Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

# MASTER THESIS

**TITLE: Containerizing ONAP using Kubernetes and Docker**

**MASTER DEGREE: Master's degree in Applied Telecommunications and Engineering Management (MASTEAM)**

**AUTHOR: Albert Toro Marin**

**ADVISOR: Jesus Alcober**

**DATE: July, 10th 2019**

**Abstract**

This document explains the deployment of virtualized network functions in a private cloud through ONAP, together with the architecture used, the machines employed and the software required.

ONAP (Open Networking Automation Platform) is an open source project of the Linux Foundation, which provides a platform for real-time orchestration and automation of physical and virtual network functions.ONAP allows designing, orchestrating and managing all the elements related to virtualized network functions.

In order to deploy ONAP, we acquired and installed a cloud infrastructure from scratch following the latest market best practices, which included setting up the servers and network devices, configuring their NICs and installing all the software required: OpenStack, ONAP, etc.

Moreover, these software solutions also have to be properly configured and installed, something that will be clarified in this paper along with the configuration of the cloud network.

ONAP, being a novel solution, is not fully developed. Meaning it has presented many challenges and setbacks we have had to overcome in order to deploy it successfully in our environment.

The final goal of this project is to create an environment in a private cloud with working virtualized network functions and to serve as a guide for other people who desire to deploy ONAP, overcoming all the obstacles we had to face and providing solutions, or at least, alternatives to the plethora of issues they might stumble across, just as we did.

Everything discussed in this document can be replicated in environments of greater scope than the ones defined in the paper, to the point that what is explained can even be applied in production scenarios by telephone operators.

## Special thanks to:

Imgflip, for providing memes when ONAP didn’t work as it should

ONAP, for making a platform almost impossible to run

The Aussie, for coping with my incessant rants when ONAP stopped working for no good reason

Liliane, for making me smile when I should’ve cried

Masmi and the Masmitos, for having helped me all this time

Whoever is currently reading this document. Will you be able to finish it?

# INDEX

# INTRODUCTION

Nowadays, one of the most heard buzzwords is virtualization, but this recurring term has been around since the 1960s when it simply meant to divide a computer's resources between different applications. With the years, the actual meaning of the term has substantially broadened [1].

ONAP (Open Networking Automation Platform) is an open source project of the Linux Foundation, which provides a platform for real-time orchestration and automation of physical and virtual network functions that allows IT producers, cloud providers, and developers to quickly automate new services and support the complete management of their life cycle [2].

Its architecture is very innovative because it is divided into two environments; a run-time framework dedicated to the design of the infrastructure (virtualized services, virtual network functions, or VNF, and physical network functions, or PNF), and another framework employed during the execution time (creation and management of service instances, VNF and PNF), making ONAP one of the preferred options of the different network providers [3].

ONAP, being a novel solution, is not fully developed. Meaning it has presented many challenges and setbacks we have had to overcome in order to deploy it successfully in our environment.

The document is organized as follows:

Firstly, we describe the architecture designed to offer these virtualized network functions.

Secondly, we present the method applied to generate the private cloud using OpenStack and the orchestrating mechanism based on Kubernetes for the automation of the deployment, load balancing, and management of applications in containers, and Rancher, which allows managing multiple Kubernetes clusters.

Lastly, we explain the results obtained and the conclusions, which to sum up, has been a satisfactory automated deployment of ONAP's private cloud.

The work presented in this paper will be the starting point of another project called "Developing and deploying advanced NFV solutions" in which the goal is to use ONAP's infrastructure in order to deploy virtualized network functions.

# CHAPTER 1. VIRTUALIZATION OVERVIEW

On this first chapter, we describe the architecture designed to offer these virtualized network functions and the opportunities of the project.

## 1.1. Opportunities and goals of the project

The aim of this project is to take advantage of virtualization technologies in order to implement cutting-edge solutions, especially in the field of network service providing and service management in the cloud, looking to get rid of the nowadays monolithic solutions for clouds and get closer to a portable, efficient, scalable, and virtualized approach.

To obtain a better insight into virtualization, you will find a brief introduction in Annex I.

Basically, this project has to serve as a guide for other people who desire to deploy ONAP, overcoming all the obstacles we had to face and providing solutions, or at least, alternatives to the plethora of issues they might stumble across, just as we did.

Even if virtualization has clear benefits, many companies have qualms in using virtualized solutions because it is a very different way to do what they are currently doing.

With this in mind, this project has to show the clear advantages virtualization has over legacy solutions implementing a fully functional cloud infrastructure.

However, in order to implement this cloud environment, many different software solutions are required, solutions that will be explained in detail in the following point.

In order to deploy ONAP, we acquired and installed a cloud infrastructure from scratch following the latest market best practices, which included setting up the servers and network devices, configuring their NICs and installing all the software required: OpenStack, ONAP, etc.

Moreover, these software solutions also have to be properly configured and installed, something that will be clarified in one of the following chapters along with the configuration of the cloud network.

To sum up, the main goal of this project is to deploy a cloud infrastructure able to run ONAP in order to design and deploy a virtual firewall, thus proving the clear advantages of virtualization over legacy solutions and machines.

## 1.2. Software platforms and solutions

In this section, the different software platforms and solutions used in the project are briefly elaborated in order to have an overall idea of what they do and why they are required.

### 1.2.1. OpenStack [4]-[7]

OpenStack is a free and open-source software platform for cloud computing, consists of interconnected components that manage different, multi-vendor, hardware pools of processing, storage, and networking resources throughout the data center.

Users can manage it through a web-based dashboard, through command-line tools, or through web applications. They can also deploy virtual machines and other instances that handle different tasks for managing a cloud environment.

Additionally, it makes horizontal scaling simple, which means that tasks that take advantage of running simultaneously can easily serve more or fewer customers in real time by just starting more instances.

And since OpenStack is an open source software, it allows changing the source code in order to tailor the solution for the needs of the user.

Typically, the infrastructure runs a platform where a developer can create software applications that are then delivered to the end users, which is represented in the Figure 1.1.

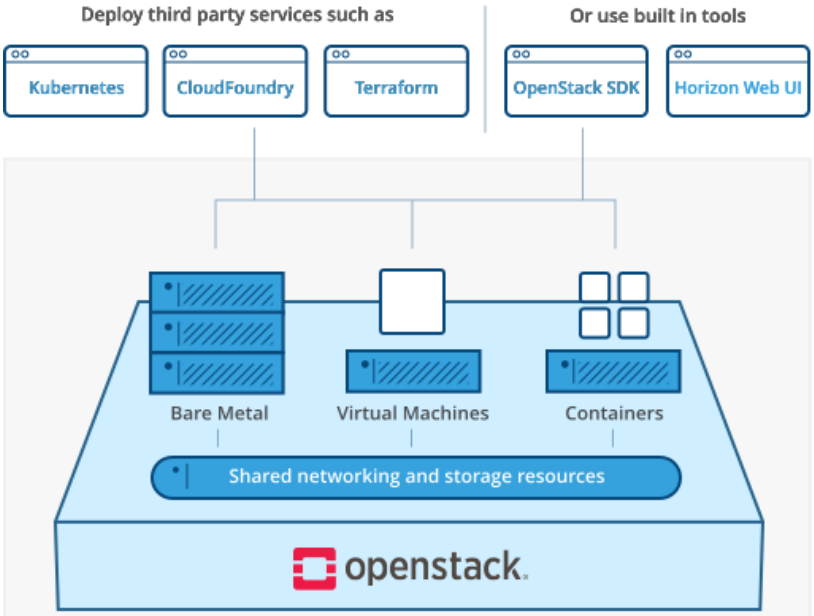


**Fig. 1.1** OpenStack basic environment

OpenStack is composed of many different moving parts called components.

There are many components, but the OpenStack community has identified nine key components that are now part of the core of OpenStack, more information on such components can be found in Annex II.

These components are distributed as a part of an OpenStack system and are maintained by the OpenStack community.

The core components can be observed as the ones in bold in Figure 1.2.

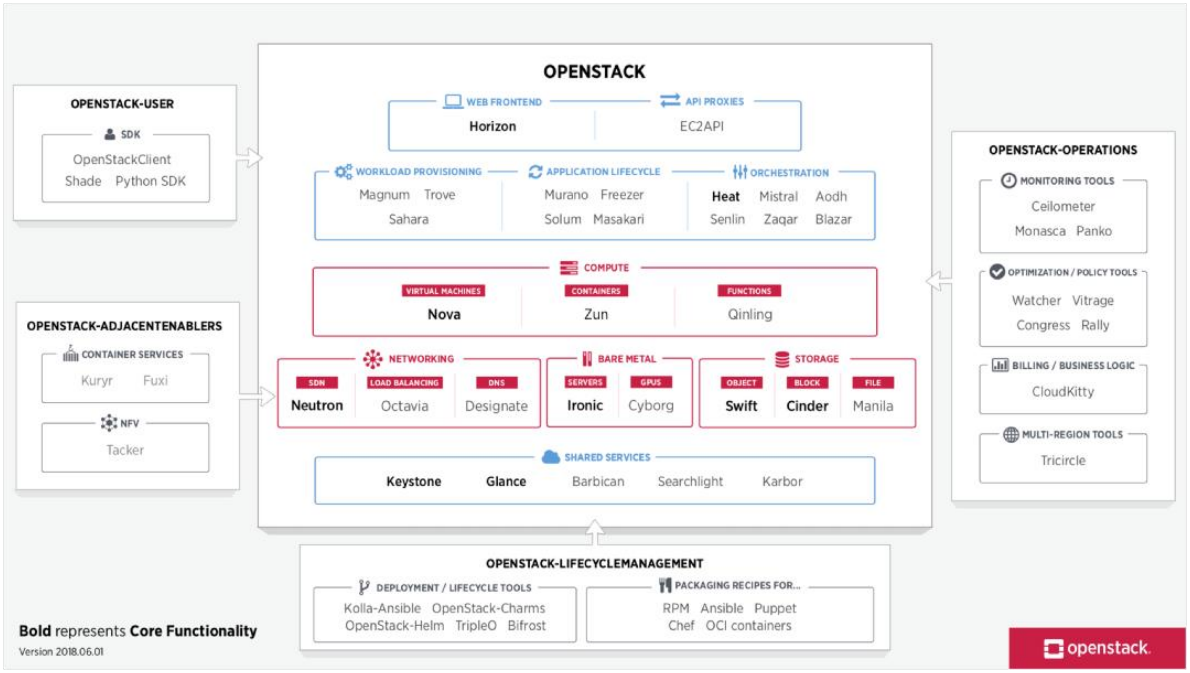


**Fig. 1.2** OpenStack component map

### 1.2.2. Ansible [8]-[11]

Ansible is an IT automation engine that automates cloud provisioning, configuration management, application deployment and intra-service orchestration.

Ansible models the IT infrastructure by describing how all the systems interact with each other rather than just managing one system at a time.

Unlike Puppet or Chef, other very famous engines, it doesn't use agents nor additional custom security infrastructure on the remote host. In its place, Ansible uses SSH, present on most IT systems nowadays.

Moreover, since it's written in Python, it's not needed to setup a client server environment before using Ansible, making it possible to run it from any machine. From the clients' point of view, there is no knowledge of any Ansible server.

Ansible works by connecting to the remote nodes and pushing out small programs, called Ansible modules, to them. These programs are written to be resource models of the wanted state of the system. Then, Ansible executes these modules and removes them when finished.

This library of modules can be installed on any machine without having to rely on external servers, daemons, or databases, making Ansible very flexible and light-weight.

### 1.2.3. Terraform [12]-[13]

Terraform is a simple tool for safely building, changing, and managing virtualized infrastructure. Terraform can control both widely used service providers like Amazon Web Services (AWS) as well as custom solutions.

Terraform needs configuration files in order to deploy and destroy the components needed to run a single application or an entire datacenter.

From these files, Terraform generates an execution plan that describes what it will do in order to reach the chosen state, and then executes it to build the planned virtual infrastructure. When the configuration files change, Terraform determines which the changes are and generates incremental execution plans to be applied afterward.

Terraform can manage low-level components such as compute instances, storage, and networking, as well as high-level components like DNS entries and SaaS features.

Basically, Terraform is a provisioning tool that enables Infrastructure as Code (IaC), which means using declarative definitions that outline the infrastructure instead of having to manually plan it and then deploy it.

IaC allows automating slow deployments that would take hours, perhaps even days, to be performed. Moreover, it increases the reliability since it plans the infrastructure to be deployed in advance, simplifying the detection of errors even before they happen.

Finally, one of the most important features of Terraform is that it enables experimentation.

Since deploying and destroying virtual instances is extremely simple with Terraform, experimental approaches can be tested, and if the changes work as intended, they can easily be scaled up to be applied in production scenarios.

### 1.2.4. Docker [14]-[17]

As it has been above-mentioned, Docker is an open-source computer program that performs operating-system-level virtualization, broadly used to create and run software packages called containers.

In Docker, these containers are isolated from each other and have their own application tools, libraries, and configuration files. However, even if they are isolated, they are able to communicate with each other through well-defined channels.

Using containers simplifies the creation of highly distributed systems by letting multiple applications, worker tasks, and other processes to run autonomously on a single physical machine or across multiple virtual machines.

These Docker containers are created from images. An image is a file comprised of multiple layers used to execute code in a Docker container that specify the precise contents of each container.

Each image has one readable and writable top layer over static layers. Layers are added to the base image to adapt the code to run in a container which may diverge depending on the application needed.

All in all, Docker serves as an efficient lightweight alternative to full machine virtualization provided by traditional hypervisors.

It is mainly used for automating the deployment of applications as portable, self-sufficient containers that can run practically anywhere on any type of server, granting the ability to build, manage and secure applications without the fear of technology or infrastructure lock-in.

Moreover, it helps to manage different applications, clouds and infrastructure at the same time it provides an easy way to update the whole process.

Docker is developed for GNU/Linux, where it uses the resource isolation features of the Linux kernel to allow independent containers to run within a single instance, avoiding the overhead of starting and maintaining virtual machines.

