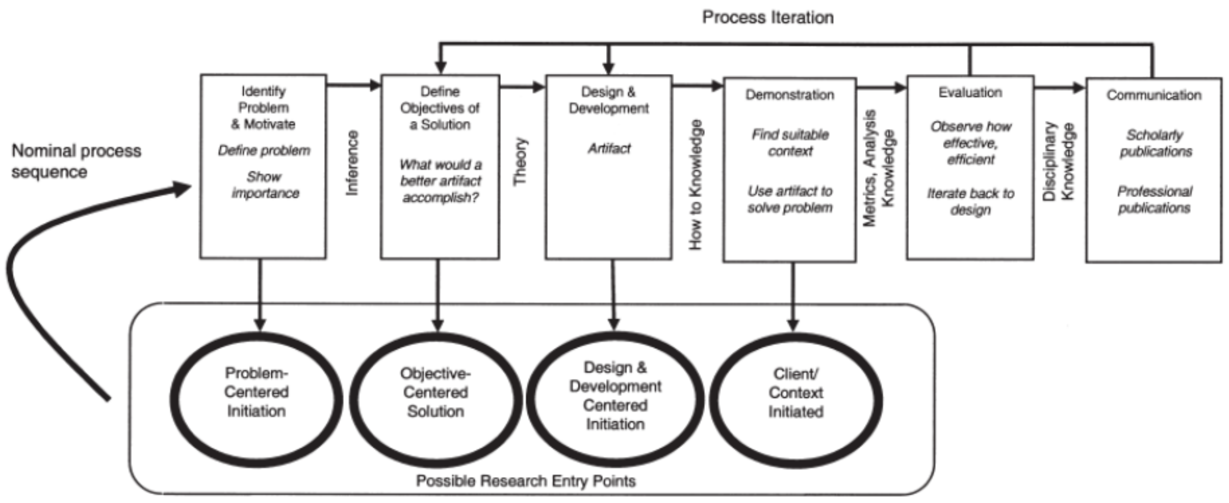

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 1: Overview of the DSRM

Based on the DSRM, the research process was divided into the following phases:

1. *Problem identification and motivation:* A systematic literature research was performed in order to identify the problems that exist in the existing IT systems and indicate the benefits of the software-defined technology at each level of an IT system.

2. *Defining the objectives of the solution:* This phase defined the objectives of the desired fully software-defined system. The objectives are translated into a list of requirements that were formed by studying the needs of a real mission critical organization.
3. *Design and development:* This phase includes the design of the artifact, which is a fully software-defined system. The design is based on the list of requirements from phase 2.
4. *Demonstration and evaluation:* The DSRM has two separate phases for the Demonstration and the Evaluation of the artifact. In this research, these two phases are included in one phase because using the artifact for solving an actual problem is infeasible during a master thesis project due to time and practical limitations. In this phase, the proposed design is presented and validated by a group of experts by using semi-structured interviews in order to define the strong and weak points of this design and suggest possible improvements.
5. *Communication:* This phase communicates the importance of this research by discussing the validation results and pointing out its contributions to the academic community and other interested organizations. This research will be published and archived into the University of Twente's repository, and the results will be presented in a Master thesis presentation.

1.5 Literature Study

To seek answers to the previously formed research questions, it is necessary to have a better insight on the software-defined technology. The first step towards this objective, was to conduct a systematic search on the available literature. Each level of the software-defined concept (infrastructure, network, computing, deployment) was thoroughly analyzed and explained. After understanding the concepts, the components and the architecture of each software-defined level, the most popular relevant tools and technologies at each level were listed, explained and compared. The literature review was mostly performed with the use of the following academic databases:

- Google Scholar
- IEEE Xplore Digital Library
- Scopus
- ResearchGate

Besides these four platforms, knowledge was gathered by studying several technical documentation, presentations and books created by the industry, and by attending some online courses explaining this technology. The learning process of writing a master thesis was also improved by studying several previous projects conducted at the University of Twente. The Google search engine was mainly used to search on websites, forums and blogs related to the software-defined technology.

1.6 Thesis Structure

This thesis is further organized as follows. Chapters 2 to 4 explain in detail the required background knowledge on the software-defined technology. The most popular tools and technologies at each level are also listed, described and briefly compared in these chapters. Chapter 2 explains the concepts related to software-defined infrastructure, Chapter 3 introduces the reader into the field of software-defined networking and Chapter 4 analyzes the technology related to software-defined computing. Chapter 5 includes the preparation of the design and the description of a specific case study, where a list of generic requirements, the architecture realization of a general software-defined system and the selection of specific tools are presented. Chapter 6 includes the actual design of the infrastructure layer of the software-defined system using the specified tools and Chapter 7 synthesizes and explains the design of the network layer. Chapter 8 includes the physical representation of the design and Chapter 9 is the validation of the design and the discussion of the validation results. Chapter 10 is the final chapter and presents the conclusions and some suggestions for future work.

2 Software-Defined Infrastructure

The introduction of cloud and virtualization was followed by an enormous number of new tools and platforms, which has led to a huge portfolio of systems for the IT enterprises which often requires even more time for maintenance. The industry has also evolved since then, and the demand for more flexible and easily accessible services has dramatically increased in the past few years.

This increasing demand, combined with the high need to cope with the continuously growing IT world has forced the organizations to seek for new and effective ways to meet the new high industry standards. As a result, more and more organizations made a step towards the software-defined approach. The level of the software-defined approach that refers to the IT infrastructure is called Software Defined Infrastructure (SDI) [5]. The Software Defined Infrastructure (which is mostly called Infrastructure as Code or IaC by the industry) is an attempt to address the high demand of IT services by maximizing the potential of the current IT infrastructure. A definition for IaC given by Kief Morris [6] is:

"*Infrastructure as Code is an approach to managing IT infrastructure for the age of the cloud, microservices and continuous delivery that is based on practices from software development* ".

2.1 General Elements

The main elements [6] of Infrastructure as Code are the definition files (also referred as code), the automation tools, the dynamic infrastructure platform and the application programming interfaces. These elements are explained with more detail below.

Definition Files

Definition files are the key element of IaC. The components of the infrastructure are defined and configured through these files. The IaC tools use these files as inputs to configure/ provision instances of the components of the infrastructure. The infrastructure components could be many things such as a server, a part of a server, a network configuration, etc.

Each IaC tool is followed by a different name for the definition files. For example, playbooks for Ansible, recipes for Chef and manifests for Puppet. The definition files are basically text files and they are treated as such. The most common formats for definition files are JSON, YAML or XML, and some tools define their own domain specific language (DSL) to allow developers to describe these files.

Dynamic Infrastructure Platform

A dynamic infrastructure platform grants the bases for provisioning and managing the main infrastructure resources, such as servers, storage and network components, and ensures that they can be programmable.

There are several available dynamic infrastructure platforms. The most known examples are the public IaaS cloud services, like Azure and AWS and private IaaS, like Openstack [7]. The infrastructure can also be managed using virtualization systems such as VMware vSphere, which do not run in the cloud. Moreover, some organizations use tools as Cobbler and Foreman [8] to manage an infrastructure entirely on bare metal physical hardware.

The dynamic infrastructure platform is not affected whether it runs in the cloud, on virtual machines or bare metal, however it is essential to be programmable, on demand and self-service. *Programmable* refers to the previously mentioned definition files and implies that the dynamic platform should support configuration and management via these files. The term *on-demand* entails that the platform provides the users with the capability to create and destroy resources instantly in a matter of minutes or even seconds. Finally, a *self-service* platform does not only offer quick deployment of resources, but also supports the ability to change and customize resources based on the user requirements.

Automation Tools

There are two different categories of automation tools for setting up the infrastructure: *provisioning* tools and *configuration* tools. Provisioning tools are used to specify and allocate the desired resources. These tools use the dynamic infrastructure platform to implement the allocation. Examples of these tools are Terraform, Openstack Heat and CloudFormation by Amazon. Configuration tools are used to configure and manage the already provisioned resources with the required dependencies and settings. There are plenty of available tools on this category, but the three most popular ones are Puppet, Chef and Ansible.

Application Programming Interfaces (API)

APIs [9] are generally used to define the programming interfaces of a software component so it can be used by other software components. In this case, the applications and the tools should be able to connect and exchange information with the underlying platform, and APIs is the main solution to this.

APIs are offered by the automation tools in order to provision and configure the resources of the infrastructure in the way described in the definition files. Even when using the off-the-shelf tools, the engineering teams must occasionally write their own custom scripts and extensions to program the tools against their API, so the used tools should support a wide variety of programming languages that the team is experienced with.

REST-based APIs are the most used as they offer remote access, ease of use and high flexibility. Many tools and dynamic platforms support some programming language-specific libraries including useful classes and structures that can be used to easily deploy the components of the infrastructure and apply operations on them.

Figure 2 illustrates an overview of the elements and the interconnections between them. The idea is that the user creates the definition files that describe the infrastructure, and these files are inserted into an automation tool that via an API instructs a dynamic platform to create and manage the infrastructure resources.

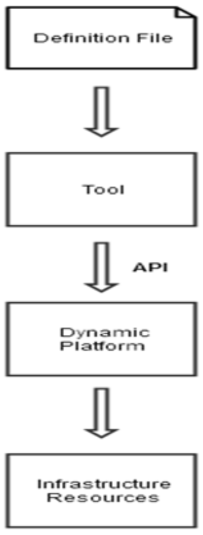


Figure 2: Main elements of the software-defined infrastructure

2.2 Benefits of Infrastructure as Code

This section summarizes the benefits [6] of using IaC to deploy, manage and update the IT infrastructure in an organization.

Easily and Fast Reproduced Systems

By using IaC, the administrators (and even developers) can provision and set up an entire infrastructure from the ground up by simply writing some scripting and definition files.

Each element of the infrastructure can be repeatedly reproduced reliably and effortlessly. The IaC scripts describe all the necessary steps for creating and provisioning the requested resource, for instance which software should be installed, its correct version, hostname etc.

Writing these scripts is much easier compared to old manual way, and the development and production of the systems becomes much faster and simple. New services and new applications can be deployed on the infrastructure easily, which has led to a more efficient software development process.

Disposable Systems

IaC has improved the old static systems into dynamic, disposable systems that are able to change in a fast and easy way. The resources of the new dynamic infrastructures are easily created, destroyed, updated, resized and relocated in the system. Nevertheless, the deployed software is able to run even when the server which it runs on, is deleted, moved or resized. Improvements and patches to the infrastructure became easier as the changes can be handled smoothly. This is crucial in large scale cloud infrastructures where the system cannot rely on the underlying hardware.

Configuration Consistency

Human errors have always caused problems to the consistency of the configuration, even when standard procedures for configuration are followed. The human factor created some slight deviations in configurations that were challenging and time consuming to debug.

The implementation of IaC fully standardizes the configuration of the infrastructure, leaving little space for human errors. As a result, the chances of encountering incompatibility problems are greatly reduced, and the execution of the running applications become more consistent and smooth.

Self-Documented Systems and Processes

IT teams have always been struggling to keep their documentation useful and accurate. Updates and improvements are implemented in a high pace so it is almost impossible for the documentation to be up to date with them. In addition, many people prefer to write the descriptive documents in their own way and in many cases they skip explanations as they consider them obvious or unnecessary for the reader. Therefore, most documents are not a fair representation of what really happens.

IaC has managed to solve this problem by enclosing and explaining all the necessary steps to execute a process in the definition files and the tools that actually carry out the procedure. An additional small piece of documentation is necessary in that case as well. The documents should be close (physically and meaningfully) to the code that they explain to help people acquire a good understanding of the underlying procedures.

Version Everything

Having the entire infrastructure codified in definition files opens the opportunity to use version control techniques to keep track of the committed changes, and roll back to previous stable version when an error occurs.

The Version Control System (VCS) offers a log file with all the implemented changes, the reason of the changes and the entity that made them. This feature is really useful for debugging purposes. Each feature of the system is identified by tags and version numbers, improving the tracing and fixing of more complicated abnormalities. Furthermore, the VCS supports the automatic start of a series of required actions upon the request of a new change, which is a feature of the continuous integration and continuous delivery approach.

Continuously Tested Systems and Processes

Automated testing is one of the most important practices that has been added to the infrastructure development with the introduction of IaC. Writing automated tests for a running infrastructure is challenging, but its correct implementation can lead to a clean, simple and functional infrastructure.

People are more confident to make changes if they receive fast feedback of the proposed changes. Testing is performed simultaneously with the development and that is crucial for an automated infrastructure, in which a small mistake can quickly cause significant damage.

Increased Efficiency in Software Development

IaC has boosted the productivity of the software developers. The software development cycle turned into a more efficient process, as the IT infrastructure can be deployed rapidly, easily and in several stages.

Developers can easily create their own sandbox system to launch and experiment with their code. Testing and security checking can take place in separate staging systems, and the application can be simply deployed and managed on the systems by using IaC tools. These tools can also delete the environments that are not used, freeing valuable computing power. Moreover, by shutting down all the unused resources the development environment remains clean and simple. That further increases the productivity of the engineering team, as they find a clean and friendly environment to deploy and they do not have to spend time erasing unused components from previous projects.

2.3 Provisioning Tools

Provisioning is the first step in order to build a concrete and functional infrastructure. Provisioning tools aim at setting up the foundational infrastructure components. Most of the provisioning tools are supported by an infrastructure vendor, such as Amazon or Google, but there are also tools that support multiple vendors. The most popular vendor specific provisioning tools are CloudFormation for AWS [10], Cloud Deployment Manager for the Google Cloud Platform [11] and Azure Resource Manager for Microsoft Azure Clouds [12]. The most popular open source provisioning tools are:

2.3.1 Terraform

Terraform [13] is an infrastructure automation tool developed by HashiCorp four years ago and it is written in the Go programming language. It is the first multi-cloud infrastructure tool that allows the user to automate and set up infrastructure elements from several cloud vendors simultaneously, as well as custom in-house solutions.

Terraform describes the infrastructure through the configuration files which are written in its own developed domain-specific language called Hashicorp Configuration Language (HCL). These files are compatible to JSON and are used to deploy the requested resources. These files can be easily shared and reused to create the same environment elsewhere.

Terraform also provides execution plans, which describe the procedure that is followed in order to reach the desired state of the infrastructure. The execution plan first gives an overview of that happens by the time it is called and then Terraform actually sets up the infrastructure by executing this plan. In addition, Terraform is able to create a graph of the infrastructure resources by parallelizing the creation and modification of any non-dependent resource. The use of the execution plan combined with the produced resource graph provides more automation towards changes with less human involvement, as the user has more insight on the Terraform's functionality, avoiding possible human errors.

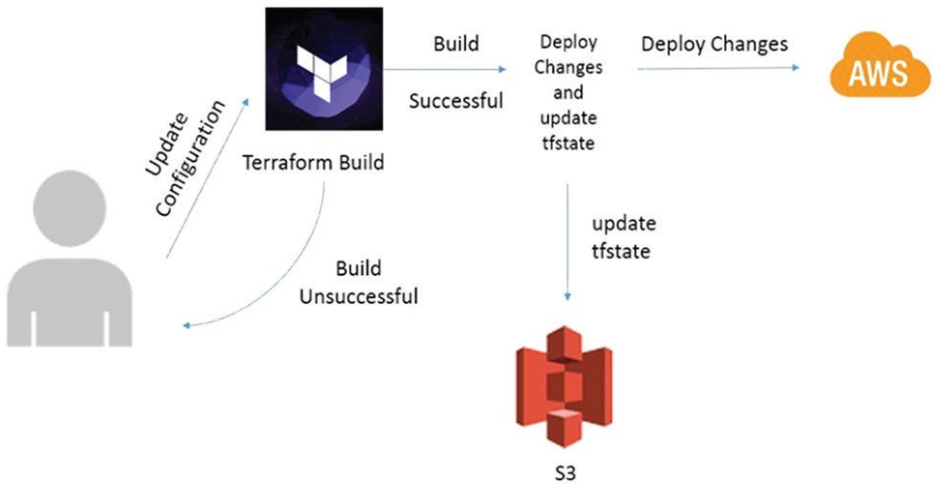


Figure 3: Sample Terraform workflow [14]

Terraform stores the state of the managed infrastructure in a local file called terraform.tfstate. This file can also be stored remotely which is useful when working in a remotely distributed team. This local state is used to create the execution plans and make the necessary infrastructure changes. After each performed operation, Terraform refreshes the state in order to match the actual real time infrastructure. Figure 3 shows a common Terraform workflow. In this example, AWS is used as a dynamic cloud infrastructure platform and S3 is used to back up the tfstate file.

2.3.2 Openstack Heat

Heat [15] is a product of the Openstack Foundation and is the main tool of the Openstack Orchestration program. Like the previous tools, Heat uses template files in the form of text files to deploy and set up multiple cloud resources of the desired IT infrastructure.

The infrastructure resources that can be used include: servers, volumes etc. The Openstack Telemetry [16] is also supported by Heat so that the user can include in the template file a scaling group [17] as a possible resource. In addition, the user can declare the connections and the dependencies between the resources. Heat uses these relationships in order to call the Openstack APIs that are responsible for the creation of the infrastructure, as prescribed by the user. The user can easily change the infrastructure by simply modifying the template file, and then Heat takes all the necessary steps to adjust the infrastructure to the desired state. Heat also deletes all the unused resources after application finishes its execution.
Heat supports an Openstack-native Rest API as well as a Cloud Formation-compatible Query API and the Heat template files are highly integrated with popular software configuration management tools such as Puppet and Chef. The Heat team is currently making the Heat template format compatible with the AWS Cloud Formation template format, so that many existing Cloud Formation templates can also be run on Openstack.

2.3.3 Comparison of the provisioning tools

A comparison of the described provisioning tools is illustrated in Table 1. The tools are compared based on their availability, the support of specific platforms and the used configuration language.

Metrics	Terraform	Openstack Heat	Cloud Formation	Cloud Deployment Manager	Azure Resource Manager
Availability	Open source	Open source	Closed source	Closed source	Closed source
Supported Platforms	Multiple platforms	Openstack	AWS	Google Cloud	Azure
Configuration Language	DSL (HCL)	DSL (HOT)	YAML, JSON	YAML	JSON

Table 1: Comparison of the provisioning tools

Terraform and Openstack Heat are open source software, whereas the rest of the tools are proprietary solutions, which are free for setting up a limited number of resources. Terraform is the only tool from the list that supports multiple dynamic platforms, enabling the combination of different resources from a variety of vendors. The other solutions are attached to a specific dynamic platform such as Openstack, AWS, Google Cloud and Azure. Terraform and Openstack Heat use domain specific languages for managing the configuration, which are YAML-based, while the rest of the tools use YAML or JSON for describing their definition files.

2.4 Configuration Tools

Once the elements of the infrastructure are provisioned, they need to be configured. Configuration management tools are used for this purpose. There are many available tools on the market, and each one of them has its own advantages and disadvantages. However, all of them serve the same goal; to configure the deployed resources according to the configuration settings. The most popular open source configuration tools based on the numbers of commits and stars in GitHub, are described in the following section.

2.4.1 Chef

Chef [18] is a configuration management tool that helps automate the IT infrastructure. Chef can manage infrastructures in the cloud, on bare metal as well as in a hybrid environment. Chef is a cl oud agnostic tool that works with many popular cloud service providers, such as Microsoft Azure, AWS, Openstack and Cloud Platform. The first version of Chef was developed in Ruby, however the latest version is partly written in Erlang and Ruby. Chef can support infrastructures up to 10.000 nodes.

The main components that form Chef are:

- *Chef workstation*: System that is used by the user to interact with Chef. With the Chef workstation, the user is able to develop cookbooks and recipes, manage the nodes of the infrastructure, synchronize the chef repository and upload cookbooks and other files to the chef server. The user can interact with the chef server using knife, which is a command line tool. The chef repository stores everything that is related to the chef server and nodes. Chef supports multiple workstations for a single chef server.

- *Chef Client node*: A virtual or physical machine that is managed by Chef. Chef can also manage nodes located in the cloud. Each node has to include an agent known as chef client in order to interact with the chef server. The configuration of a node is performed through a built in tool called Ohai, which is used to describe the node attributes to the chef client.
- *Chef Server*: A system that holds everything that is essential for configuring the nodes. The server stores the cookbooks, the used policies on the nodes and some metadata that describe the nodes that are managed by Chef. The Chef client which is installed on each node asks for the configuration details, such as recipes and templates from the server, and then applies the configuration to the specified node.

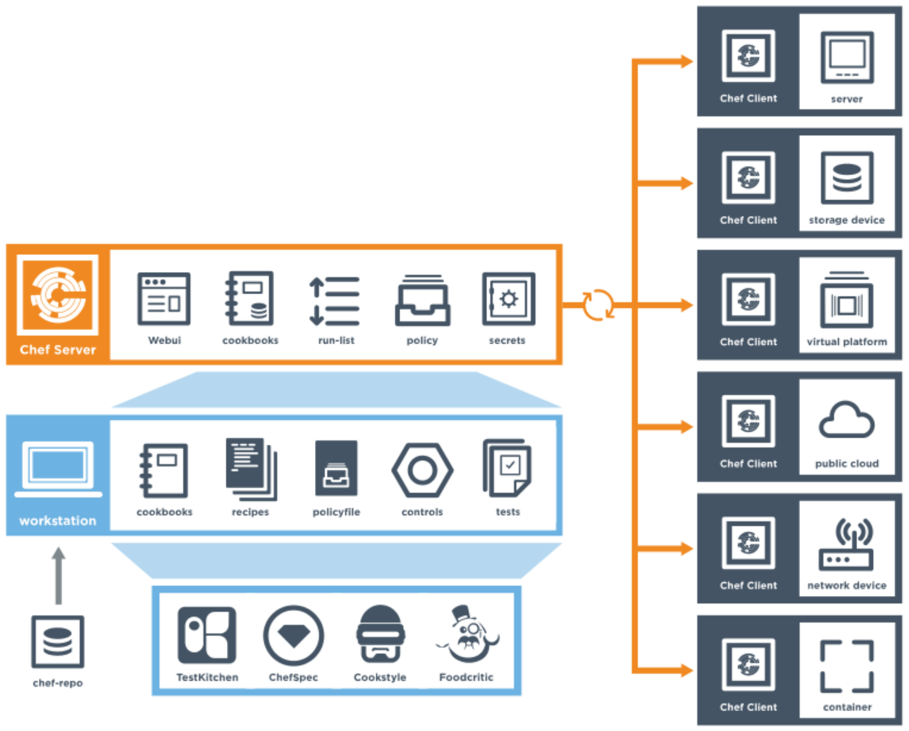


Figure 4: Basic structure of Chef [19]

Chef transforms infrastructure into code by using text files called cookbooks, which are the fundamental unit to configure and distribute policies in Chef. Cookbooks define complete scenarios, and include everything that is essential to run this scenario. Cookbooks are used to group and organize recipes. Recipes are basically scripts written in Ruby that specify the required resources and the order of their application [20]. Ruby is chosen as the reference language for creating cookbooks, with the support of an extended DSL for specialized resources. The chef client is equipped with a variety of resources to support the most common infrastructure scenarios, nevertheless the DSL can always be extended whenever there is more resources with more capabilities are needed.

Chef comes into two versions: commercial and open source. The commercial version is called Enterprise Chef and offers high availability deployment support. It is equipped with some additional features regarding security and reporting. The open source version has almost all the features of the commercial version except for the extra security and reporting. In addition, open source Chef does not support the installation of components on multiple servers.

2.4.2 Puppet

Puppet [21] is another popular configuration management tool that helps organize and configure servers. Puppet executes the configuration plans through an abstraction layer that describes the configuration elements as generic objects.

The user has to declare the resources and their attributes and that are given to Puppet as input to properly configure the resources. Puppet receives the catalog of the described resources and compares the existing state of the resources with the described one. Then it decides which actions need to be taken in order to reach an agreement between the requested state and the current state of the resources. This approach is declarative [22], since the user declares how the configuration should look like and then Puppet takes all the necessary actions to reach that intended configuration. This is the main difference with Chef, which follows a procedural approach in which the user has to describe the necessary steps to reach the desired state.

In Puppet, the resource definition files are called manifests and are written in a DSL, quite similar to Ruby. However, the user cannot simply write Ruby code in the manifests and have them executed. The manifests can be executed over and over again resulting to the same results that always match the described state.

Puppet has two main components for configuring servers: the puppet agent and the puppet master. The puppet program itself is called Puppet agent when it runs in a daemon mode on the server. The puppet master is a daemon that runs on the master server of the cluster and defines which configurations would apply to which server and also stores all the configuration information in a central location. The puppet agent asks the configurations from the puppet master at specific time intervals, and when there is a need of change the puppet agent actually implements the change. The communication between these two components is performed over a secure encrypted channel by using the SSL protocol.

2.4.3 Ansible

Ansible [23] is another powerful configuration management tool. The uniqueness of Ansible compared to other management tools is that it is also used for deployment and orchestration. Ansible is especially developed to be simple, secure, reliable and easy to learn. It offers a variety of features for an expert user but it is equally accessible to less skilled users.

Ansible does not use agents, and no additional software should be installed on the remote servers in order to manage them. Ansible manages the remote machines by using the remote management frameworks that already exists natively on the OS, for instance, SSH for Linux and UNIX machines and WinRM for Windows machines. The absence of agents results in less resource consumption on the managed machines when Ansible is not operating on them. Ansible also improves security by functioning in a push-based model where the remote machines receive only the necessary parts of the code (called modules), and the remote machines cannot interact or interfere with the configuration of the other machines. These features made Ansible suitable for high security and high performance systems.

In Ansible, the definition files to configure, automate and manage the IT infrastructure are called Playbooks. These files are written in YAML format and they describe how to perform an operation by clearly stating what should be done by each component of the infrastructure. Each Playbook consists of a list of *plays* that describe the automation process to a set of hosts, called the *inventory*. Each play includes several tasks that refer to a single host or a group of hosts in the inventory. Each task calls a module, which is a small piece of code that performs a specific job. The tasks vary from simple jobs to complex operations. Ansible can also enclose Playbook tasks into units known as *roles*. Ansible uses roles to apply commonly used configurations in several scenarios in a rapid and easy way.

Ansible was developed in a way that facilitates extensibility. The user has always the possibility of extending the native 450+ Ansible modules by writing his own modules. The built-in modules are written in Python and PowerShell, but the user can use any programming language to develop new ones, with the only restriction that they have

JSON as input format and produce JSON as output format. In addition, Ansible can be extended to support dynamic inventory, which allows the Playbooks to be executed on a group of machines and infrastructure that are not constant and statically defined, but can run a public or private cloud provider that supports the dynamic creation and deletion of the resources. Ansible supports most of the well-known cloud providers and can always be extended to support new providers by simply writing a custom program (in any programming language) that gives a JSON inventory definition as output.

Ansible is an open source project promoted by Red Hat. The paid commercial version of Ansible is called Red Hat Ansible Tower, and offers management to complex multi-tier deployments by adding control and technical support to Ansible supported systems.

2.4.4 Saltstack

Saltstack [24] is a configuration management tool that is also used for orchestrating the infrastructure. It configures, changes and updates the IT infrastructure through a central repository. It can operate on physical, virtual and cloud servers.

Like the previous IaC tools, Saltstack aims to automate the administrative and code deployment tasks and reduce the chance of human error, by removing the manual processes as much as possible. In order to achieve that, Saltstack uses both the *push* and the *pull* method to configure the servers. It pulls configuration files and code from a central repository such as Github, and then it pushes these files to the servers remotely.

Saltstack has two main components: the Salt master and the Salt minion. The master is the central server and all minions are connected to it to get instructions. The connection between the master and the minions is encrypted based on cryptographic hashes. The minions can be commanded by the master after using public key authentication. The minions can run without a master, but the full potential of Saltstack is leveraged in a master-minions network. The user can push updates and configuration files through the master to the minions, or schedule the minions to check the master at specific time slots and pull available updates and configurations. The Saltstack's management architecture is highly event-driven and offers self-dependence and healing to the system, as it leverages both the push and pull methods for updating and recovering from errors.

Saltstack includes some other important features, such as the Salt reactors, agents, minions, grains and pillars. The Salt reactors are responsible for listening for new events on the minions, while the Salt agents use secure shell to run commands on the target nodes. The minions are agents themselves that are installed on the remote servers to

push commands. The Salt grains provide valuable information about the managed servers and the Salt pillars are the configuration files.
Saltstack is different from the other configuration management tools because it is fast and can operate in a multithreaded manner, so that it can perform multiple tasks simultaneously. In addition, Saltstack is developed in Python and uses it to write the configuration scripts. However, it can also render scripts developed in other languages such as YAML and JSON. That makes Saltstack a language agnostic tool that is easily accessible to a wide audience of software developers.

Saltstack is an open source framework that is operated from the Command Line Interface (CLI). The paid edition is called Saltstack Enterprise and comes with some additional features, such as a GUI support and the support for Windows, macOS and Solaris servers. The Enterprise version can also store the events in a database, offering an auditable history of the events to the user.

