



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona



Validation and Extension of Kubernetes-based Network Functions (KNFs) in OSM for Cloud Native (CN) applications in 5G and beyond

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by
Adrián Pino Martínez

In partial fulfillment of the requirements for
Master's degree in Advanced Telecommunication Technologies (MATT)

Directors

Pouria Sayyad Khodashenas
Muhammad Shuaib Siddiqui

Advisor

Xavier Hesselbach Serra

Barcelona, January 2021



Contents

List of Figures	4
1 Introduction	9
1.1 Motivation	9
1.2 Structure of the work	10
1.2.1 Work plan	10
1.2.2 Gantt diagram	10
1.3 Incidents and Modifications	11
2 Cloud Native in the telco sector: State of the art	12
2.1 Cloud Native and 5G previous background	12
2.2 Cloud Native: challenges and enablers	14
2.3 SDN, NFV and ETSI NFV MANO	17
2.3.1 SDN and NFV	17
2.3.2 NFV Architectural framework	18
2.3.3 ETSI NFV MANO	20
2.4 Evolution of Network Functions (NF): from VNF to CNF	21
2.4.1 Past	21
2.4.2 Current status	22
2.4.3 Possible future approach	22
3 Kubernetes	23
3.1 Why use Container Orchestrators?	23
3.2 What is Kubernetes?	23
3.3 K8s features	23
3.4 K8s architecture	24
3.4.1 K8s main components	25
3.4.2 K8s main building blocks	26
3.4.3 K8s Services	27
3.5 Alternatives to K8s	28
4 Our NFV Implementation	29
4.1 Open Source MANO as NFVO and VNFM	30
4.1.1 OSM features	31
4.1.2 OSM architecture	31
4.1.3 OSM and K8s	32
4.2 Openstack as VIM	33
4.3 Kubernetes as VIM	35
5 Experimentation	36
5.1 OSM and OpenStack's Testbed	36
5.2 OSM and K8s' Testbed	43
5.3 OSM-K8s Lessons learnt	49
6 Perform validation and benchmarking	51

6.1	Related Works in the literature	51
6.2	Images used in the simulations	51
6.3	KPI 1: Deployment Process Delay (DPD)	52
6.4	KPI 2: CPU and RAM	53
6.5	KPI 3: Max number of VNFs/KNFs supported using same resources	54
7	Conclusions and future work	56
7.1	Conclusions	56
7.2	Future work	56
	References	58
	Appendices	61
A	OSM Descriptors: NSDs and VNFDs	61
A.1	VNFD "cirros_knfd" (It contains a kdu (helm-chart) with a cirros docker image).	61
A.2	VNFD "cirros_vnfd" (It contains a vdu with a cirros cloud image).	61
A.3	NSD "cirros_2knf_nsd" (It contains 2 "cirros_knfd" KNFs).	62
A.4	NSD "cirros_2vnf_nsd" (It contains 2 "cirros_vnfd" VNFs).	62
B	Code Fragments	64
B.1	KNF Instantiation	64
B.2	KNF DPD time	64
B.3	VNF Instantiation	65
B.4	VNF DPD time	66

List of Figures

1	Gantt diagram of the project	10
2	CN-journey	12
3	5G-usage-scenarios	13
4	CN-Pillars	14
5	architecture-evolution	15
6	VMs vs Containers	16
7	SDN vs NFV	17
8	ETSI NFV Architectural Framework	18
9	Cloud Native evolution	21
10	Kubernetes logo	23
11	Kubernetes Architecture	24
12	Example of a K8s Deployment	27
13	Accessing a pod using a K8s service	27
14	k8s-logo	29
15	OSM Service Platform View	30
16	OSM Architecture	31
17	OSM-K8s deployment example	32
18	Helm Chart's structure example	33
19	Openstack architecture	34
20	OpenStack testbed	36
21	OSM version and Pods needed to run OSM v8	37
22	OSM Dashboard	38
23	OSM Dashboard: Overview	38
24	OSM Dashboard: Registered VIM's section	39
25	OSM Dashboard: Openstack account details	39
26	OSM Dashboard: Instantiating a NS based on VNFs	40
27	OSM dashboard: NS status	41
28	Openstack Network Topology of a NS with 4 VNFs	41
29	Openstack instances section	42
30	OSM K8s testbed	43
31	Adding a dummy VIM to OSM	44
32	Registered K8s Clusters in OSM	44
33	K8s repositories registered in OSM	45
34	Example of the structure of two Helm Charts	45
35	Example of a KNFD, written in .yaml	46
36	OSM Dashboard: Instantiating a NS based on KNFs	47
37	OpenStack testbed	48
38	NS based on 4 KNFs from K8s point of view	48
39	Instantiation of a NS based on KNF Sequence diagram	49
40	OSM Dashboard: helm-related error	50
41	CirrOs' docker image	52
42	Deployment Process Delay (DPD) of a NS varying the number of VNFs/KNFs	52
43	CPU Usage of a NS varying the number of VNFs/KNFs	53
44	RAM Usage of a NS varying the number of VNFs/KNFs	54

List of Acronyms and Abbreviations

3GPP	3rd Generation Partnership Project
5GC	5G Core
5GPPP	5G Public Private Partnership
API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
CLI	Command Line Interface
CN	Cloud Native
CNCF	Cloud Native Computing Foundation
CNF	Cloud Native Virtual Network Function
COE	Container Orchestration Engine
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DPD	Deployment Process Delay
DevOps	Development and Operations
E/NMS	Element/Network Management System
E2E	End-to-End
EC	European Commission
ECM	Ericsson Cloud Manager
EM	Element Management
ETSI	European Telecommunications Standards Institute
HA	High Availability
IM	Information Model
IMT	International Mobile Telecommunications
IP	Internet Protocol
ISG	Industry Specification Group
K8s	Kubernetes
KDU	Kubernetes Deployment Unit
KNF	Kubernetes-based Virtual Network Function
KNFD	Kubernetes Network Function Descriptor
KPI	Key Performance Indicators
LCM	Life Cycle Manager
LF	Linux Foundation
MANO	NFV Management and Orchestration
NBI	Unified Northbound Interface
NF	Network Function
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NFVO	NFV Orchestrator
NSD	Network Service Descriptor

NSI	Network Slice
NSO	Network Service Orchestrator
ONAP	Open Network Automation Platform
OS	Operating System
OSM	Open Source Mano
OSS/BSS	Operation and Business Support Systems
RAM	Random Access Memory
RO	Resource Orchestrator
SBA	Service Based Architecture
SDN	Software Defined Networking
SDS	Software Defined Storage
SF	Service Functions
SFC	Service Function Chaining
URLCC	Ultra-Reliable and Low Latency Communications
VCA	VNF Configuration and Abstraction
VDU	Virtual Deployment Unit
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFD	Virtual Network Function Descriptor
VNFFG	VNF Forwarding Graph
VNFM	VNF Manager
Web UI	Web User Interface
eMBB	Enhanced Mobile Broadband
eSBA	Enhanced Service Based Architecture
mMTC	Massive Machine-Type Communications

Revision history and approval record

Revision	Date	Purpose
0	23/09/2020	Document creation
1	13/11/2020	Inputs by Pouria
2	30/11/2020	Inputs by Xavier
3	18/12/2020	Inputs by Pouria
4	08/01/2021	Inputs by Xavier
5	25/01/2021	Last inputs by Xavier

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Adrián Pino Martínez	adrian.pino@i2cat.net
Pouria Sayyad Khodashenas	pouria.khodashenas@i2cat.net
Muhammad Shuaib Siddiqui	shuaib.siddiqui@i2cat.net
Xavier Hesselbach Serra	xavier.hesselbach@upc.edu

Written by:		Reviewed and approved by:	
Date	25/01/2021	Date	28/01/2021
Name	Adrián Pino Martínez	Name	Muhammad Shuaib Siddiqui
Position	Project Author	Position	Project Supervisor

Abstract

Adopting Cloud Native (CN) into 5G telecommunications systems has been identified as a good candidate for reducing the cost, improving system agility and role out of 5G services. This is well reflected on the recent 3GPP standardization activities. Taking the cue from 3GPP, European Telecommunications Standards Institute (ETSI) has published NFV referenced architecture to adapt for the CN and enhancement to NFV framework to include Zero-Touch, Containers, Load Balancers and more as part of the reference architecture. The aim of this work is to validate container technology in ETSI-hosted MANO platform, Open Source Mano (OSM) in a CN environment. In order to validate the performance while assessing the limits, advantages and drawbacks, KNFs behaviour is benchmarked using a OSM-standalone K8s' testbed compared with an OSM-OpenStack, where VNFs are deployed. The results obtained in this work can help to further encourage users and operators the use of KNFs and get the most out of containerization in NFV.

Key words: NFV, 5G, Cloud Native, OSM, K8s

Acknowledgments

This work was supported by the European Union's H2020 research and innovation programme under the CAMEL project (Grant agreement No. 833611) and the Spanish national project ONOFRE 2 (TEC2017-84423-C3-1-P).

1 Introduction

The aim of this first section is to present the main scope and motivation of this work. A structure of the work is provided as well, where our work plan and a Gantt diagram of the project are included. To conclude the section, we briefly comment some issues encountered.

1.1 Motivation

Due to the evolution in virtualization technologies and networking, a new paradigm has arrived in the telco domain, the Cloud Native telco. This means a radical new approach to NFV, badly needed to meet 5G verticals. The only feasible approach for the Cloud Native Telco is to offer an evolution of Virtual Network Functions (VNFs) (e.g: a Firewall, a DHCP server, or a router) running in VMs towards Kubernetes-based Network Functions (KNFs), running in containers. It mirrors how enterprises are moving their monoliths based on VMs to containers orchestrated by Kubernetes and then (often slowly) refactoring them into microservices.

Following ETSI NFV, which is pushing for OSM as the open source MANO (OSM) system (acting as NFVO and VNFM), and taking into account its recent support through containers (via K8s), we decided to study this new NFV architecture, K8s along with OSM, which allows us to benefit from the advantages of container technology.

As explained in this research work, OSM can work with K8s following two approaches. The first one managing containers using a VIM between OSM and K8s, such as OpenStack (this means using containers inside at least a VM) reaching an OSM-OpenStack-K8s architecture. The second one follows a pure CN scenario, where OSM directly talks with K8s, and NS are instantiated via KNFs in containers.

The first solution has the disadvantage that in order to use KNFs it is necessary to have an OpenStack operating cluster, which for some users may be far, due to resources or lack of expertise in this complex cloud computing platform. In contrast, in the second solution, a user can start working with KNFs without having this OpenStack knowledge. By working only with OSM and K8s, developers and users could work with KNFs in a more direct way, favoring a faster and more efficient advancement of container-based NS.

At the beginning of this research, in the official doc. of OSM, some tutorials and videos, guide users to implement the OSM-OpenStack-K8s architecture. Nevertheless, the second approach had just a couple of notes on how to implement such a scenario. Bearing in mind that OSM is an open source initiative, we decided to explore this approach and validate it with this research.

Thus, this research aims to validate and study Kubernetes-based Virtual Network Functions (KNFs) in a pure Cloud Native (CN) scenario, and hence validate the OSM-K8s scenario. In order to validate the performance while assessing the limits, advantages and drawbacks, KNFs behaviour is benchmarked using a OSM-K8s' testbed compared with an OSM-OpenStack, where VNFs are used.