The Linux kernel's support for namespaces mostly isolates an application's view of the operating system and limit the resources for memory and CPU.

### 1.2.5. Kubernetes [18]-[21]

As it has already been explained, Docker containers have had a deep impact on the way applications are developed, deployed, and maintained.

Taking advantage of the native isolation capabilities of operating systems, containers support VM-like separation of tasks, but with less overhead and more flexibility than fully virtualized virtual machines.

These containers have created new paradigms regarding the application architectures, in which the different services that constitute an application are run in separate containers, and deployed across a cluster of physical or virtual machines.

Manually managing thousands of hundreds of containers is impossible, the only way to do this in a feasible way is with a container orchestrator, such as Kubernetes.

Kubernetes, or K8s, is a portable, extensible open-source container orchestration system for automating the deployment, scaling, and management of containerized applications and services.

Since Kubernetes is open source with relatively few restrictions on how it can be used, it can be implemented by anyone who wants to run and manage thousands of containers.

It is very important to highlight that Kubernetes does not replace Docker, but expands it. However, Kubernetes does replace some of the higher-level technologies that have emerged around Docker.

Kubernetes is mainly used in real production apps with multiple containers.

Those containers must be deployed across multiple server hosts, and multilayered security for containers can be complicated.

In this case, Kubernetes allows building application services that spawn multiple containers, schedule those containers across a cluster, scale those containers, and manage the health of those containers over time.

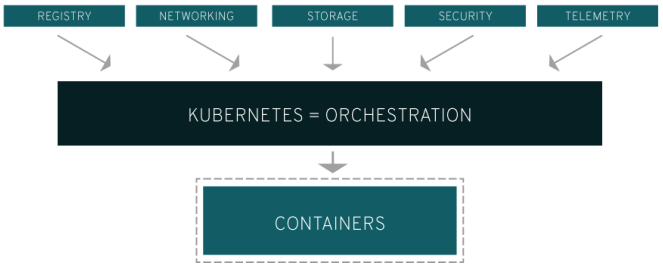


**Fig. 1.3** Kubernetes simplified diagram

Kubernetes also needs to integrate with networking, storage, security, telemetry, and other services to provide a comprehensive container infrastructure. The diagram with the main services kubernetes integrates can be seen in Figure 1.3.

Moreover, Kubernetes fixes a lot of common problems with container proliferation, which means sorting containers together into a “pod”, basic within the Kubernetes implementation.

Pods add a layer of abstraction to grouped containers, which simplifies schedule workloads and provides necessary services like networking and storage to those containers.

Other parts of Kubernetes help to load balance across these pods and ensure that the network has the right number of containers running to support the workloads. The main elements of Kubernetes in a regular scenario can be seen in Figure 1.4.

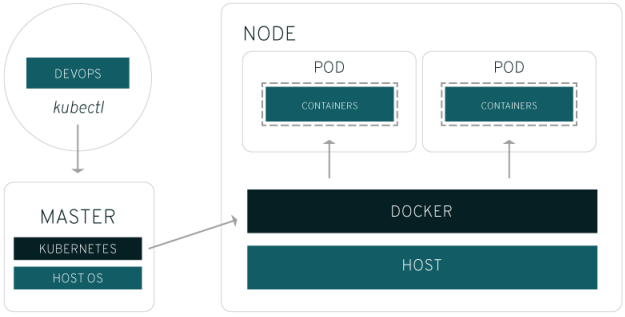


**Fig. 1.4** Kubernetes basic environment setup diagram

As in most technologies, Kubernetes has its own terms and terminology, the most important being the following:

- Master: It is the machine that controls the Kubernetes nodes. This is where all task assignments are created.
- Node: It is a group of machines that perform the requested and assigned tasks. The master controls them.
- Pod: It’s a group of one or more containers deployed to a single node. Pods abstract network and storage away from the underlying containers.
- Replication controller: This controls how many identical copies of a pod should be running somewhere on the cluster in order to load balance.
- Service: This decouples work definitions from the pods. Service proxies automatically get service requests to the right pod independently of its position in the cluster.
- Kubectl: This is the command line configuration tool for Kubernetes available for the user.

Kubernetes runs on virtually any operating system and hosted environment and interacts with pods of containers running on the nodes.

As it can be seen in Figure 1.4, the master takes the commands from an administrator and communicates those instructions to the nodes.

This handoff works with a multitude of services to automatically decide which node is best suited for the task. Then, it allocates resources and assigns the pods in that node to fulfill the requested work.

From an infrastructure point of view, there is little change to how the containers are managed. The control over those containers happens at a higher level, improving the control without the need to micromanage each separate container or node.

### 1.2.6. Rancher [22]-[24]

Rancher is an open source software platform that enables organizations to run and manage containers. It solves the operational and security issues of managing multiple Kubernetes clusters, while at the same time it provides a unified control plane.

Rancher offers a GUI to present complex features such as load balancing in an easy and understandable way.

However, Rancher has more advantages, which are the following:

- Infrastructure management: Nodes can be easily provisioned using the GUI.

- Infrastructure orchestration: Rancher uses raw computing resources from any public or private cloud. The hosts in the cloud can be a virtual machine or physical machine.

- Container scheduling/orchestration: Can be performed using Cattle, Rancher's own orchestrator, but it is now taken care by other orchestrators such as Kubernetes.

- Multiple environments: Rancher can provision and use multiple environments at the same time with one unified control plane.

- Monitoring, health-checks and logging: All of these functions are combined to improve its performance.

All of the previous points can be summarized in the Figure 1.5.

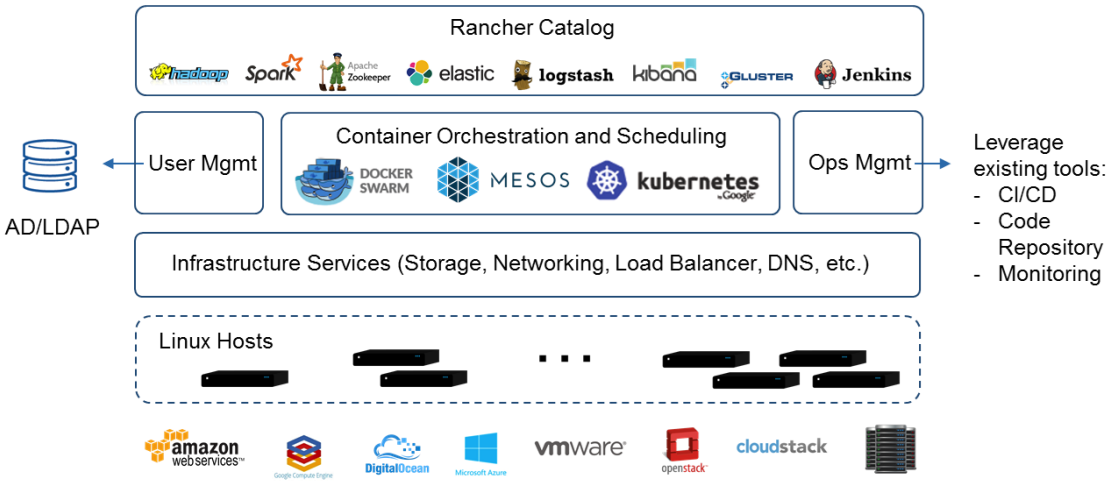


**Fig. 1.5** Rancher architecture

In addition to what has previously been exposed, Rancher implements a layer of infrastructure services designed to empower containerized applications.

Rancher's services include networking, storage, load balancer, DNS, and security. These services are usually deployed as containers so that Rancher can run on any cloud.

### 1.2.7. Helm [25]-[27]

Helm is a package manager application running on top of Kubernetes that allows defining the structure through "helm-charts", a collection of files that describe a related set of Kubernetes resources and managing it with simple commands.

Helm is very important because it allows deploying a massive number of microservices with ease. Additionally, it also simplifies their management. These microservices can be managed, updated and scaled individually, unlike a monolithic application.

In addition, issues with one microservice do not affect the functionality of the rest of components and new applications can easily be created using existing microservices.

Basically, the main advantage is the ability to accept scalability from the beginning.

Launching applications that require many microservices can easily be done using Helm and its charts.

### 1.2.8. ONAP [28]-[31]

ONAP, which stands for Open Network Automation Platform, is an open source networking project hosted by the Linux Foundation that brings the capabilities for designing, creating, orchestrating and handling the full lifecycle management of VNF, Software Defined Networking (SDN), and all of the services related to these.

In short, ONAP is the platform above the physical infrastructure layer that automates the network.

ONAP allows connecting products and services through the physical infrastructure, deploy VNFs and scale the network in a fully automated way.


**Fig. 1.6** ONAP logotype

The high-level architecture of ONAP has many different software subsystems that are part of the design environment as well as part of an execution time environment that runs what is designed in the previous environment.

Since ONAP enables VNFs, other network functions, and services to be interoperable in an automated, policy-driven, real-time environment, this allows the ability to create, design and deploy automated network services anywhere.

ONAP has the goal to bring the DevOps best practices and dynamic AGILE deployment methods to the Telecom world. This will reduce the barriers for new users to enter the world of cloud technologies and network virtualization.

It will also enable operators to quickly add new features, deploy them in real time, reduce operational costs and have much more control over the network and the services that are available. This way, the network will be able to work in a much more efficient manner and consumers will benefit from the new services and the improved experience.

As a cloud-native application based in numerous services, ONAP requires a sophisticated initial deployment as well as some post-deployment management.

The ONAP Operations Manager (OOM) is responsible for orchestrating the lifecycle management and monitoring of ONAP components. It is integrated with the Microservices Bus, which provides service registration/discovery and support for internal and external APIs and SDKs.

OOM uses Kubernetes to provide CPU efficiency and platform deployment. Its architecture and the main operations OOM can perform can be seen in Figure 1.7.

In addition, OOM enhances the ONAP platform by providing scalability and resiliency enhancements to the components it manages.

The platform on itself provides tools for service designing as well as a model-driven, run-time environment, with monitoring and analytics to support automation and service optimization.

Both design-time and run-time environments are accessed through the Portal Framework, with role-based access for service designers and operations.

The design-time framework provides a development environment with tools, techniques, and repositories for defining and describing resources, services, and other products that include policy design and implementation, SDK with tools for VNF packaging and validation.

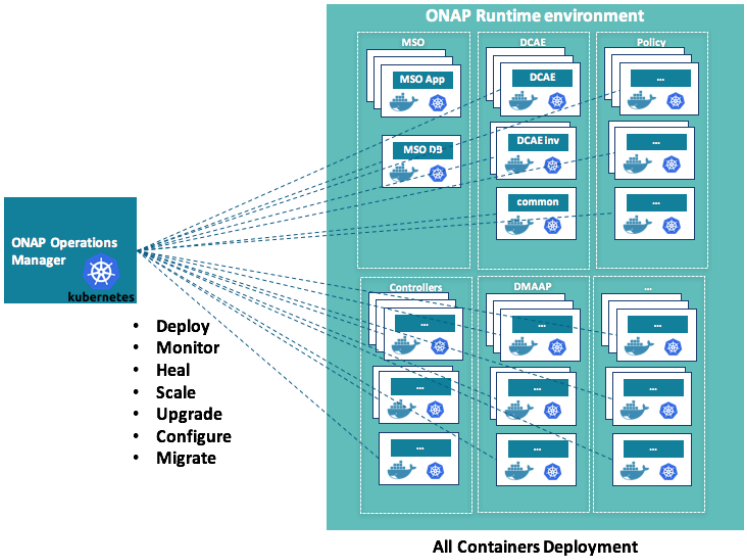


**Fig. 1.7** OOM architecture

The run-time environment applies the rules and policies distributed by the design and creation environment, as well as the controllers that manage physical and virtual networks.

The run-time service execution components are in constant communication with the automation modules, which provide real-time monitoring, analytics, alarms, and event correlation.

The complete ONAP architecture can be observed in the Figure 1.8.

The ONAP modules provide continuously updated data about existing services to service designers, who will deem if it is necessary to tune them, replicate high-performing portions of deployed services or directly creating new ones.

ONAP has a very long list of modules, however, not all of them are needed to implement the minimal functionality required to deploy virtualized network functions.

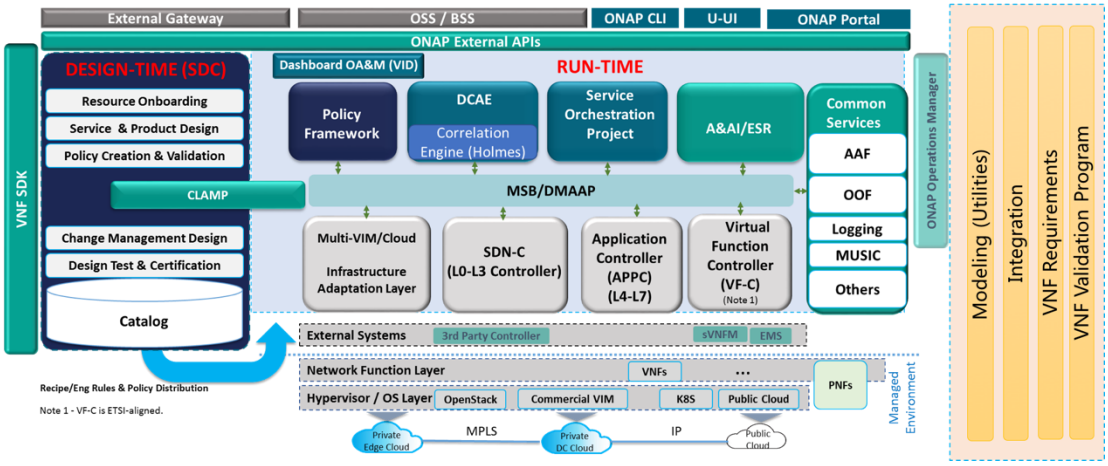


**Fig. 1.8** ONAP's architecture

# CHAPTER 2. CLOUD ENVIRONMENT DEPLOYMENT

As it has been abovementioned, ONAP requires a cloud infrastructure in order to work. This cloud infrastructure is provided by OpenStack using Kubernetes. However, we had to install the cloud infrastructure from scratch.