2.4.5 Comparison of the configuration tools

Table 2 represents a comparison between the previously mentioned configuration tools. There is a wide variety of factors, which can be used to compare this sort of tooling but for the purpose of this research the five most suitable metrics were selected.

Metrics	Chef	Puppet	Ansible	Saltstack
Configuration Language	Ruby	DSL (Ruby-based)	YAML	YAML, Python, JSON
Enterprise Version	Yes	Yes	Yes	Yes
Architecture	Master-Agent	Master-Agent	Master only	Master-Agent
Configuration Method	Pull	Pull	Push	Push and Pull
OS Support	Master (Linux) Agents (Linux & Windows)	Master (Linux) Agents (Linux & Windows)	Master (Linux) Agents (Linux & Windows)	Master (Linux) Agents (Linux & Windows)

Table 2: Comparison of the configuration tools

All the described tools provide high levels of scalability. These tools are able to manage large infrastructures with more than 10.000 nodes and the user should only declare the IP or the hostname of the nodes that require configuration, and the tools perform the desired configuration. Furthermore, all tools provide an enterprise supported version, which comes with different prices depending on the size of the infrastructure.

Chef, Puppet and Saltstack follow a master-agent architecture, while Ansible requires only the installation of the master on the server machine and no additional software is installed on the client machines. Chef and Puppet follow the pull configuration method, while Ansible uses the push method. Saltstack uses both push and pull methodes to configure the servers. It pulls configuration files and code from a central repository and then it pushes these files to the servers remotely.

Chef and Puppet use Ruby-based configuration languages for managing their definition files, which requires a basic understanding on programming in order to use them efficiently. On the other hand, Ansible and Saltstack use YAML for configuration, which is a user-friendly configuration language. Saltstack also offers support for Python and JSON.

3 Software-Defined Technology at the Network Layer

Over the past decades, server virtualization has been a remarkably beneficial technology and has managed to improve the IT infrastructure significantly by providing agility, flexibility and scalability to the systems. The success of the server virtualization formed the starting point of the research on new technologies that aim to achieve such a level of virtualization of the network infrastructure.

The new technologies in this field are the *Software Defined Networking (SDN)* and the *Network Function Virtualization (NFV)*. These approaches aim to contribute to a more virtualized network infrastructure that is dynamic and highly scalable. SDN and NFV are often considered as the same technology; however there are two independent technologies which are certainly related. NFV is mainly focused on deploying infrastructure services on commercial off-the-shelf hardware, while SDN is the tool that dynamically controls the network resources. NFV was developed by the telecommunication providers with the goal of reducing CapEX, OpEx and the power consumption of the service provider networks, while SDN tries to provide programmable network control and configuration. Therefore, the main area of operation of the NFV technology is service provider networks, whilst SDN mainly operates on datacenters and cloud environments. The following sub-sections give a more detailed description of the SDN technology as it is more relevant to the purpose of this research.

3.1 Software-Defined Networking

The traditional IP networks are widely used all around the globe; however they can cause several difficulties during the configuration and management process due to their high complexity. Each individual network device should be configured independently using in many cases some especially designed commands in order to reach the described high level network policies. Furthermore, the networks should be adaptable to dynamic load changes and occasional errors. The current IP networks have limited support for automatic configuration and error resolving actions, resulting to difficulties when dealing with a dynamic environment. The level of complexity in the present IP networks increases since they are vertically integrated. This means that the control plane (controls the data traffic) and the data plane (forwards the data in the network) are stacked together on the network devices, lowering the levels of the flexibility in the network. Another serious challenge for the network researchers and engineers is that the Internet in its current form is a huge and important part of our society, and it is highly challenging to improve it in terms of performance and physical infrastructure. Setting up new physical network infrastructure and developing new Internet protocols requires time, money and lot of engineering effort. At the same time, the Internet applications and services have become more sophisticated with more functionalities and demands, and it is critical for the current Internet to grow in terms of scalability and performance to face these newly emerged challenges.

The idea of programmable networks has been proposed as a promising solution to tackle the problems at the current network infrastructure. Specifically, the academia introduced *Software-Defined Networking (SDN)* [25] as a paradigm to lift the constrains at the current network systems. The main goal of SDN is to provide agile, scalable and programmable networks. SDN separates the control plane from the underlying hardware (routers, switches) that forwards the data into the network (data plane). By detaching the control plane from the data plane, the network devices are simply responsible for forwarding data, and the control of the network is now enclosed in a logical central controller (or network operating system – NOS). Figure 5 shows the difference between a traditional network infrastructure and a SDN network in which the control plane is independent from the data plane, and it is located in the SDN controller.

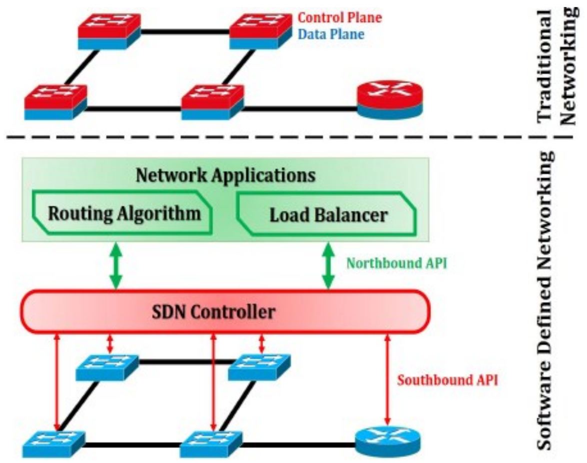


Figure 5: SDN vs traditional networks [26]

The SDN architecture is composed by three main elements:

- *SDN Applications*: Software programs that interact with the SDN controller via the API. The network infrastructure is described and constructed by these applications based on the gathered information from the SDN controller. The SDN APIs are divided into two categories: the Northbound API and the Southbound API. The *Northbound* API is the link between the SDN applications and the SDN controller. The *Southbound* API defines the communication between the controller and the physical devices.

- *SDN Controller*: It is the main component of the SDN architecture. It controls the SDN devices via the southbound API and the SDN applications through the northbound API. The purpose of the controller is to provide a programmable and intelligent network.
- *SDN devices*: They are the networking devices, such as routers and switches that forward and route the data in the network, based on the flow tables described by the SDN controller via the southbound API. The devices can be real hardware switches or virtual ones. The virtual switches are implemented using Open vSwitch (OVS) [27].

3.1.1 Southbound Interfaces

OpenFlow [28] is the most popular open southbound standard for SDN. The forwarding devices known as OpenFlow switches are mainly composed by two components: flows tables and an abstraction layer. The abstraction layer allows secure communication between the switches and the SDN controller via the OpenFlow protocol. Flow tables include several flow entries, which decide where and how the flow packets should be forwarded and processed. Basically, flow entries are match-action rules that dictate what action needs to be taken when a specific match is encountered. Flow entries generally include: 1. Match fields, which are used to match incoming packets, 2. Counters, which are used to gather information and keep statistics for the particular flow such as the number of received packets, the duration of the flow etc. and 3. Set of instructions or actions that should be implemented when a match is found. Through the OpenFlow protocol, a controller can create, add, update and remove flow entries from the flow tables on the switch.

By the time a new packet arrives at an OpenFlow switch, the packet header is compared with the matching field parts of the flow entries. If there is a match, the device takes the necessary set of actions described in the matched flow entry. This can be a reactive (triggered by a arrival of a packet) or proactive process. A match is expressed through specific values on fields within packet headers and a flow match statement can be for instance a specific source IP address or a range of IPs for the packet or the protocol of the packet being TCP or UDP. In case of no matching, the set of taken actions is described on the instructions explained in the miss flow entry of the table. The miss flow entry is an essential component for each flow table in order to handle table misses. This entry describes a series of necessary actions in case of no matching such as dropping the packet, moving on the next flow table or forwarding the packet to the controller using the OpenFlow channel.

OVSDB [29] is another popular southbound interface. OVSDB was introduced as a part of OVS, but currently is supported by several switch platforms. The network administrators use Openflow to program the flow entries of a switch and OVSDB to configure the switch, itself. Using OVSDB, allows the engineers to create, delete and configure bridges, ports and tunnel interfaces on the switch.

Other interfaces that are suitable as a southbound standard are ForCES, POF, OpFlex, Openstate, ROFL, Hardware Abstraction Layer (HAL) and Programmable Abstraction of Data – path (PAD) [28].

3.1.2 Northbound Interfaces

Unlikely the southbound API, which is focused on the hardware, the northbound API is mostly a software-based environment. In these systems, the implementation is a central driver and the standardization mostly comes later. For the southbound interface, OpenFlow is a commonly accepted standard but there is not a widely accepted northbound API yet.

Defining an open and standard northbound interface is essential for the development and the promotion of application portability among the various control platforms. Each SDN controller has its own specific northbound API, and each of them has its own particular definitions. SFNet [30] is an example of a northbound API. It converts higher level application requirements into lower level service requests. However, the functionality of SFNet is restricted to queries that only ask the congestion state of the network, such as bandwidth reservation and multicast.

It is questionable whether a single northbound API would ever be developed and accepted as the common standard, as the network applications are quite complex with plenty different requirements. For instance, a northbound interface aiming at supporting security applications is most likely different from an API focusing on routing applications.

3.1.3 SDN Controllers

The configuration and management of most of the current networks is performed by using low-level unique commands and device-focused proprietary network operating systems, such as Cisco IOS [31] and Juniper JunOS [32]. In addition, the current network operating systems are not able to provide general functionalities for the entire network and they only provide device-specific functionalities.

SDN is a promising solution that is trying to face the current networking problems by separating the control plane from the data plane. The idea is to handle the control plane separately inside a central logical network operating system (NOS), which is known as the controller. The controller is an essential component of the SDN architecture and it is the cornerstone for configuring the network based on policies described by the network administrator. It is quite similar to the traditional operating systems and it decouples the system from the lower level device specific instructions on the forwarding devices.

The controller is basically the brain of the entire network and it has an overall view of all the network devices, the connections between them and the best routes between the hosts. This global view helps the controller act rapidly, efficiently and smartly regarding the direction of the flows, the control and the network recovery in case of a link error. The controller has predefined routes for every link in the network and in case of error an alternative route is always available. Another important feature of the controller is the capability to provide network virtualization. This capability allows the creation and maintenance of virtual networks on physical infrastructure in an abstract way, similar to server virtualization. Furthermore, a controller should be able to create the requested number of flows and maintain them. Thus, the flow creation time and the manageable number of flows per second are two important indicators for the performance of the controller. The number of the supporting APIs is another major factor for the controllers. The controller should support southbound APIs to control the switches and northbound APIs to control the upper layer applications. The more APIs a controller supports, the more flexible it becomes, and especially for the northbound APIs that allow the dynamic configuration of the network by the applications based on specific requirements. For example, the support of the OpenFlow protocol is really important for the majority of the available SDN controllers. This is a matter of high importance as the OpenFlow is the widely accepted southbound API, and it is responsible of taking the necessary decisions in case of no matching in the existing entries in the flow control table on the hardware switch. The controller has to decide whether to drop the packet or create a new flow to forward it.

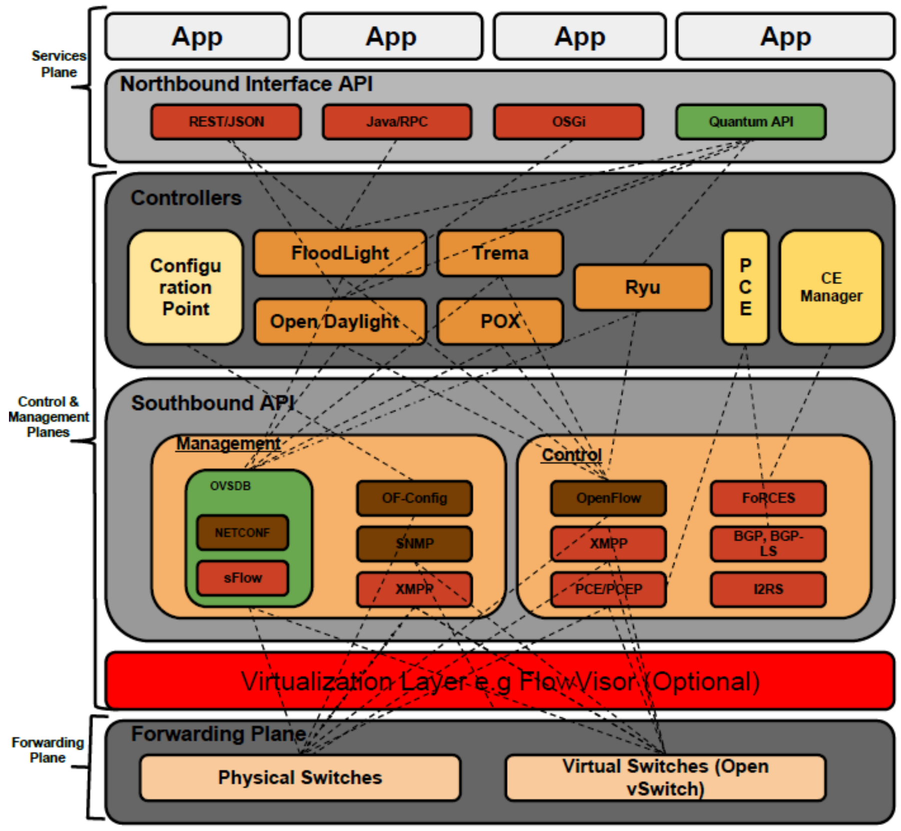


Figure 6: Interfaces and SDN controllers [38]

There is a huge variety of available controllers both commercial and open source. Some examples of commercial controllers [33] are: the Cisco Application Policy Infrastructure Controller (APIC), the HP Virtual Application Networks SDN Controller, the NEC ProgrammableFlow PF6800 Controller and the VMware NSX Controller. The most used open source controllers are POX, Ryu, Trema, Floodlight and OpenDayLight. POX [34] is a controller developed in Python, and it is focused on debugging SDN errors, developing programming models and managing virtualized network resources. *Ryu* [35] is a component-based controller, which means that it comes with a group of pre-defined components. The network operator can create custom controller applications by modifying these pre-defined components. The components can be developed in any programming language, and the choice is up to the operator and the requirements of the applications. *Trema* [36] is a controller supported by NEC labs and it aims to offer an easy to code system with high performance. Ruby is used as the scripting language to increase productivity, and C is used as the compiler language to boost performance. The *Floodlight* [37] controller includes a set of modules, and each of these modules grants a service to the other modules and to the control application via a Java API or a REST API. The controller is able to run on Linux, Mac and Windows systems.

OpenDayLight is explained with more details in the following sub-section, as it is the current most popular SDN controller among the users' community and the vendors, and further analysis seems necessary. Figure 6 illustrates the described SDN controllers with all the supported protocols and the connections between them.

Opendaylight

All the previously mentioned SDN controllers support only Openflow as the southbound protocol, however recent studies has showed that Openflow-based SDN architectures are inadequate for high-performance networks, and new architectures that support wide range of southbound interfaces should be invented.

Opendaylight (ODL) [39] was introduced by the industry to solve the previously mentioned problem. ODL is developed in Java and is supported by any hardware and operating system that has Java support. In addition, it is an open source project that tries to gather a contributing community around it to improve the code and use it for the development of commercial products.

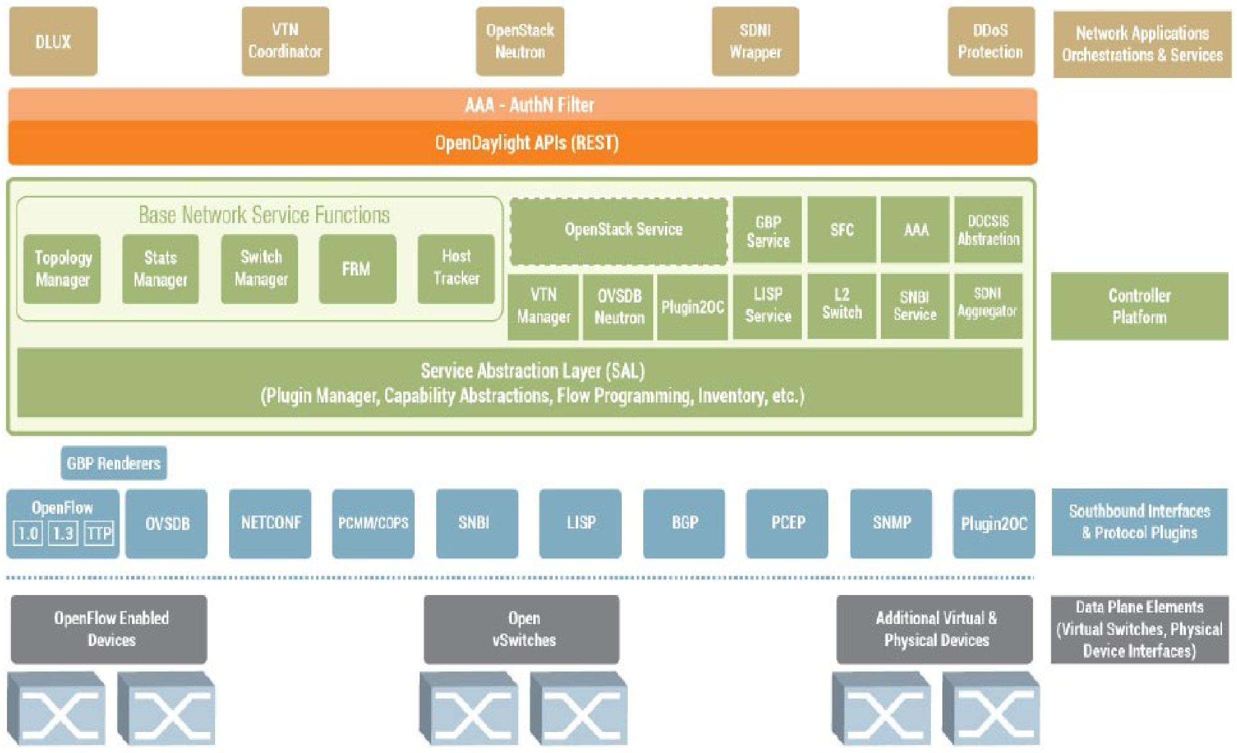


Figure 7: The architecture of Opendaylight [40]

The ODL architecture is represented at Figure 7. The architecture consists of three basic layers [40]: the network applications and services, the controller platform and the southbound interfaces. The controller controls the data plane elements, but they are a separate layer outside the controller.

A. Controller platform

The main elements of the *controller platform* are:

- *The Base Network Service Functions (BNSFs)*: These functions gather statistics and information about the components of the whole network, their state and their capabilities. They also offer access to the collected information to the network application via the northbound APIs. The current main built-in network services are: the topology manager, the statistics manager, the switch manager, the forwarding rules manager, the inventory manager and the host tracker.
- *The Platform Network Service Functions*: They are plugins services that improve the functionality of the SDN controller with specific networking tasks. Some noticeable pluggable services are the affinity metadata service, the virtual tenant network manager, the L2 switch, the service function chaining, the group-based policy and the authentication, authorization and accounting service.
- *The Service Abstraction Layer (SAL):* It is the main element of the ODL controller, as it enables the support of multiple southbound protocols via the southbound plugins. SAL also offers the Device Discovery service that is used by the Topology Manager to build the topology of the network and form the capabilities of the network components.

B. Southbound Interfaces

The main responsibility of the southbound protocols is to establish a communication between the SDN controller and the data plane elements. ODL uses the southbound plugins to support several southbound protocols that can manage, configure and monitor the data plane elements. Furthermore, the southbound protocols improve the compatibility of ODL with heterogeneous networks and other technologies. Some southbound protocols that are supported are OpenFlow, Open vSwitch Database (OVSDB), SNMP, BGP-LS/ PCEP and NETCONF.

C. Network Applications and Services

At the higher layer of the architecture, are the network applications and services that are responsible for the management, the control and the monitoring of the entire network. These applications are highly related with multiple services from the control platform, such as the VTN manager and coordinator. These applications are also responsible for the orchestration and the redirection of the traffic based on the requirements of the network. ODL offers northbound APIs that support communication between the applications of the top layer. The northbound APIs in ODL are mainly supported by the OSGi framework and bidirectional REST APIs. Some supported network applications are:

- the openDayLight User eXperience (DLUX) which is a web-based user interface.
- the VTN coordinator that offers REST APIs to the users to build and manage virtual networks.
- the SDNi Wrapper that offer communication between different SDN controllers.
- the DDoS protection which is an application that detects and protects against DDoS attacks.
- the Openstack Neutron that supports the creation of virtual networks in the Openstack cloud platform.

4 Software-Defined Computing

Software defined computing (SDC) [41] uses virtualization techniques to compute the necessary functions in the system. The virtualization technology decouples CPU and memory resources from the physical hardware, resulting to pools of resources that can be used wherever they are needed.

The industry uses two popular technologies to achieve SDC transformation of their systems: virtual machines (VMs) and containers. A *VM* is defined as an isolated software framework that runs its own operating system and applications as a physical machine. A VM in order to run requires a hypervisor which is a piece of software that is placed between the hardware and the OS. The hypervisor provides the capability to the hardware to share its resources among the VMs that run on top of it. The hypervisor is also responsible for managing and monitoring the VMs. A *container* [42] on the other hand is a unit of software that packages up code and all its dependencies, so that the application can run quickly and reliably from one computing environment to another. Containers act as sandboxes that isolate software from the environment, and grant the functionality of the application regardless of possible differences in the deployed environments. One advantage of the containerization technology over VMs is that they do not need a fully functional OS to run, like virtual machines.

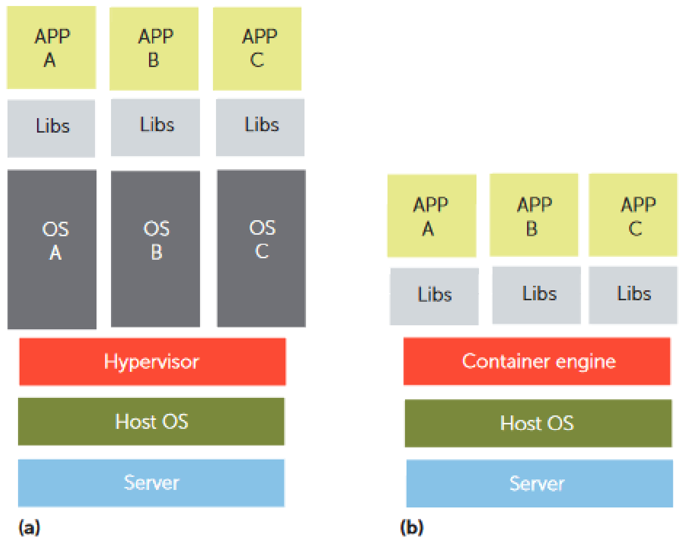


Figure 8: Comparison of hypervisor (a) and container-based (b) deployments [42]

4.1 Benefits of SDC

SDC abstracted the data center from the physical hardware and added flexibility and automation to the system, which was a huge challenge a decade ago. Some of the most important benefits of SDC are discussed in the following section [43].

Shared Resource Pooling

One of the biggest advantages of SDC is the elasticity that it offers to the system. By adding a virtual layer between the physical hardware and the operating system, the utilization of the resources becomes easier and much more flexible. The resources are grouped into logical pools and the virtual machines/ containers can use and share the same physical resource. Upon an increase in demand, the VMs /containers can be configured to use more resources, in contrast with the bare metal servers, where the resources are limited and it is difficult and sometimes even impossible to upgrade the hardware.

Server Consolidation Anywhere

A virtual server is basically a file and as file it can be deployed everywhere in an easy manner. It can be deployed in a data center and serve almost all the traffic, on the laptop of a software developer for running tests and developing code or it can run in the cloud, where users worldwide can manage it through a browser.

Automation

The levels of automation in the new systems have dramatically increased with the introduction of SDC. The system can automatically migrate VMs and containers in case of error with no actual impact on the user experience. The system also automatically monitors the performance and the traffic load on the VMs/ containers, and takes the necessary actions to reach balance and efficiency between the hosts and the requested resources. In addition, the system can automatically start new resources in case of high demand, and shut down unused resources in periods of low demand.

Prioritizing

SDC provides the user with the capability to give higher priority to some servers over others. The administrator can set a minimum threshold for resources of a critical server, ensuring that the server always has the necessary resources to perform some essentials functions. Furthermore, the administrator can determine the order in which the VMs/

containers mitigate in case of error. Standard system cannot be served by one-sized solution for everything, and SDC can solve this by providing the necessary granularity to manage each server independently.

4.2 SDC Tools

The main elements of the SDC technology, as already mentioned, are virtualization techniques, such as VMs and the relatively new containerization technology. The main software, that creates and monitors the VMs, is the hypervisor. The hypervisor is basically a software layer that allows several VMs with different operating systems to run simultaneously at a physical machine. The hypervisor can be divided into two categories: the Type1 and the Type2 hypervisors. Type1 hypervisors, also known as bare metal hypervisors, run directly on the hardware, while the Type2 run as applications on an existing operating system on the hardware. The most used Type1 hypervisor solutions are KVM, Xen, VMWare ESXi, while the most used Type2 hypervisors are Microsoft Hyper-V and Oracle VirtualBox. The most popular tool for creating and deploying containers is Docker. Other popular tools for running containers are lxc, runc and rkt.

KVM

KVM (Kernel Based Virtual Machine) [44] is a part of Linux and was developed to run on x86 machines. KVM has a kernel module called kvm.ko that transforms the Linux kernel into a hypervisor allowing the VMs to run directly on hardware. The user can use the Virtual manager (GUI) and virsh applications (CLI) to control and manage the created VMs.

Xen

Xen [45] is one of the most popular Type1 open-source hypervisors. It also comes in a commercial version from Citrix, and Oracle VM is another paid implementation of Xen. Xen is based on the para-virtualization technology, in which the host guest operating systems are aware that they do not run on their specified hardware. This requires some extra modification of the guest OS, but it comes with increased performance. The Xen Project platform is compatible with many cloud platforms such as Openstack and Cloudstack.

VMWare ESXi

VMWare is the biggest player in the hypervisor market in terms of revenue and total share. VMware offers both types of hypervisor. It offers ESXi, which is a Type1 hypervisor, and VMWare Fusion which is a Type 2 hypervisor and runs on desktops and laptops. VMWare ESXi can be downloaded for free [46]. ESXi is bare metal hypervisor that runs on a physical server directly. That increases its efficiency compared to hosted architectures and can partion hardware in an effective way to reduce costs for the customers, by increasing consolidation.

Oracle VirtualBox

VirtualBox [47] is a Type2 hypervisor that runs on Linux, Windows, Macintosh and Solaris hosts. It can run both on 32-bit and 64-bit host operating systems, which makes it highly portable. The VMs can be imported and exported in the VirtualBox by using the OVF (Open Virtualization Format). It also supports OVFs that were created by a different virtualization software.

Docker

Docker [48] is the main tool for creating, deploying and running containerized applications. Docker is a bit similar to a virtual machine hypervisor, but unlike a hypervisor, Docker allows the applications to use the same Linux kernel as the system that they run on, and the only requirement is to include components, such as dependencies and libraries which are not installed on the host machine. This increases the performance and the scalability, and also lowers the size of the application. Docker is an open source framework with a huge community that supports and contributes to it.

5 Preparation of the design - Case Study

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.

5.1 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* [22] *:* 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.

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*	*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*

Table 3: Requirements of the system

5.2 Overview of the Design

The purpose of this chapter is to combine the acquired knowledge regarding the software-defined technology of Chapters 2- 4 with the requirements identified in Chapter 5.1 with the higher goal of designing a software-defined system from the ground up.

Figure 9 depicts an overview of the high-level steps required for the design of the software-defined system. This design procedure is the most appropriate approach based on the personal understanding of the software-defined technology in general and the interconnections between the relevant tools. Step 1 is the provisioning of the infrastructure, therefore a suitable tool should be selected. In step 2, a dynamic platform that offers SDN features is selected, while the choice of an adequate infrastructure configuration tool takes place in step 3. Afterwards, in step 4 the entire infrastructure layer should be checked whether it offers support of the SDN / NFV technologies, and step 5 is the selection of a SDN controller for the management and configuration of the entire network of the system. A more detailed description of the design steps and the tool selection can be found in the following sections.

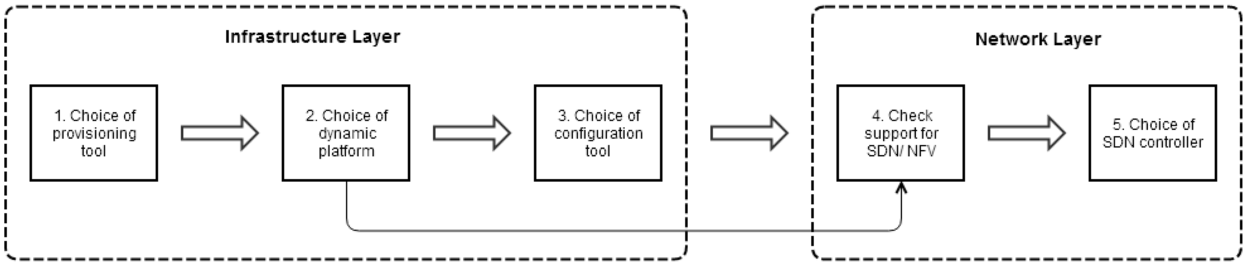

Infrastructure Layer
1. Choice of provisioning tool
2. Choice of dynamic platform
3. Choice of configuration tool
Network Layer
4. Check support for SDN/ NFV
5. Choice of SDN controller

Figure 9: Design steps for the software-defined system

The realization of the infrastructure and the network layer of a generic software-defined system is shown in Figure 10. This illustration is focused on the fundamental components and concepts of a software-defined system and the interactions between them, and it is highly based on the design steps from Figure 9. The arrows in Figure 10 depict the communication between the components, which is performed mainly by using APIs.