1.2 Structure of the work

1.2.1 Work plan

The work is structured in the following manner:

- In Section 2, an overview of the state of the art of both Cloud Native and 5G is provided. The intention of this section is to give an overview of the context needed to understand the work described later on. The current challenges and enablers for Cloud Native and 5G are presented, describing in deep the ones with special relation with the present work.
- Section 3 focuses in the current industry de facto standard container orchestrator, Kubernetes. Its architecture along with its main components are described.
- The object of Section 4 is to explain the NFV network model that we have studied on this research. The reasoning that we have followed to choose OSM and Openstack/K8s as our main components is exposed.
- In Section 5 we present the experimentation part of our work. The chosen configuration is provided along with a detail analysis of the one based on K8s, (the main focus of this research). A subsection which conveys some of our observations regarding the K8s testbed is included.
- Section 6 aims to show the achieved results in the experimental part of the work. To do so, different KPIs are used to compare and benchmark the pure Cloud Native scenario that has been set versus the classical solution based on VMs.
- Finally, Section 7 summarizes the main conclusions of the work as well as future research lines.

1.2.2 Gantt diagram

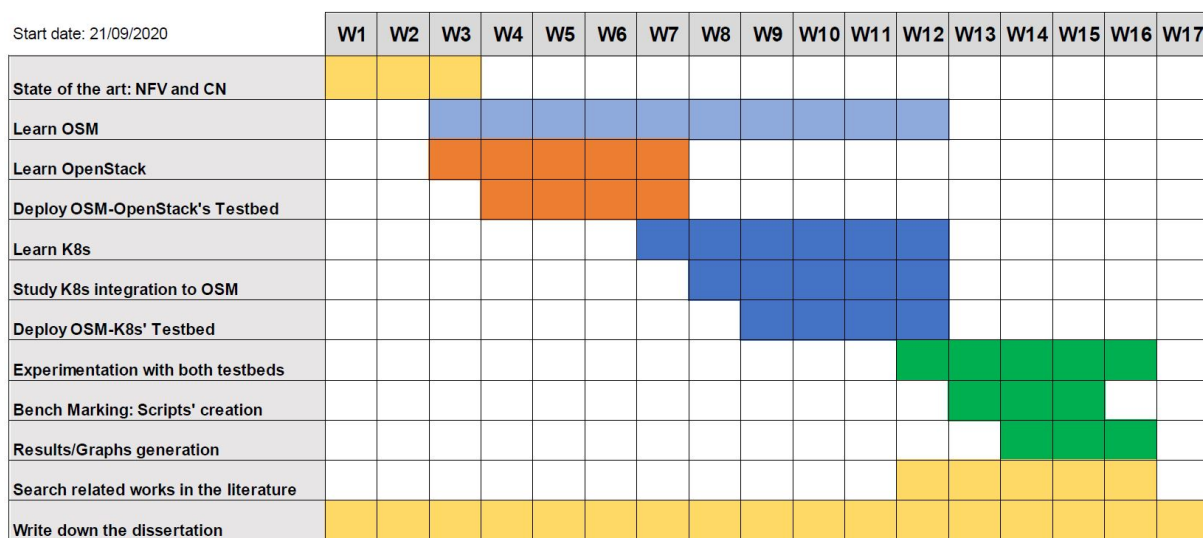


Figure 1: Gantt diagram of the project

1.3 Incidents and Modifications

To the best of our knowledge, this is the first time that a validation of standalone K8s integrated with OSM is provided.

Having said that, at the moment of this research, OSM (via its official documentation) did provide just some notes on how to start implementing such scenario. Hence, some of the difficulties faced during the development of this work rely on the fact that in some phases we were in a blocking phase, where we had to perform a back and forth communication with some OSM's members via e-mail and via its official slack channel (as OSM is a rich Open Source community, users help each other via this channel) and we could finally solve some of the issues encountered. Currently, they have enriched the documentation on this subject, fixing some of the gaps that we found.

Apart from that, due to limited resources (especially the OpenStack cluster), our experimentation had to be done with light images, but the results can be extrapolated without major problems.

2 Cloud Native in the telco sector: State of the art

The aim of this section is to provide an overview on the Cloud Native (CN) concept and its application in the telco world. It will be examined how Cloud and Telcos are merging in what we can call the new "Cloud Native 5G paradigm".

We will study how concepts such as network softwarization, virtualization, microservices, orchestration, services architectures, infrastructure-as-code, automation and continuous integration/delivery pipelines are affecting 5G ecosystem and changing the scene.

To conclude this section, ETSI NFV vision along with its Architectural Framework will be reviewed, as well as an analysis of the evolution of Network Functions (NF) in the telco industry.

2.1 Cloud Native and 5G previous background

The main endeavour from the IT industry has been boosting development agility and cost efficiency of services. Two main enablers for this goal are virtualization and softwarization, where a given functionality runs in software instead of hardware, aiming to lower the cost and service operation, reducing the time to market for new services and introducing higher flexibility.

Cloud Native's journey began with the introduction of virtualization technologies (Xen Hypervisor, Intel VT-x, KVM hypervisor, Linux Containers, and etc.) which rapidly were adopted by the IT industry (vmware, AWS, Openstack, etc.). The wide-spread adoption of virtualization paradigms in the data center has been a main driver for the great evolution we have experienced over the past years. Containers, as an important tool for cost reduction and improved Agility, came to play with the introduction of Docker, leading to huge architectural and philosophic changes (Microservices, DevOps). Figure 2 illustrates the evolution of the technology as what we know as "The Cloud", since the beginning until what we have reached now: The Cloud Native paradigm.

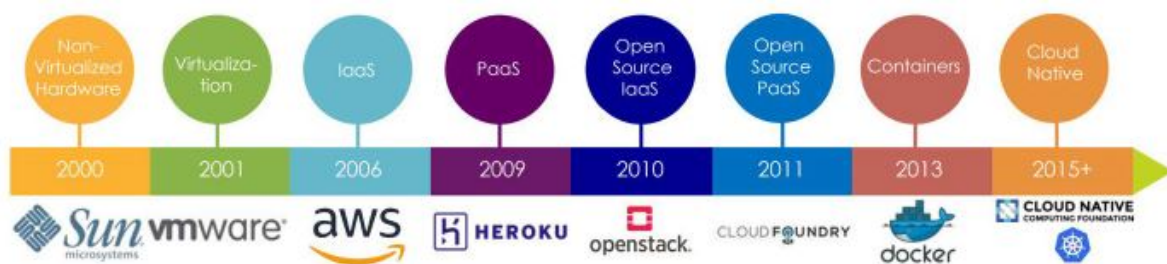


Figure 2: Cloud Native's journey, [1]

Motivated by the benefits of the data centres evaluation, the telecommunications industry started to experiencing its own modernization journey.

On the one hand, boosted by the publication in 2012 of "NFV – Introductory White Paper" at the SDN and OpenFlow World Congress in Darmstadt-Germany, a new and disruptive reference architecture was defined, [2].

This architecture was based on Network Function Virtualization (NFV) along with complementing Software Defined Networking (SDN) technology and the adoption of an All-Internet Protocol (IP) transport network, making the telecommunications industry to adopt cloud technologies, transforming the network and building a new ecosystem. These concepts will be studied with more detail in section 2.3.1.

On the other hand, the fifth generation of communication systems (5G) where various verticals, e.g. transport, health, media and entertainment, industry and automotive, are expected to be enabled by the 5th Generation mobile network. The advanced communications of 5G will bring massive machine-type communications (mMTC), enhanced mobile broadband (eMBB), and, ultra-reliable and low latency communications (URLLC), which correspond to the 5G usage scenarios defined by ITU-R, [3] on International Mobile Telecommunications (IMT) 2020 and beyond (IMT-2020).

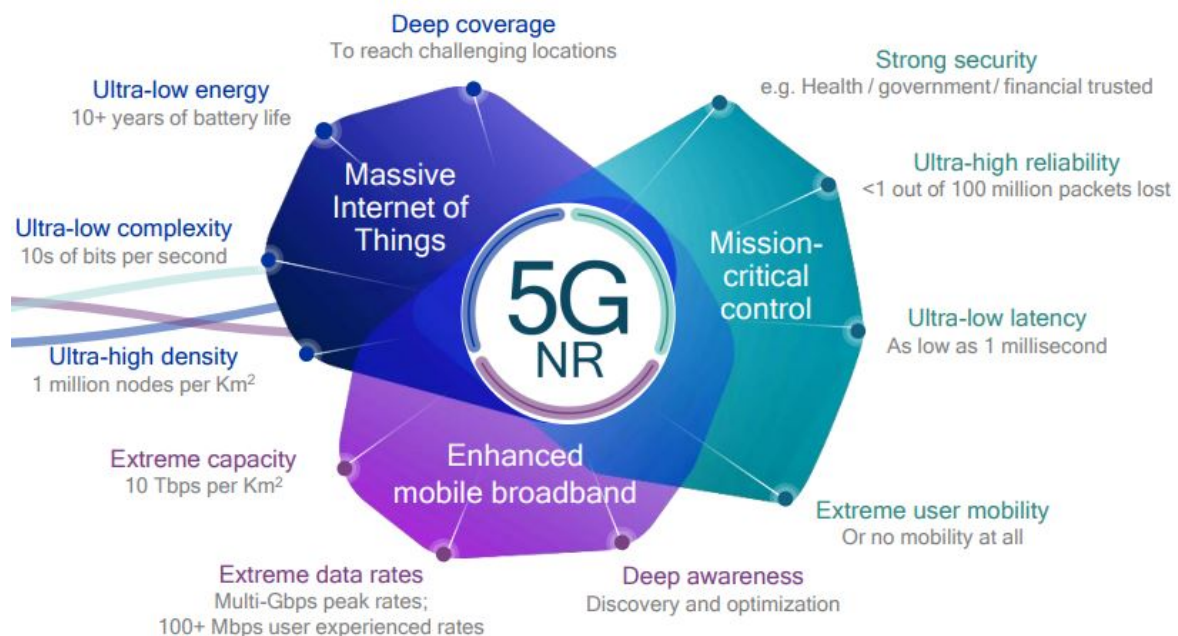


Figure 3: Scalability to address diverse service and devices, [4]

5G enables significantly more capacity and higher user data rates than today's capabilities, so as to meet the growing demands of users. In addition, an important goal of 5G is to provide increased resilience, continuity, and much higher resource efficiency including a significant decrease in energy consumption.

The 5G Key Performance Indicators (KPIs), such as 1,000 capacity, 10 to 100X higher typical user data rate, better/increased/ubiquitous coverage, service creation in minutes, and end-to-end (E2E) latency of < 1 ms, are summarized by the 5G Public Private Partnership (5G PPP), the European leading association on 5G [12], and are represented in Figure 3.

It is important to highlight that no single technology alone will be able to meet all 5G KPIs listed above, and of course not all of these requirements will be required for every 5G single application. The general understanding is that 5G is the network of networks.

Having said that, **adopting Cloud Native into 5G telecommunications systems have been identified as a good candidate for reducing the cost, improving system agility and role out of 5G services.** This is well reflected on the recent 3GPP standardization activities. The 3rd Generation Partnership Project is an umbrella term for a number of standards organizations which develop standards for mobile telecommunications. The proposed Service Based Architecture (SBA) in Release-15 (Rel-15) and the enhancement on Service Based Architecture (eSBA) in Rel-16 (which extends the service concept from the 5GC (5G Core) control plane to the user plane function) are two good examples of this action.

Taking the cue from 3GPP, European Telecommunications Standards Institute (ETSI) has published NFV referenced architecture to adapt for the Cloud Native and enhancement to NFV framework to include Zero-Touch, Containers, Load Balancers and more as part of the reference architecture [5].

In the rest of this section, we will focus more on the definition of cloud native and we will provide some insight about the challenges that has to be addressed.

2.2 Cloud Native: challenges and enablers

Cloud Native (CN), is the name of the approach to design, build and run applications/virtual functions that fully exploit the benefits of the cloud computing model. The CN approach refers to the way applications are created and deployed, not where they are executed, [6].

It is based on the principle of decomposing an application into a set of microservices that can be developed and deployed independently to accelerate and optimize the DevOps(a set of practices that combines software development (Dev) and IT operations (Ops)) life cycle, [7], of software systems. The microservices are packaged into light-weight containers which are scheduled to run on compute nodes by a container orchestrator. Figure 4 shows the main pillars of cloud native.