In this chapter we explain the network setup and installation, which took around 2 weeks, the virtual disk creation, which took around 1 day, and OpenStack's deployment, which took approximately a month.

## 2.1. Network setup and installation

The network is composed of the following devices, which can be seen in Figure 2.1. A complete description of all the hardware characteristics can be seen in Annex III.

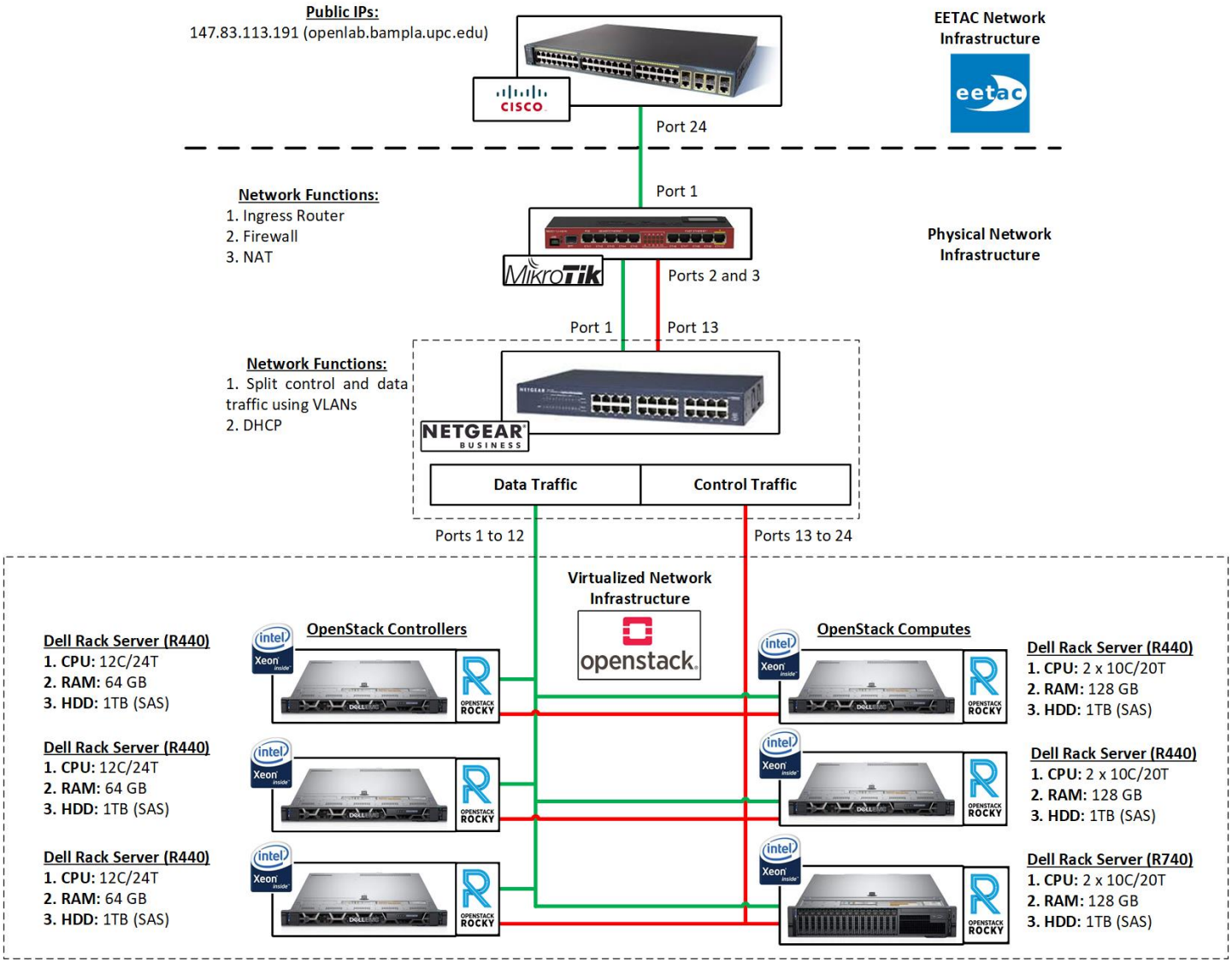


**Fig. 2.1** Cloud infrastructure and physical network

The list of elements of the private cloud is the following:

- An UPC router: It will offer a public IP for the scenario and allow users to remotely connect to the cloud.

- MikroTik router: It will use the UPC public IP provided by the UPC router. It will act as the firewall and NAT of the cloud.

- Category 6 Ethernet cables: These cables allow 10 gigabit Ethernet connections.

- 2 VLANs: One VLAN is for control (red wire with tag 101) and the other one is for data (green wire with tag 102)

- Netgear switch: It splits the control traffic and data traffic, which will be allocated in two different VLANs. It will also perform DHCP tasks.

- 3 OpenStack controllers: The ONAP controllers consist of 3 powerful Dell Rack Server R440. These servers are connected to the switch and to the OpenStack computes at the same time. These servers will generate the control traffic.

- 3 OpenStack computes: The ONAP computes consist of 2 powerful Dell Rack Server R440 and one heavy-duty Dell Rack Server R740. These servers are connected to OpenStack controllers through the switch. These servers will generate data traffic.

This environment has two networks, the outer network and the inner network.

The outer network belongs to the UPC, and therefore, we have no control over it.

The inner network is split using two VLANs, one dedicated to control and another one dedicated to data.

Each VLAN has its own range of IPs, which is 192.168.100.0/24 for data and 172.16.100.0/24 for control.

The IP addresses 192.168.100.1 and 172.16.100.1 are reserved for the router. The rest of IPs are assigned to the servers in an incremental order starting in 192.168.100.10 and 172.16.100.10 respectively.

## 2.2. Virtual disk creation and operative system installation

When we set up the environment, we proceeded to create the virtual disks in the HDD disks.

In order to do so, we used Dell's firmware to create a recognizable virtual disk inside the HDD disk, and then, we installed the operative system in each server.

The operative system we have selected is Ubuntu server 18.04.02 LTS because it supports Kubernetes for process-container coordination, management, and scalability.

During the installation, we had to configure the network interfaces and the SSH clients of each server.

## 2.3. OpenStack deployment

As it has already been mentioned, ONAP is a cloud-native application, and because of that, it needs a cloud in order to be deployed.

There are several different ways to create this private cloud, but we have decided to use OpenStack due to its characteristics and open-source nature.

The version of OpenStack we have used has been Rocky because it was the stable release when we started with the installation. Now, there is a new version called Stein, but since updating OpenStack is pretty forthright this procedure can be performed anytime.

Instead of manually setting up OpenStack and its virtual machines, we have used Ansible. With Ansible, configuring and deploying the whole scenario in an automated way has been straightforward, saving us hours of manual labor and granting us the possibility to replicate the setting elsewhere.

Moreover, thanks to Ansible, we didn't have to change the GitHub code for installing OpenStack. We just had to modify Ansible's inventory file, known as inventory host file, to point to our private servers. [31]

Once the file had been modified, we did the following steps.

### 2.3.1. Prepare deployment host

Before we could start with the OpenStack installation, we had to update the operative system in the deployment host, the server that will install OpenStack to the rest of machines. Then, we created the SSH keys for Ansible and downloaded the OpenStack GitHub repository.

### 2.3.2. Prepare target hosts

Like in the previous step, we updated the operative system, configured the SSH keys for Ansible, the storage volumes, and the virtual network interfaces.

### 2.3.3. Configure deployment

Ansible references some files that contain mandatory and optional configuration directives, so before we could run the Ansible playbooks, we had to modify these files.

In total, we have modified three files, which are the following:

- OpenStack_user_config.yml: File with the configuration of the different users, machines that comprise the cloud and their IP addresses.
- User_secrets.yml: File with the configuration of the passwords and keys of the different users.
- User_variables: File with the different variables that users will need in order to access OpenStack.

To sum up, we have modified the target host networking to define bridge interfaces and networks, the list of target hosts on which to install OpenStack, the virtual and physical network relationships for OpenStack Networking (Neutron) and the passwords for all the different services.

It’s important to highlight the importance of handling the credentials because Ansible playbooks do not manage changing passwords in an existing environment. Changing passwords and rerunning the playbooks will fail and might destabilize the OpenStack environment.

### 2.3.4. Run playbooks

Once we finished the configuration, and after we checked the integrity of the files, the deployment host ran the playbooks in the machines, thus installing OpenStack.

We didn’t encounter any errors nor remarkable problems during this installation.

### 2.3.5. Verify OpenStack's installation

When the installation process finished, we verified its correct operation checking the API offered by OpenStack, listing the containers running and also confirming that the OpenStack's dashboard had been deployed.

Figure 2.2, which shows Horizon's login screen, proves OpenStack's correct operation.

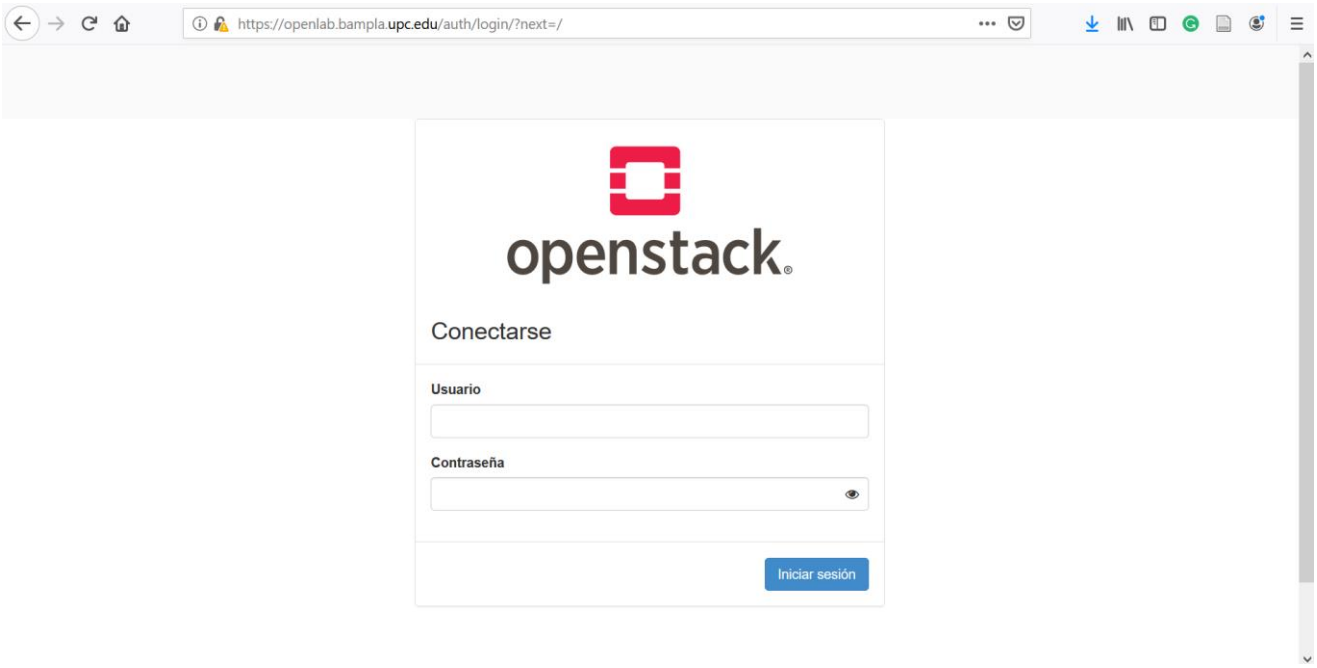


**Fig. 2.2** OpenStack Horizon login screen

For further information regarding OpenStack's installation, you can find more information in [31].

### 2.3.6. Certificate installation to OpenStack

By default, OpenStack creates SSL certificates that cannot be trusted by third-parties because they are self-signed.

Since ONAP requires trustworthy certificates in order to accept incoming HTTP connections, we have emitted a valid certificate using Let's Encrypt, a non-profit authority run by the Internet Security Research Group [32].

To install this certificate, Let's Encrypt first validated the OpenStack server, then we enabled the web root for the certification bot, generated the certificates with the bot, copied the files to the host and finally applied these certificates in the proxy [32].

Figure 2.3 proves that the Let's Encrypt certificate works correctly since our connection to the dashboard is secure.

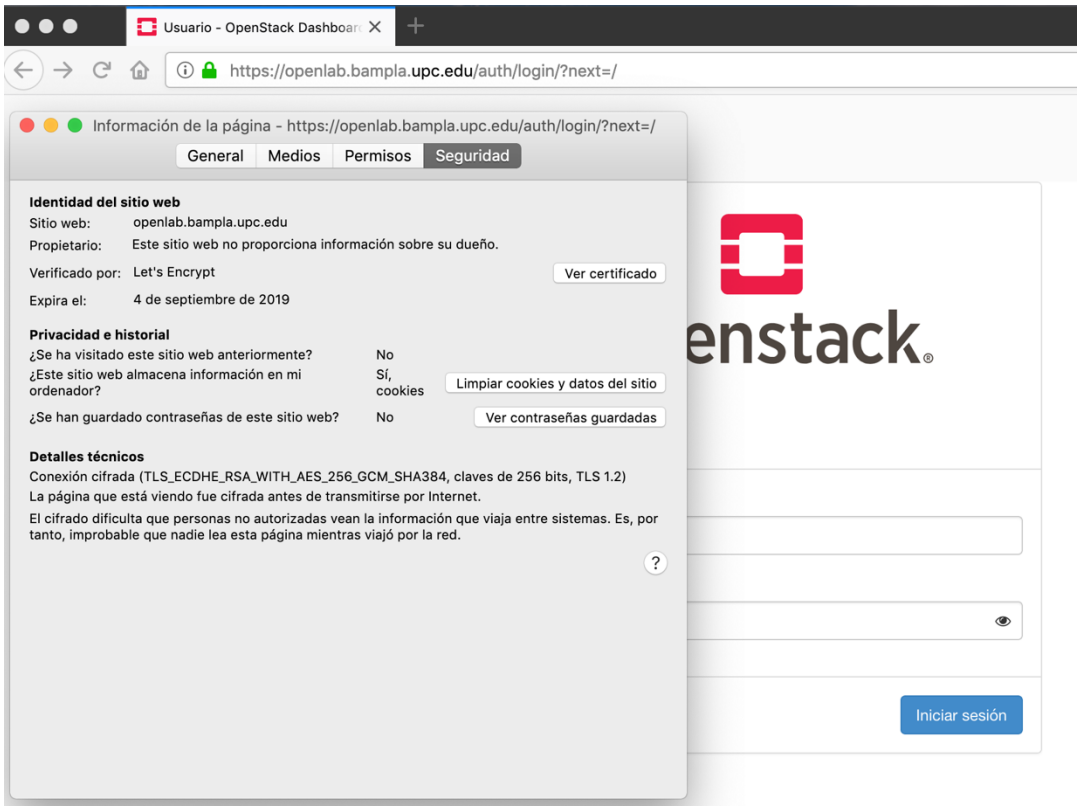


**Fig. 2.3** OpenStack Horizon secure login screen

# CHAPTER 3. OOM INSTALLATION

Once we had OpenStack up and running, the next step was to install OOM, the ONAP Operations Manager.