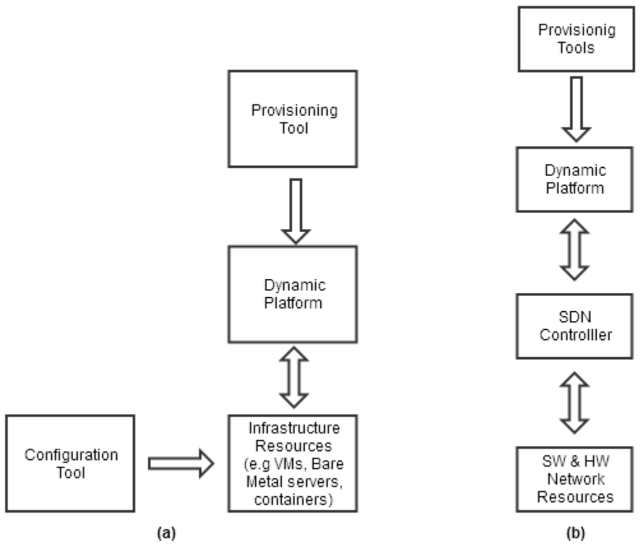


Figure 10: Realization of the infrastructure (a) and network (b) layer of a generic software-defined system

The main components of the infrastructure layer are:

- *Provisioning Tool*: The user describes the desired infrastructure resources in the definitions file that differ based on the selected tool, and the provisioning tool translates these files via API calls into instructions for a dynamic platform that sets up the requested resources.

- *Dynamic Platform*: The main building block of the infrastructure. It is responsible for setting up the computing resources of the system. The dynamic platform should be compatible with a declarative provisioning tool in order to achieve central control and management of the system. The dynamic platform can be a cloud platform or a software tool that is able to provision computing resources.
- *Infrastructure Resources*: The main computing elements of the system where the applications are deployed. They can be bare metal or virtual servers along with clusters of containers.
- *Configuration Tool*: Configures and installs the required software/ applications on the provisioned computing resources.

While the network layer is composed of:

- *Provisioning Tool*: The same provisioning tool as the infrastructure layer. In this case, the user describes the capabilities of the network resources and is able to group the computing resources into separate networks, which can be physical or virtual.
- *Dynamic Platform*: The same dynamic platform as the infrastructure layer. It should be able to create virtual and physical networks and equip them with the necessary network resources.
- *SDN Controller*: The central "brain" of the network layer that controls, manages and configures the network resources. The controller should be able to control and configure physical and virtual network resources such as switches, and should provide northbound APIs that communicate with the dynamic platform. The idea is that the dynamic platform creates the networks and the controller manages them.
- *Network Resources*: The main elements that provide the desired functionality of the network layer. The typical example of network resources are the virtual and hardware switches, which are responsible for forwarding data between the networks. Other common network resources are firewalls, data diodes and load balancers.

5.3 Selection of Tools

The selection of tools that are used in the designed system is mainly based on the list of requirements of Chapter 5.1. 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 shown in Figure 9. A complete list of the selected tools can be found in Table 4.

Scope	Tool	Reasoning
Provisioning	Terraform	• Open source • Supports multiple dynamic platforms. • The alternative options are vendor specific solutions.
Dynamic Platform	Openstack (VMs)	• Open source • Private cloud solution • Extensive support by Terraform • Provides SDN functionality (Neutron)
	Cobbler (Bare Metal)	• Open source • Supported by Terraform • Alternatives are proprietary solutions that do not offer private functionality.
Configuration	Ansible	• Open source • Agent-less deployment • Fast and reliable • Secure as it uses SSH to push modules • Supports dynamic inventories • Already used in Thales
SDN Controller	OpenDaylight	• Open source • Support of wide variety of southbound protocols, including Openflow, OVSDB, NETCONF • Openstack compatible

Table 4: An overview of the selected tools

5.3.1 Provisioning Tool

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 at Table 1 in Section 2.3.3. 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.

5.3.2 Dynamic Platform

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 [50]. 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 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 [51] is the SDN Openstack project that provides networking-as-a-service (NaaS) in virtual compute environments. Furthermore, Openstack Magnum [52] is an Openstack project that enables the use of container orchestrators such as Kubernetes[1] [53] 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 [54]. 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 [55] 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.

5.3.3 Configuration Tool

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

1. Kubernetes is an open source software that automates, scales and manages the deployment of containerized applications. Kubernetes is mainly used with Docker to build and manage containers.

inventories [56], 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.

5.3.4 SDN Support and Controller

A programmable network infrastructure is also crucial for the proper functionality of the new software-defined system. A virtualized network layer is more adjustable to changes and can offer higher levels of scalability when compared to the traditional physical networks, which are highly dependent on the hardware. Therefore, the entire infrastructure layer should enable the use of SDN technology.

As it is mentioned in Section 5.3.1, Openstack Neutron offers several SDN features to the system. Neutron can create, delete and configure virtual networks and the connections between them. In addition, Neutron can virtualize some particular network components, such as switches, gateways, routers, firewalls and DNS. Openstack uses the Open vSwitch technology to virtualize the network layer. Open vSwitch creates the virtual switches that connect the virtual machines to each other, forming virtual networks. Basically, the virtual machines connect to virtual ports on virtual bridges on top of a virtual switch. Furthermore, most of the networking features of Openstack are also supported by Terraform, offering the desired declarative approach (Requirement 2) to the network layer.

Openstack supports the connection of a virtual network to an external physical network. However, it is unable by default to manage and configure the physical networks. The configuration and management of the physical switches requires the additional support of a SDN controller. OpenDaylight (ODL) was selected among the available SDN controllers. ODL is the only available SDN controller that provides supports for a wide range of southbound protocols, such as Openflow, OVSDB and NETCONF. The Openflow and OVSDB plugins enable ODL to manage both virtual and hardware switches. In addition, ODL is compatible with Openstack, forming a complete solution for the network layer that combines both the virtual and the physical networking infrastructure into one functional system. Moreover, the open-source nature of ODL fulfills the requirements for this case study (Requirement 3).

6 Infrastructure Design

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 in Chapter 7.

6.1 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.

6.1.1 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 (Figure 11). 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 [60] 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 [61] 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 [62]
  - *I/O*: Number of physical I/O that the VM can use[63].
- *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 [58]*: 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.

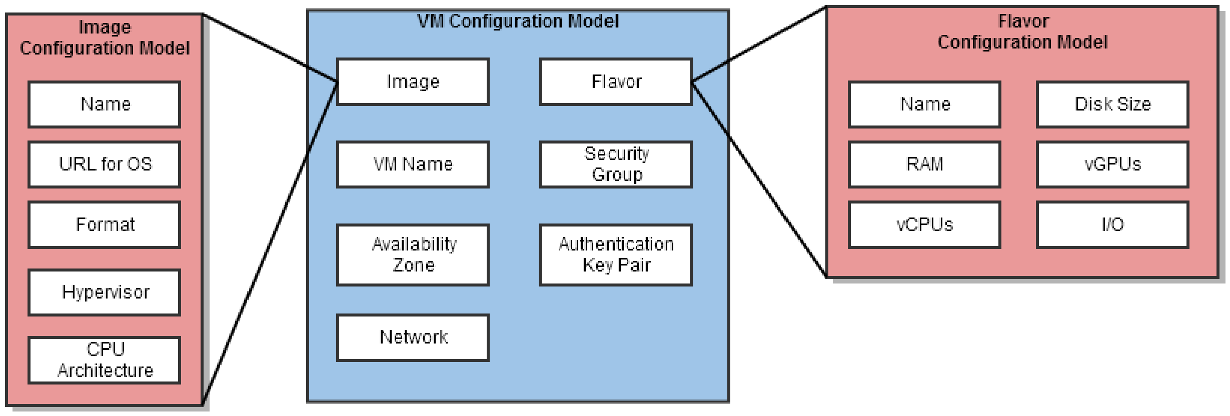


Figure 11: VM configuration model

More technical details are available in Appendix A.

6.1.2 Provisioning Bare Metal Machines

Terraform uses the Cobbler provider [64] to provision bare metal machines. Cobbler performs provisioning by using the PXE standard to boot the machines over the network. PXE [65] 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 [66] 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.

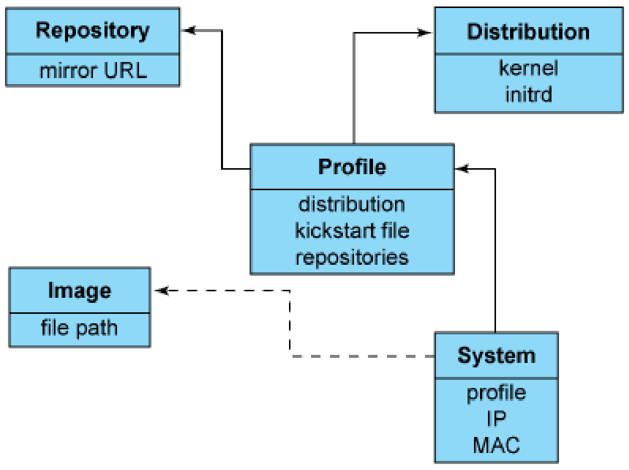


Figure 12: Overview of the objects in Cobbler [66]

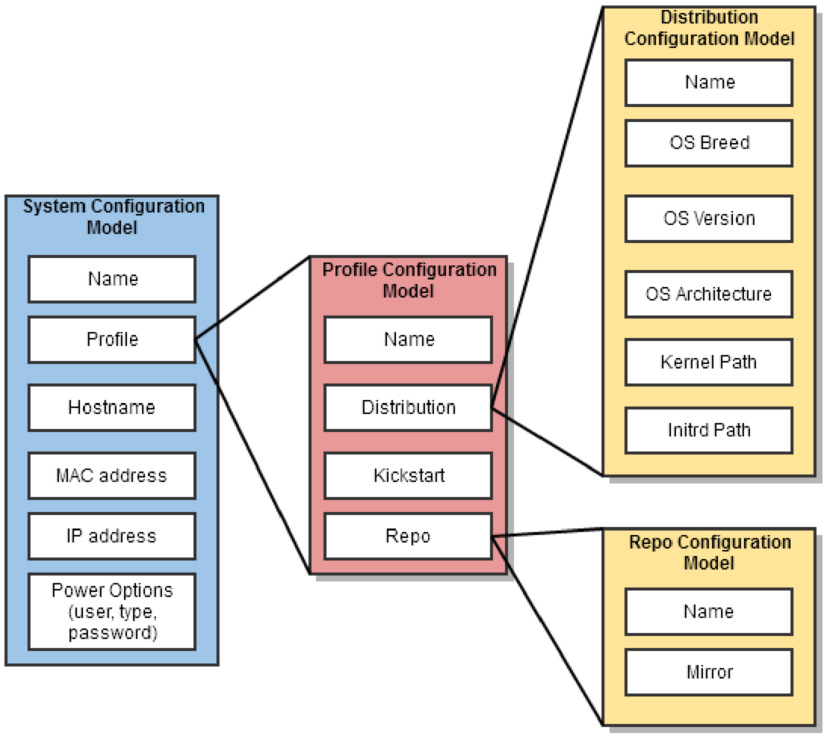


Figure 13: Bare Metal Configuration Model

Terraform uses the Cobbler provider to translate all the Cobbler objects to the corresponding Terraform resources for provisioning bare metal machines (Figure 13). The *system* [67] 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 [68] 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 [69] 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 *repo* resource: The user creates repos [70] 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.

More technical details can be found in Appendix B.

6.1.3 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 [71]). The descripted Magnum features are integrated in Terraform.

The configuration of a cluster [72] 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 [73] 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).

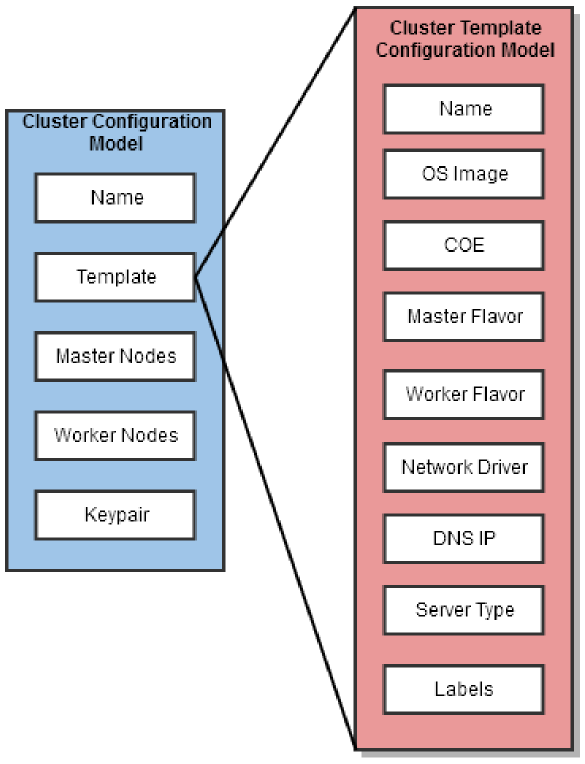


Figure 14: Container cluster configuration model

More technical details can be found in Appendix C.

6.2 Configuration

The configuration of the provisioned resources is the next step towards the development of a functional software-defined system. The automation of the configuration procedure is performed with Ansible, which is installed on the central control nodes and can configure and install software on the machines by pushing out small programs, called *modules*. Once modules are installed, Ansible uses SSH to execute them and removes them automatically after their execution.

In this specific case study, Ansible should manage virtual machines created with Openstack and bare metal servers provisioned by Cobbler. As a result, there is a fluctuation at the inventory of Ansible as new hosts are spinning up and shutting down in response to the demands of the system. Ansible supports these options via a dynamic external inventory system [56]. Ansible offers two different techniques for supporting dynamic inventory: the inventory plugins and the inventory scripts. The inventory plugins are a more recent solution and they use the latest updates of the Ansible's core. However, only Openstack has support for an inventory plugin and Cobbler should be managed using the inventory scripts.

Ansible should be able to select hosts for configuration based on specific attributes such as the OS distribution, the CPU architecture, network interfaces, etc. Ansible supports this feature by using *facts* [74], which are basically system properties that are collected by Ansible when it is executed on a machine. Facts can be collected into a file as output for reporting the state of the system or they can be used into a playbook for making runtime decisions.

More technical details on the configuration with Ansible are available in Appendix D.

6.3 Architecture Realization of the Infrastructure Layer

The purpose of this section is to map the described selected tools from the previous chapters to the generic components described in Figure 10a and explain the interconnections between them in order to realize the architecture of the infrastructure layer. The architecture realization of the supported infrastructure layer is shown in Figure 15. 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.

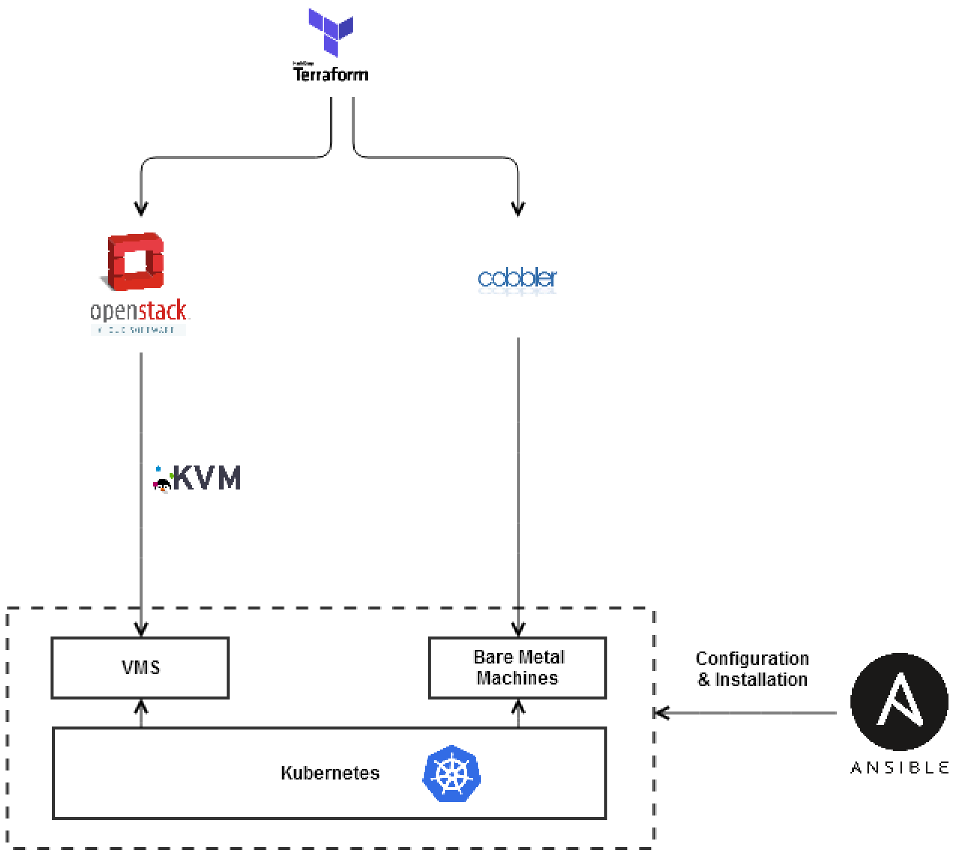


Figure 15: Architecture realization of the supported infrastructure layer

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 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.

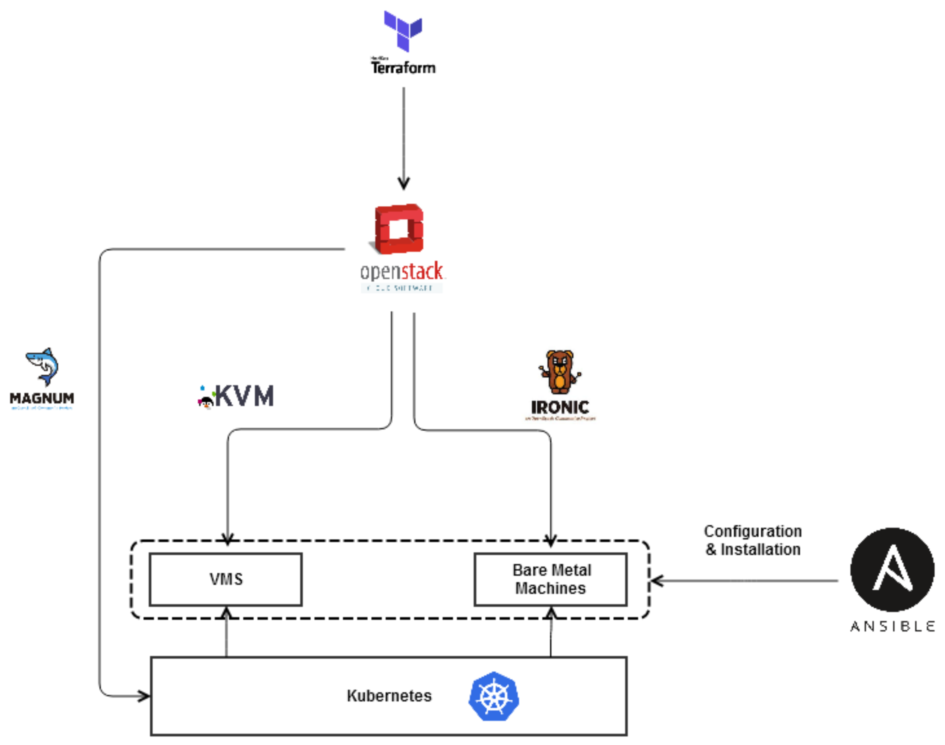


Figure 16: Architecture realization of the future possibly improved infrastructure layer

Figure 16 depicts 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.

7 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 software-defined 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.

7.1 Creating virtual networks

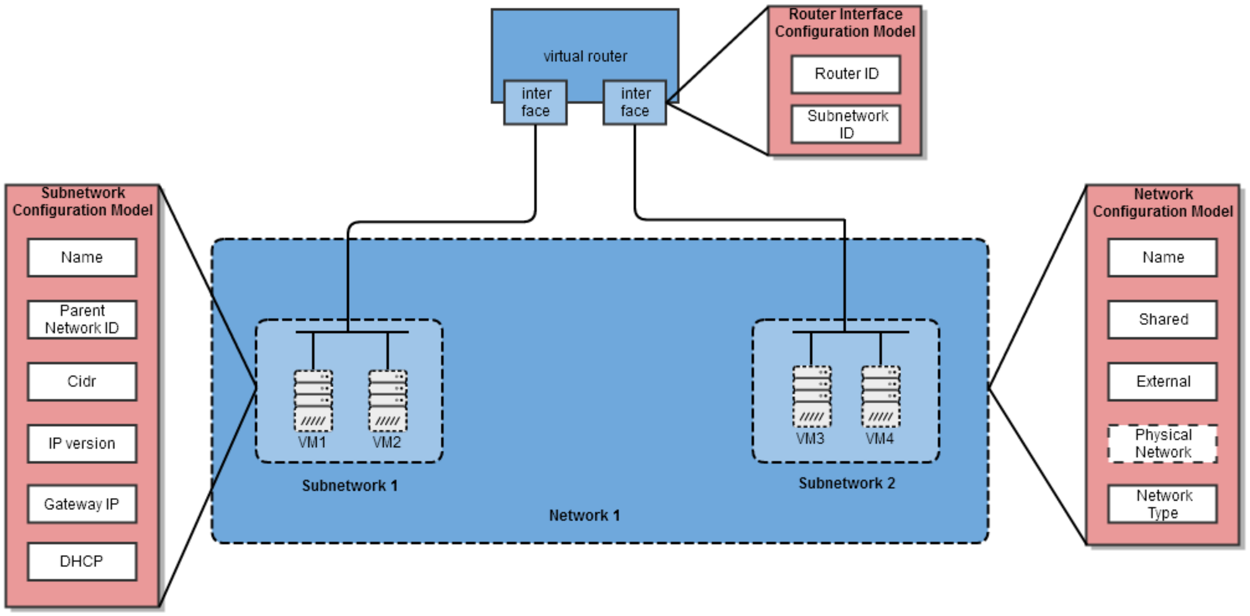


Figure 17: Configuration models of the network components

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 [75] 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 [76] 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 [77]. The subnetworks are connected to a virtual router by using the *router interfaces* [78], 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* [79]. 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.

More technical details are available in Appendix E.

7.2 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.

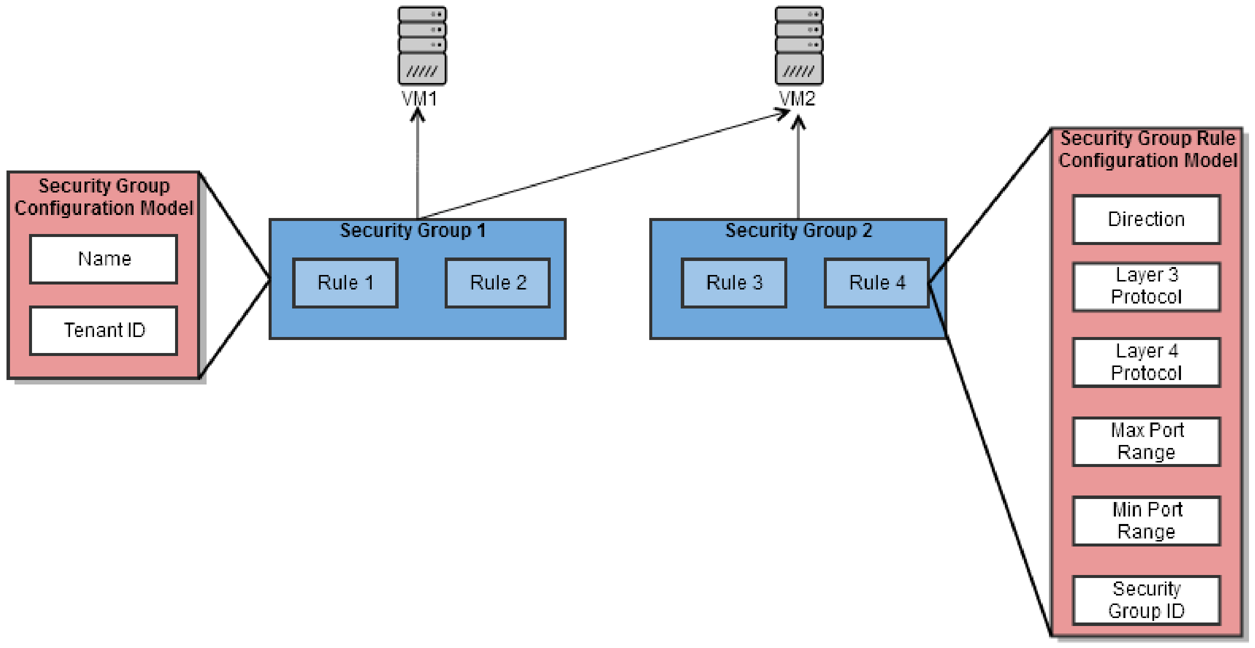


Figure 18: Security group configuration model

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.

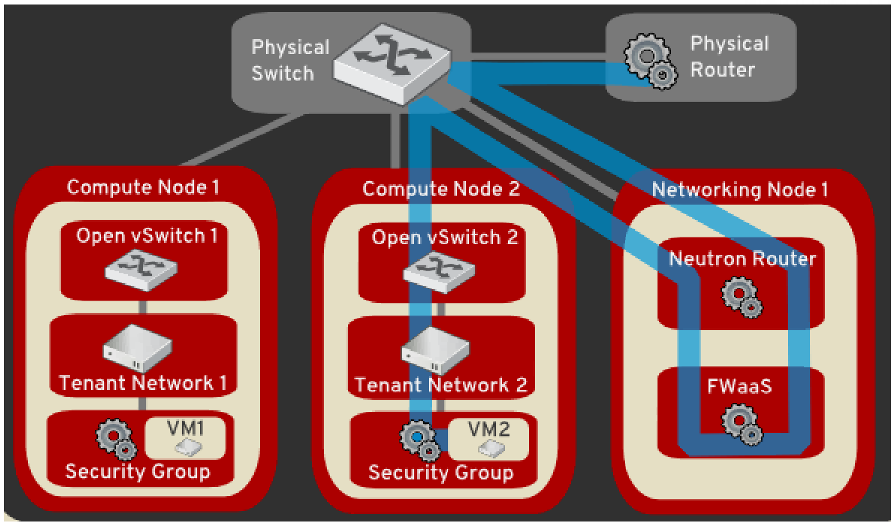


Figure 19: FWaaS and security group protection [83]

More technical details about security groups can be found in Appendix F.

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.

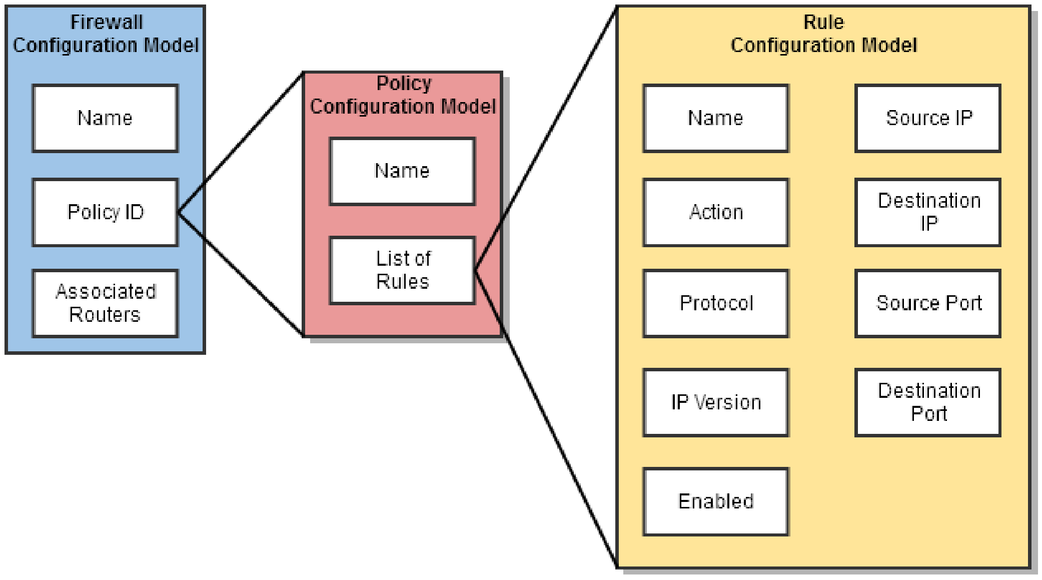


Figure 20: FWaaS configuration model and related concepts

Firewalls, policies and rules are created using unique separate Terraform resources, whose configuration is depicted in Figure 20. The configuration of a firewall [84] consists of:

- *Name*: A unique name for the firewall resource.
- *Policy ID*: The ID of the used policy resource. The user can create several policies [85] using the corresponding Terraform resource. A policy resource consists of:
    - *Name*: A unique name for the policy resource.
    - *Rules*: An array of one or more firewall rules that characterize the policy. Rules are created using a different Terraform resource, whose configuration include:
        - *Name:* A unique name for the rule resource.
        - *Action:* The user defines the action in case of a firewall match (e.g., allow or deny)
        - *Protocol:* The protocol on which the firewall functions *(*e.g., TCP, UDP, ICMP)
        - *IP version:* E.g., IPv4 or IPv6
        - *Source IP:* The source IP address on which the firewall rule operates.
        - *Destination IP:* The destination IP address on which the firewall rule operates.
        - *Source port:* The source port on which the firewall rule operates.
        - *Destination port:* The destination port on which the firewall rule operates.
        - *Enabled*: The user can define whether the firewall is enabled or disabled.
- *Associated Routers*: A list of routers that this firewall should be deployed on.