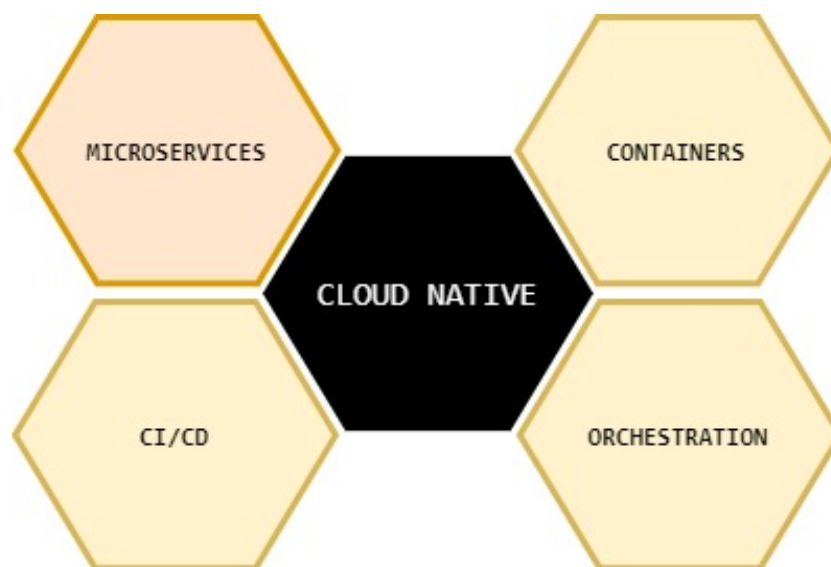


Figure 4: Cloud Native pillars.

The Cloud Native (CN) approach is a huge philosophical and cultural shift as well as a technical challenge. CN technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. With it, engineers are able to make high-impact changes frequently and predictably with minimal toil [8].

To boost the benefits of CN, several associations and foundations have been formed. The most important one is the **Cloud Native Computing Foundation (CNCf)**, which is an open source software foundation under the umbrella of Linux Foundation (LF), that is pushing to container technology, aligning the tech industry around its evolution.

CNCf provides structure and constraints for Cloud Native, promoting that the applications developed and architected are designed in microservices and deployed as containers. To be properly classified as Cloud Native, microservices need to be “stateless” meaning there has to be separation of processing from the associated data and the storage of the data in Cloud.

Orchestration, meaning the automated configuration, coordination, and management of computer systems and software, plays a critical role in CN. As the de facto solution, **Kubernetes**, (the CNCf-hosted Open Source container orchestrator) plays a crucial role in actively scheduling and optimizing resource utilization while providing observability, resiliency and an immutable infrastructure, [9].

To understand better the role of orchestration, we need to think that although it is true that Cloud Native today is based on microservices, this type of architecture has not always been followed. The classic solution was to follow a monolithic architecture. Breaking the monolithic applications into microservices can also lead to challenges in developing, deploying and operating many more entities. As a result, DevOps along with automation tools, i.e. orchestration, is needed for managing the increased complexity.

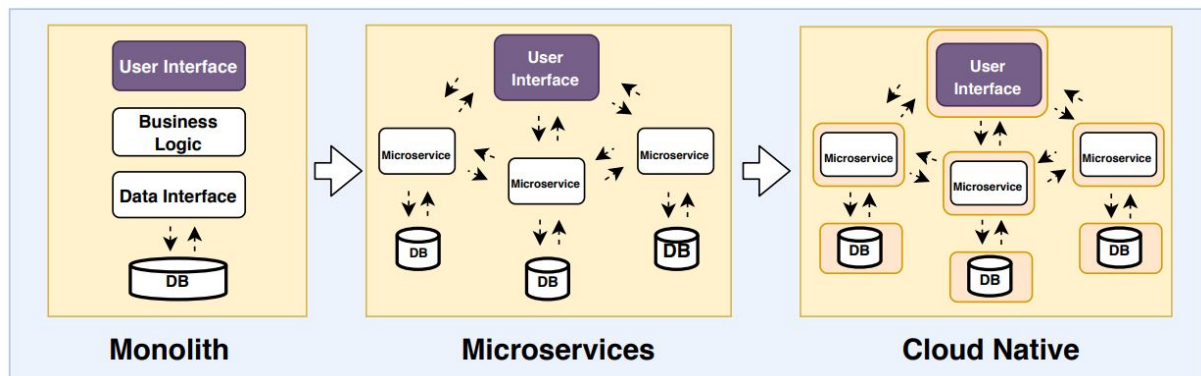


Figure 5: Architecture evolution to Cloud Native.

Microservices are an increasingly popular architecture for building large-scale applications. Rather than using a single, monolithic codebase, applications are broken down into a collection of smaller components called microservices, as it can be seen in figure 5. When architecting following microservices principles, all the functions are deployed individually and with clear interfaces and dependencies. Each microservice, must provide well-defined REST architectural style.

This approach offers several benefits, including the ability to scale individual microservices, keep the codebase easier to understand and test, and enable the use of different programming languages, databases, and other tools for each microservice. This again verifies the need of having strong orchestration tool to automate the overall management of the system.

As mentioned before, **one of the key enablers of cloud native is containerization**, which is a relatively new type of virtualization. Both virtual machines (the classic solution) and containers can help to realize network softwarization and to make the most of available resources. Containers are gaining tremendous popularity today, but virtual machines have been, and continue to be, used in data centers of all sizes.

A virtual machine (VM) is an emulation of a computer system. Simply, it makes it possible to run what appear to be many separate computers only on a single hardware.

The operating systems (OS) and their applications share hardware resources from a single host server, or from a pool of host servers. Each VM requires its own underlying OS which runs over a virtualized resource. A hypervisor, or a virtual machine monitor, is software, firmware, or hardware that creates and runs VMs. It sits between the hardware and the virtual machine and is necessary to virtualize the server.

VMs, however, can take up a lot of system resources. Each VM runs not just a full copy of an OS, but a virtual copy of all the hardware that the operating system needs to run, as we can see in Figure 6. This quickly adds up to a lot of RAM and CPU cycles. That is still economical better when compared to running separate actual computers. However, efficiency can still be improved, as for some applications it can be overkill, which led to the development of containers.

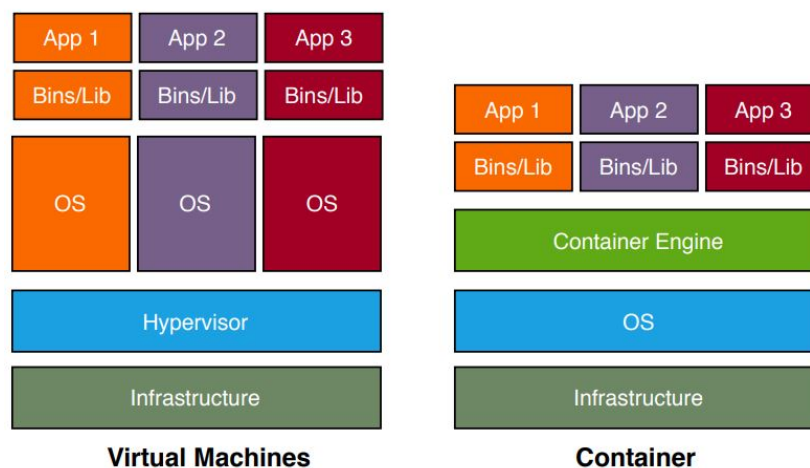


Figure 6: VM and container architecture.

A **container** is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A container image is a lightweight, standalone, executable package of software that includes everything

needed to run an application: code, runtime, system tools, system libraries and settings. The lifecycle of a container is managed by what is commonly called a container runtime.

While virtual machines brings abstraction to the hardware, containers bring abstraction to the operating system. Thus, container are more efficient because each container shares the host OS kernel and, usually, the binaries and libraries, too. This makes containers lighter weight than virtual machines where every instance has its own OS, binaries and libraries as we mention in the previously. Lastly, it is important to mention that containers only use the hardware resources they need at the running time, so there is no reservation of resources as in the case of VMs.

In this case, why containers have not been used before? The answer relies on its complexity. In 2013 an open-source software to develop and deliver software in containers is released, namely **Docker**.

Docker is the tool for managing and deploying microservices in containers. Each microservice can be further broken down into processes running in separate containers, which can be specified with Dockerfiles and Docker-Compose configuration files. Combined with an orchestration tool such as Kubernetes, each microservice can be then easily deployed, scaled, and collaborated on by a developer team. Specifying an environment in this way also makes it easy to link microservices together to form a larger application.

Before closing this part, it is worth clarifying that the last pillar of CN, (Figure 4), **Continuous Integration and Continuous Delivery (CI/CD)** is out of the scope of this work. Nevertheless, CI/CD pipelines are vital in DevOps; the technical goal of CI is to establish a consistent and automated way to build, package, and test applications. Lastly, CD automates the delivery of applications to selected infrastructure environments. More info can be found here, [10].

2.3 SDN, NFV and ETSI NFV MANO

2.3.1 SDN and NFV

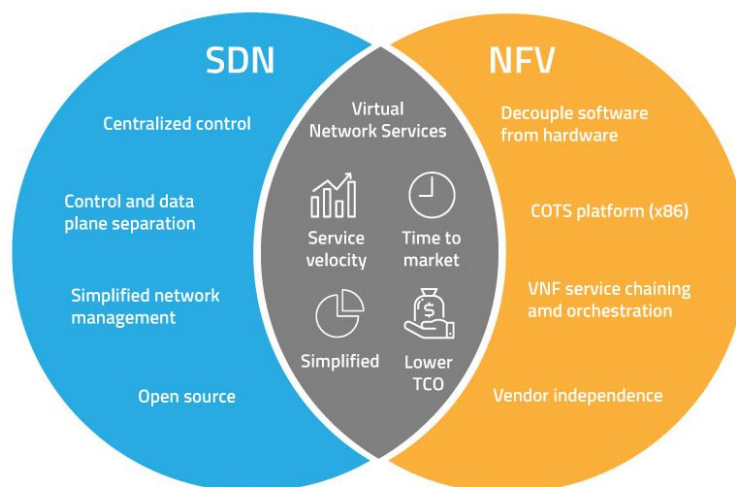


Figure 7: SDN and NFV concept applicability, [11]

While networks have been moving towards greater virtualization, with the true decoupling of the control and forwarding planes, as advocated by Software-Defined Networking (SDN) and Network Functions Virtualization (NFV), that network virtualization has become more of a focus.

Software-Defined Networking (SDN) is an emerging network architecture based on the idea of decoupling the control and data planes; SDN exploits a logically centralized network controller, which works in the control plane, handling the allocation of traffic to network elements in an isolated data plane.

Network Function Virtualization (NFV) foresees to implement the network function (networking connectivity and services) of a network device in a software package running in virtual machine(s) / container(s), to allow for network functions being decoupled from the hardware. The NFV introduces flexibility and allows quick installation/re-configuration of network functions by simply installing/upgrading software package(s).

The mentioned SDN and NFV technologies are mutually beneficial but are not dependent on one another. However, the reality is SDN makes NFV more compelling and vice-versa, as we can see in Figure 7. Usually, SDN provides network automation that enables policy-based decisions to orchestrate network traffic (L2-3), while NFV focuses on the services, ensuring the network's capabilities are align with the virtualized environment (L4-7 services including firewalling and server load balancing).

2.3.2 NFV Architectural framework

In 2012, the NFV Industry Specification Group (ISG) was founded by seven telco network operators within the European Telecommunication Standard Institute (ETSI). The creation of ETSI NFV ISG led to the foundation of NFV's basic requirements and architecture.