As it has been abovementioned, OOM is essentially ONAP's cloud, required in order to manage the whole ONAP's life-cycle. However, OOM does not manage the services, VNFs nor the virtual infrastructure [30].

OOM's deployment has been performed using Kubernetes and Rancher following these steps [34].

## 3.1. Virtual machine configuration

Since there were known deployment issues in the Casablanca release (April 15, 2019), making the deployment nearly impossible due to its unsolved problems, we have adapted the Dublin OOM release (June 6, 2019) deployment to Casablanca, which has different requirements and installation techniques, such as the usage of the Rancher Kubernetes Engine (RKE) that solved the aforesaid problems.

First of all, a key pair is required to access the created OpenStack VMs. Also, this key will be used by RKE to configure the VMs for Kubernetes.

OOM needs a certain number of virtual machines in order to be deployed, which are the following [34]:

- Kubernetes workers: OOM needs 12 virtual machines with the m1.xlarge flavor (16 GB of RAM, 8vCPU, and 100GB of storage). These machines are the computes of the ONAP cloud.

- Kubernetes control plane: OOM needs 3 virtual machines with the m1.large flavor (8 GB of RAM, 4vCPU and 50GB of storage). These machines control the Kubernetes workers

- Network File System server: OOM requires only one virtual machine with this role, which also has an m1.large flavor. It allows remotely updating and storing files in the cloud.

- RKE server: OOM requires only one virtual machine with this role, which also has an m1.large flavor. This machine hosts the RKE server, which allows monitoring the status and health of the Kubernetes pods.

An example of worker and controller can be seen in Figure 3.1.

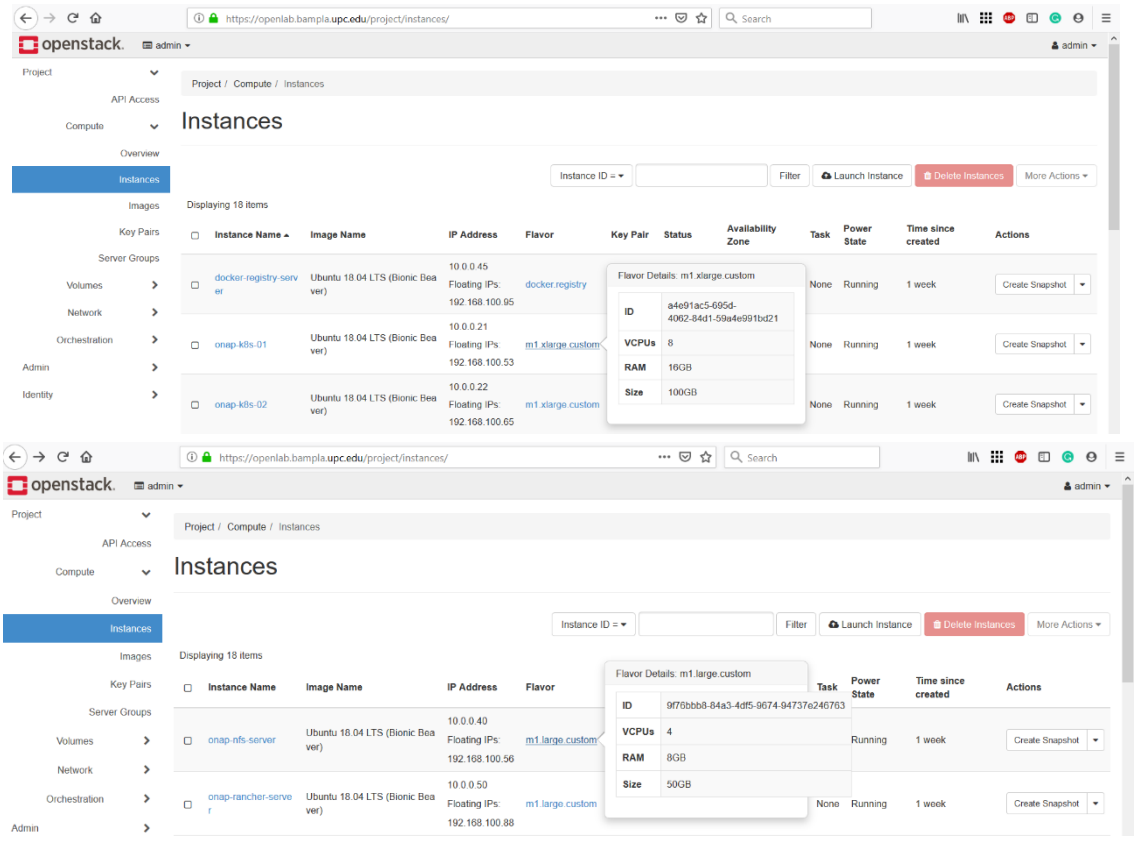


**Fig. 3.1** Worker and controller as seen in OpenStack

In order to do so, we first created the 3 OpenStack virtual machines to host the Highly-Available Kubernetes control plane. It's important to highlight that ONAP workloads are not scheduled on these Control Plane nodes.

Then, we created the 12 OpenStack virtual machines to host the HA Kubernetes workers. ONAP workloads will only be scheduled on these nodes.

By splitting the VMs in control plane machines and worker machines, we reduce the overall processing power required in the workers so more resources can be dedicated to ONAP rather than the management of the machine, improving, in the end, ONAP's performance [34].

Finally, we created the NFS server and the RKE server.

Nevertheless, since many pods are sensitive to synchronization and latency, we deployed an extra virtual machine with 4vCPUs, 4GB of RAM and 250 GB of storage to work as a Docker registry to store all the Docker images in a local repository in order to avoid synchronization issues and speed up the installation process.

To get all the images necessary for a fresh installation, we used a script which made a pre-pull of all the required Docker images of the different ONAP modules. The total amount of storage used by these images is around 100GB

Thanks to the Docker registry addition, we managed to reduce the installation time around a 60%.

The resulting architecture can be seen in Figure 3.2.

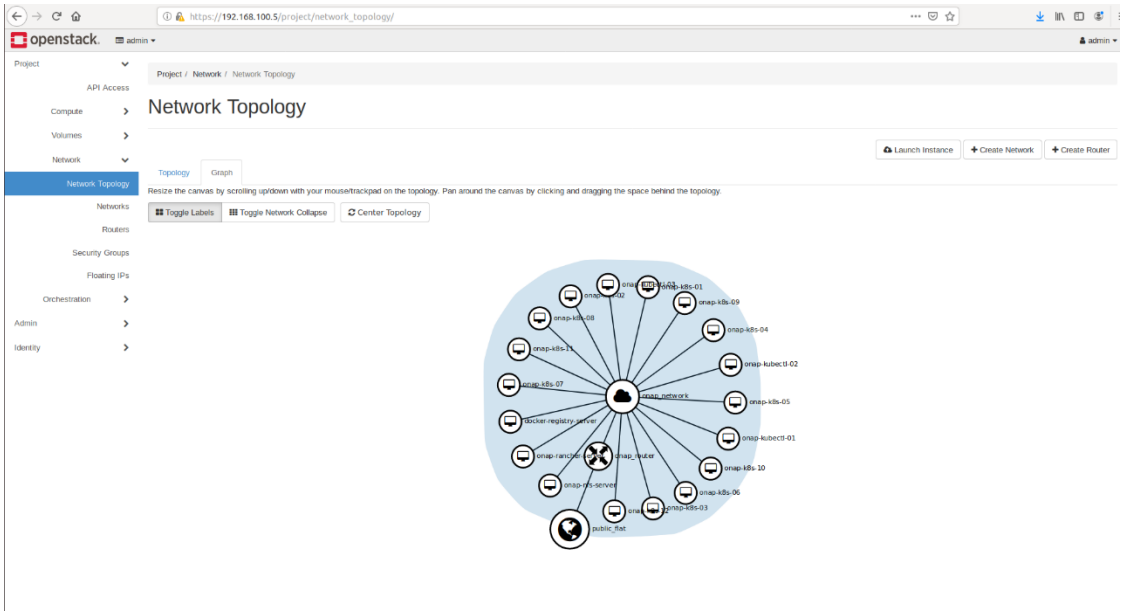


**Fig. 3.2** Private cloud architecture

Additionally, in order to automate this installation process and to be able to reproduce this scenario anywhere, all of these machines have been created and configured in OpenStack with a custom Terraform script.

This Terraform script has been developed by us in order to quicken the deployment procedures of the virtual machines, a very useful feature not contemplated by ONAP.

Thanks to this Terraform script we created, the cluster can be created and fully provisions with the proper configuration in just 10 minutes instead of the about an hour manual procedure we would have done if we had followed the ONAP's guide.

The most important parts of the script are explained in the following points.

It is worth mentioning that the script is based on variables, which simplifies the process of replication.

### 3.1.1. Configure credentials for OpenStack

This part of the script configures all the credentials needed in order to be able to use OpenStack from Terraform. It's the most essential step and must be done before anything else.

```
provider "openstack" {
  user_name     = "${var.openstack_user_name}"
  tenant_name   = "${var.openstack_tenant_name}"
  password      = "${var.openstack_password}"
  auth_url      = "${var.openstack_auth_url}"
  region        = "${var.openstack_region}"
  endpoint_type = "${var.openstack_endpoint_type}"
}
```

### 3.1.2. Create SSH key pair

The SSH key pair is compulsory to create the virtual machines. Without it, OpenStack is not able to create the machines.

A new key pair is needed to recreate the scenario somewhere else. They cannot be reused.

```
resource "openstack_compute_keypair_v2" "onap_key" {
  name = "onap_key"
}
```

### 3.1.3. Create security group

The OpenStack security group will enable the ICMP and SSH traffic in the virtual machines.

All the IPv4 and IPv6 ICMP ports are open so the correct functioning of the nodes can be ensured using ICMP packets.

Additionally, the rest of TCP and UDP packets are also allowed in all ports, both in IPv4 and IPv6.

```
resource "openstack_compute_secgroup_v2" "onap_secgroup" {
 name        = "onap_secgroup"
```

```
# IPv4 rules
rule {
  ip_protocol = "icmp"
  from_port   = -1
  to_port     = -1
  cidr        = "0.0.0.0/0"
}

rule {
    ip_protocol = "tcp"
    from_port   = 1
    to_port     = 65535
    cidr        = "0.0.0.0/0"
}

rule {
    ip_protocol = "udp"
    from_port   = 1
    to_port     = 65535
    cidr        = "0.0.0.0/0"
}
```

```
  # IPv6 rules
  rule {
    from_port   = -1
    to_port     = -1
    ip_protocol = "icmp"
    cidr        = "::/0"
  }

  rule {
      ip_protocol = "tcp"
      from_port   = 1
      to_port     = 65535
      cidr        = "::/0"
  }

  rule {
      ip_protocol = "udp"
      from_port   = 1
      to_port     = 65535
      cidr        = "::/0"
  }
}
```

### 3.1.4. Create Kubernetes Control Plane VMs

Using the previous settings, we proceeded to create the Kubernetes control plane virtual machines.

```
resource "openstack_compute_instance_v2" "kubectl_vm" {
  count             = "${var.kubernetes_control_plane_VMs}"
  name              = "${format("onap-kubectl-%02d", count.index + 1)}"
  key_pair          = "${openstack_compute_keypair_v2.onap_key.name}"
  user_data         = "${file("scripts/openstack-k8s-controlnode.sh")}"
  image_name        = "${var.openstack_image_bionic64}"
  flavor_name       = "${var.openstack_flavor_large}"
  security_groups   = ["${openstack_compute_secgroup_v2.onap_secgroup.id}"]

  network {
    port            = "${openstack_networking_port_v2.kubectl_port.*.id[count.index]}"
    access_network = true
  }
}
```

### 3.1.5. Attach floating IP to the control plane virtual machines

Once created, it's very important to assign each machine a different floating IP to allow external access. Otherwise, without a floating IP, the machines will be isolated inside the cloud.

```
resource "openstack_compute_floatingip_associate_v2" "kubectl_vm" {
  count            = "${var.kubernetes_control_plane_VMs}"
  floating_ip      = "${openstack_networking_floatingip_v2.kubectl_floating_ip.*.address[count.index]}"
  instance_id      = "${openstack_compute_instance_v2.kubectl_vm.*.id[count.index]}"
}
```

### 3.1.6. Create Kubernetes Worker VMs

Using the previous settings, we proceeded to create the Kubernetes worker virtual machines.

```
resource "openstack_compute_instance_v2" "kubectl_vm" {
  count            = "${var.kubernetes_control_plane_VMs}"
  name             = "${format("onap-kubectl-%02d", count.index + 1)}"
  key_pair         = "${openstack_compute_keypair_v2.onap_key.name}"
  user_data        = "${file("scripts/openstack-k8s-controlnode.sh")}"
  image_name       = "${var.openstack_image_bionic64}"
  flavor_name      = "${var.openstack_flavor_large}"
  security_groups  = ["${openstack_compute_secgroup_v2.onap_secgroup.id}"]

  network {
    port           = "${openstack_networking_port_v2.kubectl_port.*.id[count.index]}"
    access_network = true
  }
}
```

### 3.1.7. Attach floating IP to the Kubernetes worker virtual machines

Just like the control plane machines, it's very important to assign floating IPs to allow external access.

```
resource "openstack_compute_floatingip_associate_v2" "kubectl_vm" {
  count            = "${var.kubernetes_control_plane_VMs}"
  floating_ip      = "${openstack_networking_floatingip_v2.kubectl_floating_ip.*.address[count.index]}"
  instance_id      = "${openstack_compute_instance_v2.kubectl_vm.*.id[count.index]}"
}
```

### 3.1.8. Create Rancher Kubernetes Engine (RKE) VM

With both the control plane and worker machines deployed, we created the RKE server.

```
resource "openstack_compute_instance_v2" "rancher_vm" {
  count             = "${var.rancher_kubernetes_engine_VM}"
  name              = "onap-rancher-server"
  key_pair          = "${openstack_compute_keypair_v2.onap_key.name}"
  user_data         = "${file("scripts/openstack-ranchernode.sh")}"
  image_name        = "${var.openstack_image_bionic64}"
  flavor_name       = "${var.openstack_flavor_large}"
  security_groups   = ["${openstack_compute_secgroup_v2.onap_secgroup.id}"]

  network {
    port            = "${openstack_networking_port_v2.rancher_port.*.id[count.index]}"
    access_network = true
  }
}
```

The RKE is a light-weight Kubernetes installer that supports installation on bare-metal and virtualized servers [35].