More technical details about the configuration of FWaaS can be found in Appendix G.

7.3 Connectivity with physical networks

In order to form a functional network layer that supports a wide variety of use cases, it is necessary to be able to connect the virtual network infrastructure created by Neutron with the current physical networks (Requirement 14).

The implementation of a SDN controller is required for the configuration and management of both the virtual and the physical network infrastructure. The virtual network infrastructure consists of virtual machines and containers connected to each other via virtual switches such as Open vSwitch, while the physical infrastructure contains bare metal servers that are connected via real hardware switches such as ToR (Top of the Rack) switches. Opendaylight is the selected SDN controller, as it offers a wide range of southbound protocols, including Openflow and OVSDB, and it is highly compatible with Openstack. The selected network switches (virtual and physical) should support the VXLAN technology and the OVSDB and Openflow protocols in order to provide the required connectivity and be properly controlled by the ODL controller.

7.3.1 VXLAN and VTEPs

The connectivity between virtual and physical networks is provided by using VXLAN technology. VXLAN [87] offers scalability, extensibility and flexibility, and provides multi-tenancy across the data centers by extending the Layer - 2 connectivity over Layer - 3 segments, making it a suitable solution for cloud environments. VXLAN handles the high scalability of the ever-expanding cloud systems by extending the 12-bit segment ID of the simple VLAN technology to a 24-bit one, providing around 16 million IDs for networking [88]. VXLAN uses VTEPs (VXLAN Tunnel Endpoint) to map end devices to VXLAN and encapsulate and de-encapsulate VXLAN traffic. A VTEP can be either a hardware VTEP (e.g., a ToR switch) or a software VTEP (e.g., OVS). A VXLAN tunnel can be established between a hardware and a software VTEP.

7.3.2 OVSDB and Openflow support

The switches (virtual and physical) should support both the OVSDB and the Openflow protocol to be correctly configured by the controller. The components of a physical switch that supports Openflow and OVSDB can seen in Figure 21.

OVSDB is used for the configuration of the switch (e.g., configuration of bridges, ports, tunnel interfaces). In an OVSDB enabled switch, there is always an OVSDB server, which is controlled by the OVSDB client. The OVSDB client is deployed both on the switches and the SDN controller. The server maintains an OVSDB database schema. For the virtual switches (OVS), the schema is called OVS database schema, while for the physical switches, the schema is called HWVTEP schema [89]. The schema stores in several tables control and statistical details gathered by the OVSDB clients on the switches and the SDN controller. The OVSDB client either on the controller or on the switches, monitors the schema for modifications to this information and can add or delete rows to the schema. This information provides a method through which the switches and the SDN controller can exchange information [90]. For instance, the physical switches capture MAC addresses of entities in the physical network and push this information to the schema so that the SDN controller that is connected to that physical switch can access the MAC addresses. The other way around, the SDN controller can collect MAC addresses of entities from the virtual network and push these data to the schema. As a result, the physical switches with connections to the SDN controller can access the MAC addresses.

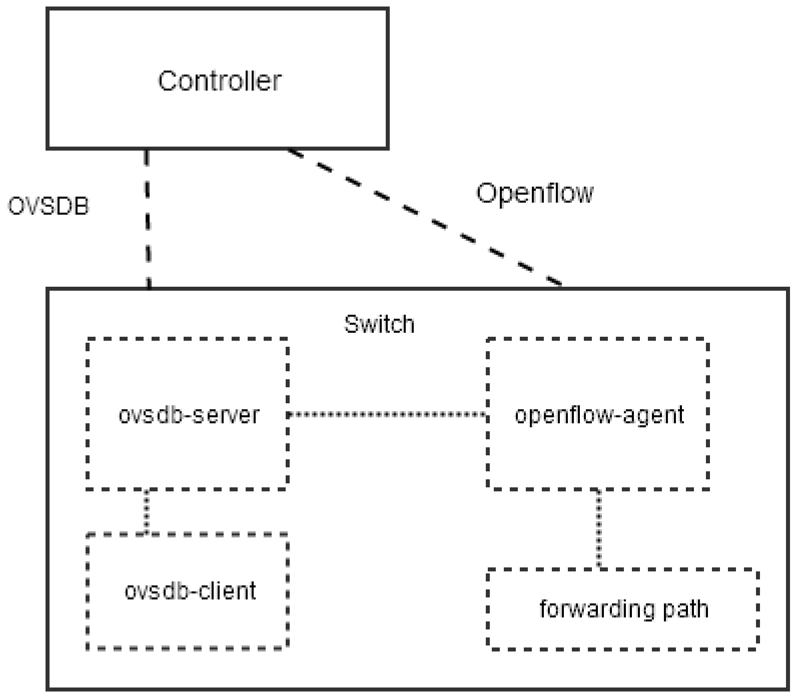


Figure 21: Components of a Openflow & OVSDB physical switch

In contrast, the Openflow protocol is used to control the forwarding pipeline of the switch by managing the flow entries, which are match-action rules that dictate what action needs to be taken when a specific match is encountered. An Openflow switch is equipped with an Openflow agent that creates the connections to the SDN controller. The Openflow agent speaks with the database on the switch to get information about the configuration of the switch.

The architecture of an OVS (Figure 22) is different from the architecture of a physical switch. The OVSDB server remains the same, but the OVSDB client and the Openflow agent are integrated into an entity called ovs-vswitchd, which is basically a daemon that implements the virtual switch.

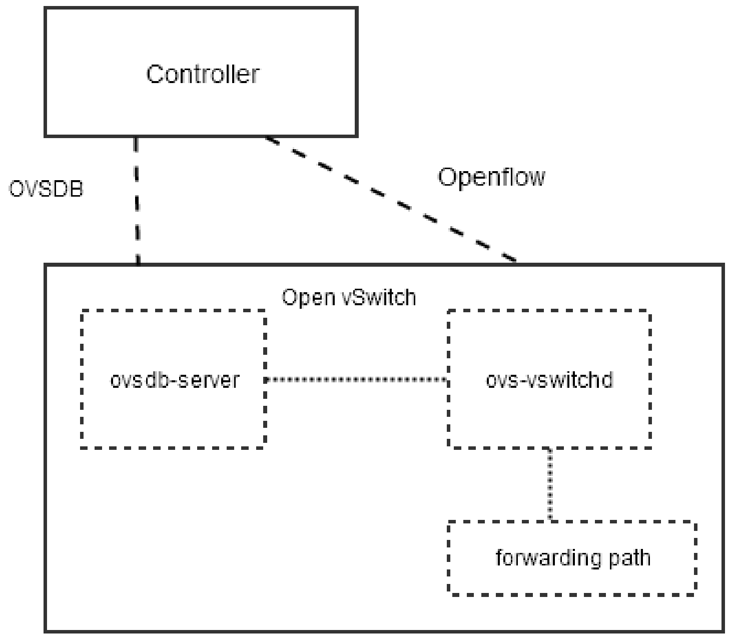


Figure 22: Components of an Open vSwitch [91]

7.3.3 L2 Gateways

Openstack Neutron needs to have the L2 Gateway [92] plugin installed in order to support the connectivity with physical networks. The L2 gateway plugin is a relatively new service that basically bridges the virtual VXLAN tenant networks created by Neutron to physical VXLAN network via the OVSDB Hardware VTEP enabled physical switch.

The L2 Gateway solution has two main components [93]:

- *L2GW service plugin:* It is deployed on the control node of the system. It is responsible for notifying the L2GW agent and normal L2 OVS agents running on compute hosts about changes in the network and share information about the VTEP IPs between them.
- *L2GW agent:* It is deployed on a network node. The main responsibility of this agent is to connect the OVSDB server running on the hardware switch and change the database based on information received from the L2GW service

plugin, so it basically acts as an ovsdb-client. The control node and the network node could be the same machine.

7.3.4 Configuring the connections

Opendaylight should have an overview of all of the devices that are part of the network. For that purpose, ODL uses *Transport Zones* which are logical groups of all the devices that are part of the network. In the Transport Zone format, the devices are called *Tunnel Endpoints (TEPs)*. The user should declare all the physical switches in the Transport Zone by passing the ID of the device. An example of Transport Zone JSON file and more technical details can be found in Appendix H. The TEPs of the virtual switches are automatically passed to Opendaylight by Neutron and they should not be declared into the Transport Zone file.

The next step is the configuration of the ToR switches, which is highly dependent on the vendor of the switch. However, the main configuration steps are almost the same, namely the configuration of the interfaces on the switch that connect to the bare metal machines and the configuration of the connection with the manager software (Opendaylight in our case).

The following configuration step is the creation [94] of a L2 Gateway on the ToR switch. The user should first create a provider VLAN network that has external routing facility and is accessible by users/ tenants. The next step is the creation of a subnetwork on top of this network. The IP range of this subnetwork should include the IP addresses of the bare metal machines that consists the physical network.

After the creation of the Opestack subnetwork, the user should create a L2 gateway and relate it to the corresponding physical TOR device by using the specific commands. The commands to do this are represented in Appendix H.

The final step is the connection of the L2 Gateway to a provider Openstack network, which would result in the creation of the VXLAN tunnels. The command to implement this can be found in Appendix H. If there are several ToR switches, the user can create several L2Gateway connections to each of the switches. The L2Gateway support is a relatively new feature of Openstack Neutron and it is not supported by the current version of Terraform.

7.4 Architecture Realization of the Network Layer

A suggested architecture realization of the network infrastructure is depicted in Figure 23. This architecture maps the components of the generic software-defined network layer from Figure 10b to the specific products and features described in Chapter 7. For the integration of Opendaylight with Openstack, the networking-odl plug-in has to be installed on Openstack. This plug-in passes the Openstack network configuration to the Opendaylight controller and basically offloads all the networking tasks of Neutron to Opendaylight. REST APIs are used for the communication between Openstack and Opendaylight. Opendaylight uses NetVirt [95] which is a network virtualization solution for supporting the Neutron features of Openstack.

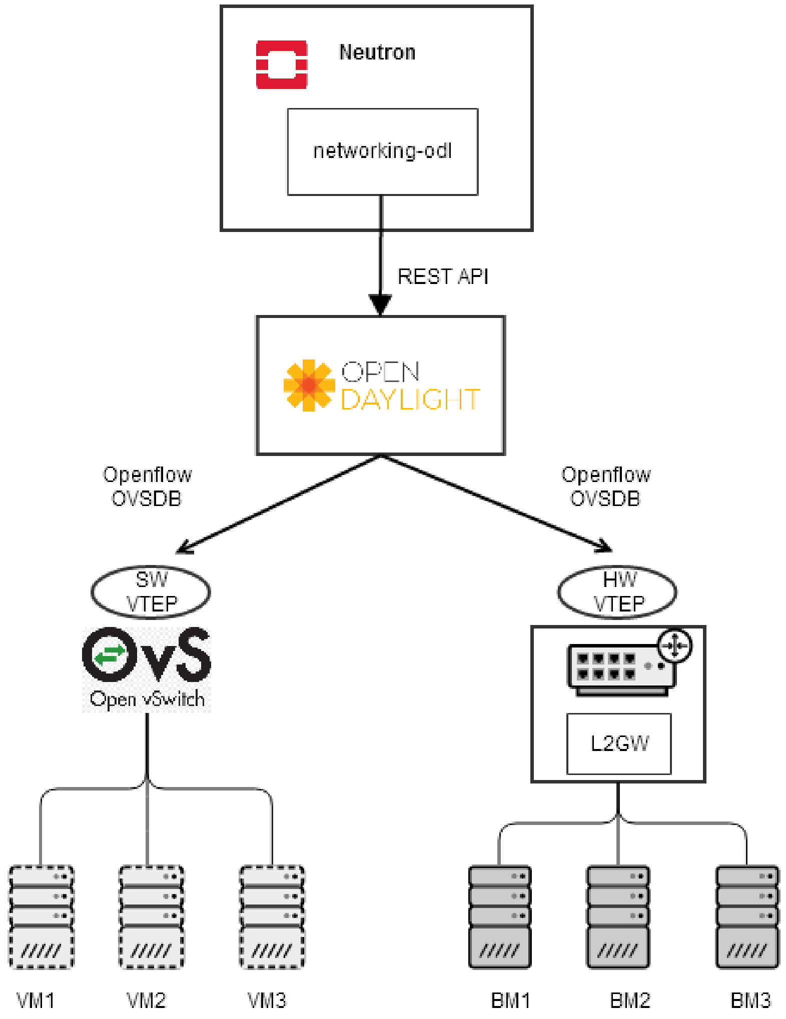


Figure 23: Architecture realization of the network layer

The connectivity between the virtual networks and the physical networks is provided by deploying VXLAN tunnels between the virtual and the hardware switches. In order to achieve that VXLAN connectivity, Neutron deploys a L2GW on the physical switches. Opendaylight configures the OVS switches (SW VTEP) created by Openstack-Neutron and the physical ToR switches (HW VTEP) using the Openflow and OVSDB protocols. The physical switches should support Openflow, OVSDB and VXLAN encapsulation.

7.5 Traffic flow within the network

In order to have a better understanding of the network infrastructure, the traffic flows between different components of the network are displayed and explained. A simple network topology was defined for that purpose. This topology consists of a control node, a network node, two compute nodes, two bare metal servers and an external component (e.g., a radar for the Thales case). A hardware switch that supports Openflow, OVSDB and VXLAN encapsulation, provides the required connectivity between all these components.

The OpenDaylight controller is deployed on the control node, while the networking features of Openstack, such as the L3 router, are installed on the network node. The control, the network and the two compute nodes have Open vSwitch installed, whereas none of the two bare metal servers support Open vSwitch and they form an external VXLAN network. Each compute node hosts two virtual machines (VM1 and VM2 on compute node 1 and VM3 and VM4 on compute node 2 accordingly). VM1, VM2 and VM3 are in the same Openstack tenant VXLAN network (Network 1). Network 2 is a provider VXLAN network that consists of the external bare metal servers 1&2. VM2 is also part of Network 2. The external component (e.g., the radar) forms a flat provider network (Network 3).

Name	**Type**	**Components**
Network 1	*VXLAN tenant*	*VM1, VM2, VM3*
Network 2	*VXLAN provider*	*Bare Metal 1 &2, VM4*
Network 3	*Flat provider*	*External Component*

Table 5: Networks of the topology

Four traffic scenarios were defined. The first three scenarios use the default functionality of Neutron for managing the L2 and L3 traffic and the functionality of the Opendaylight controller is not used. Scenario 1 shows the traffic that flows between the virtual machines, while scenario 2 depicts the traffic between the bare metal servers and the virtual machines. Scenario 3 examines the exchanged traffic between the radar and both the virtual and the bare metal machines. In Scenario 4, the Opendaylight controller is used to manage entirely the virtual and the hardware switches. This scenario leverages the functionality of the SDN controller to forward traffic between different networks.

Scenario 1: Traffic between virtual machines

In this scenario VM1 exchanges data with VM2. These two VMs are in the same network and they share the same host. As a result, the traffic will be forwarded by the virtual switch in the compute node 1. The traffic flow between VM2 and VM3 is depicted with the green line. These VMs are still in the same network but they are deployed on different hosts. Therefore, the exchanged data should pass also through the physical switch that connects the two compute nodes. On the other hand, the traffic between VM3 and VM4 has also to pass through the L3 router on the network node in order to arrive to its destination. This happens because VM3 and VM4 are in different networks (VM3 in Network 1 and VM4 in network 4).

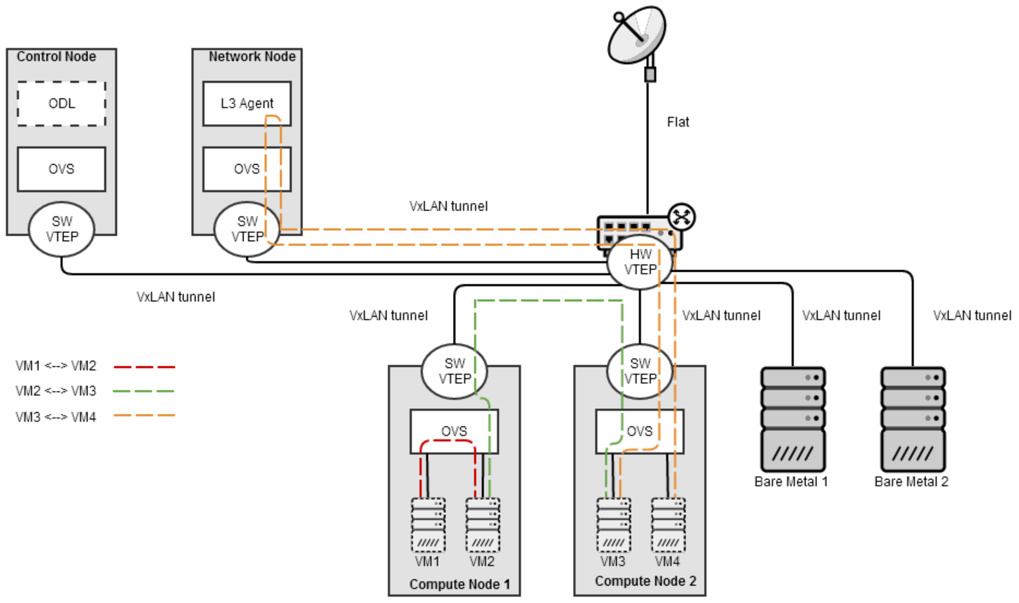


Figure 24: Traffic flows for scenario 1

Scenario 2: Traffic between bare metal machines and virtual machines

The two bare metal servers are in the same VXLAN provider network (Network 2) and they can directly communicate to each other through the physical switch. VM4 is also a part of Network 2, so it can exchange traffic with the bare metal servers (e.g. Bare Metal 1 in Figure 25) through the Open vSwitch and the physical switch. L3 router is not used in this case, as the two components are in the same network. The VXLAN provider network has connectivity with the Openstack components via the L2 GW that is assigned on the physical switch. However, that does not apply for the traffic flow between VM1 and bare metal 2. These components are in two separate networks, thus the L3 router on the network node is used for forwarding the traffic to the proper destination.

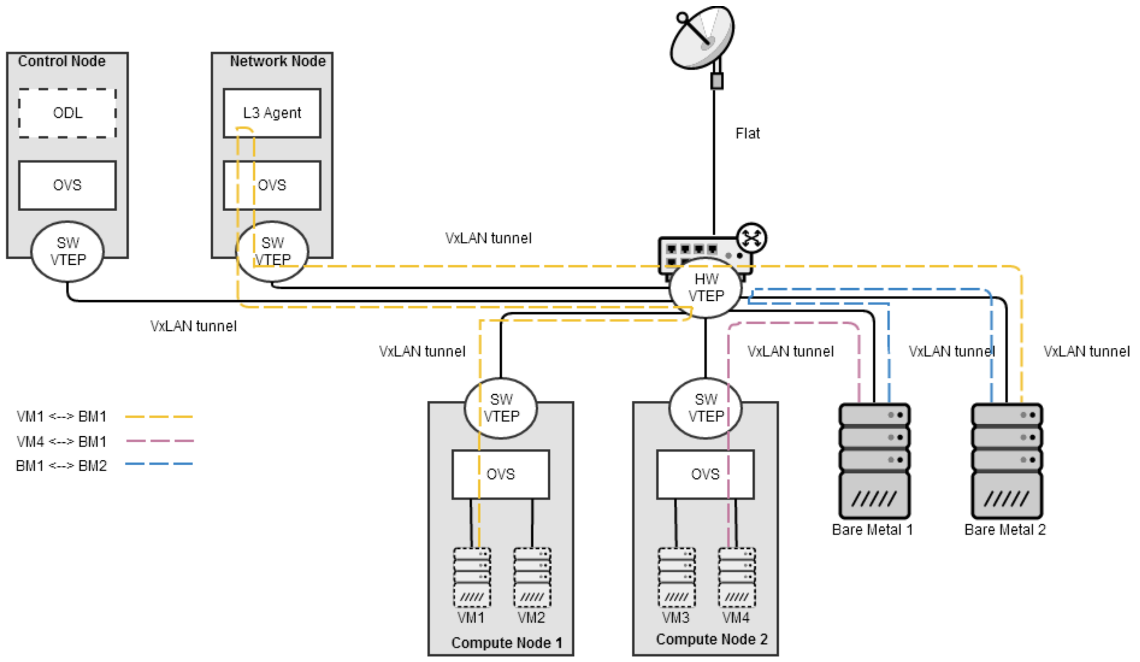


Figure 25: Traffic flow for scenario 2

Scenario 3: Traffic with an external component

The external component (e.g., a radar) is in a separate flat provider network (Network 3). As a result, all the traffic that is exchanged between the radar and the other components of this topology should be directed to the L3 router in order to be properly forwarded.

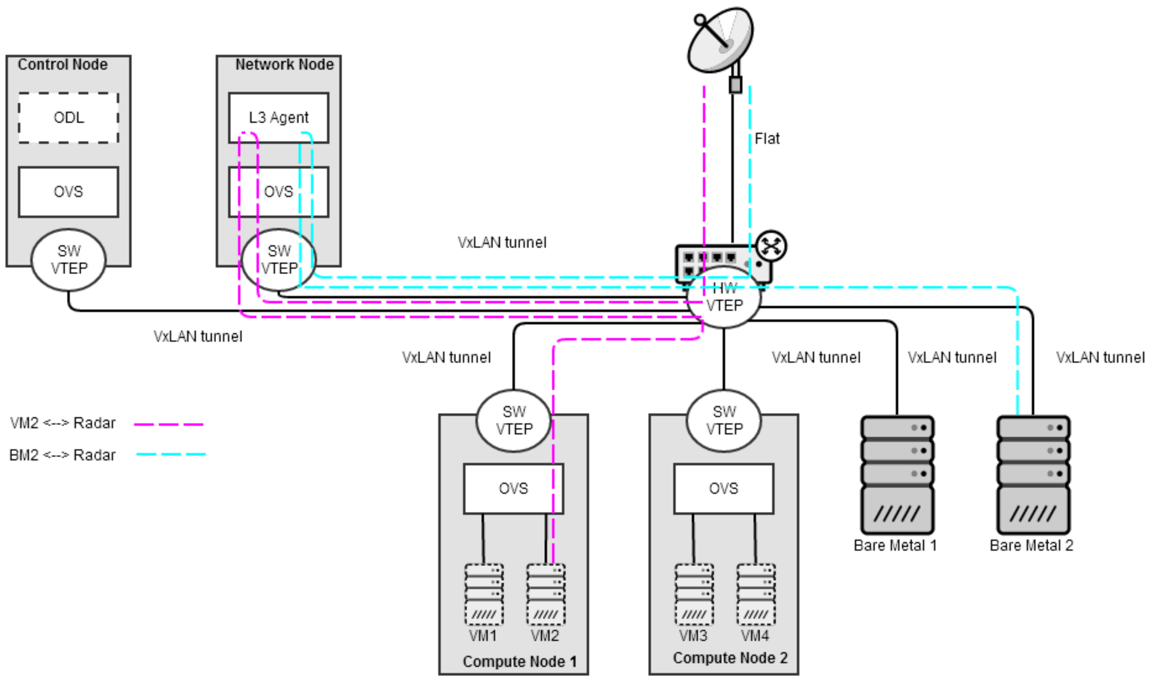


Figure 26: Traffic flows for scenario 3

Scenario 4: Traffic managed by Opendaylight

The L2 and L3 forwarding is managed entirely by the Opendaylight controller in this scenario. This eliminates the need for the Neutron L3 agent for routing. The controller uses Openflow and OVSDB to configure the switches to route the incoming IPv4 packets. IPv6 packets are not supported [96]. As a result, the L3 forwarding is performed on the switches and not on the network node in which the Neutron L3 agent is deployed. Floating IPs are also supported by Opendaylight driver, however, the FWaaS feature is not currently supported.

As it is shown in Figure 27, traffic between different networks is not forwarded to the Neutron L3 agent anymore. Instead the switches take care of routing packets between instances that are part of separate networks.

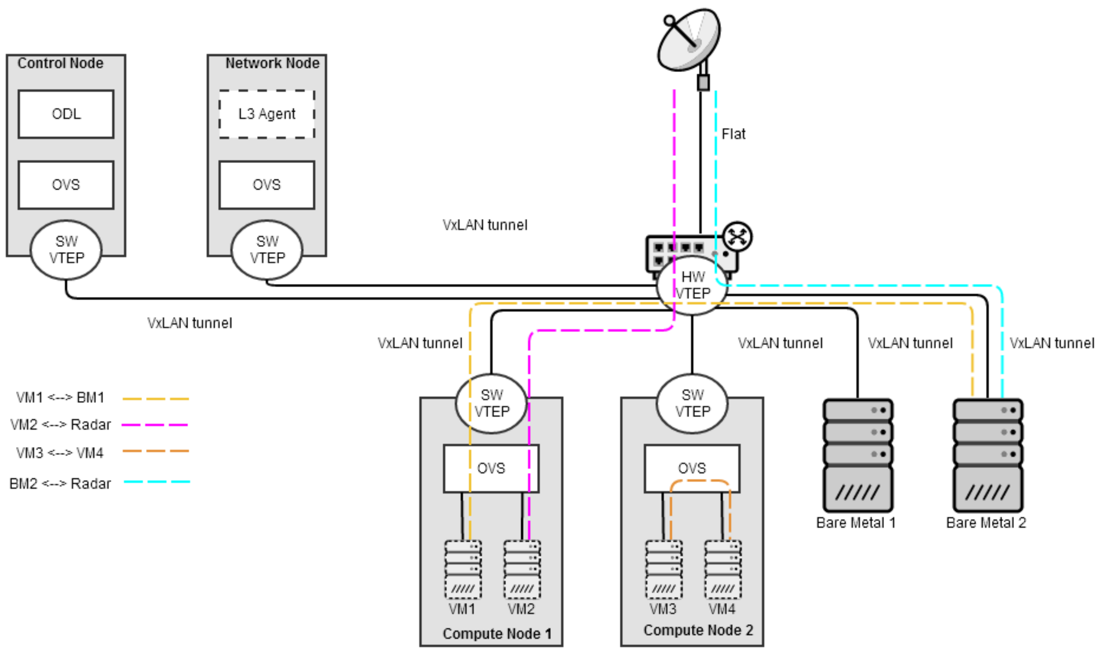


Figure 27: Traffic flows for scenario 4

8 Physical Architecture of the Design

In this chapter the physical architectures of the proposed designs are described. The main physical components of the system in both cases are a controller node, a network node, the computing nodes which are divided into the virtualization and the bare metal nodes, and a physical switch that provides the required connectivity between the nodes. The functionality of the network node can be integrated into the controller node, however for redundancy reasons, it is recommended to have them separated. The minimum hardware requirements for the nodes of the system can be found in Appendix I.