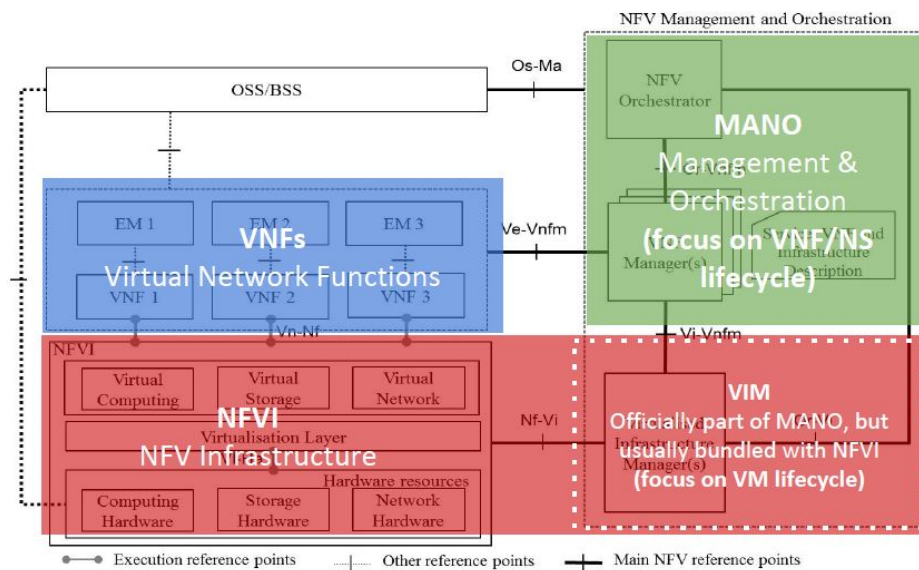


Figure 8: ETSI NFV Architectural Framework [14]

The NFV Architectural framework, [13], represented in Figure 8, classifies the functional blocks and the main reference points between those blocks. In this architectural framework there is not specified any detailed implementation. The main execution reference points are represented with solid lines, which are in the scope of NFV. By dotted lines are represented the other reference points, which are available in current deployments but require extension to handle NFV.

It is important to highlight that, as previously mentioned, **this architectural framework do not specify any detailed implementation**. Thus, many combinations of different blocks can be formed; for instance, on the one hand, VMware vCloud Director, OpenStack, and OpenVIM are some examples of VIMs. On the other hand, ONAP (Open Network Automation Platform), OSM (Open Source MANO) and Cloudify MANO are some open source MANO's systems. Finally, some vendors' proprietary MANO's implementations are ECM (Ericsson Cloud Manager), and CloudBand (Nokia).

From the last figure, the main functional blocks are the following:

- **Virtualised Network Functions (VNF)**. Implementation of an NF that can be deployed on a Network Function Virtualisation Infrastructure (NFVI). Multiple VMs can host a single component of the VNF or the whole VNF can be deployed in a single VM. Some examples of VNFs include NAT, DNS and firewalls. The Element Management (EM), is the block which performs the management of one or more VNFs.

As we will explain in section 2.4, due to the evolution of the technology with the arrival of CN in the world of telcos, a new concept has emerged: CNF.

A **Cloud Native Virtual Network Function (CNF)** is a network function designed and implemented to run inside containers. CNFs inherit all cloud native architectural and operational principles including K8s lifecycle management, agility, resilience, and observability. Therefore, we can notice that the natural evolution will come from VNFs to CNFs.

- **NFV Infrastructure (NFVI)**. Totality of all hardware and software components which build up the environment in which VNFs are deployed, managed and executed. A virtualization layer above the physical resources is created to abstract the hardware resources. From the VNF's perspective, the virtualisation layer and the hardware resources look like a single entity providing the VNF with desired virtualised resources.

In the NFVI we can include the **Virtual Infrastructure Manager (VIM)**, whose concept will be reviewed later (Officially part of MANO but usually bundled with NFVI, as it focus on VM lifecycle). The NFV Infrastructure can span across several locations (multiple/different VIMs), [13].

- **NFV Management and Orchestration (MANO)**. Manages the Network Service (NS) lifecycle and coordinates the management of VNF lifecycle (supported by the VNFM, whose concept will be reviewed later as well) and NFVI resources (supported by the VIM) to ensure an optimized allocation of the necessary resources and connectivity. [15].
- **Operation and Business Support Systems (OSS/BSS)**. It stands to the Operations Support Systems and Business Support Systems of an Operator.

2.3.3 ETSI NFV MANO

NFV Management and Orchestration (MANO) is the architectural framework for the management and orchestration of all resources in the cloud defined by ETSI. It is one of the main blocks in the ETSI NFV architectural framework, as we saw in Figure 8. It has the role to manage the NFVI and orchestrate the allocation of resources needed by the VNFs and the NSs (The chain of a set of VNFs creates a NS).

In fact, NFV MANO (in specific, NFVO-VIM relation) is the main focus of this research. As we will see later on in the next sections, two testbeds sharing the same NFV MANO System (Open Source MANO) with different VIMs (Openstack and K8s) will be analyzed in order to validate and compare NSs based on VNFs and CNFs (called KNFs as well).

ETSI NFV MANO is broken up into three functional blocks:

- **NFV Orchestrator (NFVO):**
 - On-boarding of new Network Service (NS), VNF Forwarding Graph (VNF-FG) and VNF Packages
 - NS lifecycle management (including instantiation, scale-out/in, performance measurements, event correlation, termination)
 - Global resource management, validation, and authorization of network functions virtualization infrastructure (NFVI) resources by engaging with the VIMs directly through their north bound APIs
 - Policy management for NS instances
 - Creates end to end service among different VNFs
 - Open Source MANO (OSM) is the de facto NFVO fixed by ETSI MANO standards.
- **VNF Manager(VNFM):**
 - Lifecycle management of VNFs: creates, maintains and terminates VNF instances installed on the VMs or containers
 - Overall coordination and adaptation role for configuration and event reporting between NFVI and the traditional element / network management system (E/NMS)
- **Virtualized Infrastructure Manager (VIM):**
 - Manages life cycle of virtual resources in one NFVI domain
 - Controlling and managing the NFVI compute, storage and network resources, within one operator's infrastructure sub-domain: Creates, maintains and tears down VMs/-containers from physical resources in an NFVI domain
 - Collection and forwarding of performance measurements and events
 - OpenStack is the de facto VIM adopted by the market.
 - With the rise of CNFs, K8s is gaining importance as VIM.

2.4 Evolution of Network Functions (NF): from VNF to CNF

In order to understand the current status and to try to foresee the possible future evolution, it is worth to understand the cloud solution that has been widely adopted on the market in recent years, taking into account both the weaknesses of this framework and the evolution of virtualization technologies. 5GPPP's "Cloud Native and 5G Verticals' services" white paper, [8], conveys the point of view of the European Commission (EC) and the industry in a very insightful manner.

In figure 9, we can observe the evolution from the classic solution based on VNF implemented to run inside VMs, to a new architecture based on CNF implemented inside. As we can see, this shift is a process that cannot be done instantly, instead, a gradual evolution must take place. Therefore, we have reached a period of coexistence between VMs and containers that is going to last few years at least.

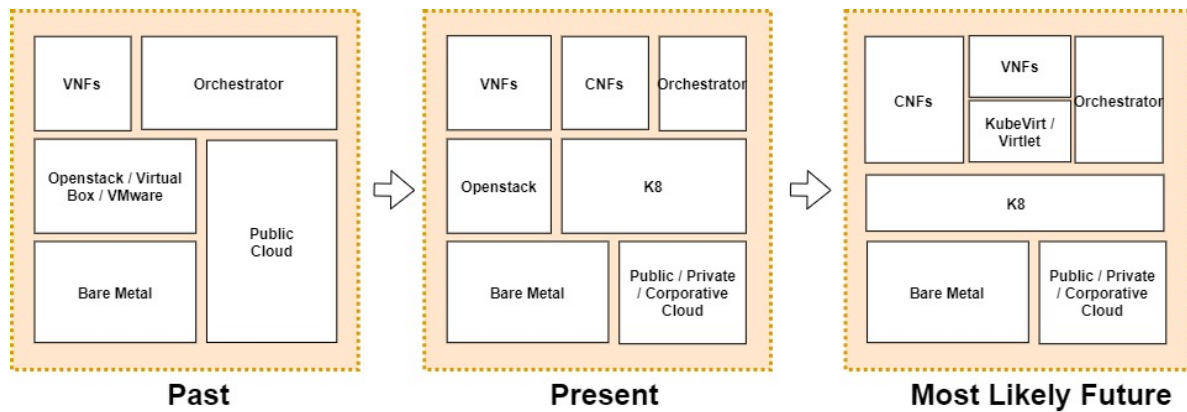


Figure 9: Evolution of the Virtualization Solutions used in Telcos.

2.4.1 Past

The classic solution is based on running VMs on top of Bare Metal/Public Cloud. AWS, Microsoft Azure, or Google Cloud are some of the most important public cloud providers. Hypervisors such as VMware or VirtualBox have been widely adopted. Openstack has been used as the de facto cloud computing platform. The architectural framework that has been adopted in the Telcom sector follows the NFV MANO which, as stated before, is based on:

- Virtual Infrastructure Manager (VIM) e.g. OpenStack
- Network Function Virtualisation Orchestration (NFVO) e.g. OSM
- Virtual Network Function Manager (VNFM) e.g. OSM

Concerns of this framework

- First of all, In Multi-domain orchestration environments (very common in 5G services), managing several VIMs (e.g. Openstack) in a multi-cloud environment is a complex and hard task. Therefore, this a problem that needs to be addressed.

- Secondly, it exists a problem to manage multiple VNFs in a consistent way as well. We are facing the hard dependency 1:1 between hardware and element managed system, that exist in physical world.
- Thirdly, at implementation level, it is also hard to combine different blocks from different vendors.

These concerns can be solved if we move forward into a Cloud Native solution, based in a VIM such as Kubernetes. This involves an evolution neither easy nor instantaneous from VNFs to CNFs.

2.4.2 Current status

The only feasible approach for the Cloud Native Telco is to offer an evolution of VNFs towards CNFs (Cloud Native Network Function), likewise called KNFs (Kubernetes Network Functions). To do so, containers are a great approach to utilize. It mirrors how enterprises are moving their monoliths to Kubernetes and then (often slowly) refactoring them into microservices. For this to be economic, there need to be incremental gains in resiliency, scheduling, networking, and development velocity as more network functions become Cloud Native.

It is important to understand that the fact of moving from VMs to Containers is a process that cannot be done instantly, instead, a gradual evolution must take place. Besides, many legacy solutions as well cannot be implemented with K8s, so we have reached a period of coexistence between VMs and containers, as we can see in Figure 3, that is going to last few years at least.

From the telco point of view, most of the current prototypes and projects are based on a pure Openstack ecosystem and are including the capability of running Kubernetes along with Openstack. It is important to clarify that the intelligence is still going to be centralized in the VNFM.

2.4.3 Possible future approach

The most likely future approach is based entirely on K8s (No longer place to Openstack). To address uses cases where VMs are a must (where some VNFs have not been ported to CNFs yet or cannot be ported), some software such as KuberVirt or Virtlet could be used on top of K8, as we can see in Figure 9.

It is reasonable to assume that Cloud Native capabilities will be explored and leveraged via technologies such as Kubernetes while legacy will continue playing an important role. Hereunder are several topics that need to be explored.

The question here is whether or not to set Kubernetes as 5G default orchestrator. To ensure virtualization technology neutrality and portability across different use cases and future environments, Kubernetes is used as an industry de facto standard container orchestrator. Kubernetes can be either deployed on the bare metal servers or on top of some virtualization technology.

Some predict Kubernetes can become the operating system for 5G networks. Indeed, thanks to its flexibility, such an orchestration engine will accelerate the adoption of container network functions (CNF) by operators.