With RKE, Kubernetes installation is simplified, regardless of the operating system and platform, that is to say, it can run anywhere, even outside the cluster [35].

Nevertheless, this is highly inadvisable because of the number of rules that would have to be applied both in the switch and the router in order to let the RKE server connect to the cluster.

```
nodes:
- address: 192.168.100.68
  port: "22"
  internal_address: 10.0.0.11
  role:
  - controlplane
  - etcd
  hostname_override: "onap-kubectl-01"
  user: ubuntu
  ssh_key_path: "~/.ssh/onap-key"
- address: 192.168.100.82
  port: "22"
  internal_address: 10.0.0.12
  role:
  - controlplane
  - etcd
  hostname_override: "onap-kubectl-02"
  user: ubuntu
  ssh_key_path: "~/.ssh/onap-key"
- address: 192.168.100.56
  port: "22"
  internal_address: 10.0.0.13
  role:
  - controlplane
  - etcd
  hostname_override: "onap-kubectl-03"
  user: ubuntu
  ssh_key_path: "~/.ssh/onap-key"
- address: 192.168.100.67
  port: "22"
  internal_address: 10.0.0.21
  role:
  - worker
  hostname_override: "onap-k8s-01"
  user: ubuntu
  ssh_key_path: "~/.ssh/onap-key"
```

Because of that, we have created the RKE server inside the cluster. Moreover, we will use this server as access point from where we will run the rest of scripts. The RKE works with an additional file called cluster.yml, in which all the different Kubernetes machines are configured.

As it can be seen, we specify the private IP (inside the cloud), the internal IP (inside the cluster), the port, which is the SSH port, the name of the machine, the user to access the machine, the role of the machine and the SSH key to be used.

Unlike the Terraform scripts, the cluster.yml does not work with variables, so manual tweaks and changes would be required in order to use this file in another environment.

### 3.1.9. Attach floating IP to the RKE server

Since the RKE server is the access point to the cluster, a floating IP is required to remotely connect to it.

```
resource "openstack_compute_floatingip_associate_v2" "rancher_vm" {
  count            = "${var.rancher_kubernetes_engine_VM}"
  floating_ip      = "${data.openstack_networking_floatingip_v2.rancher_floating_ip.address}"
  instance_id      = "${openstack_compute_instance_v2.rancher_vm.*.id[count.index]}"
}
```

### 3.1.10. Setting up an NFS

Finally, we created the NFS server and made it visible to the whole private cloud, a very important step since this server is the one responsible of sharing the storage amongst the Kubernetes clusters.

This server is the NFS Master and the Kubernetes worker machines are the NFS Slaves.

However, this Master is outside of the Kubernetes cluster because this ensures that it will not compete for resources with the Kubernetes control plane machines nor the worker machines.

```
resource "openstack_compute_instance_v2" "nfs_server_vm" {
  count            = "${var.nfs_server_VM}"
  name             = "onap-nfs-server"
  key_pair         = "${openstack_compute_keypair_v2.onap_key.name}"
  user_data        = "${file("scripts/openstack-nfs-server.sh")}"
  image_name       = "${var.openstack_image_bionic64}"
  flavor_name      = "${var.openstack_flavor_large}"
  security_groups  = ["${openstack_compute_secgroup_v2.onap_secgroup.id}"]

  network {
    port           = "${openstack_networking_port_v2.nfs_server_port.*.id[count.index]}"
    access_network = true
  }
}
```

### 3.1.11. Attach floating IP to the NFS server

This server must also be accessible from the outside, so we have attached a floating IP to it.

```
resource "openstack_compute_floatingip_associate_v2" "nfs_server_vm" {
  count           = "${var.nfs_server_VM}"
  floating_ip     = "${openstack_networking_floatingip_v2.nfs_floating_ip.*.address[count.index]}"
  instance_id     = "${openstack_compute_instance_v2.nfs_server_vm.*.id[count.index]}"
}
```

It's very important to remark that since Terraform configures the machines, it also installs all the dependencies required, such as Docker, running scripts inside the virtual machine to be deployed.

## 3.2. Configure Rancher Kubernetes Engine (RKE)

When we deployed all the machines required for the OOM, we configured them and created the different relationships with the Rancher Kubernetes Engine.

As RKE is based on configuration files and scripts, we had to run a single command that fed on the cluster.yml file commented above.

Basically, the command was *rke up*, and it configured the machines with the values inside the cluster.yml file, thus creating the virtual infrastructure.

Inside this file we specified the configuration of the different nodes of the cluster, an example of one node can be seen in the following code snippet:

```
nodes:
- address: 192.168.100.67
  port: "22"
  internal_address: 10.0.0.11
  role:
  - controlplane
  - etcd
  hostname_override: "onap-kubectl-01"
  user: ubuntu
  ssh_key_path: "~/.ssh/onap-key"
```

```
network:
  plugin: canal
authentication:
  strategy: x509
ssh_key_path: "~/.ssh/onap-key"
ssh_agent_auth: false
authorization:
  mode: rbac
ignore_docker_version: false
kubernetes_version: "v1.13.5-rancher1-2"
private_registries:
- url: nexus3.onap.org:10001
  user: docker
  password: docker
  is_default: true
cluster_name: "onap"
restore:
  restore: false
  snapshot_name: ""
```

The RKE configuration scenario is represented in Figure 3.3, where a single machine, in our case, inside the cluster, configures the rest of virtual machines.

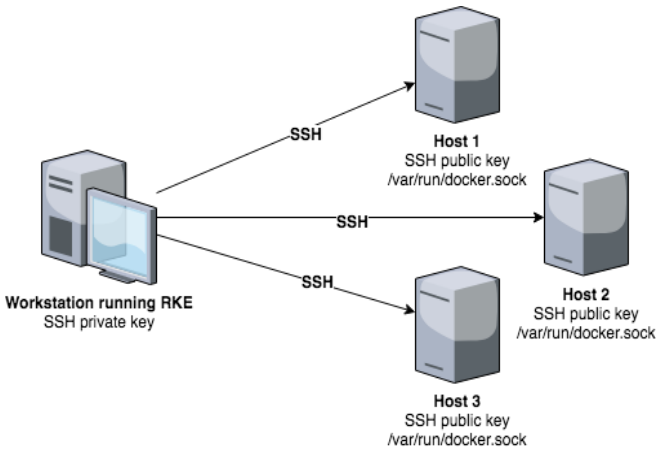


**Fig. 3.3** RKE configuration scenario

## 3.3. Rancher Graphic User Interface installation

RKE, being an engine, comes without a graphic user interface, which makes the management and monitoring of the different virtual machines tedious and inefficient since it has to be done using a console line interface.

In order to solve this problem, since the cluster is actually a Kubernetes cluster deployed by the Rancher engine, we installed a Rancher server inside the cluster and imported the existing infrastructure to it [36].

To install the Rancher server, we created another virtual machine with Docker and 4GB of RAM, and then, we ran the following command:

```
└─▪ sudo docker run -d --restart=unless-stopped -p 80:80 -p 443:443 rancher/rancher
```

After having run that command, the Rancher server got deployed along with the web graphic interface.

Within the webpage, we configured the rest of the server, its agents, and successfully imported the existing Kubernetes cluster.

To import it, the Rancher gave us *kubectl* commands that we had to run inside the RKE server to install the agents in all the cluster machines, and once the agents were installed, the Rancher server automatically fetched their health and started monitoring them [37].

The result of the import can be seen in Figure 3.4, which shows Rancher's graphic dashboard.

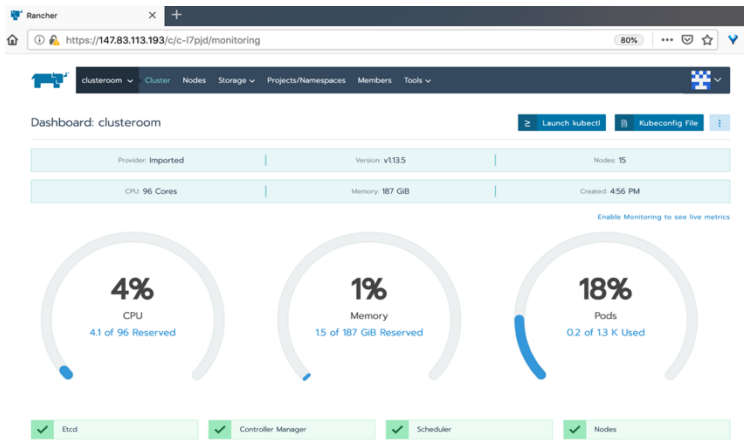


**Fig. 3.4** Rancher's graphic dashboard

Figure 3.5 shows Rancher's list of nodes. Notice that the nodes are the same as in Figure 3.2, the architecture provided by OpenStack, proving that the import has correctly been performed.

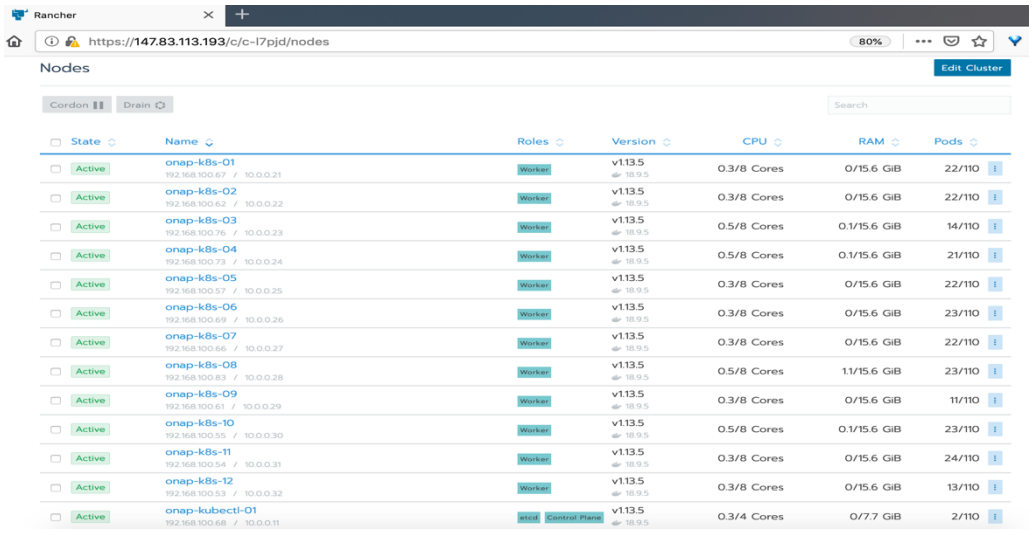


**Fig. 3.5** Rancher's list of existing nodes

Even if the cluster import has been successful, it's important to remark that modifying the Kubernetes cluster, for example, adding or removing nodes, upgrading the cluster version and changing Kubernetes component parameters, has to be done outside of Rancher in the RKE server [37].

However, the incorporation of this server to the cloud has been a very important addition because it has facilitated the control and management of the cluster, something rather unfeasible just by using the Rancher engine.

## 3.4. Kubectl and Helm installation

The last step was to install Kubectl and Helm clients.

Kubectl is required in order to apply Kubernetes commands and to be able to configure, create and manage the Kubernetes pods.

In these pods, the different ONAP modules will be deployed using some proprietary Helm charts.

It's not needed to install neither the Kubectl client nor the Helm client in all virtual machines, only on the RKE server. The rest of machines only have Kubernetes so the RKE server can communicate with them.

It is worth mentioning that the engine responsible of installing Kubernetes inside the virtual machines of the cluster is the RKE server itself.

Once the RKE server had both clients properly working, we proceeded to install the different ONAP modules required for the minimal deployment.

## 3.5. ONAP deployment

With the virtual machines correctly instantiated and with OOM installed, we proceeded to deploy ONAP's platform with Helm and its charts.

We decided to install all ONAP's modules in the first deployment to make sure that all of them work correctly.

Making this full deployment takes around 8h to complete, less than 5h with the Docker registry virtual machine.