8.1 Supported design

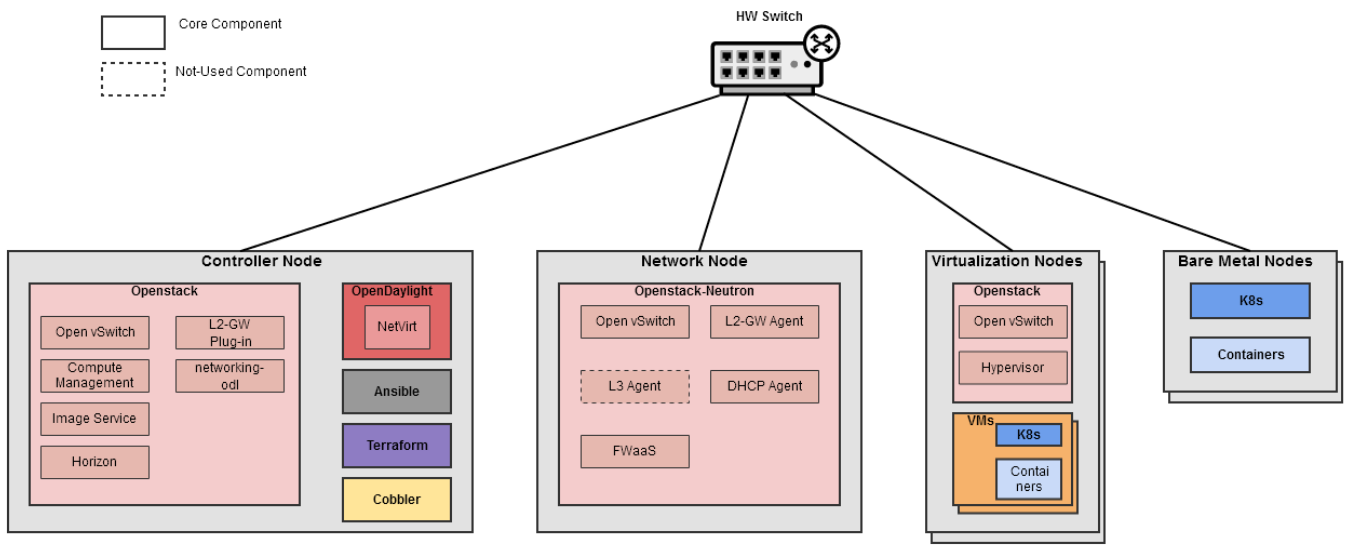


Figure 28: Physical Architecture of the current design

Figure 28 depicts the physical architecture of the supported design (Figure 15). The controller node in the current design hosts the following software components:

- The main management features of the Openstack platform, such as the compute management service, the image management service and the web user interface (Horizon).
- Open vSwitch agent that comes with the installation of Openstack, and is also deployed on the network and virtualization nodes.

- L2-GW service plug-in that speaks with the L2-GW agent on the network node.
- Networking-odl plug-in that passes the Openstack network configuration into Opendaylight controller
- OpenDaylight controller that has NetVirt installed.
- Ansible for configuring the compute nodes and installing Kubernetes on them.
- Cobbler for the provisioning of the bare metal computing nodes.
- Terraform for writing the HCL files that set up the resources of the infrastructure.

The networking service (Neutron) of Openstack is installed on the network node. Neutron installs the following software on the network node:

- Open vSwitch agent
- L3 agent that basically creates the virtual routers in the system and enables L3 connectivity between nodes of different virtual networks. However the functionality of the L3 agent is replaced by the ODL controller
- FWaaS agent that deploys virtual firewalls on the virtual routers.
- L2-GW agent that is basically an ovsdb client that speaks with the ovsdb server on the hardware switch.
- DHCP agent that assigns IP addresses to the components of a virtual network

Virtualization nodes are compute nodes that are responsible for hosting virtual machines. Openstack compute services and the selected hypervisor (e.g., KVM) are deployed on the compute nodes. Openstack deploys virtual machines on the virtualization nodes and Kubernetes is deployed on them by Ansible.

In contrast, bare metal nodes are computing nodes that are used as pure bare metal solution and do not run virtual machines. Ansible is used to deploy Kubernetes on the bare metal nodes and configure the VLAN interfaces. The created containers run directly on the bare metal machine. The L2-GW is used to provide connectivity between the bare metal nodes and the virtual networks created by Openstack.

8.2 Possibly improved design

Figure 29 shows the physical architecture of the possibly improved design (Figure 16). In this design, the controller node hosts the same Openstack services as the current design with some additional software:

- Magnum that creates Kubernetes clusters on the compute nodes.
- Heat that is used by Magnum to deploy the Kubernetes clusters.

- Ironic that provisions the bare metal computing nodes. Cobbler is not installed on the controller node in this design.

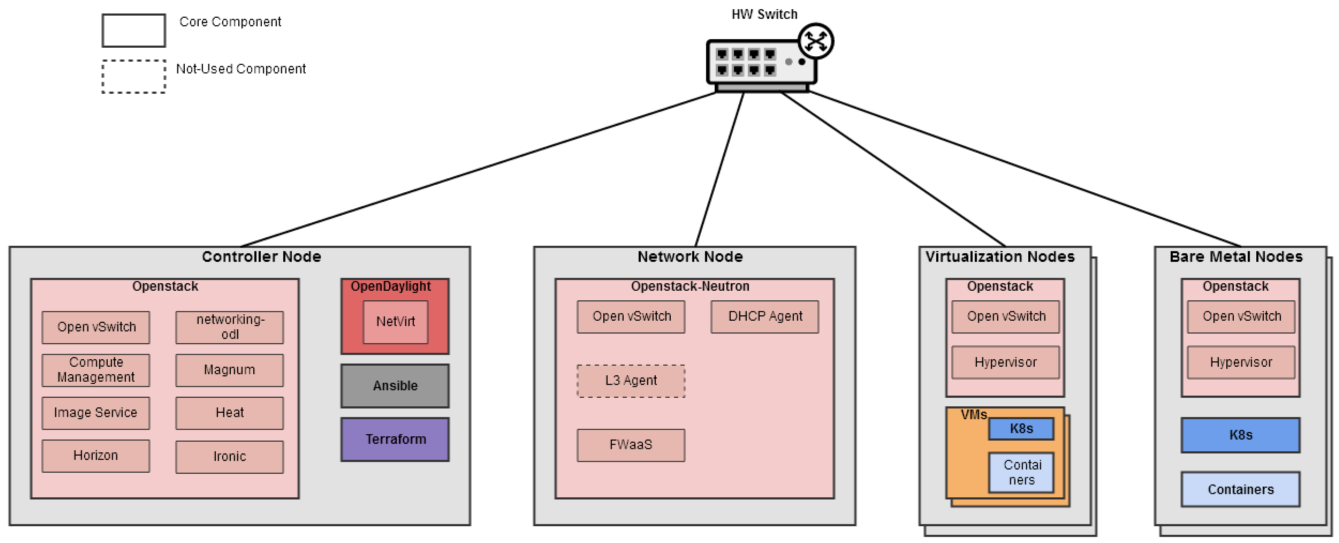


Figure 29: Physical architecture of the possibly improved design

The network node and the virtualization nodes host almost the same components as the current design with the exception of the L2-GW agent, which is not required in this design. Kubernetes cluster in this case are created by Magnum and Ansible is only used for some additional configuration.

The bare metal nodes in this design have the Openstack compute (Hypervisor) and networking services (Open vSwitch) installed on them. The installation of Open vSwitch on the bare metal nodes, integrates them with the rest of the Openstack networks making the L2-GW unnecessary. Kubernetes is installed on these nodes by using Magnum in combination with Ironic, which installs by default the required Openstack components on these nodes.

9 Validation

This chapter describes the validation process for the designs of this research, which is based on the expert opinion validation method [3]. Several interviews were conducted in order to gain the opinion of the experts on the proposed design. The validation approach and the validation results are presented in this chapter.

9.1 Validation approach

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 [3] 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 the design in a minimum testing environment and comparing it with a current system is not possible either. Setting up a minimum version of this design requires specific hardware, such as several bare metal machines and specialized physical switches. The lack of this specialized hardware in combination with time limitations makes this validation method impractical for this design. Furthermore, comparing the design with an existing system of a critical mission organization in regards to some quantitative aspects such as performance is practically impossible. For instance, in our case study, Thales has a wide variety of systems for each use case and each system uses different technology that differs highly from the proposed design. As a result, there is no direct match to an existing Thales system for proper comparison.

The qualitative validation approach is based on a series of semi-structured interviews. Seven IT experts in four relevant IT fields took part in the interviewing process. These four areas of expertise are: Cloud Architecture, Infrastructure & Software Architecture, Systems Design and Networking .The working experience of the interviewees varies and has an average of 20 years. The majority of the participants (5 out of 7) have been working for Thales for several years, providing useful insight into the Thales projects and systems, while two participants combine plethora of external knowledge with the Thales internal insight.

Interviewees	Company	Profession	Experience in IT
Interviewee 1	Thales (external)	Cloud Architect	~40
Interviewee 2	Thales	System Engineer	~20
Interviewee 3	Thales	Software Architect	~20
Interviewee 4	Thales	Infrastructure Architect	30
Interviewee 5	Thales	Cloud Architect	22
Interviewee 6	Thales	Systems Architect	~30
Interviewee 7	Thales	Network Engineer	5

Table 6: List of experts with their experience in years

The interviews were performed face-to-face with the participants and the duration of each interview was approximately one hour. The participants' responses were archived using written notes. The notes were analyzed afterwards and were formally documented in shorts reports. The validation interview and the resulting interview reports are available in Appendix J and Appendix K respectively.

The interview consisted of ten semi-structure questions focusing on several major quality characteristics such as functionality, performance, modifiability, reusability, traceability, learnability, maturity and potential costs. These aspects are derived from the ISO 25010 [98] standard, which is the main standard for evaluating the quality of a software product. Before conducting each interview, a short description of the software-defined technology along with the proposed design was sent to each participant approximately two weeks in advance. The interviewees were able to ask questions regarding the design and the used technology during that two weeks period and also during the first ten minutes of the interview.

9.2 Results

The results gathered from the seven conducted interviews are presented in this chapter. The results are categorized into nine sections based on the topic of each interview question, discussing respectively the proposed current and future infrastructure layer, the proposed network layer, performance, scalability, upgradability, reusability, traceability, and the maturity of the software-defined technology.

9.2.1 Current and future proposed design of the infrastructure layer

All experts agree that both designs conceptually manage to meet the defined generic requirements. They both provide VMs, bare metal machines and containers in one system. However, the majority of the interviewees believe that Openstack is a complex and heavy platform that adds another layer of abstraction, but a few experts characterized it as a reasonable solution for an organization that searches for a private cloud solution.

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. Solutions that focus entirely on one open-source tool, tend to include high risk levels. The selection of open-source tooling can be crucial for the organization in case the selected tool becomes obsolete and eventually is not supported anymore.

According to the experts, a configuration management tool, such as Ansible, is required in this design in order to push updates and patches to the system, however they believe that this design is a temporary solution for the problem. The long term solution should be an immutable system that is self-healed and patched. Moreover, some interviewees claimed that the future design should be based entirely on containers and microservices that would completely replace virtual machines. On the other hand, there are experts who believe that virtual machines would be still needed in the future for running legacy applications that require high levels of security.

9.2.2 Design of the network layer

According to the experts, the network infrastructure meets in principle the requirements that were set. The SDN controller provides control over physical and virtual switches offering connectivity between physical and virtual networks. In addition, the network administrators are able to control the traffic within the network via the northbound REST API by writing their own applications and scripts, making the entire network infrastructure programmable. The supported security features such as the virtual firewalls and security groups are also useful additions of these design. However, one expert mentioned that learning and using this new network design would be a challenge for the engineers as it requires a completely different mindset from the currently used network layer. This network design would be solid and useful only if the engineers change their way of designing and building the networks.

Some experts expressed their concern about Openstack regarding its complexity and size, which indirectly affects the choice of the SDN controller. The experts described Opendaylight as a reasonable choice as it is highly compatible with Openstack. However, in case Openstack is replaced, the selection of the SDN controller might also have to change. In addition, a few experts referred to Kubernetes and its interaction with the SDN controllers. They commented that it would be handy in future to have Kubernetes controlling and managing both the physical and virtual switches via the northbound API of the controller.

The majority of the experts believe that this network design still misses some important specialized network requirements in order to be used by Thales in production. One of these requirements is the efficient support of multicast traffic between the nodes of the network, which is not addressed in this research. Furthermore, one participant addressed the compatibility of this design software-defined design with some specialized in-house appliances such as hardware firewalls and dedicated proxy servers. The same expert is also skeptical whether Thales needs the large virtual networks offered by VXLAN and he mentions that additional research is required on the VXLAN technology in order to identify the exact benefits that it offers.

9.2.3 Performance

All experts mentioned that real-time testing is required to acquire a clear overview of the performance of this design. The majority of the interviewees believe that in principle the performance of the network layer should not cause performance issues and they cannot spot any potential bottlenecks. However, experimentation and testing with the described specialized OVSDB hardware switches is required to determine their exact failover time and latency. According to some experts, the virtual switches offer decent performance levels, which in some cases can be even higher than the performance of the currently used hardware.

Potential performance issues might arise at the infrastructure layer. This design is based on the virtualization technology, which by default adds some performance penalties due to the additional abstraction layer. In addition, the performance of this design might be jeopardized by the high complexity that is introduced by Openstack and the big number of software interfaces that it creates. All these software interfaces should function concurrently on a regular basis, which might increase the start-up times in case of an error. Overall, the experts evaluated the performance as a crucial factor when deciding whether a theoretical design should be implemented or not and actual testing is necessary for determining the efficiency of a system.

9.2.4 Scalability and Upgradability

Most experts mentioned that in principle this design is designed to be highly scalable. Terraform offers an overview of the entire infrastructure and is able to discover if it is up to date. Scalability is also improved with Openstack, which is able to manage, scale and upgrade the virtual part of the design. One interesting comment made from some interviewees is that this design can easily scale up but it would be challenging to scale it down. Scaling up requires a large IT infrastructure for provisioning the resources that is mostly found in big data centers. This design cannot scale down easily due to Openstack, which requires at least three to four servers to run properly. Another expert mentioned that the scalability of this design is still manually enforced by the user by changing values into he configuration files, and it is not a fully dynamic solution yet.

The upgradability of this design is also at the desired levels according to the majority of the interviewees. The used declarative tools such as Ansible, Terraform and Kubernetes, push updates and patches whenever it is required. This new system is easier to update compared to the old one where everything has to be reinstalled whenever a new update is required. Nevertheless, one interviewee expressed his concerns about the upgradability of this design. He believes that updating resources in this design would be a challenge for the engineers as they would have to deal with all the dependencies between the involved software interfaces. Updating one component of the infrastructure would require updating several other dependent components, which requires time and effort.

9.2.5 Reusability

Most of the interviewees agreed that the proposed design can be used by many different organizations that run a variety of applications. The usability of this design depends on the size of the IT infrastructure of the organization. This design mostly refers to organizations that want to create large cloud-based systems. As a result, this design might not be the most suitable solution for Thales, where most of the projects are based on small-sized systems. This design might be used within Thales only for setting up the infrastructure of large battleships, which is relatively uncommon. Furthermore, some interviewees mentioned that this design should be able to combine and manage all the different security domains that exist between the different Thales projects, in order to be used in production.

According to one expert, the reusability of this design is also dependent on the area of expertise of the organization. Some components of this design might need to change based on the use case. For instance, an organization that decided to follow a container-based approach, will not need VMs and the entire part of this design that is responsible for the creation of the VMs will be obsolete. On the other hand, an organization that needs legacy applications will keep the part that creates VMs.

9.2.6 Traceability

Most of the interviewees mentioned during the interviews that system error tracking system is better described with the term ‘observability’ than the term ‘traceability’. Observability is not covered in this design and some additional software components should be added in order to form a proper monitoring pipeline. The goal of this pipeline would be to monitor hardware and software components as one entity and not as individual components. The administrators would be able to get insights on the configuration of the components and identify and fix the errors. However, some experts believe that error tracking would be challenging even with the addition of the monitoring pipeline. This design consists of several different software components that would complicate the error tracking of this system.

All the experts agreed that the software-defined technology improves and facilitates the documentation of the system. Code files are used for describing the system, making the system self-explanatory. Comments are used on top of the code to better describe the functionality of the infrastructure resources. In addition, the code files can be uploaded to a version control system such as Git in order to keep track of the changes and roll back to previous versions.

9.2.7 Learnability and Complexity

Overall, the interviewees agreed that the complexity of this design is relatively high due to the big number of involved software components. Many experts also referred to Openstack as a really complex system that requires time and effort for proper understanding and operation.

During the discussion about the required learning curve, several different opinions were expressed. Most experts agree that the learning curve would be high at the beginning but it would decrease in the long term. For the short term, it would be challenging to learn and use the system because it combines many different and difficult to grasp technologies such as Kubernetes, Openstack and software-defined networking. Furthermore, some interviewees mentioned that the initial effort would be high as the proposed design requires a change of mindset from the engineers. The way of designing and building the new system is completely different from the current way of thinking. The new system requires better collaboration between the development and the operation teams. The teams should consist of more DevOps people that have a good understanding of the entire stack. Some experts believe that it would mostly challenge the development teams because they will have to learn and understand the infrastructure concepts, while some other interviewees claimed that the operation teams would have to put more effort into learning and managing this new technology due to its high complexity. Nevertheless, most interviewees concluded that on the long term this technology would be easier to learn and use as the people would have become familiar with it. However, some experts disagree by mentioning that the learning curve of this technology would not decrease in the future because of the involved complexity, which will cause problems during the testing and maintenance procedure, especially in the network part.

According to some experts, the age of the engineers is another important factor that affects the required learning curve. One expert commented that the required knowledge would be significantly lower for a senior engineer compared to the learning curve of a young inexperienced engineer. On the other hand, a few experts believe that the younger the engineer the lower the required learning curve. Older people are used to the old way of thinking that would be hard to change, whereas it is easier for the young engineers to immediately embrace the new mentality that is introduced by the software-defined technology.

9.2.8 Costs

In the discussion about the potential implementation costs of this design, several conflicting viewpoints were expressed. Several interviewees believe that the implementation costs would be even lower than the current values. This new technology runs on COTS hardware, which is generally cheaper than the currently used specialized hardware. In addition, the physical OVSDB-enabled switches that are used in this design are relatively expensive at the moment, but their price will decrease in the future following the generic rule of every brand new technology. Some experts also added that the high levels of automation that are introduced by this new technology would save time and effort from the engineers, resulting into reduced expenses. Furthermore, this technology is mainly based on the virtualization technology, which is highly efficient with the utilization of the available resources preventing over- and under-utilization of the hardware.

On the other hand, some interviewees mentioned that the implementation of this design would be unfeasible for Thales due to the high levels of complexity, which are mainly caused by Openstack. These experts believe that this technology requires extensive training of the engineers, which would be expensive. People should become experts on this technology in order to understand and manage all the included software components. Moreover, the licenses for the selected tools would be high-priced even though the used technology is open source. Thales always selects the most supported software version, especially in case the in-house knowledge is inadequate for managing and debugging the selected tools. According to one expert, an actual business plan is required in order to accurately estimate the potential implementation costs of this design and decide whether this technology is actually worth implementing.

9.2.9 Maturity

All the interviewees agreed that the software-defined technology has already been used in production by several organizations but it is still considered immature for the purposes of Thales. Thales prefers to use technologies that are highly established in the market, while software-defined technologies are relatively new and are still under improvement. The majority of the interviewees found the infrastructure layer much more mature than the network layer, which was characterized as a promising technology with plenty of useful functionalities. As a conclusion, all the interviewees acknowledged the potential of the software-defined technology and mentioned that follow-up research is required in order to further investigate important aspects such as redundancy, safety, performance and specialized features at the network and infrastructure layer.

9.3 Discussion of the results

The interviewees mentioned that the proposed designs of this research are really complex systems, as they include numerous software interfaces resulting to a system that is difficult to understand and debug. An approach for decreasing the overall complexity of the system would focus on the design of several smaller sub-systems that better define the responsibilities of each project of the organization. Applications with different life cycles would be split into different sub-systems, and the system designer would mainly focus on connecting all these sub-systems together rather than designing one large system that fulfills the requirements of all the projects of the organization.

The interview results showed that the proposed designs can easily scale up to thousands of servers; however it is challenging to scale down to small number of servers with these designs. Openstack is the main reason for this, as it requires at least three servers for the minimum installation. As a result, the potential of these designs can be fully leveraged by organizations that have large cloud-based IT infrastructures, and organizations with smaller IT infrastructures could better focus on solutions that use microservices.

In addition, these designs can adequately serve the requirements of the current market by providing support for virtual machines, bare metal machines and containers into one system, however many experts think that the future infrastructure will probably be a fully immutable system that will be mainly based on containerized applications. As a result, the proposed design can be characterized as a medium term solution, and several components of this design might change in future in order to achieve long term functionality. The components that have higher chances to be replaced in future are Openstack and Ansible, because Openstack is a complex and heavy system and configuration management tools such as Ansible would be unnecessary in an centralized immutable system.

Regarding the learning curve of this design, several different viewpoints were presented and overall the experts agreed that the required learning curve is dependent on the experience and the age of the engineers that are available in the organization. These different viewpoints can be explained by the fact that the experts have different backgrounds and different ages. Experts who are more familiar with the concepts of the software-defined technology are confident that the learning curve would be relatively low, while experts with less experience with the software-defined concepts were skeptical about the required learning curve. Overall, it is clear that organizations with employees that have a good understanding of the entire software development cycle and are familiar with the software-defined techniques have better chances of successfully building and maintaining a fully software-defined system, whereas companies that lack this kind of expertise will face some challenges in the initial phase of understanding and setting up the system.

Another remarkable point that can be derived from the results concerns the implementation costs of these proposed designs. The experts opinions are again controversial on this topic, which is justified by the fact that the costs are highly dependent on the learning curve; the higher the learning curve the higher the costs. The hardware costs for implementing these designs should be lower in the long term, as the software-defined technology is based on COTS hardware, which is relatively cheap and easy to find. Therefore, the hardware costs might be even lower than the current values and the determinant factor during the calculation of the potential costs would be the in-house expertise of the engineers.

From the results it can also be seen that the overall maturity of the software-defined technology is considered inadequate for mission critical organizations such as Thales. These organizations have several critical requirements that need support for specific features and certain performance values must be reached before using the technology into production level environments. That implies that further research and testing is required on several fields and especially on performance, which is a crucial aspect for the success of an IT system.

10 Conclusions

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.

10.1 Answers to research questions

On this section, a short summary for each sub-question is presented, concluding to an answer to the main research question.

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

   The software-defined technology is an umbrella term that is identified in several layers of an IT system. This research mainly focused on the infrastructure (SDI) and the network layer (SDN). The computing layer (SDC) and deployment layer (SDD) are considered parts of the infrastructure layer.

   The SDI technology uses definition/ code files for describing and automating the provisioning and configuration process of the infrastructure. These code files are passed as inputs to specific tools (IaC tools) that are responsible for provisioning and configuring the requested infrastructure resources. The IaC tools do not provision the resources themselves, instead they instruct a dynamic platform via APIs to set up and manage the needed resources. The majority of the provisioning tools are vendor-specific solutions, such as CloudFormation for AWS, Cloud Deployment Manager for the Google Cloud Platform and Azure Resource Manager for Microsoft Azure Clouds. Terraform is presented as a more general solution that supports several dynamic platforms and is highly extensible, serving several use cases. The most used tools for the configuration of the infrastructure are Chef, Puppet, Ansible and Saltstack. All described configuration tools are highly scalable and their selection highly depends on the problem that needs to be solved and the current system of the organization.

   SDN is the main software-defined technology at the network layer. NFV is another significant technology in this field, but it is not thoroughly analyzed in this research project as it is mainly used by the telecommunication providers. SDN abstracts the control plane from the hardware, making the network devices only responsible for forwarding data into the network. The control of the entire network is assigned to a central software controller that offers APIs for communicating with the network applications and the network devices. The network applications are used or developed by the network administrators for managing and constructing the network infrastructure based on the gathered information from the controller. The network devices are responsible for forwarding the data into the network and can be either virtual or physical devices. There are several available open-source controllers and Opendaylight stands as the most popular controller at the moment. Opendaylight supports several communication protocols and is supported by an active open-source community, making a great free solution for a variety of use cases.

VMs and containers are the main technologies at the computing level, forming the fundamentals for building an entire software-defined system. There are several vendors for the creating virtual machines with VMware being the leading one, while Docker is the main framework for building and managing containerized applications.

2. **What does the SDx technology offer at each level of an IT system?**

At the infrastructure layer, the engineers are able to set up the entire infrastructure by describing the desired resources in the definition files, which is relatively simple and fast, resulting to increased efficiency of the entire software development cycle. The developers can use the IaC tools to easily create their own sandbox environment for testing without the need of the operation team, which increases their productivity by saving valuable time and effort. The use of the software-defined technology also improves the scalability and upgradability of the infrastructure. The engineers can easily create, delete, resize and relocate the infrastructure resources by simply changing some lines in the configuration files. The functionality of the deployed software applications is maintained regardless the applied changes. In addition, the use of code files improves the configuration consistency, as the engineers use a standardized configuration format, which leaves little space for potential human errors. Furthermore, the software-defined technology leads to self-explained systems that require little additional documentation. The code is enriched with comments that describe in detail the resources of the infrastructure at each part of the code. Last but not least, having the entire infrastructure described by code files, enables the engineers to use a central version control system for keeping track of the changes and rolling back to previous versions in case of an error.

The SDN technology enables the control and management of the entire network infrastructure from a central point with the use of the SDN controller. This simplifies the configuration and the debugging of the network resources, eliminating the need of configuring each specialized device individually using specialized commands. This technology also improves the scalability of the network infrastructure, as the administrator is able to easily provision new virtual resources based on the traffic demands of the underlying network. The administrator can combine virtual and physical network resources into one infrastructure providing support for a variety of use-cases. The SDN technology also offers programmability to the network, as the configuration of the network is performed via code files or GUIs. The administrator can automate several routine network operation procedures by writing case-specific scripts that are passed to the SDN controller, which will lower the operational costs of the network infrastructure. The hardware costs are also reduced, as this technology is based on "white box" hardware, which is cheap and easy to find.

At the computing level, the software-defined technology adds a virtualization layer between the hardware resources and the operating system, which offers elasticity and flexibility to the system. The utilization of the resources becomes much more efficient as resources are created or deleted based on the demand. The virtual computing resources are described into code files, which can be easily shared among different teams. The engineers can set up their resources following the same steps in many different environments, ranging from local development laptops to large cloud systems. Furthermore, the engineers have the capability of prioritizing their resources, giving higher priority to some resources over others.

3. How to build a reference architecture for a generic software-defined system?

In order to answer this question, a problem analysis approach was performed that aimed to identify the main components of the reference architecture of a generic software-defined system. The problem analysis was performed via a literature study that focused on the difficulties that the engineers face during the construction of the existing IT infrastructure. After the identification of the problem, a further literature study was conducted in order to acquire the necessary knowledge on the software-defined concepts and tools at each level of an IT system.

The reference architecture is based on the concepts of the software-defined technology and presents the interconnections between them. The reference architecture is mainly focused on the infrastructure and the network layer. The computing level is part of the infrastructure layer. The main components of a generic software-defined system are: a provision tool, a dynamic platform, the infrastructure and network resources, a network controller and a configuration tool. The administrator describes the desired infrastructure resources into code files that are inserted to the provisioning tools, which instructs via a REST API the dynamic platform to build and manage the required resources. The provisioning tools can create infrastructure and network resources. The configuration tool is used for additional configuration of the resources. At the network layer, a central controller is added to control the entire network infrastructure. The controller can communicate with the dynamic platform via the northbound APIs and configures the network resources via the southbound APIs.

4. How to build a software-defined system for a specific use case?

For answering this question, the requirements for the specified use case were identified and listed. A list of requirements was compiled after discussing with the engineers of the case-study organization about the current needs and the future plans of the company. To facilitate the design process, the requirements were categorized into infrastructure, network and general requirements, and the importance level of each requirement was determined. Based on the requirements, specific software-defined tools were selected that refer to the components of the described reference architecture of a generic software-defined system.

The case study for this research project is a large high tech company that is mainly focused on the privacy and the performance of the IT system. Openstack was selected as the main dynamic platform for managing and building the infrastructure resources, while Terraform and Ansible are the chosen provisioning and configuration tools, respectively. Opendaylight is the proposed SDN controller for this specific use case.

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

The expert opinion was used to validate the design of this research project as the most suitable validation method for this specific case study. The design was presented and explained to seven experts from four relevant IT fields: Cloud Architecture, Infrastructure & Software Architecture, Systems Design and Networking. The experts expressed their opinion on several aspects of the design through semi-structured face-to-face interviews. All the interviewees acknowledged the potential of the software-defined technology and mentioned that the current stage of the technology is relatively immature and further research is required to acquire a clear picture on the performance of this design.

Main question: "How can the software-defined technology be used to improve a static IT infrastructure of an organization?"

As a conclusion to the research, it should be mentioned that software-defined technology can be used in several layers of an IT system ranging from the infrastructure to the deployment layer. The software-defined technology improves the scalability, upgradability and the documentation of the system by using code files that are easy to understand and use, and improves the entire development cycle of the organization by automating the provisioning and configuration of the system resources. Overall, the software-defined technology is a popular and relativity new topic that introduced several new concepts and tools. These tools are already being used by a big number of organizations, but insights on some crucial factors are still missing for a number of organizations that focus on specialized products. This research project can be used as a reference point for understanding these concepts and the interconnections between them, and could be the basis for follow-up research projects that would help organizations in making the right steps towards the implementation of the software-defined technology into production level environments.