3 Kubernetes

The goal of this section is to get the flavour of Kubernetes, the current industry de facto standard container orchestrator. Its definition will be given, as well as its architecture and main components, not without first highlighting the need for this type of tool in a container-based architecture.

3.1 Why use Container Orchestrators?

Although we can manually maintain a couple of containers or write scripts for dozens of containers, in a complex scenario with multiple services and multiple containers:

- Containers communication is neither direct nor easy (Use of Docker-compose tool)
- Scaling is a difficult task
- Setting up of services is a manually and complex task
- Manual work of fixing if a node crashed
- Distributing traffic was challenging

Therefore, container orchestrators are essential for effective management of the infrastructure. Among them, Kubernetes has become the de facto standard, as previously mentioned.

3.2 What is Kubernetes?

Kubernetes is an open-source container orchestration tool which automates container deployment, container scaling and container load balancing, grouping containers into logical units.

Kubernetes is also referred to as K8s, as there are 8 characters between k and s.



Figure 10: Kubernetes logo.

3.3 K8s features

K8s main features can be grouped in the following way:

- **Automatic bin packing.** Kubernetes automatically schedules containers based on resource needs and constraints, to maximize utilization without sacrificing availability.
- **Self-healing.** Kubernetes automatically replaces and reschedules containers from failed nodes. It kills and restarts containers unresponsive to health checks, based on existing rules/policy. It also prevents traffic from being routed to unresponsive containers.

- **Horizontal scaling.** With Kubernetes applications are scaled manually or automatically based on CPU or custom metrics utilization
- **Service discovery and Load balancing.** Containers receive their own IP addresses from Kubernetes, while it assigns a single Domain Name System (DNS) name to a set of containers to aid in load-balancing requests across the containers of the set
- **Automated rollouts and rollbacks.** Kubernetes seamlessly rolls out and rolls back application updates and configuration changes, constantly monitoring the application's health to prevent any downtime
- **Secret and configuration management.** Kubernetes manages secrets and configuration details for an application separately from the container image, in order to avoid a re-build of the respective image. Secrets consist of confidential information passed to the application without revealing the sensitive content to the stack configuration, like on GitHub
- **Storage orchestration.** Kubernetes automatically mounts software-defined storage (SDS) solutions to containers from local storage, external cloud providers, or network storage systems.
- **Batch execution.** Kubernetes supports batch execution, long-running jobs, and replaces failed containers

3.4 K8s architecture

K8s follows a master-worker architecture, where the control plane is managed by the master. The worker machines, called nodes, run the containerized applications. Therefore we have a clear separation between control plane and the workload in the data plane.

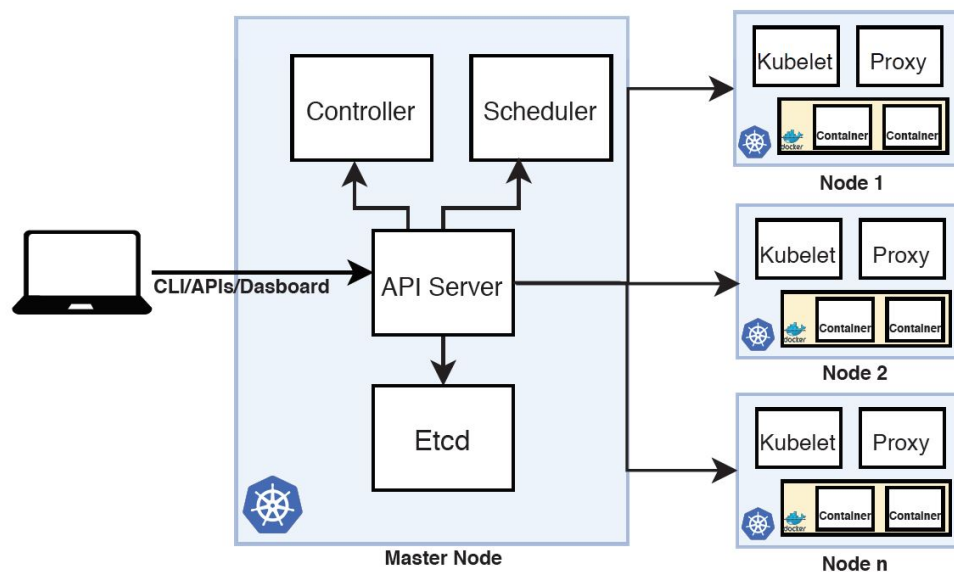


Figure 11: Kubernetes architecture.

In Figure 11, we present a diagram of a Kubernetes cluster with its components tied together. Each of the components is reviewed below. As we can notice, we can access the K8s API Server via the Dashboard, via kubectl (the Kubernetes CLI) and directly accessing the REST API.

Different configurations can be deployed; single-master single-node, single-master multi-node, multi-master multi-node (High Availability (HA) scenarios), or even an all-in-one single-node (testing purposes).

3.4.1 K8s main components

As a Kubernetes cluster can be deployed in different ways, to better explain its components we are going to divide them in two groups regarding its domain; master node and worker node components.

Master Node Components.

The master node provides a running environment for the control plane responsible for managing the state of a Kubernetes cluster, and it is the brain behind all operations inside the cluster. The control plane components are agents with very distinct roles in the cluster's management. In order to communicate with the Kubernetes cluster, users send requests to the master node via a Command Line Interface (CLI) tool, a Web User-Interface (Web UI) Dashboard, or Application Programming Interface (API).

A master node has the following components: **API server, scheduler, controller managers, etcd.**

- All the administrative tasks are coordinated by the **kube-apiserver**, a central control plane component running on the master node. The API server intercepts RESTful calls from users, operators and external agents, then validates and processes them.
- The role of the **kube-scheduler** is to assign new objects, such as pods, to nodes. During the scheduling process, decisions are made based on current Kubernetes cluster state and new object's requirements.
- The **controller** managers are control plane components on the master node running controllers to regulate the state of the Kubernetes cluster. Controllers are watch-loops continuously running and comparing the cluster's desired state (provided by objects' configuration data) with its current state (obtained from etcd data store via the API server). In case of a mismatch corrective action is taken in the cluster until its current state matches the desired state.
- To persist the Kubernetes cluster's state, all cluster configuration data is saved to **etcd**. However, etcd is a distributed key-value store which only holds cluster state related data, no client workload data.

This components can be distributed among different master-nodes, useful in a HA scenario, or can be located in the same machine in a more straight-forward one.

Worker Node Components

A worker node provides a running environment for client applications. Though containerized microservices, these applications are encapsulated in Pods (smallest unit in K8s), controlled by the cluster control plane agents running on the master node. Pods are scheduled on worker nodes, where they find required compute, memory and storage resources to run, and networking to talk to each other and the outside world.

A worker node has the following components: **Container runtime, kubelet, kube-proxy. To conclude, addons are presented.**

- Although Kubernetes is described as a "container orchestration engine", it does not have the capability to directly handle containers. In order to run and manage a container's lifecycle, Kubernetes requires a **container runtime** on the node where a Pod and its containers are to be scheduled. Kubernetes supports many container runtimes: Docker, containerd, and more.
- The **kubelet** is an agent running on each node and communicates with the control plane components from the master node. It receives Pod definitions, primarily from the API server, and interacts with the container runtime on the node to run containers associated with the Pod. It also monitors the health of the Pod's running containers.
- The **kube-proxy** is the network agent which runs on each node responsible for dynamic updates and maintenance of all networking rules on the node. It abstracts the details of Pods networking and forwards connection requests to Pods.
- **Addons** are cluster features and functionalities not yet available in Kubernetes, therefore implemented through 3rd-party pods and services. Some examples are DNS, Dashboard, monitoring and logging addons.

3.4.2 K8s main building blocks

To add a bit of more context, the main building blocks in K8s are defined below.

- **Pods.** As defined earlier, there are the smallest and simplest K8s object. It is the unit of deployment in Kubernetes, which represents a single instance of the application. A Pod is a logical collection of one or more containers, which are scheduled together on the same host, share the same network namespace, have access to mount the same external storage (volumes).
- **Labels.** Key-value pairs attached to Kubernetes objects. Labels are used to organize and select a subset of objects, based on the requirements in place. Many objects can have the same Label(s). Labels do not provide uniqueness to objects. Controllers use Labels to logically group together decoupled objects, rather than using objects' names or IDs.
- **ReplicaSets.** Help to scale the number of Pods running a specific container application image. Scaling can be accomplished manually or through the use of an autoscaler.
- **Label Selectors.** Controllers use Label Selectors to select a subset of objects.
- **Deployments.** K8s objects that include Pods and ReplicaSets. Deployments allow seamless application updates and downgrades through rollouts and rollbacks. They are directly

managed in the Deployment Controller (part of the master node's controller manager in the master node), to ensure that the current state always matches the desired state.

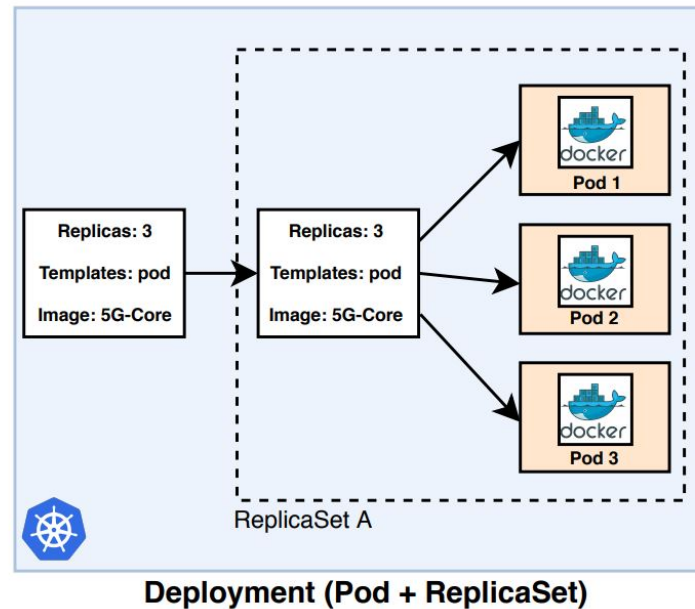


Figure 12: Example of a K8s Deployment

3.4.3 K8s Services

K8s services are the abstract way to expose an application running on a set of pods as a network service. Kubernetes services give Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

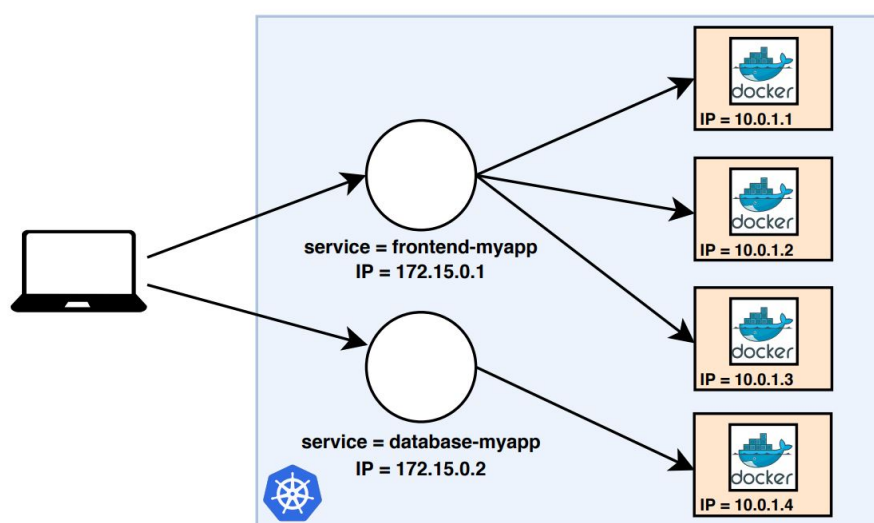


Figure 13: Accessing a pod using a K8s service

The motivation of using them because of the volatile nature of containers; e.g: if some set of pods (backends) provide functionality to other pods (frontends) inside a cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload? The answer is using K8s services.