Once we ensured the correct functioning of the different modules, as the virtual Firewall use case only requires the minimal deployment, we decided to redeploy it with the minimal modules enabled for the simplest use case to work, thus not overloading the servers. [38]

Making this minimal deployment takes around 4h to complete, less than 2h with the Docker registry virtual machine.

It's very significant to highlight the importance of monitoring the critical points during the installation in order to ensure all the pods from the modules start correctly.

Once the installation is over, another important step is to check that the NFS server is up and running and properly working, that all the worker VMs share the same storage, and that the pods inside them are not restarting.

So as to check the health of the different pods, we have used the following command:

```
kubectl –n onap get pods ––all–namespaces | grep –v Running | grep –v Complete
```

If a pod is not running, it's mostly because one of its dependencies, which is another pod, is not ready yet. Starting again the dependencies of the not completed pods will most likely solve the problem. In order to do so, the following command was used:

```
kubectl –n onap delete ––grace–period=0 ––force pod <pod–id>
```

However, to avoid these restarting pods, it's very important to respect the order of installation, something that the official Casablanca's documentation overlooks.

In order to respect this order, we used a custom script called *cd.sh* that automatically deploys all ONAP's modules taking into account the order selected by us. [39]

This order can be seen in the following snippet of code. Apart from the order, we can also appreciate the number of pods required to be in running state to consider the module as ready for the next one to start the deployment.

```
DEPLOY_ORDER_POD_NAME_ARRAY=('consul msb dmaap dcaegen2 aaf robot aai so sdc sdnc vid policy portal log appc clamp cli')
# don't count completed pods
DEPLOY_NUMBER_PODS_DESIRED_ARRAY=(4 5 11 11 13 1 15 10 12 11 2 8 6 3 5 5 1)
# account for podd that have varying deploy times or replicaset sizes
# don't count the 0/1 completed pods – and skip most of the ResultSet instances except 1
# dcae boostrap is problematic
DEPLOY_NUMBER_PODS_PARTIAL_ARRAY=(2 5 11 9 11 1 11 10 12 11 2 8 6 3 5 5 1)
```

The command to run the script has been the following:

```
sudo ./cd.sh –b master –e onap –p false –n docker–registry–server:10001 –f true –c false –d false –w true –r false
```

Where '–b' specifies the release, the '–e' specifies the environment, the '–p' specifies whether if a docker prepull has to be performed or not, the '–n' stipulates the portal to be used in the pulling stage, the '–f' determines whether if it has to be a full deployment or not, the '–c' indicates if a new repository has to be cloned, the '–d' orders to delete the previous OOM deployment, the '–w' forces to apply workarounds to avoid existing bugs and finally, '–r' removes the OOM deployment at the end of the script [39].

Instead of downloading the official git repository through the script, we used a local repository with the Casablanca release previously downloaded via a git clone in the machine in order to have more control over the new versions to avoid unwanted changes that might reduce the functionality or even destabilize the whole deployment. Because of that, the variable '–b' can be set to any of the releases, but only if the '–c' variable is set to false.

As it has already been mentioned, we created our own docker registry where we prepull all the ONAP images prior to the deployment. That is the reason for which the '–p' variable is set to false.

Even though the '–f' flag is set to true, as we had modified the script, we are not deploying a full ONAP but only the minimal modules required so as to instantiate the vFW use case.

One of the biggest problems when deploying ONAP, was the number of workarounds we had to implement in order to make it work correctly. Thankfully, with the '–w' flag set to true, we can apply them all at once automatically.

Finally, when all the modules are installed, the script checks every single component to make sure it works fine and returns the overall status of the deployment. In order to perform this health check, the following command was run by the script:

```
sudo ./oom/kubernetes/robot/ete-k8s.sh onap health
```

Apart from the tests the script performs, to ensure the correct installation of the different modules, we ran health checks inside the RKE server, which gave us a list with the status of all the modules.

```
ubuntu@onap-rancher-server:~$ helm list
NAME            REVISION        UPDATED                         STATUS          CHART           NAMESPACE
onap            18              Thu Jun 27 00:03:44 2019        DEPLOYED        onap-3.0.0      onap
onap-aaf        13              Thu Jun 27 00:03:45 2019        DEPLOYED        aaf-3.0.0       onap
onap-aai        11              Thu Jun 27 00:03:48 2019        DEPLOYED        aai-3.0.0       onap
onap-appc       3               Thu Jun 27 00:03:56 2019        DEPLOYED        appc-3.0.0      onap
onap-clamp      2               Thu Jun 27 00:03:57 2019        DEPLOYED        clamp-3.0.0     onap
onap-cli        1               Thu Jun 27 00:03:59 2019        DEPLOYED        cli-3.0.0       onap
onap-consul     17              Thu Jun 27 00:04:00 2019        DEPLOYED        consul-3.0.0    onap
onap-dcaegen2   14              Thu Jun 27 00:04:02 2019        DEPLOYED        dcaegen2-3.0.0  onap
onap-dmaap      15              Thu Jun 27 00:04:05 2019        DEPLOYED        dmaap-3.0.0     onap
onap-log        4               Thu Jun 27 00:04:08 2019        DEPLOYED        log-3.0.0       onap
onap-msb        16              Thu Jun 27 00:04:09 2019        DEPLOYED        msb-3.0.0       onap
onap-policy     6               Thu Jun 27 00:04:16 2019        DEPLOYED        policy-3.0.0    onap
onap-portal     5               Thu Jun 27 00:04:19 2019        DEPLOYED        portal-3.0.0    onap
onap-robot      12              Thu Jun 27 00:04:21 2019        DEPLOYED        robot-3.0.0     onap
onap-sdc        1               Thu Jun 27 09:42:17 2019        DEPLOYED        sdc-3.0.0       onap
onap-sdnc       8               Thu Jun 27 00:04:25 2019        DEPLOYED        sdnc-3.0.0      onap
onap-so         10              Thu Jun 27 00:04:29 2019        DEPLOYED        so-3.0.0        onap
onap-vid        7               Thu Jun 27 00:04:34 2019        DEPLOYED        vid-3.0.0       onap
```

However, the graphic interface of Rancher allows us to see the persistent volumes of the different modules, as it can be seen in Figure 3.6.

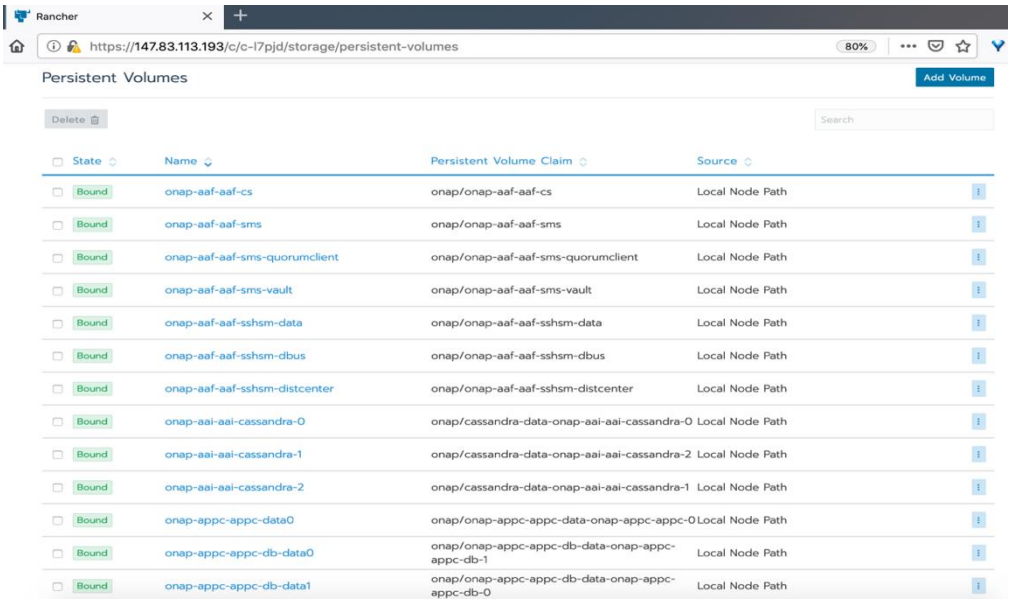


**Fig. 3.6** Rancher's persistent volumes

## 3.6. Keystone compatibility issue

Keystone, OpenStack's identity service provider, has been evolving throughout the different versions of OpenStack. The latest version, which is the one our version of OpenStack employ, is Keystone v3.

We have deployed ONAP's Casablanca release, which has a very limited Keystone compatibility, being only compatible with Keystone v2, a deprecated version of the core. This has supposed a major setback since the platform requires v2 to deploy the network functions and we have v3 [40].

After having contacted ONAP Orange OpenLab, the laboratory we have used as reference to choose our OpenStack's version, we found out that in order to overcome these compatibility shortcomings, they used a private third-party Keystone wrapper.

Since that wrapper was private, we weren't able to use it for our OpenStack scenario so as to deploy the virtualized network functions.

It is expected that all of these issues will be solved in ONAP's Dublin release, since it will give native support to Keystone v3. However, this will suppose an OOM and ONAP upgrade of our environment in order to implement these changes, with all the possible failures and setbacks attached to it [41].

Since the current Dublin release is not stable, we tried to use a second OpenStack with the Ocata release, which implements both Keystone versions. This OpenStack was already deployed in another UPC's laboratories, but we didn't succeed in ONAP getting the necessary configurations in order to connect to an different OpenStack than the one where ONAP is deployed.

# CONCLUSIONS

As it has been seen throughout the entire project, thanks to OpenStack and Rancher, it's possible to deploy a Kubernetes cluster in a reasonable period of time, especially thanks to the custom Terraform scripts we have developed in order to automate the whole deployment process.

Even though ONAP's official documentation describes these procedures in a very manual and confusing way, mainly because they focus on designing and implementing services, they do not explain how to deploy the infrastructure needed.

It's also very important to highlight the complexity of ensuring a completely functional full ONAP deployment, since what we have explained in this document didn't ascertain the correct functioning of more specific points, such as the connection between the SO module and OpenStack's Keystone.

These issues arise when deploying the actual service in this environment, what forced us to rollback, change the configuration, and deploy the infrastructure again.

A direct consequence of this work is a potential reduction of costs for mobile operators since it greatly reduces the amount of time required to deploy this platform.

Along with these monetary savings, there is also a positive impact regarding the environment. Less monolithic machines performing tasks that are now virtualized thanks to this paper imply a reduced power consumption, which ultimately translates into a benefit for the environment.

This project is very complete as is, but in the future, it would be very interesting to automate with Ansible the cluster initialization, currently done with RKE, and the kubectl and Helm installation in the Rancher-server machine, which has been performed manually, in order to save more time and obtain an ONAP environment ready to be deployed just by running a few scripts, which would, in the end, save even more money.

Similarly, there is the necessity of updating ONAP's release to Dublin so as not to need create intricated workaroundss since Dublin's release will support Keystone v3.

The work that has been presented in this document will be the starting point of another project called "Developing and deploying advanced NFV solutions" in which the goal is to use ONAP's infrastructure in order to deploy virtualized network functions.

# BIBLIOGRAPHY

[1] Charles David Graziano. A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project, https://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=3243&context=etd [Accessed: 15/05/2019]

[2] The Linux Foundation Projects. ONAP Platform, https://www.onap.org/platform-2 [Accessed: 15/05/2019]

[3] Cloudify. What is ONAP and what does it mean for you? https://cloudify.co/onap/what-is-onap/ [Accessed: 15/05/2019]

[4] OpenStack. Official OpenStack webpage, https://www.openstack.org/ [Accessed: 17/05/2019]

[5] Opensource, What is OpenStack? https://opensource.com/resources/what-is-openstack [Accessed: 17/05/2019]

[6] OpenStack. Official OpenStack webpage, https://www.openstack.org/software/ [Accessed: 17/05/2019]

[7] Ubuntu, What is OpenStack? https://www.ubuntu.com/openstack/what-is-openstack [Accessed: 17/05/2019]

[8] Ansible. Official Ansible webpage, https://www.ansible.com/overview/how-ansible-works [Accessed: 20/05/2019]

[9] Cloudacademy. What is Ansible? https://cloudacademy.com/blog/what-is-ansible/ [Accessed: 20/05/2019]

[10] Networklore. What is Ansible? https://networklore.com/ansible/ [Accessed: 20/05/2019]

[11] Edureka. What is Ansible? https://www.edureka.co/blog/what-is-ansible/#advantages_of_using_ansible [Accessed: 20/05/2019]

[12] Terraform. Official webpage, https://www.terraform.io/intro/index.html [Accessed: 21/05/2019]

[13] Scottlogic. Infrastructure as Code - Getting Started with Terraform, https://blog.scottlogic.com/2018/10/08/infrastructure-as-code-getting-started-with-terraform.html [Accessed: 21/05/2019]

[14] Docker. Official Docker webpage, https://www.docker.com/why-docker [Accessed: 22/05/2019]

[15] Webopedia. What is Docker? https://www.webopedia.com/TERM/D/docker.html [Accessed: 22/05/2019]

[16] Docker. Official Docker user guide, https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/ [Accessed: 22/05/2019]

[17] Searchitoperations, What is Docker Image? https://searchitoperations.techtarget.com/definition/Docker-image [Accessed: 22/05/2019]

[18] Kubernetes. Official Kubernetes webpage, https://kubernetes.io/ [Accessed: 23/05/2019]

[19] Kubernetes. Official Kubernetes documentation https://kubernetes.io/docs/home/ [Accessed: 23/05/2019]

[20] Infoworld. What is Kubernetes? Container orchestration explained, https://www.infoworld.com/article/3268073/what-is-kubernetes-container-orchestration-explained.html [Accessed: 23/05/2019]

[21] Redhat. What is Kubernetes? https://www.redhat.com/en/topics/containers/what-is-kubernetes [Accessed: 23/05/2019]

[22] Rancher. Official Rancher webpage, https://rancher.com/what-is-rancher/overview/ [Accessed: 24/05/2019]

[23] Marksei. What is Rancher? Containers in the age of Cattle https://www.marksei.com/rancher/ [Accessed: 24/05/2019]