10.2 Limitations

This research project was restricted by the following limitations:

- *Lack of specialized hardware*: An actual implementation of the proposed design was not possible due to the specialized hardware that is required. The involved high prices of this hardware were a deterrent factor in the scope of a Master thesis project.
- *Time limitation*: Conducting tests for measuring and analyzing all the software quality aspects, was infeasible due to the time limitation of a Master project.
- *Limited academic literature*: The software-defined technology gathered the attention of the academic world the last couple of years; as a result the amount of available academic work is limited.
- *Limited practical expertise:* During the validation interviews, most experts mentioned that they are aware of the concepts of the software-defined technology, however they lack of practical experience with the involved tools,

which limited their feedback on the conceptual level and the approach of the design, and their feedback on the potential practical issues was based on assumptions.

- *Unsynchronized documentation:* The software-defined technology is improving rapidly and the relevant documentation is occasionally not up to date with the latest versions of the tools, which created challenges during the research procedure.
- *High uncertainty levels*: The software-defined technology is characterized by constant releases of new versions for the existing tools and the regular development of brand new tools that aim to improve the existing situation. This adds high levels of uncertainty during the process of tool selection, as the selected tool might be replaced in the close future by another solution, which would result to a completely different architecture.

10.3 Contributions

This research project offers several contributions to the academic world and practice:

- This Master thesis described, analyzed and slightly compared the software-defined concepts and the tools at the infrastructure layer. This knowledge already existed in technical documentations and books written by industry experts but it had never been presented in an academic project. As a result, this Master thesis project presented the concepts and tools of the software-defined infrastructure in an academic document for the first time.
- The details of each layer of a software-defined system were included in several separate academic papers, although there was no academic document that included an overview of the all concepts and tools of the entire software-defined term. This research project can be seen as an academic survey that presents a detailed overview of all layers of a software-defined system in combination.
- This research can be beneficial for the industry because it shows the interconnections between the concepts and tooling of the software-defined infrastructure and the software-defined networking with a real use case example that uses the existing software-defined tools. The majority of the technical documentation includes examples of how to use the tools at each level, but not examples that combine technologies from the network and the infrastructure layer into one design.
- This project presents a logical way of designing a software-defined system from the ground up, which can help the organizations identify the existing problems

and form a list of requirements that describe the desired future functionality of the software-defined system. Based on these requirements and the described reference architecture, they can select the most appropriate software-defined tools for their use case.
- This research can be used as a reference point for understanding the concepts of the software-defined technology, and both the academic community and the industry world can use it as a basis for future work.

10.4 Recommendations

Based on the personal understanding of the technology and the opinions from some experts, several recommendations are proposed:

- The selection of specific tools and technologies is highly based on each organization and the specific time they want to go on production. The designs proposed in this research are short to medium term solutions and they mainly refer to companies that aim to deliver a production level infrastructure immediately or in a short time period. Companies that seek for long term solutions should focus mostly on microservices architectures that are based on the containerization technology, as these technologies are highly scalable, less complex and easy to develop and deploy. It is highly possible that containers would completely eliminate the need of virtual machines, as they are constantly improving and they might also able to serve legacy or high-security applications that are currently supported only by virtual machines.
- Mission critical organizations should further investigate the software-defined technology in regards to some specialized features and appliances such as multicast, firewalls and dedicated proxies.
- Organizations should keep in mind that some parts of the proposed design, such as the configuration management tools, might drastically change in the future and can be entirely replaced by technologies that aim to provide a fully immutable and self-healing system.
- Organizations that have several projects with several separate IT infrastructures should focus on designing small software-defined systems that specifically serve the requirements of each project. The organization should focus on finding a mechanism to connect this small sub-systems rather than designing one large software-defined system that fulfills the requirements of all projects. These large systems are complex and difficult to debug, as they include many software interfaces between the involved components.

- The organizations should invest time during the selection of the open-source tool for their use case, because there is a significant chance for the chosen tool to become obsolete in the future resulting to an unusable infrastructure.
- The organizations should wait before choosing a software-defined solution for their existing network infrastructure, as the SDN technology is relatively immature and the involved specialized hardware is highly priced at the moment. The prices of this specialized hardware will probably drop in the future, following the generic rule of every brand new technology.
- The organizations should focus on hiring more people that have a DevOps mindset and are able to understand and work with both infrastructure and development concepts and processes. The software-defined technology combines technologies from the entire development cycle, and experienced DevOps teams are crucial for the success of an organization that uses software-defined techniques.

10.5 Future work

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

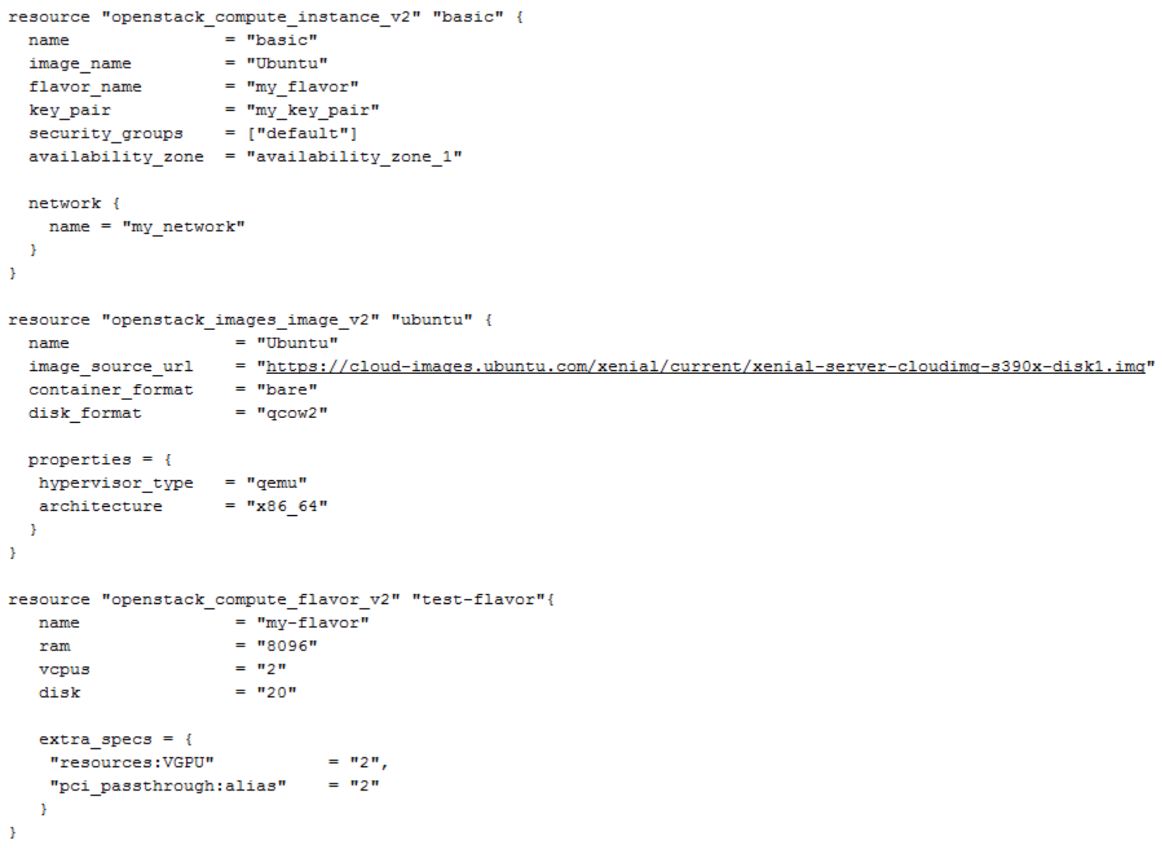
- *Performance:* Further research is required on the overall performance of the system. Thorough insights of the performance of the system can be gained by building a fully functional prototype of the proposed designs with all the necessary hardware and software components and testing the prototype with 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 Metal^3 [99], which is a Kubernetes API for managing bare metal hosts and Digital Rebar [100], 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.

Appendix A

Creating VMs with Terraform – Openstack

Terraform can create VMs in Openstack by combining several resources from the Openstack provider. A Terraform file that includes all the required resources to create a VM is shown in Figure 30. This file includes three separate resources: the openstack_compute_instance_v2 resource, which is the main required resource for creating a Openstack VM, the openstack_images_image_v2 resource and the openstack_compute_flavor_v2 resource. In the compute_instance resource the user is able to define a name for the VM (e.g., basic), a name for the selected image (e.g., Ubuntu), a flavor name (e.g., my_flavor), a key pair, the preferred security groups (e.g., the default security group), an availability zone where the VM will run on and the name of the network where the VM is assigned to.



```
resource "openstack_compute_instance_v2" "basic" {
  name                = "basic"
  image_name          = "Ubuntu"
  flavor_name         = "my_flavor"
  key_pair            = "my_key_pair"
  security_groups     = ["default"]
  availability_zone   = "availability_zone_1"

  network {
    name = "my_network"
  }
}

resource "openstack_images_image_v2" "ubuntu" {
  name                 = "Ubuntu"
  image_source_url     = "https://cloud-images.ubuntu.com/xenial/current/xenial-server-cloudimg-s390x-disk1.img"
  container_format     = "bare"
  disk_format          = "qcow2"

  properties = {
   hypervisor_type    = "qemu"
   architecture       = "x86_64"
  }
}

resource "openstack_compute_flavor_v2" "test-flavor"{
   name                = "my-flavor"
   ram                 = "8096"
   vcpus               = "2"
   disk                = "20"

   extra_specs = {
    "resources:VGPU"             = "2",
    "pci_passthrough:alias"      = "2"
   }
}
```

Figure 30: Terraform file for creating VMs

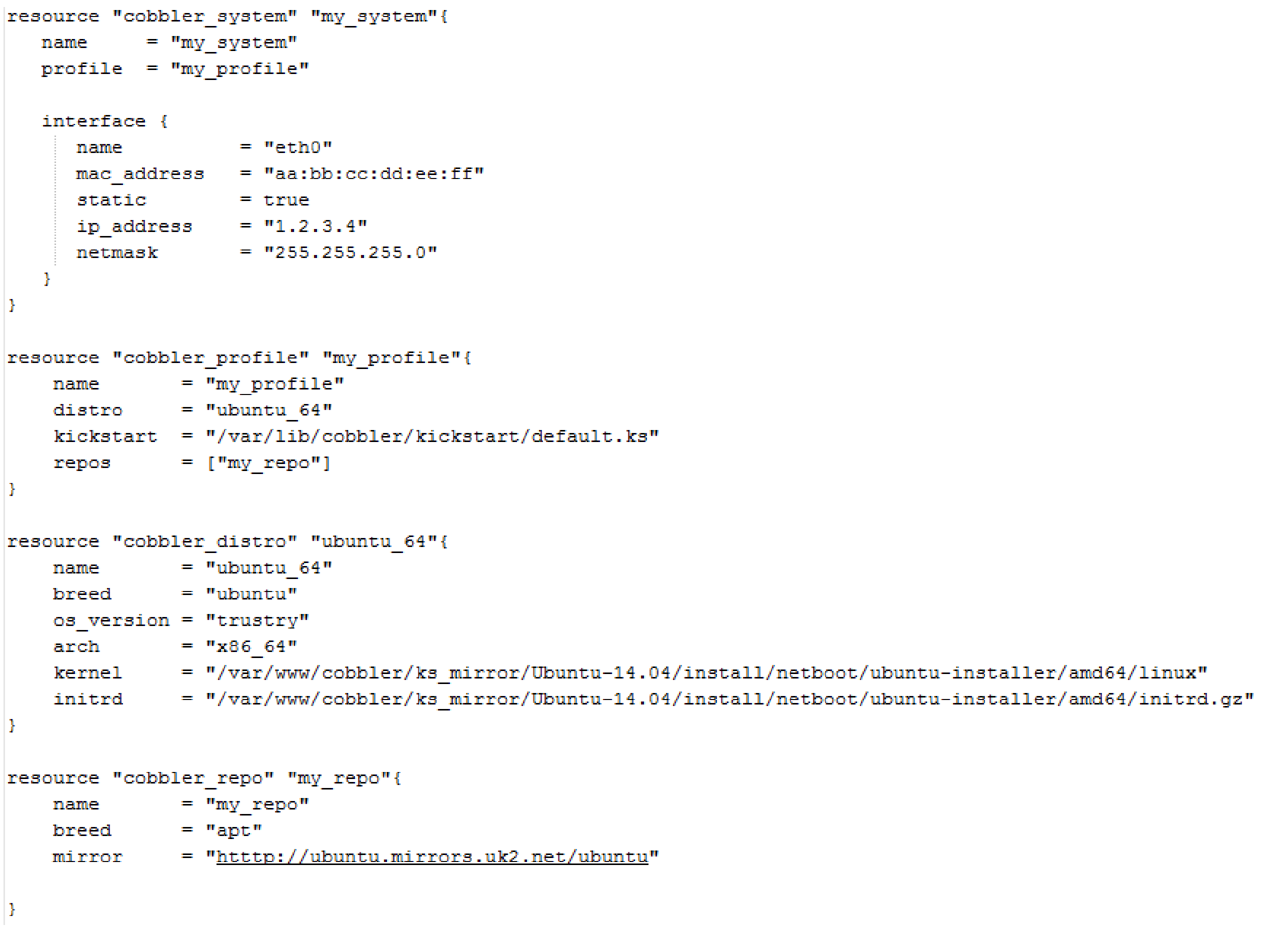
The user should first use the openstack_images_image_v2 resource to create images that would be later used during the creation of the VMs. In the image_source_url field the user can pass the url of the preferred OS that Openstack would download and use, and in the disk_format field the user should declare the format of the image (e.g. iso, qcow2, vhd, etc.). In the properties field, the type of the hypervisor (e.g. qemu, vmware, xen, lxc) and the CPU architecture of the VM (e.g. x86_64, arm, ppc65) can be passed as key-value pairs. The user should also define a name for the image that would be used by the compute_instace resource in the image_name field when creating a VM.

The openstack_compute_flavor_v2 resource is needed for setting the capabilities of the VMs. The name of the flavor resource is set in the name field, which is used in the flavor_name field of the compute_instance resource to configure the VMs. In the ram, vcpus and disk field, the user can determine the size of available ram, the number of virtual cores and the capacity of the VMs accordingly. The system should also support the selection of virtual GPUs and I/O devices. The extra_specs field is used for that purpose. The user declares the desired number of vGPUs by typing resources:VGPU= <number of vGPUs> in the extra_specs field. The vGPU support is not enabled by default, and the supported GPU devices should be defined on the compute nodes that carry the corresponding hardware. Practically the same procedure applies when enabling the support for I/O devices. In the extra_specs field, the user should add the line pci_passthrough:alias = <number of devices> to allow full access of physical PCI devices to the VMs. Similar to the vGPUs, the PCI devices are not enabled by default and configuration is required on the compute and control nodes for enabling the PCI passthrough feature and set aliases [59] for the PCI devices. 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.).

Appendix B

Provisioning Bare Metal Machines with Terraform – Cobbler

The main resources for provisioning a bare metal machine with Terraform using the Cobbler provider are: the cobbler_system resource, the cobbler_profile resource and the cobbler_distro resource. A Terraform file that includes all the necessary resources for provisioning a bare metal machine can be found in Figure 31.



```
resource "cobbler_system" "my_system"{
   name     = "my_system"
   profile  = "my_profile"

   interface {
      name           = "eth0"
      mac_address    = "aa:bb:cc:dd:ee:ff"
      static         = true
      ip_address     = "1.2.3.4"
      netmask        = "255.255.255.0"
   }
}

resource "cobbler_profile" "my_profile"{
    name       = "my_profile"
    distro     = "ubuntu_64"
    kickstart  = "/var/lib/cobbler/kickstart/default.ks"
    repos      = ["my_repo"]
}

resource "cobbler_distro" "ubuntu_64"{
    name       = "ubuntu_64"
    breed      = "ubuntu"
    os_version = "trustry"
    arch       = "x86_64"
    kernel     = "/var/www/cobbler/ks_mirror/Ubuntu-14.04/install/netboot/ubuntu-installer/amd64/linux"
    initrd     = "/var/www/cobbler/ks_mirror/Ubuntu-14.04/install/netboot/ubuntu-installer/amd64/initrd.gz"
}

resource "cobbler_repo" "my_repo"{
    name       = "my_repo"
    breed      = "apt"
    mirror     = "htttp://ubuntu.mirrors.uk2.net/ubuntu"

}
```

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

The user should first use the cobbler_distro resource in Terraform to create a distribution within Cobbler. In the kernel and initrd field, the user must indicate the exact paths to the kernel and the initrd on the filesystem accordingly. These paths do not exist in the file system of Cobbler by default and the user has to manually download the image of the preferred OS, mount it, import it into Cobbler and point out the created paths into the distribution resource. The user should also determine in the distribution resource, a name for the distribution, the breed of the OS (e.g., Redhat, Fedora, Centos, Ubuntu etc.), the version of the OS (e.g., trusty) and the architecture of the OS (e.g., i385, x86_64, ia64 etc.) in the corresponding fields.

The next step is to associate the distribution with the profiles. The cobbler_profile resource is responsible for that purpose. The user has to choose a name for the profile in the name field and can pass the name of the preferred distribution into the distro field. In addition, the exact path to the preferred kickstart file should be declared into the kickstart field. The kickstart file should be created outside the scope of Cobbler and a kickstart file usually contains information regarding the initial configuration of the machine, such as the mode of installation (graphical or text), keyboard layout and system language, disk partitioning, the root password etc. An example of a kickstart file is depicted in Figure 32. Moreover, the user can assign repos to a profile in the repos field. The repos are created with the use of the cobbler_repo resource and their names are passed to the repos field in the profile resource. The user also should give the url address for the mirror repository and the breed of that repository (e.g., apt, yum, rsync).

The last step is to link the machines with the preferred profiles for installation. The cobbler_system resource serves that objective. In the name field the user should select an appropriate name for the system, and in the profile field the user should declare the name of the preferred profile that was created previously with the cobbler_profile resource. The hostname of the machine can also be assigned in the hostname field. In the interface block, the user can choose the machine for provisioning by declaring the MAC address of the machine in the corresponding field and the preferred IP address of the machine after the provisioning, is passed into the ip_address field.

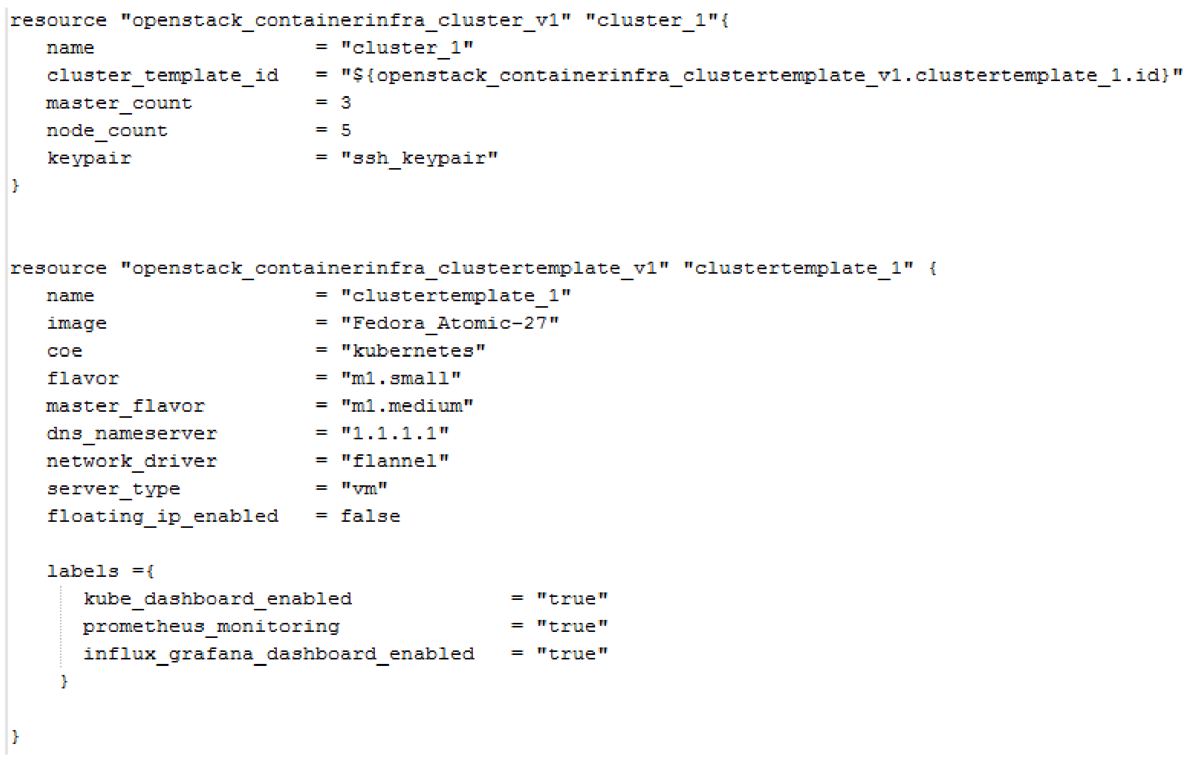
```
#System bootloader configuration
bootloader --location=mbr
#Partition clearing information
clearpart --all --initlabel
#Run the Setup Agent on first boot
firstboot --disable
#Activate X
xconfig --startxonboot
#Use network installation
url --url=$tree
#additional repostories get added here
$yum_repo_stanza
#Reboot after installation
reboot
#System keyboard
keyboard us
#System language
lang en_US
#System timezone
timezone  America/New_York
#Root password
rootpw --iscrypted $default_password_crypted
#Install OS instead of upgrade
install
#Clear the Master Boot Record
zerombr
#Allow anaconda to partition the system as needed
autopart

%packages
@base
@base-x
firefox
flash-plugin
$desktop_pkg_group
%end

%post
#create a default user to log in X
useradd desktop-user
passwd -d desktop-user

#adds the yum repositories to the installed system
$yum_config_stanza
#cobbler final steps
$SNIPPET('kickstart_done')
%end
```

Figure 32: An example of a kickstart file

Appendix C

Provisioning Clusters of Containers with Terraform – Magnum

The main Terraform resources for creating container clusters in Openstack are the openstack_containerinfra_clustertemplate_v1 and openstack_containerinfra_cluster_v1 resources. Figure 33 illustrates a Terraform files that uses these resources to provision a Kubernetes cluster.



```
resource "openstack_containerinfra_cluster_v1" "cluster_1"{
   name                    = "cluster_1"
   cluster_template_id     = "${openstack_containerinfra_clustertemplate_v1.clustertemplate_1.id}"
   master_count            = 3
   node_count              = 5
   keypair                 = "ssh_keypair"
}


resource "openstack_containerinfra_clustertemplate_v1" "clustertemplate_1" {
   name                    = "clustertemplate_1"
   image                   = "Fedora_Atomic-27"
   coe                     = "kubernetes"
   flavor                  = "m1.small"
   master_flavor           = "m1.medium"
   dns_nameserver          = "1.1.1.1"
   network_driver          = "flannel"
   server_type             = "vm"
   floating_ip_enabled     = false

   labels ={
      kube_dashboard_enabled              = "true"
      prometheus_monitoring               = "true"
      influx_grafana_dashboard_enabled    = "true"
    }

}
```

Figure 33: Terraform file for provisioning container clusters of containers

The user should first describe the parameters of the cluster in the openstack_containerinfra_clustertemplate_v1 resource in Terraform. A name for the cluster template should be declared in the name field, while the preferred OS image that is used for the nodes of the cluster, is selected in the image field. The name of the preferred COE should be passed into the coe field. In addition, the user can choose the capabilities of the worker nodes and the master of the cluster by selecting values in the flavor and the master_flavor fields correspondingly. The user should also declare the IP address of a DNS server for the cluster and choose an appropriate network driver (e.g., flannel) in the corresponding field.

In the server_type field the user can select the type of the servers for the cluster nodes. The available options are “vm” and “bm” for virtual machines and bare metal serves respectively. Moreover, in the labels block, the user can enable or disable important external features of the selected COE. For the Kubernetes case, Magnum supports the Kubernetes dashboard and Prometheus with Grafana for monitoring. The user can enable or disable these features by typing true or false in the respective fields.

The next step is the actual deployment of the cluster by referring to the attributes defined in the particular cluster template. The Terraform openstack_containerinfra_cluster_v1 resource is used for creating the clusters. The user should choose a name for the cluster in the name field and should associate the cluster template by assigning the respective id of the template to cluster_template_id field. Furthermore, the user has the option to determine the size of the cluster by declaring a specific number for the master nodes and the worker nodes in the corresponding fields. Additionally, the selection of a specific key pair (e.g., ssh key pair) for the communication between the nodes is done via the keypair field.

Appendix D

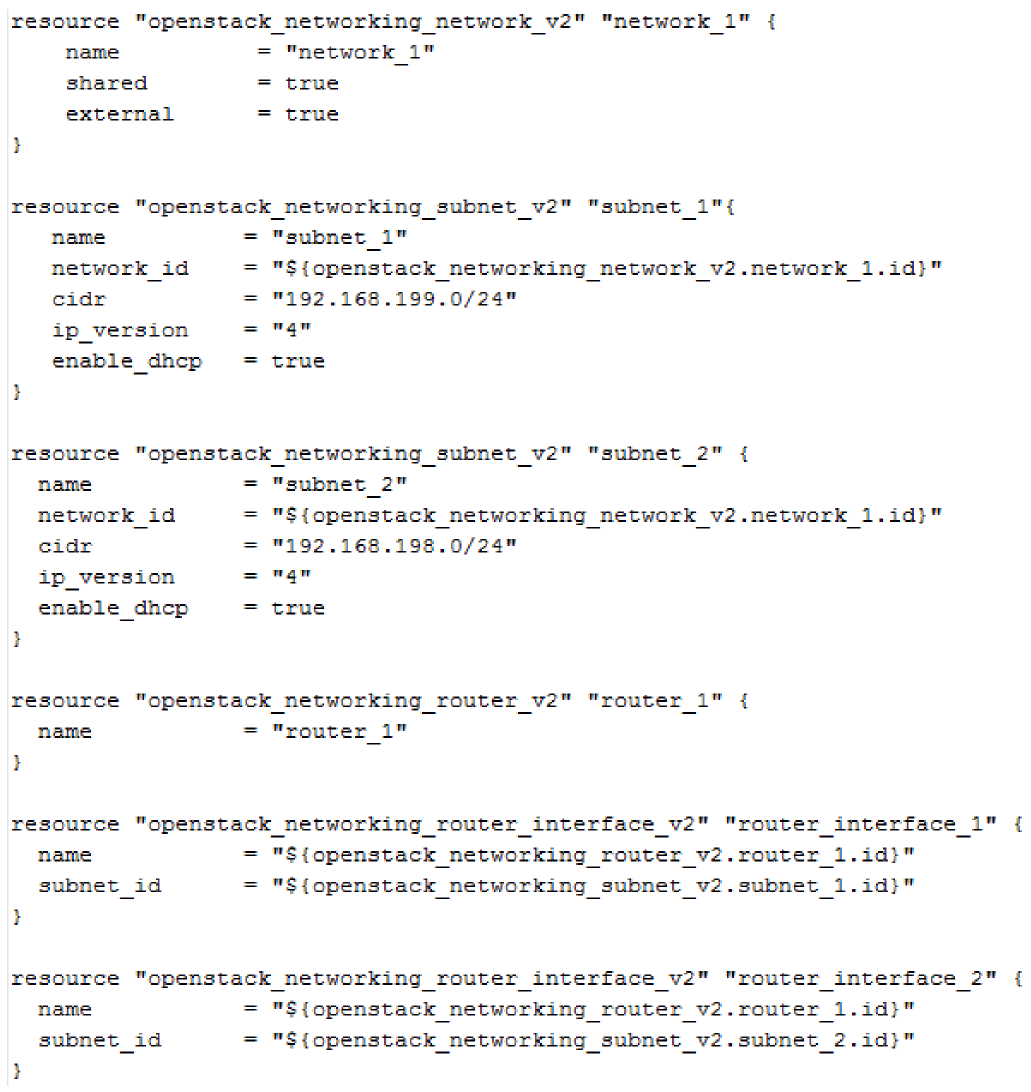
Configuration with Ansible

Ansible uses the inventory plugins and the inventory scripts to manage dynamic inventories. Openstack has support for an inventory plugin, while Cobbler is managed using inventory scripts. For the Cobbler case, the user should download the respective script and place it into the hosts’ directory of Ansible. In addition, the user has to include a cobbler.ini file into the ansible directory to let Ansible know where the Cobbler server is located. Openstack has support for both plugin and script dynamic inventories. The dynamic inventory script works in the same way as Cobbler, while the inventory plugin should first be enabled in the Ansible configuration file, and a separate yaml file should be declared that includes a plugin field where the enabled plugin is declared. This yaml file should be called every time an Ansible playbook is executed.