3.5 Alternatives to K8s

As mentioned before, Kubernetes is the current de facto standard container orchestrator engine (COE). Of course, is not the only option available in the market. According to CNCF's Cloud Native Landscape, there are more than 109 tools to manage containers, but 89% are using different forms of Kubernetes, [20].

Next we are going to quick recap the two important alternatives to K8s:

- **Docker Swarm.** Open source solution coming from Docker's team. Thus, any software, services, or tools that run with Docker containers, has full support in Swarm. It uses the same command line from Docker.

Docker Swarm offers a simple solution that is quick to get started with. Simpler to use and to install than K8s, works fine with developers aiming fast deployments and simplicity. Not recommended in production environments.

- **Mesos.** Open source software that provides efficient resource isolation and sharing across distributed applications or frameworks. Mesos architecture is based on a master-slave approach, where daemons run task on the slaves. At a first glance could look like K8s, but its complexity is greater.

It works very well with large systems and it is designed for maximum redundancy. Highly recommended for companies that use multi-cloud and multi-region clusters as well. Mesos is recommended existing workloads are being used such as Hadoop, Kafka etc. Interestingly, Mesos is currently being adapted to add a lot of the K8s concepts and to support the K8s API. So it will be a gateway to getting more capabilities for the K8s applications.

- **Openshift.** OpenShift is a family of containerization software products developed by Red Hat. Its flagship product is the OpenShift Container Platform, an on-premises platform as a service built around Docker containers orchestrated and managed by Kubernetes on a foundation of Red Hat Enterprise Linux, [21].

It has stronger security policies than Kubernetes, but limited to Red Hat Linux distributions. Talking about CI/CD, neither platform provides a full CI/CD solution. Both tools need to be integrated with extra tools. Nevertheless, this integration with OpenShift could be easier, as it offers a certified Jenkins container available to run the CI tasks, acting as CI server.

4 Our NFV Implementation

The purpose of this section is to explain the NFV network model, following the architecture by the ETSI, that we have studied on this research. To start with, we will briefly explain the scenario from a high level, where we can find the pieces of the ETSI NFV puzzle. Later on, we will explain why we have chosen each of the blocks.

Firstly, and as we can see in Figure 14, we find **Open Source MANO (OSM)**, acting as the NFVO Orchestrator, which will perform the deployment of network services (NS) and virtual network functions (VNFs). This tool, also acts as the virtual network function manager (VNFM), monitoring the life cycle and management of virtualized network functions.

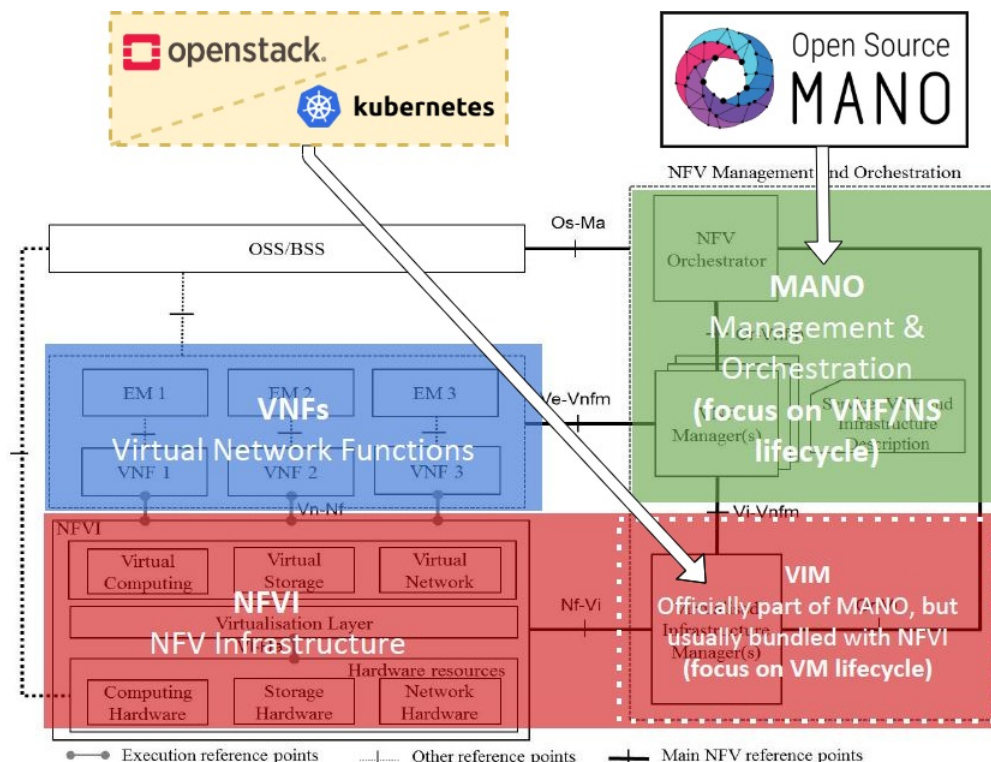


Figure 14: Our NFV Implementation

The virtual infrastructure manager (VIM) component will be implemented through two different ways. In Figure 14 we can find on the one hand, **Openstack**; the classic and widely solution adopted by the industry. On the other hand, **Kubernetes**; the new sheriff in town. Both components will control and manage the computing resources, storage and network resources (NFVI). Although the main focus of this research is to validate KNFs (through K8s), we decided to compare and benchmark KNFs against VNFs (through Openstack).

Just to give a bit more of context, it is worth to highlight that Network Services (NS) can be formed by an operator using one or more service functions (SF), following the so-called ETSI Service Function Chaining (SFC) architecture. To do so, VNF forwarding graphs are needed (VNFFG). This tasks can be specified in a MANO's system such as OSM.

4.1 Open Source MANO as NFVO and VNFM

Open Source MANO (OSM) is an ETSI-hosted initiative to develop an **Open Source NFV Management and Orchestration (MANO)** software stack aligned with ETSI NFV. OSM's goal is the development of a E2E Network Service Orchestrator (E2E NSO) driven by an open source community with production quality for telco services, capable of modelling and automating real telco-grade services, with all the intrinsic complexity of production environments, [22].

As mentioned earlier, **OSM acts as the NFVO Orchestrator**, which performs the deployment of network services (NS) and virtual network functions (VNFs). This tool **also acts as the virtual network function manager (VNFM)**, monitoring the life cycle and management of virtualized network functions. Additional layers, such as service orchestration are also required for operators to enable true NFV services. Therefore, OSM is a key tool, because it contains two of the key components of the ETSI NFV architecture and strictly follows the ETSI NFV stack. In Figure 15 we can find OSM's Service Platform View. An interesting point to highlight is that a NS can span across the different domains identified (virtual, physical and transport).

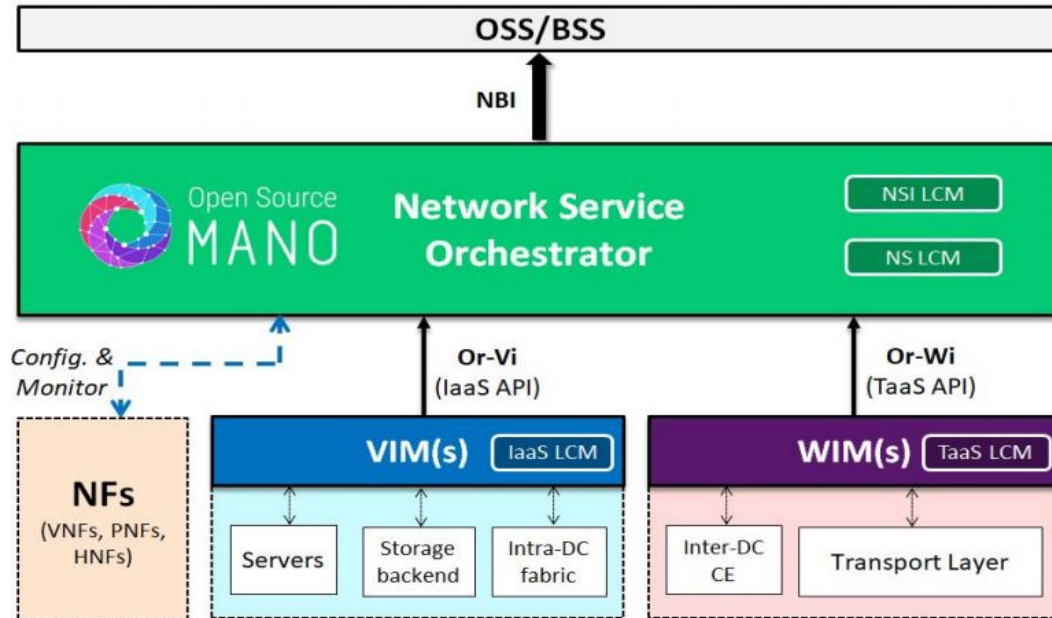


Figure 15: OSM Service Platform View, [23]

At the moment of this work, OSM is under Release EIGHT, which is the result of more than 4 years of evolution. Since Release SEVEN, **OSM started to support Kubernetes-based Network Functions (KNFs) orchestration** (very limited support at this point), by leveraging existing or new kubernetes clusters. This was a major milestone in an industry that considers Kubernetes as the most likelihood underlying infrastructure for next generation 5G services, as we mentioned in Section 2.4.3.

Release EIGHT brings a set of new features allowing to improve the orchestration of diverse virtualization environments, including PNFs, a number of different VIMs for VNFs, and Kubernetes for KNFs, [24].

4.1.1 OSM features

The key features of OSM are the following:

- An **Information Model (IM)**, which is capable of modelling and automating the full life-cycle of Network Functions, Network Services (NS) and Network Slices (NSI). OSM's IM is aligned with ETSI NFV, and is completely infrastructure agnostic. Thus, OSM can be used indistinctly no matter what VIM or transport technologies are being used.
- An **Unified Northbound Interface (NBI)**, based on NFV SOL005.
- The extended concept of "Network Service" in OSM, so that an NS can span across the different domains identified (virtual, physical and transport), easily noticeable in Fig 15.
- **OSM can also manage the lifecycle of Network Slices**, assuming when required the role of Slice Manager, extending it also to support an integrated operation.

4.1.2 OSM architecture

Next, we are going to present OSM's architecture. In Figure 16 we have the main blocks of its architecture. As we the different blocks communicate between them via a kafka bus. These blocks are mapped into kubernetes pods (if we have installed OSM using the K8s flavour).