[24] Rancher. Official Rancher documentation, https://rancher.com/docs/rancher/v1.6/en/ [Accessed: 24/05/2019]

[25] Helm. Offial Helm documentation, https://helm.sh/docs/ [Accessed: 27/05/2019]

[26] Hackernoon. What is Helm and why you should love it https://hackernoon.com/what-is-helm-and-why-you-should-love-it-74bf3d0aafc [Accessed: 27/05/2019]

[27] Boxboat. Helm and Kubernetes deployments, https://boxboat.com/2018/09/19/helm-and-kubernetes-deployments/ [Accessed: 27/05/2019]

[28] ONAP. Official ONAP webpage, https://www.onap.org/ [Accessed: 28/05/2019]

[29] Cloudify. What is ONAP? https://cloudify.co/onap/what-is-onap/ [Accessed: 28/05/2019]

[30] OOM. OOM User guide, https://onap.readthedocs.io/en/latest/submodules/oom.git/docs/oom_user_guide.html#user-guide-label [Accessed: 28/05/2019]

[31] ONAP. Platform Architecture, https://www.onap.org/platform-2 [Accessed: 28/05/2019]

[32] OpenStack. OpenStack-Ansible documentation, https://docs.openstack.org/openstack-ansible/rocky/ [Accessed: 28/05/2019]

[33] Openpower. Integrating OpenStack with Let's Encrypt, http://openpower.ic.unicamp.br/blog/integrating-openstack-ansible-with-lets-encrypt.html [Accessed: 29/05/2019]

[34] ONAP. ONAP on Kubrenetes with Rancher, https://docs.onap.org/en/casablanca/submodules/oom.git/docs/oom_setup_kubernetes_rancher.html [Accessed: 29/05/2019]

[35] Rancher. Official Rancher documentation, https://rancher.com/docs/rke/latest/en/https://docs.onap.org/en/dublin/submodules/oom.git/docs/oom_setup_kubernetes_rancher.html#run-rke [Accessed: 30/05/2019]

[36] Rancher. Official Rancher webpage, https://rancher.com/quick-start/ [Accessed: 31/05/2019]

[37] Rancher. Official Rancher documentation https://rancher.com/docs/rancher/v2.x/en/cluster-provisioning/imported-clusters/ [Accessed: 31/05/2019]

[38] Github. Oficial OOM repository. Minimal onap deployment, https://github.com/onap/oom/blob/master/kubernetes/onap/resources/environments/minimal-onap.yaml [Accessed: 03/06/2019]

[39] Gerrit. Official ONAP repository, https://git.onap.org/logging-analytics/tree/deploy/cd.sh [Accessed: 03/06/2019]

[40] ONAP. Casablanca release notes, https://docs.onap.org/en/casablanca/release/ [Accessed: 04/06/2019]

[41] ONAP. SO's Jira, https://jira.onap.org/browse/SO-18 [Accessed: 05/06/2019]

[42] Opensource.What is virtualization? https://opensource.com/resources/virtualization [Accessed: 06/06/2019]

[43] Redhat.What is virtualization?
https://www.redhat.com/en/topics/virtualization [Accessed: 06/06/2019]

[44] Vmware. Understanding Full Virtualization, Paravirtualization, and Hardware Assist
https://www.vmware.com/es/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html [Accessed: 06/06/2019]

[45] Networks & Servers. Full Virtualization explained
https://networksandservers.blogspot.com/2011/11/full-virtualization-explained.html [Accessed: 06/06/2019]

[46] Anandtech. Hardware virtualization https://www.anandtech.com/show/2480 [Accessed: 06/06/2019]

[47] SearchServer. Paravirtualization advantages, disadvantages and use cases
https://searchservervirtualization.techtarget.com/answer/Paravirtualization-advantages-disadvantages-and-use-cases [Accessed: 06/06/2019]

[48] Vmware. Paravirtualization pros and cons
https://communities.vmware.com/thread/260788 [Accessed: 06/06/2019]

[49] Bayt. What is Hardware Virtualization and what are the Benefits of Hardware Virtualization. https://m.specialties.bayt.com/en/specialties/q/80150/what-is-hardware-virtualization-and-what-are-the-benefits-of-hardware-virtualization/ [Accessed: 06/06/2019]

[50] Sumo logic. Containerization: Enabling DevOps Teams. https://www.sumologic.com/blog/devops/how-containerization-enables-devops/ [Accessed: 06/06/2019]

[51] Webopedia. Containerization.
https://www.webopedia.com/TERM/C/containerization.html
[Accessed: 06/06/2019]

[52] Linuxnix. WHAT IS CONTAINERIZATION IN DEVOPS?
https://www.linuxnix.com/what-is-containerization-in-devops/
[Accessed: 06/06/2019]

[53] SearchNetworking. network functions virtualization
https://searchnetworking.techtarget.com/definition/network-functions-virtualization-NFV
[Accessed: 06/06/2019]

[54] Juniper. What is Network Functions Virtualization?
https://www.juniper.net/us/en/products-services/what-is/network-functions-virtualization/ [Accessed: 06/06/2019]

[55] Sdxcentral. What is NFV – Network Functions Virtualization https://www.sdxcentral.com/networking/nfv/definitions/whats-network-functions-virtualization-nfv/ [Accessed: 06/06/2019]

[56] Mikrotik. RB2011UiAS-2HnD-IN https://mikrotik.com/product/RB2011UiAS-2HnD-IN [Accessed: 06/06/2019]

[57] Netgear. 10-Gigabit Smart Managed Pro Switch Series https://www.netgear.es/business/products/switches/smart/XS728T.aspx [Accessed: 06/06/2019]

[58] Dell. PowerEdge R440 Rack Server https://www.dell.com/en-us/work/shop/povw/poweredge-r440 [Accessed: 06/06/2019]

[59] Dell. PowerEdge R740 Rack Server https://www.dell.com/en-us/work/shop/povw/poweredge-r740 [Accessed: 06/06/2019]


# ANNEX

**TITLE: Containerizing ONAP using Kubernetes and Docker**

**MASTER DEGREE: Master's degree in Applied Telecommunications and Engineering Management (MASTEAM)**

**AUTHOR: Albert Toro Marin**

**ADVISOR: Jesus Alcober**

**DATE: July, 10th 2019**

# ANNEX I: TYPES OF VIRTUALIZATION

## I.1. Hardware Virtualization

Hardware virtualization is the process of running a virtual instance of a complete or partial computer system in a layer abstracted from the actual hardware, thus hiding the physical characteristics of the computing platform to the upper layers [42].

For these upper layers, which usually manage applications running on top of the virtualized machine, it appears as if they were on their own dedicated machine, where the operating system, libraries, and other software are unique to the virtualized machine and unconnected to the operating system which runs below. Hardware virtualization is often used to create IT services that are traditionally bound to hardware, in such a way that optimizes the use of resources of the machine and allows distributing its capabilities amongst the different virtual environments, also called virtual machines [42] [43].

Ultimately, virtual machines are based on computer architectures that provide the functionality of a physical computer, but their implementation relies upon the level of virtualization desired.

In fully virtualized machines, the machine simulates enough hardware to allow an unmodified operative system (OS) run isolated from other instances [44].

Full virtualization requires every feature of the hardware to be reflected in the virtual machine, something that might not be required depending on the application [44].

The full virtualization technique grants the potential to combine existing systems on to newer ones with increased efficiency using the Virtual Machine Manager (VMM), a paramount element in all kinds of virtualizations. The representation of a fully virtualized machine can be seen in Figure 1 [44] [45].

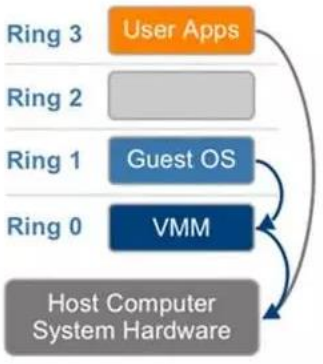


**Fig. 1** Representation of a fully virtualized machine

It effectively trims down the operating costs related to repairing and enhancing older and less competent systems, ultimately reducing the physical space and augmenting the overall performance.

But the virtualization software makes applications often run somewhat slower on virtualized systems because the hypervisor has to process additional data, which in the end means that part of the computing power of a physical server and its related resources have to be reserved for the hypervisor [46].

To sum up, the VMM brings some performance degradation caused by the extra processing.

Moreover, the hypervisor must contain the interfaces to the resources of the machine, that is to say, it must have the device drivers. If a machine has hardware resources the hypervisor has no driver for, the virtualization software can't be run on that machine [46].

In hardware-assisted virtualization, the hardware provides enough support to enable building isolated virtual machines. The representation of this kind of hardware virtualization can be seen in Figure 2 [46].

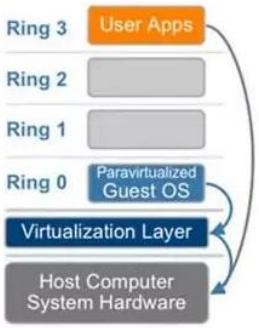


**Fig. 2** Representation of a hardware-assisted virtual machine

In this case, the hardware is not simulated because it uses the hardware provided by the host machine.

It is much more efficient than fully virtualized machines, but since it requires to employ the machine's processors, sometimes there is no explicit support with the hardware [44].

This sort of virtualization uses the unmodified guest operating system, which involves many virtual machine traps, and thus high CPU overheads, limiting scalability and the efficiency of server consolidation.

Finally, in paravirtualization, the virtual machine is presented as a software interface which is similar, but not identical, to the underlying hardware/software interface. This special interface can only be used to modify the operative system. For this to be possible, the host's operative system's source code must be available, otherwise, this sort of virtualization cannot be implemented [44] [47].

Paravirtualization enhances performance by decreasing the number of VMM calls and prevents the unnecessary use of privileged instructions.

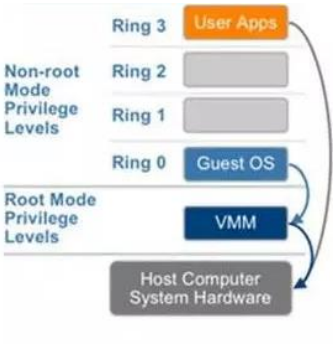


**Fig. 3** Representation of a paravirtualized machine

The representation of a paravirtualized machine can be seen in Figure 3. Additionally, it allows running many operating systems on a single server.

Generally, this method is considered as the most beneficial technique since it improves the performance per server without the operating cost of a host operating system [47] [48].

However, paravirtualization requires the operating system to be modified in order to interact with the paravirtualized interfaces. This typically limits support to open source operating systems and some proprietary operating systems [47] [48].

As paravirtualization cannot support unmodified operating systems, its compatibility and portability are poor [48].

Paravirtualization can also introduce significant support and maintainability issues in production environments since it requires deep operative system kernel modifications. [48]

An essential feature of virtualization are *hypervisors*, which is the term that defines the software, hardware or firmware in charge of handling the virtualization process in the machine, that is to say, it creates, runs and destroys virtual machines.

Hypervisors are split into two classes, which can be seen graphically in the Figure 4. [49]:

- Type one, “native” or "bare metal" hypervisors run guest virtual machines directly on the system's hardware. An example of a type one hypervisor would be Xen.

- Type two, or "hosted" hypervisors behave like traditional applications that can be started and stopped like a normal program. An example of a type two hypervisor would be VirtualBox.

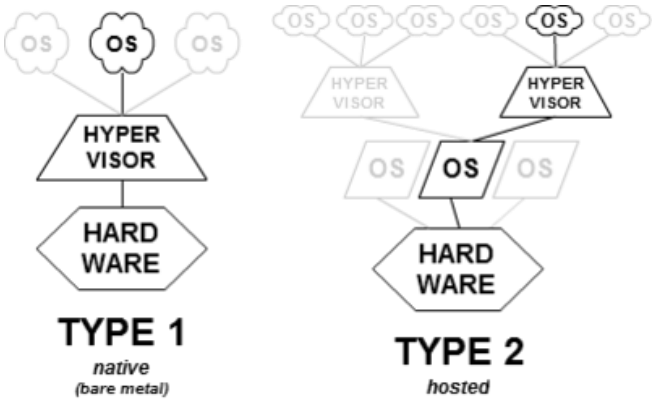


**Fig. 4** Representation of the different hypervisors

## I.2. Operating-system-level virtualization

Operating-system-level virtualization, more commonly known as containerization, is a type of lightweight virtualization in which applications are encapsulated inside a container [50].

These containers are partitions, virtual environments or jails that look like computers from the point of view of the applications.

Regular operating systems can see all the system's resources and can allow or deny access to programs.

However, with containerization, it is possible to run programs within containers, to which only parts of the operating system resources are allocated [51].

A program inside a container can only see the designated resources and believes them to be all that is available.

This allows running any application in any suitable physical machine without having to worry about the dependencies.

Several containers can be created on each operating system, to each of which a subset of the computer's resources are allocated. Each container may contain any number of computer programs, and these programs can run concurrently, separately or even interact with each other [51] [52].

Containerization also enables working with identical development environments and stacks, facilitating the use of stateless designs.

The cornerstone for containerization lies in the LinuX Containers (LXC) format, which is a user-space interface for the Linux kernel containment features [52].

As a result, containerization only works in Linux environments and can only run Linux applications.

Oppositely, with traditional hypervisors, applications can run on Windows or any other operating system that supports the hypervisor. A graphical comparison between containers and traditional hypervisors can be seen in Figure 5 [49].

Another difference with containerization is that containers share the Linux kernel used by the operating system running the host machine, which means that any other container running on the host machine will also be using the same Linux kernel. This sharing makes deploying multiple containers on a single host both quick and extremely efficient [51].

However, for this to be possible, composition and clustering are required.

Computer clustering is where a set of computers are connected and work together so that they can be viewed as a single system.

Similarly, container cluster managers handle the communication between containers, manage resources and manage task execution.

Lately, containerization has gained a lot of renown thanks to the open-source container application known as Docker.