Appendix E

Creating networks with Terraform – Openstack Neutron

The main Terraform resources for creating networks components in Openstack are shown in Figure 34.



```
resource "openstack_networking_network_v2" "network_1" {
    name           = "network_1"
    shared         = true
    external       = true
}

resource "openstack_networking_subnet_v2" "subnet_1"{
   name           = "subnet_1"
   network_id     = "${openstack_networking_network_v2.network_1.id}"
   cidr           = "192.168.199.0/24"
   ip_version     = "4"
   enable_dhcp    = true
}

resource "openstack_networking_subnet_v2" "subnet_2" {
  name            = "subnet_2"
  network_id      = "${openstack_networking_network_v2.network_1.id}"
  cidr            = "192.168.198.0/24"
  ip_version      = "4"
  enable_dhcp     = true
}

resource "openstack_networking_router_v2" "router_1" {
  name            = "router_1"
}

resource "openstack_networking_router_interface_v2" "router_interface_1" {
  name            = "${openstack_networking_router_v2.router_1.id}"
  subnet_id       = "${openstack_networking_subnet_v2.subnet_1.id}"
}

resource "openstack_networking_router_interface_v2" "router_interface_2" {
  name            = "${openstack_networking_router_v2.router_1.id}"
  subnet_id       = "${openstack_networking_subnet_v2.subnet_2.id}"
}
```

Figure 34: Terraform file for creating Openstack network

Networks can be created by using the openstack_networking_network_v2 resource in Terraform. The user has to select a name for the network in the respective field, and in the shared field the user can decide whether the network is accessible by any other tenants/ users. In the external field, the external routing facility of the network can be determined. The possible values are true and false and false is the default option. In the segments block, the type of the network can also be defined in the network_type field. The default option, which is VXLAN, is not displayed and the other available choices are flat and GRE.

Subnetworks can be created on top of a network with the openstack_networking_subnet_v2 Terraform resource. The user should set a name for the subnetwork and pass the network id of the parent network in the corresponding fields. In the cidr field, the user can assign a value for the IP range for this subnetwork (e.g., 192.168.199.0/24). The user also has the option to select the IP version of the network (IPv4 or IPv6), assign an IP for the subnetwork gateway and enable or disable the support of DHCP. DHCP is enabled by default. Compute instances (VMs) can be later created in the specified subnetworks by using the network field in the compute instance resource of Terraform.

The openstack_networking_router_v2 resource should be used for the creation of the virtual router. The name field is the only mandatory field for the creation of this resource. The router resource should be combined with the openstack_networking_router_interface_v2 resource in order to connect the router to a subnetwork. In the router and subnet id fields the user should declare the ids of the router and the subnetwork that should be connected.

```
resource "openstack_compute_instance_v2" "instance_1"{
    name                = "instance_1"
    image_name          = "Ubuntu"
    flavor_name         = "my_flavor"
    key_pair            = "my_key_pair"
    security groups     = ["default"]
}

resource "openstack_networking_floatingip_v2" "fip_1" {
    pool = "my_pool"
}

resource "openstack_compute_floatingip_associate_v2" "fip_1" {
    floating_ip = "${openstack_networking_floatingip_v2.fip_1.address}"
    instance_id = "${openstack_compute_instance_v2.instance_1.id}"
}
```

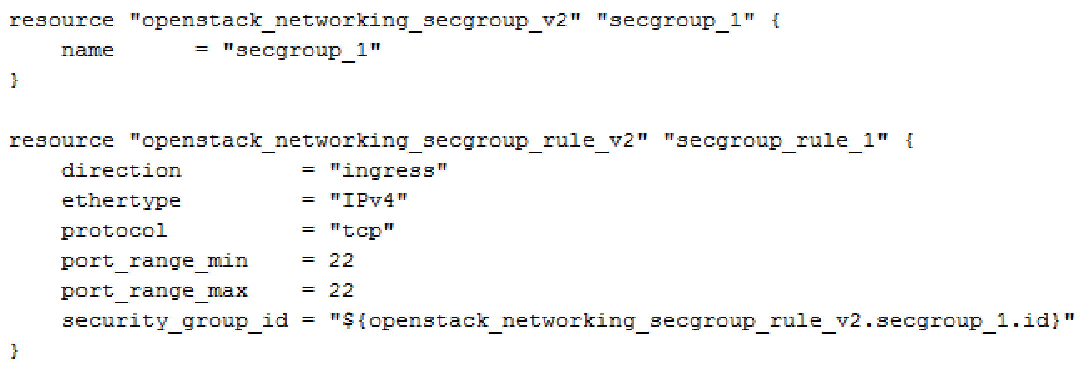
Figure 35: Terraform file for creating floating IPs

Floating IPs can be created in Terraform with the openstack_compute_floatingip_v2 [79] resource (Figure 35). The user should only declare the name of the pool from which to obtain the floating IP (e.g., public pool). The floating IP is assigned to an instance via the openstack_compute_floatingip_associate_v2 [80] resource, where the id of the floating IP and the instance should be filled in in order for the assignment to take place.

Appendix F

Creating security groups networks with Terraform – Openstack

The security groups can be created with the openstack_networking_secgroup_v2 Terraform resource, while the rules are created with the openstack_networking_secgroup_rule_v2 resource. In the security group rule resource, the user should choose the direction of the rule (ingress or egress), and the layer 3 protocol type in the ethertype field (IPv4 or IPv6). In addition, the allowed layer 4 protocol (tcp, udp, icmp) for a specific port range can be selected in the protocol field. In the port_range_min/max field the user can determine the lower and the upper part of the allowed port range (between 1 and 65535), and in the security_group_id field the id of the security group that this rule will be part of, should be declared. The user can create several rules with the security group resource and assign them into one or more security groups created by the security group resource in Terraform.



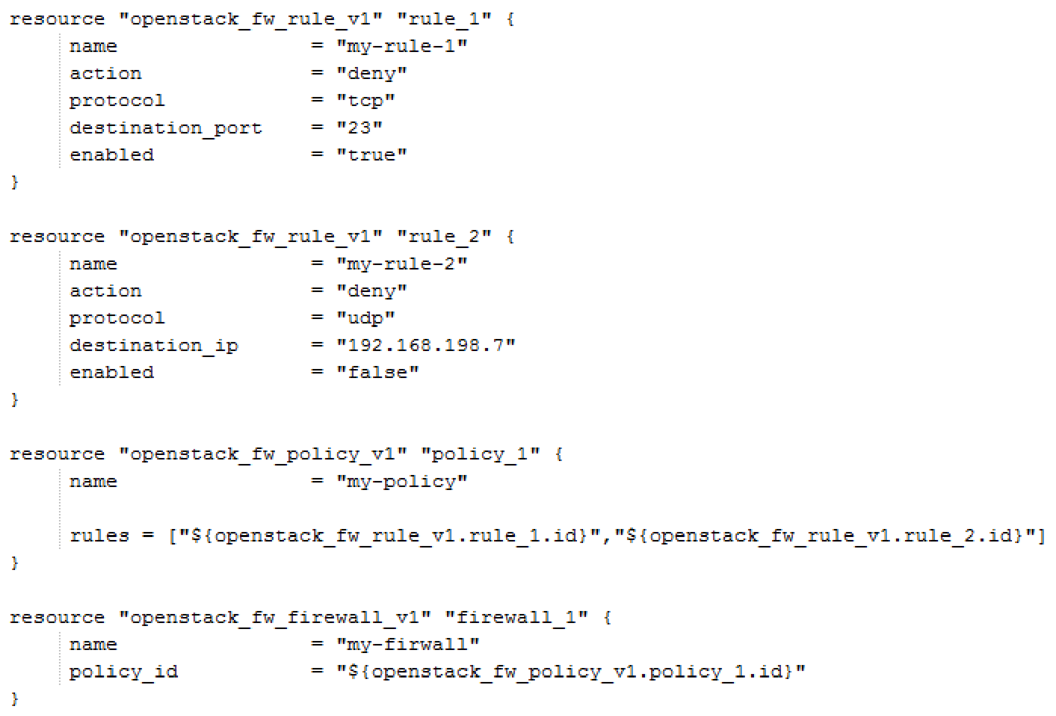
```
resource "openstack_networking_secgroup_v2" "secgroup_1" {
    name      = "secgroup_1"
}

resource "openstack_networking_secgroup_rule_v2" "secgroup_rule_1" {
    direction         = "ingress"
    ethertype         = "IPv4"
    protocol          = "tcp"
    port_range_min    = 22
    port_range_max    = 22
    security_group_id = "${openstack_networking_secgroup_rule_v2.secgroup_1.id}"
}
```

Figure 36: Terraform file for creating security groups

Appendix G

Creating Firewalls with Terraform – Openstack

Rules are created in Terraform with the openstack_fw_rule_v2 resource. The user should select the protocol (tcp, udp, icmp, any) on which the firewall functions and the implemented action in case of a firewall match in the respective fields. The available actions are allow and deny. In the source and destination port field, the corresponding ports on which the firewall rule operates are declared. There is also the enabled field, where the user can enable or disable the specific rule. True is the default option. Policies are created with the openstack_fw_policy_v1 resource where the user passes an ordered array of one or more rules. The actual firewall is created via the openstack_fw_firewall_v resource, where the user should declare the id of the policy that would be deployed on that firewall. Additionally, the firewall can be associated with specific routers at the associated_routers field. Figure 37 illustrates a Terraform file that uses these resources to create a firewall.

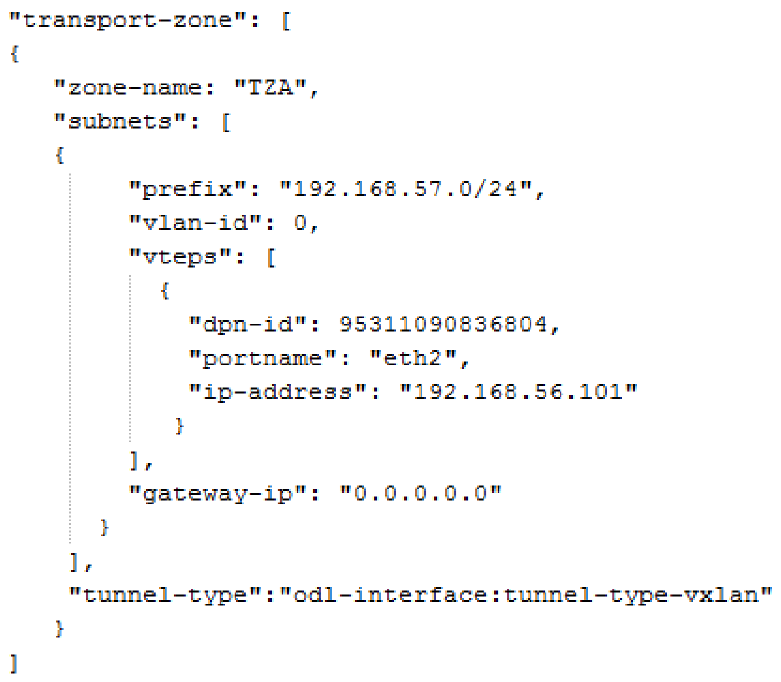


```
resource "openstack_fw_rule_v1" "rule_1" {
     name                    = "my-rule-1"
     action                  = "deny"
     protocol                = "tcp"
     destination_port        = "23"
     enabled                 = "true"
}

resource "openstack_fw_rule_v1" "rule_2" {
     name                    = "my-rule-2"
     action                  = "deny"
     protocol                = "udp"
     destination_ip          = "192.168.198.7"
     enabled                 = "false"
}

resource "openstack_fw_policy_v1" "policy_1" {
     name                    = "my-policy"

     rules = ["${openstack_fw_rule_v1.rule_1.id}","${openstack_fw_rule_v1.rule_2.id}"]
}

resource "openstack_fw_firewall_v1" "firewall_1" {
     name                    = "my-firwall"
     policy_id               = "${openstack_fw_policy_v1.policy_1.id}"
}
```

Figure 37: Terraform file for creating firewalls

Appendix H

Configuring the connections



```
"transport-zone": [
{
    "zone-name: "TZA",
    "subnets": [
    {
         "prefix": "192.168.57.0/24",
         "vlan-id": 0,
         "vteps": [
           {
             "dpn-id": 95311090836804,
             "portname": "eth2",
             "ip-address": "192.168.56.101"
            }
         ],
         "gateway-ip": "0.0.0.0.0"
      }
     ],
     "tunnel-type":"odl-interface:tunnel-type-vxlan"
    }
]
```

Figure 38: Transport Zone example

The user should pass the dpn-id of the device into the Transport Zone in order to declare it. The dpn-id is the datapthid of the br-int in decimal format. The Transport Zone example that is shown in Figure 38 creates a Transport Zone named TZA of type VXLAN. This Transport Zone includes a TEP with dpn-id 95311090836804, a tunnel interface named eth2 and local endpoint ip 192.168.57.101.

The user before creating the L2 gateway should create and configure a Neutron network and subnetwork. The IP of the bare metal machines should be in the same subnetwork created by the subnet create command.

```
openstack network create mynet1 --tenant_id $(opensatck project list |grep '\sadmin' | awk '{print $2}') --provider:network_type vxlan
openstack subnet create mynet1 11.11.11.0/24 --name net1-snet1
```

The next step is the creation of the L2 gateway and its allocation of the physical TOR device. The user should use the neutron l2-gateway-create command to create the gateway by passing the name of the ToR device in the name argument in the device flag and the id of the current used openstack project in the tenant_id flag.

```
neutron l2-gateway-create gw1 --tenant_id $(openstack project list | grep '\sadmin' | awk '{print $2}')
--device name=${PS_NAME},interaface_names=${INTERFACE_NAME}
```

The final step is the connection of the L2 gateway to the provider network. The neutron l2-gateway-connection-create command is used for that purpose. The user should give as arguments the name of the provider network and the name of the previously created L2Gateway.

```
neutron l2-gateway-connection-create gw1 mynet1 --default-segmentation-id 0
```

Appendix I

Hardware Requirements

The minimum hardware requirements [97] for the controller and the network nodes are:

- A 64-bit x86 processor with at least 1 core
- 16 GB of RAM
- 40 GB of disk space
- A minimum of 2 NICs of 1 Gbps. Additional NICs are needed for bonded interfaces or tagged VLAN traffic.

While the minimum requirements for the computing nodes are:

- A 64-bit x86 processor with at least 4 cores. The AMD-V or the Intel VT hardware virtualization extensions should be enabled on the virtualization compute nodes.
- 6 GB of RAM
- 40 GB of disk space
- A minimum of 2 NICs of 1 Gbps. Additional NICs are needed for bonded interfaces or tagged VLAN traffic.

Appendix J

Validation Interview

Participant's Information and Agreement

Name: **Date:**

Job Title: **Company:**

Years of Experience in the IT sector:

I agree to be listed in the master thesis with my name, job title and company, in the chapter concerning the validation of the design:

- ☐ Yes
- ☐ No

Questions

1. How would you evaluate the functionality of the current proposed design? Does this design serve all the high importance requirements?

2. How would you evaluate the functionality of the future-possibly improved design? Does this design serve all the high importance requirements?

3. How you evaluate the functionality of the network infrastructure?

4. Can you give your opinion on the potential performance [e.g. latency, utilization of resources] of this design (current and future)? Do you recognize any parts of the system that might affect the performance positively or negatively?

5. What do you think about the scalability and upgradability of the system?

6. What do you think about the reusability [can it be used in a wide variety of use cases] of the design ?

7. What do you think about the traceability (documentation of the system, error tracking etc.) of the design?

8. What do you think about the complexity and the required learning curve [in short and long term] from an average engineer in order to use and manage this system? What problems do you think they might face during this process ?

9. Do you think that the implementation of the design is feasible regarding the potential costs (e.g. hardware, people) ?

10. What do you think about the maturity of this design and the software-defined technology in general?

Appendix K

Interview Reports

Interviewee 1

Interviewee 1 is a Cloud Architect working at Thales as an external consultant. He is highly experienced in cloud systems and IT in general with around 40 years of experience in this field.

According to Interviewee 1, the current proposed design seems to be solid and serves all the requirements that were set. The designing approach of this system is the appropriate one and has a natural flow of thinking. This design serves the current goals of the current situation in the industry because it combines VMs, bare metal machines and containers in one system. However his main concern lies on Openstack because it is heavy and quite complex, but it is a reasonable solution for an organization that wants a private cloud solution and needs support for all the previously mentioned technology. In his opinion, this design most likely would change in the future to a system that is fully based on microservices and containers and has no need for virtual machines. In that case, Openstack most likely will be replaced by another tool. Regarding the use of Cobbler, Interviewee 1 sees no obvious drawback but he is not really experienced with this tool.

The future possibly improved design is also solid and again serves all the requirements that were set. The only difference with the previous one is that it uses Ironic instead of Cobbler. Ironic is a project that started as an Openstack service but now is a stand-alone service that is used by other tools as well. For example the Kubernetes cluster API uses Ironic to provision bare metal machines that run Kubernetes. This Kubernetes cluster API cannot provision VMs and is a bleeding edge technology that is a completely container based solution. Ansible and other similar configuration management tools might also not be used for future designs that provide immutable systems that are self-healed and patched, but for the current situation a configuration tools such as Ansible is necessary to push updates and patches to the system.

Interviewee 1 is not an expert when it comes to networking, but he thinks that this design manages to fulfill the requirements and offers connectivity between the virtual and physical networks. The SDN controller manages to control both the virtual and the physical switches. The addition of the controller enables us to write our own applications and scripts for controlling the traffic in our networks, because it offers the northbound APIs for that purpose. It make sense to use Openstack Neutron for creating the virtual networks as it is especially designed for that purpose and it also offers virtual services like firewalls and routers. It is nice that on top of that network we can still deploy Kubernetes with its own network functionality. It would be really interesting if in some years from now Kubernetes will be able to speak to the controller through the northbound API and provide all this network functionality, as Kubernetes networking is quite limited in its current version.

Interviewee 1 has not a clear picture of the potential performance of this design. He does not spot any specific bottlenecks or potential performance issues but he is not willing to dive into conclusions before testing is performed. The complexity of Openstack might cause some performance problems but he has not practical experience with Openstack in order to be entirely sure.

The scalability and upgradability of this design seems to be at a very good level. Terraform has an overview of the system and can discover if it is up to date. Openstack can manage, scale and upgrade the virtual part of the design adding extra scalability and Ansible is there to push updates and patches when it is necessary.

Interviewee 1 believes that this design can be used by many different organizations and it is highly dependent on the area of expertise of the organization. Some components might need to change depending on the use case. For instance, an organization that decided to follow a container-based approach will not need VMs and the entire part of this design that has to do with VMs will not be necessary any more. On the other hand, an organization that needs legacy applications needs the VMs part, but in his opinion VMs will go obsolete in the future. Currently they offer better security but the security of the containerization technology has also made great progress the last couple of years. Overall, this design is a good solution for companies that want a combination of functionalities and a good medium term solution. Companies that want to use up to edge technologies should focus more on container-based solutions and some components of this design might be replaced.

Interviewee 1 prefers the term observability instead of the term traceability. Observability is a big topic on its own and it is not properly addressed in this design. The main concept is to monitor hardware and applications in total and not as individual components. In order to have proper observability in this design, an extra component is needed. This component should be a monitoring pipeline that decouples the different layers in the system.

In the question regarding the complexity and the required learning curve, Interviewee 1 answered that the more the tools the biggest the learning curve. Openstack is a really complex system that requires time and effort in order to learn and use it properly. For the short term, it will be difficult for an average engineer to learn and use this system because it combines many difficult technologies such as Kubernetes, Openstack, software-defined networking. Generally speaking, this new software-defined world is really complex as the engineers need to have a good understanding of the full stack. This new world requires more DevOps people and not only people who are experts on their fields. People except their technical knowledge, should also change their general mentality. On the long term, this technology would be used with much less effort, but currently it is quite difficult to learn because it is relatively new.

Regarding the potential costs for implementing this design, Interviewee 1 mentions that they highly depend on the version of the software that the organization would choose to use. Most of these used tools have the open source free version and the enterprise one. The selection of the version is highly related to the available expertise inside the company. If the company has people who are experienced with this kind of technology then they would most likely go with the free version. On the other hand, companies that are inexperienced with this kind of technology, they would most likely choose the enterprise version that provides expert support. Regarding the hardware, this design definitely saves a lot of money because it uses COTS hardware which is at least less complex. These new hardware switches that support Openflow and OVSDB are relatively expensive now but they will become cheaper in the upcoming years. Every brand new technology is overpriced during its first release.

In Interviewee 1's opinion the technology that is used for the infrastructure layer is mature enough for a production level environment. Terraform is some sort of standard for that purpose but he cannot really grade the maturity of Openstack. Some years ago, Openstack was not mature enough but in that time private clouds were not a popular solution. Most of the organizations used to choose public cloud solutions. However this has started to change the last couple of years and many companies have moved to private cloud solutions. So maybe the maturity of Openstack has also increased till then. The SDN seems to be in its early stages but it is a promising technology that offers really useful functionalities. Companies should focus on how quickly they want to go to into production. If they want to go to production immediately this design can offer them what they seek for. A company that wants to go to production some years from now should really focus on this software defined technology as well. However they most likely should replace some components of this design. Overall, companies should focus on this technology if they want to achieve a scalable, upgradable and immutable system that will last several years,.

Interviewee 2

Interviewee 2 is currently working at Thales as a System Engineer and has changed several roles during his career. He has been working for Thales for around 20 years.

Interviewee 2 thinks that the proposed design meets the set requirements. There is a high level description of the infrastructure from a central location, which is performed by Terraform. However, there are two tools for managing and creating the infrastructure resources: Openstack and Cobbler. In the near future, it should be one tool for building the infrastructure. This solution is not the ideal one and can be characterized as a temporary solution for the problem. Ansible seems to be necessary in this design for configuring the resources and installing software on them (e.g. Kubernetes), but for the near future Ansible does not seem necessary. Its role might be replaced by a central component. Ansible is still quite a manual approach for configuration.

Interviewee 2 finds the future design better than the current design as it has one dynamic platform for controlling everything. The more the tools the higher the complexity of the system. The central provisioning and description of the infrastructure is still there with Terraform. It is nice that there is only one tool for creating VMs, Bare metal servers and Kubernetes containers. The desired solutions should have as less as possible interfaces between the components to reduce complexity. The advantage of this solution is also its main downside. Openstack controls everything, so there is high dependence on it. Solutions that focus mostly on one open source tool are quite risky because the choice of these tools might be the wrong one. A wrong selection might lead to a tool that is not supported anymore and eventually will disappear.

Regarding the network infrastructure, Interviewee 2 mentions that it differs highly from the current networking infrastructure of Thales. People should first change their way of designing and building the networks. If they manage to change their mindset, then this design seems solid. Everything is centrally controlled: both the physical and the virtual elements. His main concern lies on the compatibility of some used network appliances such as firewalls and dedicated proxies with the software-defined technology. Can these appliances be part of a software-defined network or should be replaced? In addition further research is needed on the VXLAN technology because it is relatively new. Does Thales actually needs these large networks created by VXLAN and what are the benefits of it?

According to Interviewee 2, this design is highly based on the virtualization technology, which by default adds some performance penalties due to the extra abstraction layer. The performance issues most of the times arise from the network or the computing servers. In this design the network seems to be efficient and the performance problems might arise from the computing environment. At the moment we do not have a clear picture of the performance as no performance studies have been conducted. Thales should invest time and effort on conducting additional studies on the performance with real world testing environments. Benchmark testing environments should be created that would provide a good reference point for future testing. That would detect and indicate the components of the system that cause performance issue. Overall, this design should work in practice but we do not know how efficient it will be. Based on the performance we can decide whether an actual implementation of this design is worth the effort.

The scalability and the upgradability of the design are at proper levels according to Interviewee 2 because this system was designed to scale easily. However this design is not fully dynamic yet when it comes to scalability. The optimal solution would be a design that scales optimally based on performance. The application will ask for the required capabilities, such as CPU, RAM, storage and the system will allocate the application to the node with the highest possible performance. In this proposed design, the scaling is somehow manually enforced by the user. The optimal design should provide automatic scalability, which would be enforced by some rules that describe how the applications would use the infrastructure. In this design, Kubernetes does that, but it is only one component of the solution. This design is the initial approach that would lead the way on how to reach the future optimal scalability.

Interviewee 2 mentions that this design can be used for several projects within Thales. His main concern is how this design would handle the several different security domains that exist within Thales. Currently each domain has its own infrastructure. The desired situation is to have one infrastructure for every domain. If this proposed design can manage to handle these security domains properly, then it should be used for many different projects.

When answering the question about the traceability of the design, Interviewee 2 indicated that there is no error tracking in this design and a monitoring pipeline should be added. The ideal solution in his opinion would be a resilient system that detects the hardware and software failures and automatically recovers from them without the human intervention. These failures most likely would affect the performance of the system, but the system should be able to determine how much it was affected and take the necessary actions to secure performance. The downtime of the system should remain low as much as possible. In addition, a decision making component should be included to make the best possible option in each case and reach the best possible solution. Regarding documentation, Interviewee 2 finds this design self-explanatory. The code files are the documentation and the engineers can use a version control tool to check the version of the infrastructure and keep track of the changes over time.

When it comes to learning curve, Interviewee 2 thinks that an initial effort is surely required by the engineers. Especially due to the fact that this design is way different from the current one, and the way of designing and building infrastructure differs highly from the current one. The learning curve should not be a problem for an experienced engineer. A junior engineer with only a few years of experience might face problems. The engineers should have a good understanding of each component of the system. The engineers should understand the functionality and the reasoning for using each component such as VMs, bare metal machines and Kubernetes components. In addition, knowledge on the cloud native technology is needed. However, in the long term this system should be much easier to use. The user can set up infrastructure resources by simply configuring some definition files. The components of the system are hidden by the user and everything is arranged by the system itself.

Interviewee 2 believes that this design is surely feasible regarding the costs and its implementation will reduce the costs. More computing power will be offered with less footprint. Everything will be automated and that will save time for the engineers. The engineers will be able to easily build multiple prototypes for testing without worrying about the costs. In addition in the long term specialized hardware will be unnecessary. Everything will be able to run on “white boxes”. People definitely need training but this should not require significant costs. An experienced engineer might need one to two days of training and an inexperienced one might need one month.

In the conclusion of the interview, Interviewee 2 referred to the maturity of this design and the software-defined technology in general. He finds the infrastructure layer of this design not mature yet as many aspects of the system are not known, such as security and performance. There is no real time testing available. However this technology is headed to the correct direction and has a lot of potential. There are many available solutions and it is really interesting to see which solution will prevail in the end. The network layer on the other hand seems to be mature and production ready. Many organizations have already been using this technology for setting up their networks and they are many available solutions. Of course, more initiatives are needed but overall, Thales should definitely invest time and effort on this technology.

Interviewee 3

Interviewee 3 is a Software Architect in Thales with approximately 20 years of experience in the field of IT. The last couple of years he has also started to take on more managerial responsibilities within Thales.

Interviewee 3 thinks that the proposed design should in theory fulfill the set requirements, but he does not really like the use of Openstack because it is really complex and adds another layer of abstraction. On the other hand, he finds the future design better than the current one. The future design makes use of Ironic for provisioning bare metal machines. It would be nice if Terraform could speak directly to Ironic. However, in the future design the problem with Openstack still remains. Regarding the network part, Interviewee 3 mentions that conceptually it should work but it cannot be used by Thales because many important network requirements are still missing, such as the support of multicast.

According to Interviewee 3, the design should not face any performance issues. He does not spot any potential bottlenecks and he mentions that the virtual switches that are used are good enough for the Thales purposes. In addition, he believes that in principle the infrastructure and the network layer of this design are scalable and upgradable. However, when it comes to reusability, he believes that this design is not ready to serve the projects within Thales, but maybe it is useful for other organizations. In order to use that design for Thales, further research is required on aspects like redundancy and performance. Furthermore, he commented that tracking errors in this design is difficult because it involves many software components. The related documentation for this design is relatively easy to find because all the used tools are open source. The configuration files describe the infrastructure but some sort of extra documentation explaining in more details the system is required.

When discussing the complexity of the design, Interviewee 3 mentioned that this design is really complex because it involves several software components. He is also confident that the people within Thales are capable of learning how to use and manage this design. Development is easy in this design but the testing part would be difficult as the people that perform testing understand simple methods and commands and learning this new way of thinking would be challenging. Also solving problems in the network part would be challenging.

According to Interviewee 3, the potential costs for this design will not be high. Specifically, he mentions that the hardware costs will be minimum because this design uses COTS hardware, and cost would be reduced due to the easier maintainability. Training is definitely important and should be done but it would not be expensive.

Generally, Interviewee 3 focuses mostly on the performance and the fulfillment of the desired requirements and the costs are not his main priority.

Concluding the interview, Interviewee 3 believes that the maturity of this design and the software-defined technology in general is not at the desired level for Thales. However the maturity of this technology seems to be at the appropriate level for other organizations as they already use it in production. Interviewee 3 would like to see further research on the available dynamic platforms that are able to provision bare metal machines, and also on some other alternatives for the SDN controllers. In addition, follow-up research is required on redundancy and performance. Security is not his main concern for the time being.