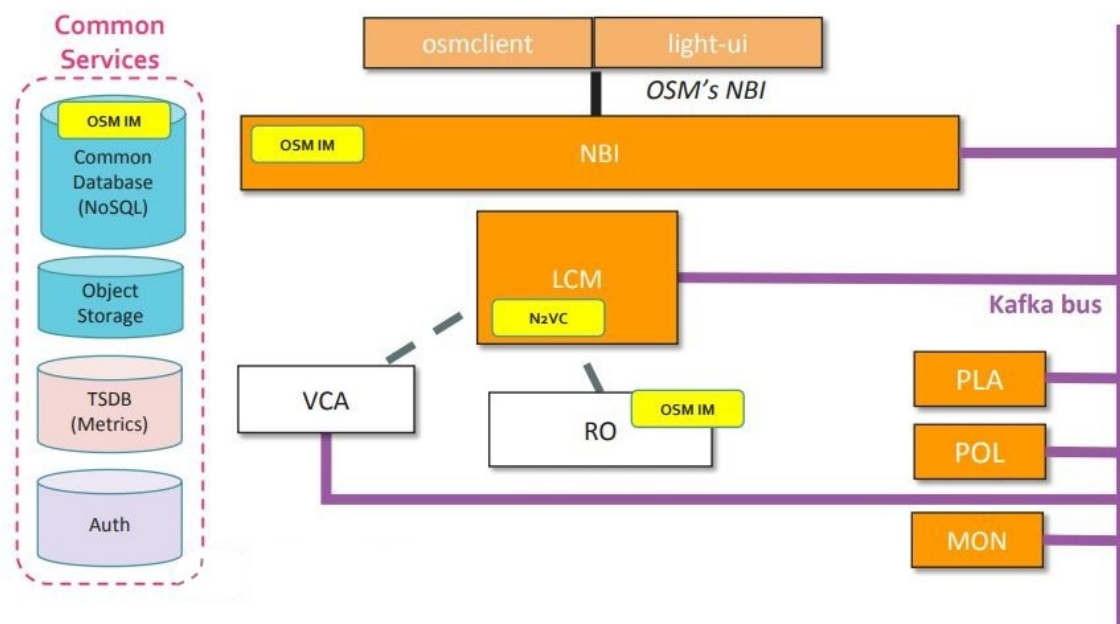


Figure 16: OSM Architecture, [23]

We could say that between LCM (Life Cycle Manager), RO (Resource Orchestrator) and VCA (VNF Configuration and Abstraction) the most part of the tasks are performed. At the left part we can find common services; the common database, the object storage, tsdb and auth which are mapped into mongodb, mysql, prometheus and keystone containers. Lastly, POL and MON are containers dedicated to monitoring, and PLA is for automated placement.

4.1.3 OSM and K8s

If we want to take advantage of container technology in network functions (via KNFs) in OSM, the way to proceed is through Kubernetes. As we have said along this document, K8 and the world of telecommunications have a promising relationship that is improving over the years. In fact, containerization is a must in many 5G Verticals (e.g: Automotive and Smart Cities).

Having said that, and as mentioned in Section 2.4, the evolution from VNFs to CNFs is neither easy nor instantaneous. Adopting container-based VIMs, such as K8s, to instantiate KNFs, is a challenging task under research (In fact, is the main goal of this work).

KNFs feature requires an operative K8s cluster. There are several ways to have that K8s running. From the OSM perspective, the K8s cluster is not an isolated element, but it is a technology that enables the deployment of microservices in a CN way, [25].

For OSM there are two modes of represent a K8s cluster:

- **K8s connected to a VIM:** Following this mode there are two possible approaches: on the one hand a K8s cluster running on VMs inside the VIM, where all VMs are connected to the VIM network. On the other hand, a K8s cluster running on baremetal and it is physically connected to the VIM network. In both approaches, we can deploy VNFs based in VMs and KNFs based in containers, giving high flexibility to the Operators.
- **K8S isolated cluster:** In this case the communication is directly between OSM and K8s without any VIM in the equation. **In fact, we consider that K8s can be considered a VIM itself, due to its functionality.** The drawback of this scenario is that, a priori, only KNFs can be deployed (not able to work in hybrid scenarios).

In Figure 17, an example of a deployment of a NS based on KNFs is presented. Furthermore, OSM's main operations and K8s' lifecycle operations supported by OSM are listed.

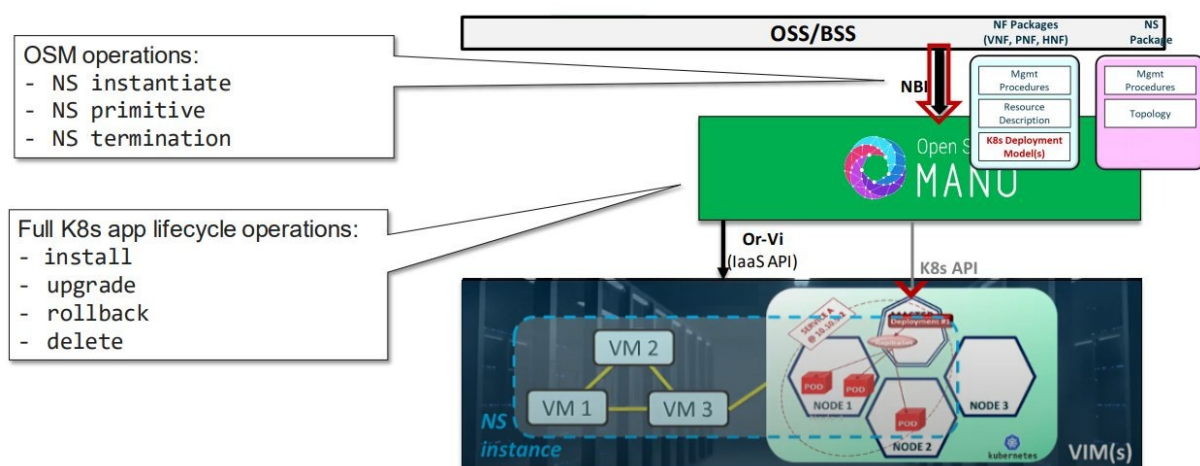


Figure 17: OSM-K8s deployment example, [26]

To on-board KNF services (hence, to deploy K8s apps), there are two ways to proceed:

- **Helm Charts.** Helm is a tool that streamlines installing and managing Kubernetes applications, widely used in DevOps (in specific, is a deployment tool used in CI/CD) to ensure automation, [27]. Hence, in a complex K8s application, we avoid to instantiate one by one the different pods, services, configurations and so on. Helm uses a packaging format called charts. A helm chart is a collection of files that describe a related set of Kubernetes resources, [28].

Helm charts in OSM imply an indirect call to the K8s API via helm.

- **Juju Bundles.** Packaged format as well, but implies an indirect call to the K8s API via Juju. Fairly powerful for inter-object configurations.

With Helm, any application (no matter how complex it can be) can be packaged in a helm-chart and deployed with simple a command. As we will explain later, is how with we have deployed KNFs.

In Figure 18, we present an example of the structure of a Helm chart that deploys an app that has three components (microservices); database tier, backend tier and frontend tier, each one with different components, configurations and images.

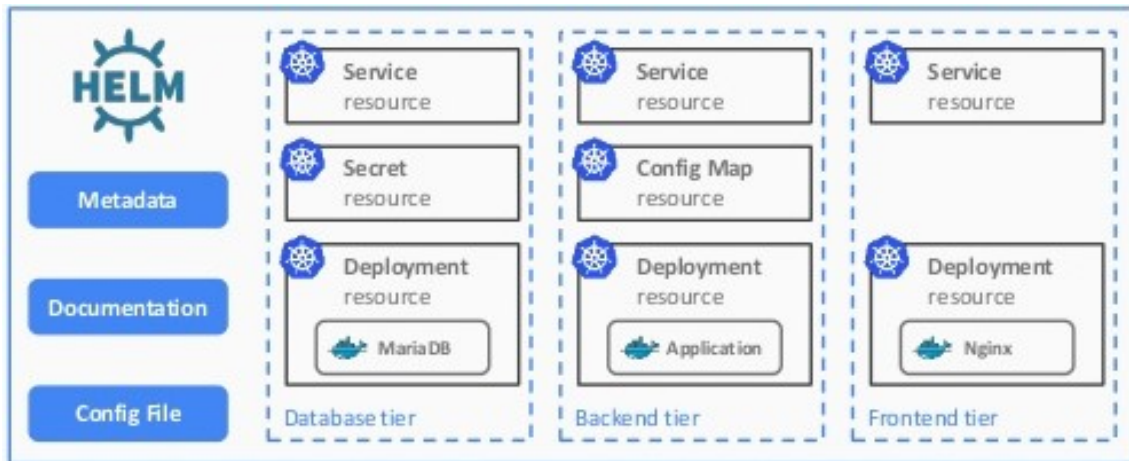


Figure 18: Example of the structure of a Helm-Chart. [27]

4.2 Openstack as VIM

Although Openstack is not the main focus of this research, we decided to use it as VIM in order to create a testbed that allow us to work with VNFs, to benchmark and compare KNFs and VNFs. Next, we are going to briefly describe what is Openstack and its main components.

Widely adopted in the IT industry, Openstack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed and provisioned through APIs with common authentication mechanisms. It can be accessed through command client or through its dashboard, giving administrators control while empowering their users to provision resources through a web interface, [29].

Beyond standard infrastructure-as-a-service functionality, additional components provide orchestration, fault management and service management amongst other services to ensure high availability of user applications.

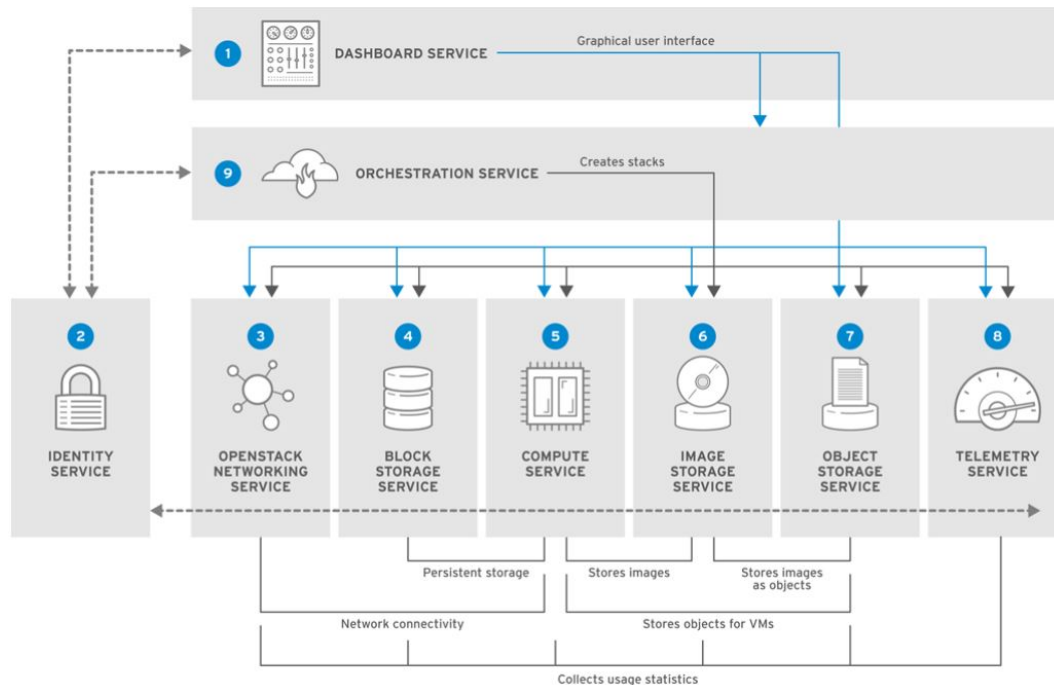


Figure 19: Openstack architecture, [30]

As seen in Figure 19, Openstack's main components are the following:

- **Dashboard: Horizon.** Web browser-based dashboard that you use to manage Openstack services.
- **Identity: Keystone.** Centralized service for authentication and authorization of Openstack services and for managing users, projects, and roles.
- **OpenStack Networking: Neutron.** Provides connectivity between the interfaces of Openstack services.
- **Block Storage: Cinder.** Manages persistent block storage volumes for virtual machines.
- **Compute: Nova.** Manages and provisions virtual machines running on hypervisor nodes.
- **Image: Glance.** Registry service that you use to store resources such as virtual machine images and volume snapshots.
- **Object Storage: Swift.** Allows users to store and retrieve files and arbitrary data.
- **Telemetry: Ceilometer.** Provides measurements of cloud resources.
- **Orchestration: Heat.** Template-based orchestration engine that supports automatic creation of resource stacks.

4.3 Kubernetes as VIM

As mentioned in Section 4.1.3, there are two approaches to deploy the K8s cluster along with OSM. Firstly, K8s connected to a VIM (e.g: Openstack). Secondly, K8s without this intermediate block, so called K8s isolated cluster/standalone.