Docker containers are designed to run on everything: from physical computers to virtual machines, bare-metal servers, OpenStack cloud clusters, and public instances.

There are other solutions, but Docker has managed to make easier containerization by offering a common toolset, packaging model and deployment mechanism that greatly simplifies the containerization and distribution of applications [52].

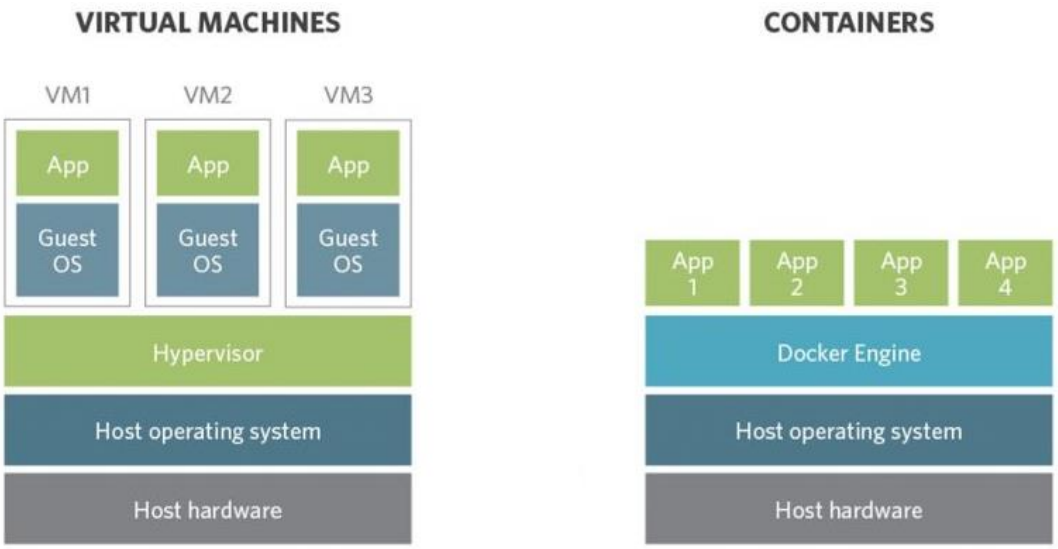


**Fig. 5** Representation of legacy Hypervisors VS Containers using Docker

Containerization has many advantages, which are the following [51] [52]:

- Multi-Cloud Platform: Containers can operate on the cloud.
- Shares the OS: As it has been above-mentioned, containers share the same operative system kernel as the host, making them more efficient than hardware virtualized machines.
- Testing and CI-CD: Containers are consistent throughout the application tests and approaches like Continuous Integration and Continuous Delivery (CI-CD).
- Portability: Containers have better portability than other hosting technologies. They can move along any system. The container configuration is also portable since it is just a file.
- Version Control: Docker offers version control that makes it easy to roll back to a previous image if a critical error happens.
- Cost-Efficiency: Containers are cost-efficient. It is possible to support many containers on the same infrastructure.
- Speed: Containers deploy faster than VMs.

But it also has some drawbacks [51]:

- Lack of Isolation: There is a flagrant lack of operative system flexibility.
- Security: Since containers share the operative system, the potential threats can easily penetrate to the system due to the lack of isolation.
- Monitoring: It is hard to monitor containers when hundreds or even thousands of them are running on the same server.
- Features: Most of the big features, such as a native monitoring system for containers, are still in development.

The drawbacks have dedicated solutions that solve them, but it is expected that in the near future there will be available native solutions that will properly assess these problems without having to rely upon third-party software.

## I.3. Network Function Virtualization

Network Function Virtualization (or NFV) is a kind of virtualization that depends on server-virtualization. However, it is not exactly the same.

The goal of NFV is to abstract network functions, called Virtualized Network Functions or VNF, allowing them to be installed, controlled and operated by software running on standardized compute nodes, avoiding like this having to use dedicated and proprietary hardware [53] [54].

Eventually, NFV decouples legacy network functions, such as network address translation (NAT), firewalls and domain name system (DNS) amongst others so they can run in any server.

NFV combines both cloud and virtualization technologies to allow a quick growth of new network services with elastic scale and automation characteristic of cloud environments [54].

NFV is prepared to consolidate and deliver the networking components needed to support a fully virtualized infrastructure, including virtual servers, storage, and other networks [53] [54].

Furthermore, NFV can work in any data processing plane or control plane in both wired and wireless network infrastructures, and its modular architecture allows automating every single layer [54].

The NFV infrastructure, or NFVI, provides the virtualization layer (hypervisors or container management systems), the physical compute, storage, and networking components to host the VNFs [55].

The NFVI is managed through the virtual infrastructure manager, or VIM, which controls the allocation of resources of the different VNFs [54].

These VNFs use the virtualized infrastructure provided by the NFVI to connect into the network and provide programmable, scalable network services [54] [55].

And finally, the management and orchestration, or MANO, is the layer responsible for the all-encompassing management and orchestration of the VNFs in the whole NFV architecture.

Basically, MANO deploys the network services through the automation, provisioning, and coordination of workflows to the VIM and VNF Managers that ultimately deploy the VNFs and overlay networking service chains [55].

MANO also connects the NFV architecture with the operation support systems (OSS) and business support systems (BSS). A representation of this infrastructure can be seen in Figure 6 [55].

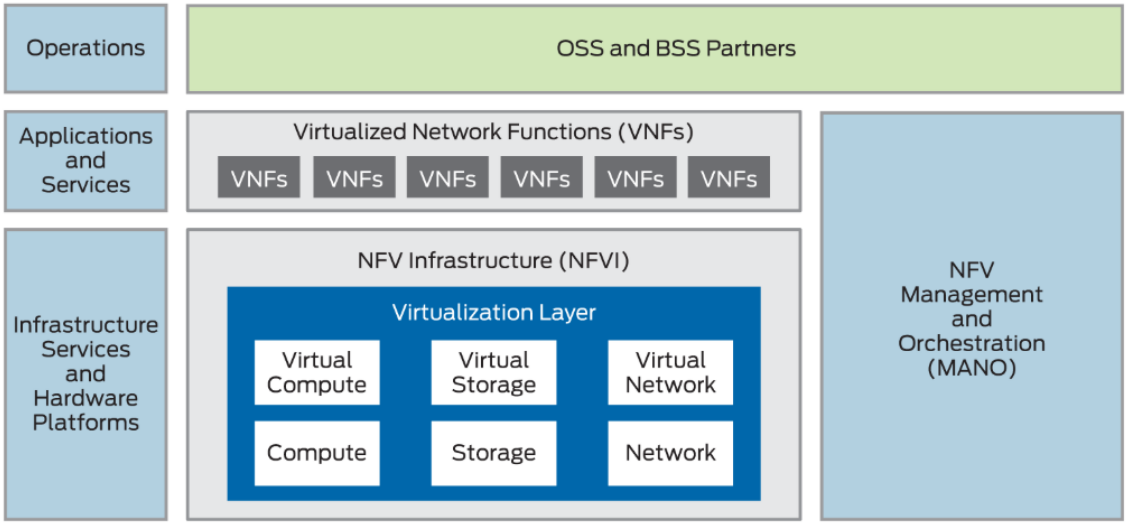


**Fig. 6** Representation of a fully virtualized infrastructure

NFV substantially speeds up delivering cost-efficient network services by removing bottlenecks caused by sluggish manual processes, also allowing new services to be deployed on demand, granting the possibility to adapt them to the customers' needs [55].

In the end, NFV reduces CAPEX since there is no need to purchase dedicated hardware, also enabling "pay-as-you-grow" models to reduce the costs related to unneeded over-provisioning [55].

Moreover, NFV also reduces OPEX because the virtualization of functions reduces the space, power and cooling requirements of the equipment [55].

Reducing the time to deploy new networking services allows smaller “Time-to-Market” times, allowing to quickly seize new market opportunities and improve the ROI of new services [55].

NFV reduces the risk of deploying new services because providers can easily try and evolve these novel services beforehand, making it possible to determine what is the best configuration for their customers.

However, even if NFV has many clear advantages, its deployment has seen hampered due to a lack of standards in MANO. NFV still has to adopt a sizeable number of standards.

Another problem is the huge variety of NFV solutions and approaches.

With so many competing possibilities, all supported by different service providers and operators, deciding which approach is the one that offers the best capabilities for the whole industry, is a nearly impossible task [53].

As a result, some service providers do not know which standards will be adopted and are more hesitant to invest in them, exacerbating the already complicated situation.

# ANNEX II: OPENSTACK CORES

- Nova: It is the primary computing engine behind OpenStack. Used for deploying and managing large numbers of virtual machines and other instances to handle computing tasks.

- Swift: It is the storage system for objects and files. Files have unique identifiers and OpenStack decides where to store this information. It also allows the system to automatically back up the data in case of the failure of a machine or network connection.

- Cinder: It is the block storage component. Allows accessing the files through a path, which might be important in scenarios where data access speed is the most important consideration.

- Neutron: It provides the networking capability for OpenStack. It ensures that each of the components of an OpenStack deployment can communicate with one another quickly and efficiently.

- Horizon: It is the dashboard behind OpenStack. It is the only graphical interface to OpenStack.

- Keystone: It provides identity services for OpenStack. It is a central list of all of the users of the OpenStack cloud, mapped against all of the services provided by the cloud, which they have permission to use. It provides multiple means of access.

- Glance: It provides image services to OpenStack. In this case, images refer to images or virtual copies of hard disks. Glance allows these images to be used as templates when deploying new virtual machine instances.

- Ceilometer: It provides telemetry services, which allows the cloud to provide billing services to individual users of the cloud. It also keeps count of each user's system usage of each component of an OpenStack cloud.

- Heat: It's the orchestration component of OpenStack, which allows developers to store the requirements of a cloud application in a file that defines what resources are necessary for that application. It simplifies managing the infrastructure needed for a cloud service to run.

# ANNEX III: SETUP HARDWARE

Essentially, our network is based on a MikroTik router, a Netgear Business class high capacity switch, and multiple Dell Rack Servers.

Each element is going to be explained in detail in the following subpoints.

## III.1. MikroTik 2011 UIAS Router

In order to create a private cloud, it's vital to isolate the servers from the public UPC network, but at the same time, they must be accessible to us so we can remotely configure and manage them.

To perform this separation, we have used an RB2011 UIAS MikroTik router.


**Fig. 1** MikroTik Router Board 2011 UIAS

This router is a low-cost multi-port network element designed especially for indoor use, and available in many different cases, with a multitude of options.

The RB2011 is powered by RouterOS, a fully featured routing operating system that gives us complete control over the device and its functions, such as dynamic routing, hotspot, firewall, MPLS, VPN, load balancing, real-time configuration and monitoring.

This router has a powerful Atheros 600MHz 74K MIPS network processor, 128MB of RAM, five Gigabit LAN ports, very important not to create bottlenecks between the private cloud and the public UPC network, and five Fast Ethernet LAN ports.

The complete specification sheet can be found in [56]

## III.2. Netgear ProSAFE Smart Managed Switch (serie XS728T)

Between the private cloud and the southbound interface of the router, there is a heavy-duty business class switch, responsible of splitting the control traffic and data traffic generated by the servers, which will be allocated in two different VLANs. Moreover, this switch will also perform DHCP tasks.


**Fig. 2** Netgear ProSAFE Smart Managed Switch (serie XS728T)

The NETGEAR XS728T is a powerful Smart Managed Switch that comes with 24 10-Gigabit Copper ports and 4 additional Dedicated SFP ports for 10G Fiber links.

This switch is a cost-effective approach to provide 10G connections to latency-dependent servers and storage systems.

Moreover, it comes with private VLAN support, protected ports to keep the cloud communications private and robust security features to avoid unauthorized access to the network.

The complete specification sheet of this device can be found in [57]

## III.3. Dell Rack Server R440

Since ONAP is an extremely demanding platform, very powerful and resilient servers are required, for both the control and data planes.


**Fig. 3** Dell PowerEdge Rack Server R440

Since ONAP is an extremely demanding platform, very powerful and resilient servers are required, for both the control and data planes.

Designed for shared access control, the Dell EMC PowerEdge R440 server is a dual–socket 1U system that has an excellent balance of performance, perfect for carrying out heavy duty tasks, and density for scale-out computing.

It is also flexible, allowing to enhance its performance with dual Intel® Xeon® Scalable processors, up to 16 DIMMs, and scalable storage that permits mixing SSDs and NVMe PCIe SSDs. The exact specifications can be found in [58]

Moreover, in order to simplify its management, these servers have embedded diagnostics, improved security elements, and automation features, helping the R440 to deliver maximum uptime and make the environment safer.

In our network, we have two different R440, 3 for control and 2 for compute.

The control R440s have 64GB of RAM, 1 12C/24T processor and 1TB of disk (HDD)

The compute R440s have 128GB of RAM, 2 10C/20T processors and 1 TB of disk (HDD)

The compute servers have better features because the ONAP compute modules are more demanding than the ones needed for control.

## III.4. Dell Rack Server R740

As it has been said before, the compute servers are under a lot of pressure, so much, that a Dell Rack Server R740 is required.


**Fig. 4** Dell PowerEdge Rack Server R740

The Dell PowerEdge R740 server is a dual-socket 2U rack system with flexible storage options and built-in security features ideal for almost any application.

Powered by Intel's novel C620 chipset, this system provides support for very demanding applications like ONAP.

Database performance is ten times better with support for NVDIMMs, and Intel's new chipset provides 27% more cores than older versions. Three double-width or six single-wide GPUs can be added to this system to enhance virtual desktop infrastructure (VDI) deployments. The complete specifications can be found in [59]

In our network, we only have one R740 with 128GB of RAM, 2 10C/20T processors and 1 TB of disk (HDD)

Since this server is bigger and it will have to perform very demanding tasks, a graphics card can be installed in order to heighten its performance and speed up its processes to accelerate the workloads.

Like the R440 server, the R740 has an improved security system, embedded diagnostic tools and automation features that make simpler its use.