Interviewee 4

Interviewee 4 works as an IT Infrastructure Architect at Thales and he has more than 30 years of experience in the IT sector.

Interviewee 4 believes that both the proposed and the future design for the infrastructure layer are really complex. Both designs consist of too many different software components and they are both based on Openstack, which is really complex and in his opinion outdated. These designs theoretically serve the requirements that were set, but they cannot be used for the Thales case. This design is meant for large IT systems and it cannot work efficiently for smaller systems. Most Thales projects consist of small systems with a small number of servers. This design requires at minimum four servers. This proposed design is a general approach that can create VMs, bare metal and containers; however Thales should mainly focus on Kubernetes and containers. Kubernetes can use Ironic to provision bare metal machines and can also create VMs. The VMs functionality in Kubernetes is not mature yet but it would improve in the future. Interviewee 4 also expressed his doubts about Terraform and whether it is the best available solution. In case, Thales needs VMs could use the old-fashion way of doing so, as there are not used often.

Regarding the network infrastructure, Interviewee 4 finds the selection of Opendaylight reasonable because it is compatible with Openstack, but he personally thinks that Openstack is not the best choice. So if Openstack is replaced from this design, then ODL might not be the optimal choice of SDN controller. Thales should focus on the integration of Kubernetes with the SDN technology and investigate the potential opportunities. A SDN controller might not even be necessary and Kubernetes might be able to handle the networking of the entire infrastructure.

According to Interviewee 4, all these software components that should function concurrently all the time, might cause high start-up times and that would jeopardize the performance. In addition, the complexity caused by all these software interfaces might cause latency problems. Interviewee 4 cannot spot any potential performance issues at the network layer. The virtual switches are capable of handling the use of Thales, offering maybe even faster performance than the currently used hardware.

Interviewee 4 mentions that this design offers higher scalability and it is meant for building large systems. Scaling down with this design is difficult. Maintainability is also challenging because it includes many different software components and It would be difficult to find and fix the errors. Furthermore, the engineer would face difficulties upgrading this design because there is too much dependency between the software components. Upgrading one component requires pushing updates to several other ones.

Interviewee 4 believes that this design cannot be used for different projects within Thales because most projects use small systems. Even in case a bigger system is required, it would be better to split it into smaller better defined systems. Applications with different life cycles should be separated into different systems with better defined responsibilities.

Interviewee 4 mentions there is not observability support in this design and a monitoring pipeline should be added to serve that purpose. Nevertheless, error tracking would be difficult because there is too much software involved. In general, documentation is improved with the use of software-defined tools.

In the discussion about the learning curve, Interviewee 4 mentioned that this design would have high learning curve both for short and long term. This design would be difficult to learn and use due to its complexity. This design requires people who are experts in this field and have a good understanding of the relations between the involved software. That requires advanced training, which is costly. This design is based on open source technologies; however if the company does not have experienced people in house, they will need to choose the supported version, which is expensive. So overall, Interviewee 4 believes that the implementation of this design is expensive due to its high complexity.

Interviewee 4 concluded the interview by stating that this design and the software-technology is not mature yet for the Thales case, but the situation will improve with further future research. He is confident that investing time and effort on this technology is the correct direction for Thales and he would like to see further research on the network part, and how Kubernetes can be used in combination with the SDN technology.

Interviewee 5

Interviewee 5 is an experienced Cloud Architect with 22 years of experience in this field. Currently he works as full-time employee at Thales, and before moving to Thales he used to work as a Cloud consultant for several different organizations. Interviewee 5 is a passionate advocate of Cloud Native technologies.

Interviewee 5 thinks that the current proposed design meets the requirements, and it should work conceptually, but he characterized it as the old way of building this kind of systems. Ironic is a useful feature but unfortunately it is not supported by this version of Terraform. Cobbler is definitely a solution for provisioning bare metal but he would like to see a more immutable solution in which Ansible is not used or it is only used to prepare the images. In an immutable system, each server is almost identical to each other and when one is unavailable, is immediately replaced with another one. On the other hand, Interviewee 5 found the future design much better than the current one because everything is controlled by Openstack. In this case Openstack is one big control plane that uses plugins services to perform the several tasks. However that makes Openstack really big in size and complexity.

Interviewee 5 finds huge potential in the design of the network infrastructure, and he especially liked the potential of the ODL controller because the network administrator can control physical and virtual switches, and it offers a REST API to control it, making it programmable. He would like to see Openstack replaced with another tool/platform, such as Kubernetes. Kubernetes might be able to control the physical switches in the future.

Regarding performance, Interviewee 5 mentions that all these virtualization layers that exist in this design might cause performance issues. Whenever an additional abstraction layer is added, it comes with a price, which can be in availability, complexity or latency. Moreover, Interviewee 5 says that the scalability of a design is always dependent on the size of the system. The proposed design can is easily scale up in theory but it requires a really big infrastructure to build on. This design is mostly meant for big cloud system and it is not designed for smaller ones. For example Openstack by default requires for 4 - 5 servers to run. Furthermore, he commented that conceptually this design should be easily upgradable because it includes Terraform, which is a declarative solution.

According to Interviewee 5, the reusability of this design depends highly on the infrastructure size of the organization. This design can work for organizations that use really big cloud-based datacenters and maybe it can also be used by Thales on really big vessels that have relatively big datacenters. However this design is not applicable for most of the Thales projects that require smaller systems.

Interviewee 5 mentions that the concepts of traceability are better defined by the term observability. Observability is really important for monitoring the components of the system and get insights on the configuration and identify the component that might cause errors. In this design there is no monitoring so an observability pipeline should be added.

Interviewee 5 believes that the learning curve for the development teams would be low because Terraform is really easy to learn and use. However, things would be difficult for the operation teams that would have to install and learn all these components and the dependencies between them. The operation team should consist of really well-trained people that have a good overview of the entire stack. Overall the learning curve for the development team would be easy, but for the operation team it would be really high. In addition, the maintainability of this system would be challenging due to the high complexity. In general, the organization should consists of DevOps team that work together in order to properly manage and use this design. Considering the potential costs, Interviewee 5 thinks that this design is not feasible for the Thales case, mainly because of Openstack, which is really complex resulting to high costs for training and maintenance.

In the final question regarding the maturity of this design, Interviewee 5 answered that the used technology is definitely mature but it includes high levels of complexity, especially Openstack. The software-defined technology in general lacks of standardization, which is a sign of immaturity. He would like to have follow-up research that focuses on the ODL controller and on an open infrastructure API for provisioning bare metal machines using immutable images. In addition, he would like to see further research on the public clouds and proprietary solution because buying hardware and setting up private clouds might be more expensive that outsourcing everything to a public cloud as long as the security requirements are met. In the conclusion of the interview, Interviewee 5 mentioned that we should design smaller and simpler solutions and not spending money and effort on big complicated systems. As a passionate supporter of the cloud native movement, he ended the interview by expressing his preference on container-based solutions mentioning that containers are the future in the IT world.

Interviewee 6

Interviewee 6 works as a System Architect at Thales and he has approximately 30 years of experience in designing IT systems.

Theoretically speaking, Interviewee 6 mentioned that the current proposed design is interesting and should work, but he has no real practical experience with these tools so he cannot be certain about the functionality of this design in practice. Cobbler is able to provision bare metal properly and Openstack seems to be really complex in his opinion. The same comments apply also for the future design. Interviewee 6 is really focused on the maturity and the complexity of Openstack. If Openstack is mature and simple enough, then this design should work. However all these technologies are relatively new and they are always changing, so further research and practical testing on Openstack are required to acquire a clear understanding of these tools.

Regarding the network infrastructure, Interviewee 6 mentions that he is familiar with the technologies in concepts, but again he lacks of practical experience with them. In principle, this network design is great but he cannot be certain if it is applicable for Thales. He would like to learn more about security, redundancy and how this technology manages some specialized network requirements such as multicast. He also has some concerns about the maturity of the ODL controller and how redundant it can be.

Interviewee 6 could not give a certain answer regarding the potential performance of the design. He cannot recognize any potential bottlenecks and needs additional real-time performance testing in order to have a clear of the efficiency of this design. Several network features, such as routing are performed by the controller and that might be a performance issue. Of course this design should combine high performance with high redundancy in order to be used in production.

According to Interviewee 6, the scalability of this design depends on the size of the available infrastructure. This design seems to perform well in large cloud systems; however the Thales case is not close to this. Thales uses small sized system with maximum four servers in size, so he is not sure how this design handles lower scalability. In his opinion, the upgradability of this design seems to be at a decent level because it uses tools such as Kubernetes and Ansible. However the main goal is to create a higher abstraction layer where we can create systems at a higher level and avoid diving into tools and configuration files.

Interviewee 6 is certain that this technology can be used for different use cases, but he is not sure about the reusability of this specific design. In order to use it in many different cases we have to create a higher abstraction layer for the applications and the system should be independent from the application. To achieve that we have to define a good mix of the limitations and identify what is feasible for our use case. Regarding the traceability of this design, Interviewee 6 mentioned that this design completely lacks observability and a monitoring pipeline should definitely be added in the future.

Interviewee 6 is highly concerned about the complexity of this design and he thinks that the required learning curve will be relatively high as it requires a good system design in advance and this system should be completely hardware independent, which is a challenging process. In order to achieve this way of thinking we need specialists who are difficult to find or train, as a result their selection should be performed carefully. The recruitment or training of experts is an expensive process. In case we do not manage to acquire the necessary experts, then we should choose the supported version of these tools which also comes with relatively high costs. Overall, a thorough and well organized business plan is needed to estimate the exact costs.

Interviewee 6 concluded that the used technologies are unstable and their maturity is not at the proper levels for Thales at this point of time. However this is a good opportunity for Thales to learn more on these kinds of technologies by conducting further research on topics such as redundancy, safety, performance and specific network features. Interviewee 6 is really curious to see if the promises of the software-defined technology would actually manage to meet reality.

Interviewee 7

Interviewee 7 is a network engineer at Thales with 5 years of experience in the IT sector. Interviewee 7 is mainly focused on the network layer combining an overview of the infrastructure layer. Interviewee 7 is familiar with the software-defined concepts and tooling.

Interviewee 7 is confident that the current proposed design covers the defined requirements and it can combine VMs, bare metal machines and containers in one system. Of course, this design is not optimal because there are many separate tools for provisioning the resources. The ideal design should have one central dynamic platform for building the entire infrastructure. This technology is really new for the use case of Thales and further research is required in order to be used into production environments. Thales uses in production only mature and as simple as possible solutions. On the other hand, Interviewee 7 finds the future design much better compared to the current one because it offers one dynamic platform for building all the required infrastructure resources. Development and debugging will be easier in this design because the engineers will only have to focus on learning and using only one tool. One downside of this design might be the complexity of Openstack but there were no alternatives that met the defined requirements.

Interviewee 7 mentions that the network infrastructure is complex as it involves many new technologies, but it is really promising and useful because we can control and build networks from a central point, which is the SDN controller. The ODL controller seems to have potential. One downside is that we need special HW switches that support OVSDB and Openflow, which are not available at Thales at the moment, but most likely we will acquire them in the future. He also likes the fact that this technology includes many security features such as virtual firewalls and security groups to control the flow of the traffic. We need testing with real HW switches to see how this system behaves in a real environment. Furthermore, this network infrastructure is based entirely on VXLAN tunneling. This technology handles quite well unicast traffic, however it is not clear how it handles multicast, which is used by many applications within Thales. The debugging process of the network might be challenging due to its complexity but Interviewee 7 thinks that the controller would be able to facilitate it.

Regarding the performance, Interviewee 7 mentions that the network part will not cause any performance issues as long as the required hardware is available. The virtual switches do not seem to cause any performance problem. Of course additional research and actual testing is required to identify and measure the failover time of the virtual switches. Actual testing with these specialized physical switches would also be interesting and it would provide a better insight on the performance of the entire network. Furthermore, the performance of the infrastructure layer should be the desired one because we are always able to select the most suitable infrastructure resources for each specific use case.

According to Interviewee 7, this software-defined system is designed to be easily scalable and it will not face any scalability issues. This design also has better upgradability compared to the currently used systems at Thales, where everything has to be reinstalled whenever a new update is released. High levels of upgradability are reached with the use of Terraform and Ansible.

Interviewee 7 mentions that this design might serve many different projects for a wide range of organizations that support different kinds of applications. For Thales this technology might be used for setting up the infrastructure for the combat management system, and maybe for building the infrastructure of a big battleship. However his main concern lies on how this technology manages the different security domains that exist at the naval domain. The customers of Thales have many different requirements that restrict the flow of data between two different security domains.

Code files are used to describe the entire infrastructure, which significantly improves the documentation of the system according to Interviewee 7. On top of the code, the engineers can write comments that describe the functionality of each resource. In addition, these code files can be version controlled by version control system such as Git. The engineers would be able to keep track of the changes and roll back to previous versions easily. Regarding the observability, he think that something is missing. Openstack offers some monitoring but an extra monitoring pipeline is needed in his opinion. However this was not the goal of this research project.

Interviewee 7 believes that the required learning curve depends on the experience and the age of the engineers in the organization. If the organization has young people, it would be easy to train them on how to use and maintain this new system. Older people have a different mindset, which might be a bit difficult to change. This new technology requires a different view of the entire system. The developers should also understand the infrastructure concepts, so some additional training for the development teams might be necessary. This technology of course is complex as it includes many different components with many different interfaces; as a result there are higher changes of errors. That would increase the required time and effort for maintenance. Overall, Interviewee 7 believes that the learning curve will not be really high and the engineers will manage to cope with this new technology.

During the discussion about the potential costs, Interviewee 7 mentioned that the hardware costs will not be high and they maybe be even lower that the current values. This technology runs on different kind of COTS hardware, which automatically reduces the costs. In addition, this specialized OVSDB switches will become cheaper in the close future. These switches are quite expensive at the moment as every new technology after its release. Furthermore, this design uses virtualization technology that utilizes the resources of the infrastructure efficiently. On the other hand, licenses for the supported version of the tool might be costly. Thales always selects the tool version that offers the best support. In total, Interviewee 7 believes that this design would be cheaper in the future and the prices will drop by the time this design is used in many different projects. Currently each project in Thales uses a different infrastructure, and this design will replace that with one infrastructure for every project.

In the conclusion of the interview, Interviewee 7 referred to the maturity of the software-defined technology. In his opinion, this technology is relatively new and it is still under improvement. This technology is used by many companies so its maturity is sufficient, but for the Thales case this technology is still immature. For Thales this is a really big step and follow up research is required. He would like to see additional research on the topics related to performance, redundancy and security. For the Thales customers performance is the most important aspect, so most of the future research should focus on this aspect.

References

1. SDx Central, “What is Software Defined Everything”, Retrieved on 5 August, 2019, from: https://www.sdxcentral.com/cloud/definitions/software-defined-everything-sdx-part-1-definition/
2. Peffers, K., Tuunamen, T., Rothenberger, M., & Chatterjee, S., “A Design Science Research Methodology for Information Systems Research”, 2007, Journal of Management Information Systems(24), 45-77
3. Roel J. Wieringa, “Design Science Methodology for Information Systems and Software Engineering”, Springer, 2014, p. 1-63
4. Alan R. Hevner, "A Three Cycle View of Design Science Research," 2007, Scandinavian Journal of Information Systems: Vol. 19: Iss. 2 , Article 4
5. 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), 2015, pp. 742-745
6. Kief Morris, “Infrastructure as Code: Managing Servers in the Cloud”, O’Reilly Media Inc., 1st Edition, 2016, p. 5 – 40
7. Openstack, “Adding Speed and Agility to Virtualized Infrastructure with Openstack”, Whitepaper, 2015, Retrieved on 27 November, 2019, from: https://www.openstack.org/assets/pdf-downloads/virtualization-Integration-whitepaper-2015.pdf
8. Foreman, Retrieved on 11 September 2019, from: https://theforeman.org/
9. 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, p. 3
10. Amazon, “AWS CloudFormation”, Retrieved on 5 August, 2019, from: https://aws.amazon.com/cloudformation/
11. Google, “Cloud Deployment Manager”, Retrieved on 5 August, 2019, from: https://cloud.google.com/deployment-manager/
12. Microsoft, “Azure Resource Manager Overview”, Retrieved on 5 August, 2019, from: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview
13. HarshiCorp, “HarshiCorp Terraform”, Retrieved on 5 August, 2019, from: https://www.terraform.io/
14. Prassanna J*, Anjali R Pawar, Neelanarayanan V, “A Review of Existing Cloud Automation Tools”, Asian Journal of Pharmaceutical and Clinical Research, vol. 10, 2017, p. 471-473
15. Openstack, “Openstack Heat”, Retrieved on 5 August, 2019, from: https://wiki.openstack.org/wiki/Heat

16. Openstack, “Telemetry”, Retrieved on 5 August, 2019, from: https://wiki.openstack.org/wiki/Telemetry
17. Amazon, “Auto Scaling Groups”, Retrieved on 5 August, 2019, from: https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html
18. Navin Sabharwal, Manak Wadhwa, “Automation through Chef Opscode: A Hand-on Approach to Chef”, Apress publications, 1st Edition, 2014, p. 5
19. Chef, “Learn Chef: An Overview of Chef”, Retrieved on 5 August, 2019, from: https://docs.chef.io/chef_overview.html
20. Gregory Katsaros, Michael Mezel, Alexander Lenk, Jannis Rake-Revelant, Ryan Skipp, Jacob Eberhardt, “Cloud application portability with TOSCA, Chef and Openstack: Experiences from a proof of concept implementation”, IEEE International Conference on Cloud Engineering, 2014
21. James Loope, “Managing Infrastructure with Puppet ”, O’Reilly Media Inc, 1st Edition, 2011, p. 1
22. Christian Endres, Uwe Breitenbucher, Michael Falkenthal, Oliver Kopp, Frank Leymann, Johannes Wettinger, “Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications”, Proceedings of the 9th International Conference of Pervasive Patterns and Applications, 2017, p. 22-27
23. Redhat, “Ansible in Depth”, Whitepaper, Retrieved on 27 November, 2019, from: http://www.cmsdistribution.com/wp-content/uploads/2016/09/Ansible-in-Depth-Whitepaper.pdf
24. Colton Myers, “Learning SaltStack”, Packt Publishing, 2nd Edition, 2016, p. v
25. Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, Steve Uhlig, “Software-Defined Networking: A Comprehensive Survey”, Proceeding of the IEEE, 2014
26. Bilal R. Al-Kaseem, Hamed S. Al-Rawshidy, “SD-NFV as an Energy Efficient Approach For M2M Network Using Cloud-Based 6LoWPAN Testbed ”, IEEE Internet of Things Journal, vol. 4, 2017
27. Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, Martin Casado, “The Design and Implementation of Open vSwitch”, Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015
28. Nick McKeon, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Lary Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner, “OpenFlow: Enabling Innovation in Campus Networks”, ACM SIGCOMM Computer Communication Review 38, 2008, p. 69-74
29. B. Pfaff, B. Davie, “The Open vSwitch Database Management Protocol”, Retrieved on 29 August, from: http://www.hjp.at/doc/rfc/rfc7047.html

30. Kok-Kiong Yap, Te-Yuan Huang, Ben Dodson, Monica S. Lam, Nick McKeown, “Towards Software-Friendly Networks ”, ApSys ’10: Proceedings of the first ACM Asia-pacific workshop, 2010, p. 49-54
31. Cisco, “Cisco IOS Technologies”, Retrieved on 5 August, 2019, from: https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-technologies/index.html
32. Juniper Networks, “Junos OS Overview”, Retrieved on 5 August, 2019, from: https://www.juniper.net/us/en/products-services/nos/junos/
33. Jim Doherty, “SDN and NFV Simplified: A Visual Guide to Understanding Software Defined Networks and Network Function Virtualization”, Pearson Education Inc., 2016, p. 210
34. GitHub, “The POX network software platform”, Retrieved on 5 August, from: https://github.com/noxrepo/pox
35. GitHub, “Ryu SDN Framework”, Retrieved on 5 August, 2019, from: http://osrg.github.io/ryu/
36. GitHub, “Trema: Full-Stack OpenFlow Framework in Ruby and C”, Retrieved on 5 August, 2019, from: http://trema.github.io/trema/
37. Project Floodlight, “Floodlight OpenFlow Controller”, Retrieved on 5 August, 2019, from: http://www.projectfloodlight.org/floodlight/
38. Rahamatullah Khondoker, Adel Zaalouk, Ronald Marx, Kpatcha Bayarou, “Feature-based comparison and selection of Software Defined Networking (SDN) controllers”, World Congress on Computer Applications and Information Systems (WCCAIS), 2014, p. 1-7
39. The Linux Foundation projects, “OpenDayLight”, Retrieved on 5 August, 2019, from: https://www.opendaylight.org/
40. Ahmad Hemid, “Facilitation of the OpenDaylight Architecture”, Computer Science Conference for University of Bonn Students (CSCUBS), 2017
41. SDx Central, “What is Software Defined Compute?”, Retrieved on 5 August, 2019, from:https://www.sdxcentral.com/networking/sdn/definitions/what-is-software-defined-compute/
42. David Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes”, IEEE Cloud Computing, 2014, vol. 1, no. 3, pp. 81-84
43. WEI-Tech Brief, “Five Game-Changing Advantages of Software Defined Computing”, Whitepaper, Retrieved on 27 November, 2019, from: https://cdn2.hubspot.net/hubfs/1774733/Tech_Briefs/Tech_Brief_-_5_Game-changing_Advantages_of_Software_Defined_Computing.pdf
44. Redhat, “Linux-KVM”, Retrieved on 5 August, 2019, from: https://www.linux-kvm.org/page/Main_Page
45. The Linux Foundation Projects, “Xen project”, Retrieved on 5 August, 2019, from: https://xenproject.org/

46. Vmware, “Vmware vSphere”, Retrieved on 5 August, 2019, from: https://my.vmware.com/web/vmware/info/slug/datacenter_cloud_infrastructure/vmware_vsphere/6_0#open_source
47. Oracle, “Virtual Box”, Retrieved on 5 August, 2019, from: https://www.virtualbox.org/
48. Opensource.com, “What is Docker”, Retrieved on 5 August, 2019, from: https://opensource.com/resources/what-docker
49. Openstack, Retrieved on 5 August, 2019, from: https://www.openstack.org/
50. Openstack, “Horizon: The Openstack Dashboard Project”, Retrieved on 5 August, 2019, from: https://docs.openstack.org/horizon/latest/
51. Openstack, “Neutron”, Retrieved on 5 August, 2019, from: https://wiki.openstack.org/wiki/Neutron
52. Openstack, “Magnum”, Retrieved on 5 August, 2019, from: https://wiki.openstack.org/wiki/Magnum
53. Kubernetes, Retrieved on 5 August, 2019, from: https://kubernetes.io/
54. Openstack, “Ironic”, Retrieved on 5 August, 2019, from: https://wiki.openstack.org/wiki/Ironic
55. Cobbler, Retrieved on 5 August, 2019, from: https://cobbler.github.io/
56. Ansible, “Working with Dynamic Inventory”, Retrieved on 5 August, 2019, from: https://docs.ansible.com/ansible/latest/user_guide/intro_dynamic_inventory.html#inventory-script-example-cobbler
57. Terraform, “openstack_compute_instance_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/compute_instance_v2.html
58. Openstack, “Select hosts where instances are launched”, Retrieved on 5 August, 2019, from: https://docs.openstack.org/nova/latest/admin/availability-zones.html
59. Terraform, “openstack_images_image_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/images_image_v2.html
60. Openstack, “Useful image properties”, Retrieved on 5 August, 2019, from: https://docs.openstack.org/glance/rocky/admin/useful-image-properties.html
61. Terraform, “openstack_compute_flavor_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/compute_flavor_v2.html
62. Openstack, “Attaching virtual GPU devices to guests”, Retrieved on 5 August, 2019, from: https://docs.openstack.org/nova/queens/admin/virtual-gpu.html
63. Openstack, “Attaching physical PCI devices to guests”, Retrieved on 5 August, 2019, from: https://docs.openstack.org/nova/stein/admin/pci-passthrough.html
64. Terraform, “Cobbler Provider”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/cobbler/index.html
65. Oracle, “PXE Booting and Kickstart Technology”, Retrieved on 5 August, 2019, from: https://docs.oracle.com/cd/B16240_01/doc/em.102/e14500/appdx_pxeboot.htm
66. IBM Developer, “Automate and manage systems installation with Cobbler”, Retrieved on 5 August, 2019, from: https://developer.ibm.com/articles/l-cobbler/

67. Terraform, “cobbler_system”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/cobbler/r/system.html
68. Terraform, “cobbler_profile”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/cobbler/r/profile.html
69. Terraform, “cobbler_distro”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/cobbler/r/distro.html
70. Terraform, “cobbler_repo”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/cobbler/r/repo.html
71. GitHub, “Kubernetes”, Retrieved on 20 Septemeber, 2019, from: https://github.com/kubernetes/kubernetes
72. Terraform, “openstack_containerinfra_cluster_v1”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/containerinfra_cluster_v1.html
73. Terrafrom, “openstack_containerinfra_clustertemplate_v1”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/containerinfra_clustertemplate_v1.html
74. UnixArena, “Ansible – How to use facts on Playbooks”, Retrieved on 5 August, 2019, from: https://www.unixarena.com/2018/08/ansible-how-to-use-facts-on-playbooks-conditional-check.html/
75. Terraform, “openstack_networking_network_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/networking_network_v2.html
76. Terraform, “openstack_networking_subnet_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/networking_subnet_v2.html
77. Terraform, “openstack_networking_router_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/networking_router_v2.html
78. Terraform, “openstack_networking_router_interface_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/networking_router_interface_v2.html
79. Terraform, “openstack_compute_floatingip_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/compute_floatingip_v2.html
80. Terraform, “openstack_compute_floatingip_associate_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/compute_floatingip_associate_v2.html
81. Terraform, “openstack_networking_secgroup_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/networking_secgroup_v2.html
82. Terraform, “openstack_networking_secgroup_rule_v2”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/networking_secgroup_rule_v2.html
83. Openstack, “Firewall-as-a-Service (FWaaS)”, Retrieved on 5 August, 2019, from: https://docs.openstack.org/neutron/pike/admin/fwaas.html
84. Terraform, “openstack_fw_firewall_v1”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/fw_firewall_v1.html

85. Terraform, “openstack_fw_policy_v1”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/fw_policy_v1.html
86. Terraform, “openstack_fw_rule_v1”, Retrieved on 5 August, 2019, from: https://www.terraform.io/docs/providers/openstack/r/fw_rule_v1.html
87. Pica8, “Improving Overlay Solutions with Hardware-Based VXLAN Termination”, Whitepaper, Retrieved on 27 November, 2019, from: https://www.pica8.com/wp-content/uploads/pica8-whitepaper-VXLAN-overlay.pdf
88. SDx Central, “What is VXLAN”, Retrieved on 16 September, 2019, from: https://www.sdxcentral.com/networking/virtualization/definitions/what-is-vxlan/
89. Open vSwitch, “Open vSwitch Manual – hardware_vtep database schema”, Retrieved on 5 August, 2019, from: http://www.openvswitch.org/support/dist-docs/vtep.5.html
90. Juniper Networks, “Understanding the OVSDB Protocol Running on Juniper Network Devices”, Retrieved on 5 August, 2019, from: https://www.juniper.net/documentation/en_US/junos/topics/concept/sdn-ovsdb-junos.html
91. VMWare Inc, “The Open vSwitch Database Management Protocol”, Retrieved on 5 August, 2019, from: https://tools.ietf.org/html/rfc7047#section-1.2
92. Openstack, “Neutron/ L2-GW”, Retrieved on 5 August, 2019, from: https://wiki.openstack.org/wiki/Neutron/L2-GW
93. Networkop.co.uk, “Openstack SDN – Interconnecting VMs and Physical Devices with Cumulus VX L2 Gateway”, Retrieved on 5 August, 2019, from: https://networkop.co.uk/blog/2016/05/21/neutron-l2gw/
94. Opendaylight, “NetVirt: L2Gateway Howto”, Retrieved on 5 August, 2019, from: https://wiki.opendaylight.org/view/NetVirt:_L2Gateway_HowTo
95. Opendaylight, “NetVirt”, Retrieved on 5 August, 2019, from: https://wiki.opendaylight.org/view/NetVirt
96. Opendaylight, “OVSDB Integration: L3Fwd”, Retrieved on 3 September, 2019, from: https://wiki.opendaylight.org/view/OVSDB_Integration:L3Fwd
97. Redhat, “Prepare for Opendaylght installation”, Retrieved on 29 August, 2019, from: https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/11/html/opendaylight_and_red_hat_openstack_installation_and_configuration_guide/prepare_for_opendaylight_installation
98. ISO 25000, “ISO/IEC 25010”, Retrieved on 30 September, from: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&limitstart=0
99. GitHub, “Metal^3 Documentantion”, Retrieved on 24 October, 2019, from: https://github.com/metal3-io/metal3-docs
100. Digital Rebar, Retrieved on 24 October, 2019, from: https://rebar.digital