However, the aim of this work is to test a pure Cloud-Native scenario, where only KNFs are instantiated (although it is true that later on this pure CN scenario will be compare with an scenario with only VNFs), as it seems to be the most likelihood scenario in the future of the telecommunications, as stated in Section 2.4.3.

Therefore, **in our approach, we have pushed to a K8s isolated cluster**, reachable from OSM. According to OSM's official documentation this corresponds to "the approach of not having a VIM" (Although we consider that K8s can be considered a VIM itself, due to its functionality and capability). Besides, using an intermediate block, from our point of view, breaks the Cloud Native vision. K8s definition, features, and architecture, have been already presented in Section 3.

It is important to remark that OSM at the moment of this research, had not any example or showcase where they show/test the "OSM-K8s isolated cluster" approach. OSM via its official documentation and its Hackfest have show-cased some examples of KNF instantiation. Nevertheless they have done it exclusively following the approach where K8s is connected to a VIM (OSM-VIM-K8s).

Hence, **one of the key points of this research is to validate the OSM-K8s standalone approach.** Likewise, no information regarding how to proceed if we want to use personalized KNFs (via helm-charts) was provided in the official doc, so another point of this research was to solve this issue. (Later on, in Section 5.2, it is explained how we have solved this problem).

5 Experimentation

In this section, an analysis of the two testbeds that we have set is provided. Originally, the scope of our work was only to focus in the OSM-K8s testbed, but in order to proper compare and benchmark this testbed we decided to create another one based on OpenStack.

We will start commenting briefly our **OSM-OpenStack testbed**, the more straight forward to implement, due to its widely adoption and maturity. It is one of the most extended and accepted VIMs by telco industry. As the scope of the work is more focused on the OMS-K8s one, we will not get into much detail in the one using OpenStack.

To continue with, a detailed explanation of the **OSM-K8s testbed** will be provided, along with a sequence diagram which illustrates the steps to be performed to instantiate NS based on KNFs (hence, based on containers as well).

5.1 OSM and OpenStack's Testbed

A schema of this testbed is presented in Figure 20. As we can see, at the left of the image we find OSM v8, which has been installed in a VM under VirtualBox. At the right part, we have an OpenStack-all-in-one cluster (version DevStack), this means that the same machine acts as master and worker (this scenario can be extended using another version of OpenStack and then adding more nodes to the cluster). OpenStack has been installed in a VM under i2Cloud's infrastructure (i2CAT's Cloud Environment). Specifications of installation details and resources needed required by each block, are presented in the table below the schema.

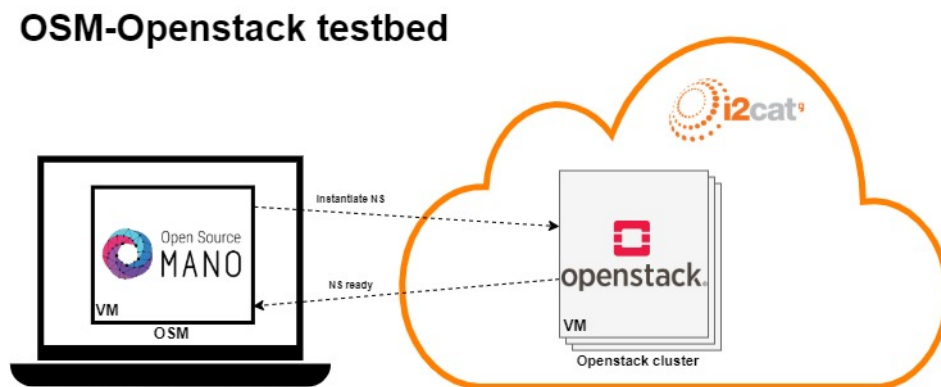


Figure 20: OpenStack testbed

Component	Installation details and resources needed
OSM	OSM v8.0.1, installation based on K8s. Installed in a VM running Ubuntu 18.04 with 2 vCPUS, 6GB of RAM and 60GB of storage
OpenStack	Devstack, v5.3.1. Installed in a VM running Ubuntu 18.04 with 2 vCPUS, 8GB of RAM and 60GB of storage

As said before, we have installed OSM v8 via K8s, which from our experience has a solid and

stable behaviour. The other installation option is based on containers as well, but with a different orchestrator, docker swarm)

In Figure 21 we present a screenshot from the VM where we have installed OSM v8. In this screenshot, we can see both the current OSM version and the different pods needed to implement OSM's functionality, distributed in different namespaces.

From one side, the ones that every single kubernetes cluster needs to run (the ones implementing K8s architecture, previously represented in Section 3.4). These pods, are within the namespace "kube-system". On the other side, a second group of pods within the namespace "osm", implementing OSM's architecture, represented in Figure 16. Lastly, grouped in the namespace "openebs", we have the ones used by OSM to storage.

```

adrian@adrian-osm:~$ osm version
Server version: 8.0.1+g9ed90df.dirty 2020-07-01
Client version: 8.0.1+gc33aecf.dirty
adrian@adrian-osm:~$
adrian@adrian-osm:~$ kubectl get pods --all-namespaces

```

NAMESPACE	NAME	READY
controller-933929c3-bbbd-4aa5-83e0-da15355c63e1	controller-0	2/2
kube-system	coredns-5c98db65d4-ptngh	1/1
kube-system	coredns-5c98db65d4-t9pl6	1/1
kube-system	etcd-adrian-osm	1/1
kube-system	kube-apiserver-adrian-osm	1/1
kube-system	kube-controller-manager-adrian-osm	1/1
kube-system	kube-flannel-ds-kznf7	1/1
kube-system	kube-proxy-vtw6l	1/1
kube-system	kube-scheduler-adrian-osm	1/1
kube-system	tiller-deploy-659c6788f5-67v4j	1/1
openebs	maya-apiserver-6866b7b55-v6l8f	1/1
openebs	openebs-admission-server-77d489c57d-krtfp	1/1
openebs	openebs-localpv-provisioner-8447b496c7-fsnmm	1/1
openebs	openebs-ndm-h7tsk	1/1
openebs	openebs-ndm-operator-f64c5bb7-58pnf	1/1
openebs	openebs-provisioner-557d489db8-nssgf	1/1
openebs	openebs-snapshot-operator-6d48b74fdb-jxp5j	2/2
osm	grafana-755979fb8c-ff8vp	2/2
osm	kafka-0	1/1
osm	keystone-bff5d9bcf-9zsdt	1/1
osm	lcm-5c88dcd5-dhpcj	1/1
osm	light-ui-58b444dfdf-gmfj5	1/1
osm	mon-66856dfb7f-x4nwl	1/1
osm	mongo-0	1/1
osm	mysql-0	1/1
osm	nbi-7bfbf4dfb-pl475	1/1
osm	pol-7b4968cfd9-q5rvp	1/1
osm	prometheus-0	1/1
osm	ro-57b9b777cb-4tgsq	1/1
osm	zookeeper-0	1/1

Figure 21: OSM version and Pods needed to run OSM v8

To follow with, in Figures 22, and 23 we can find two screenshots taken from OSM's dashboard. The first is the logging menu, and the second one is the overview section. In this last one, in the left column we find the different sections. At the right a quick recap of the onboarded NS and VNF packages, and its running instances.

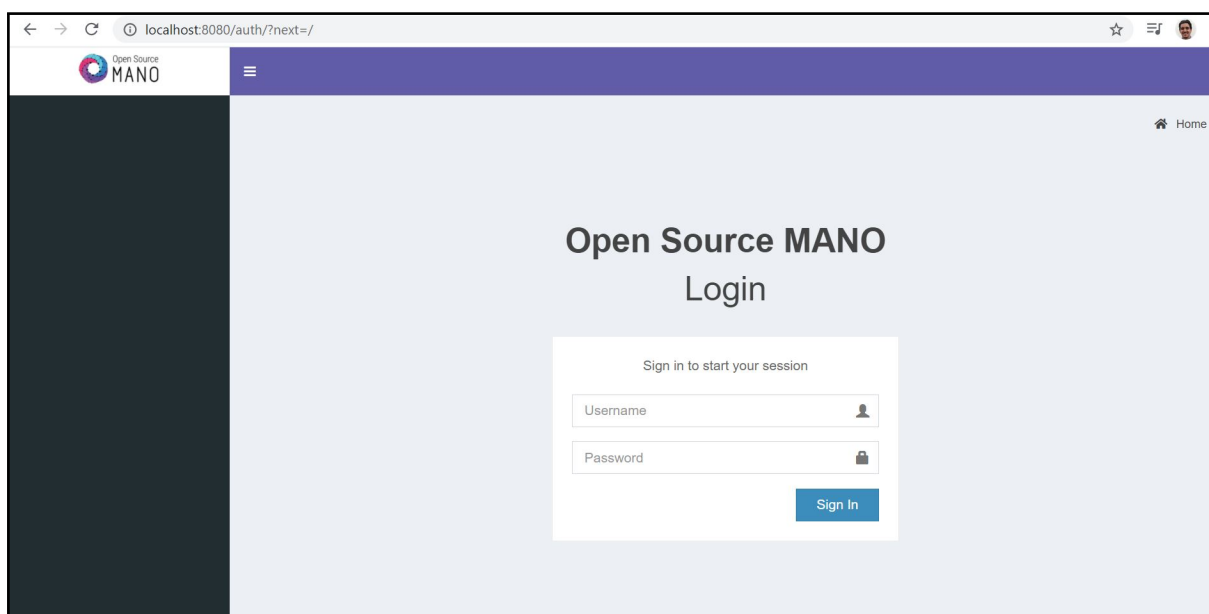


Figure 22: OSM Dashboard

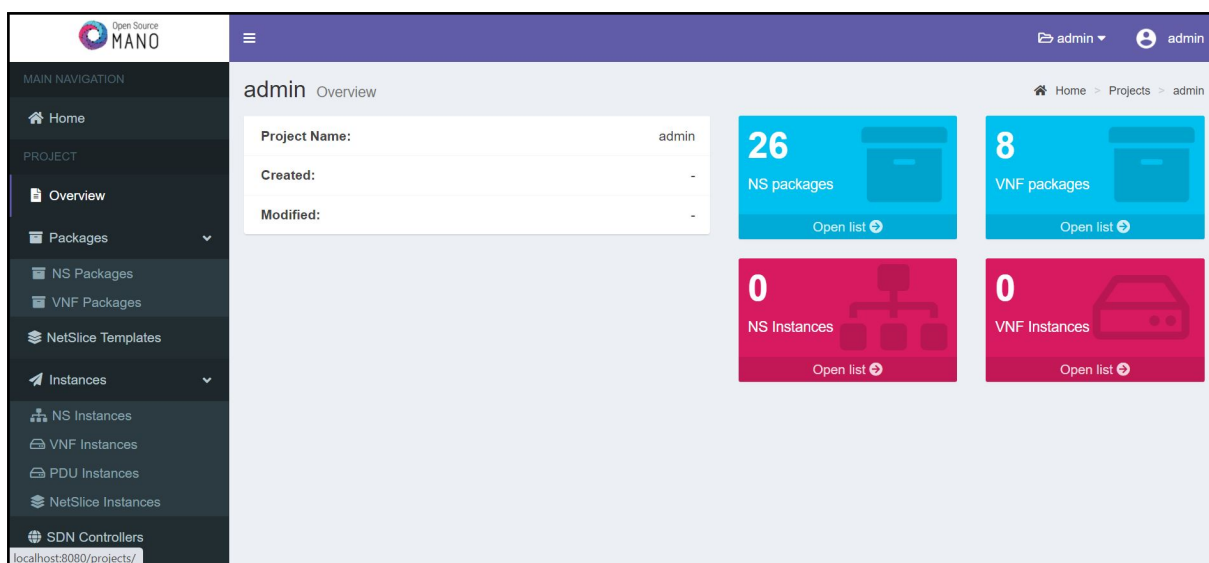
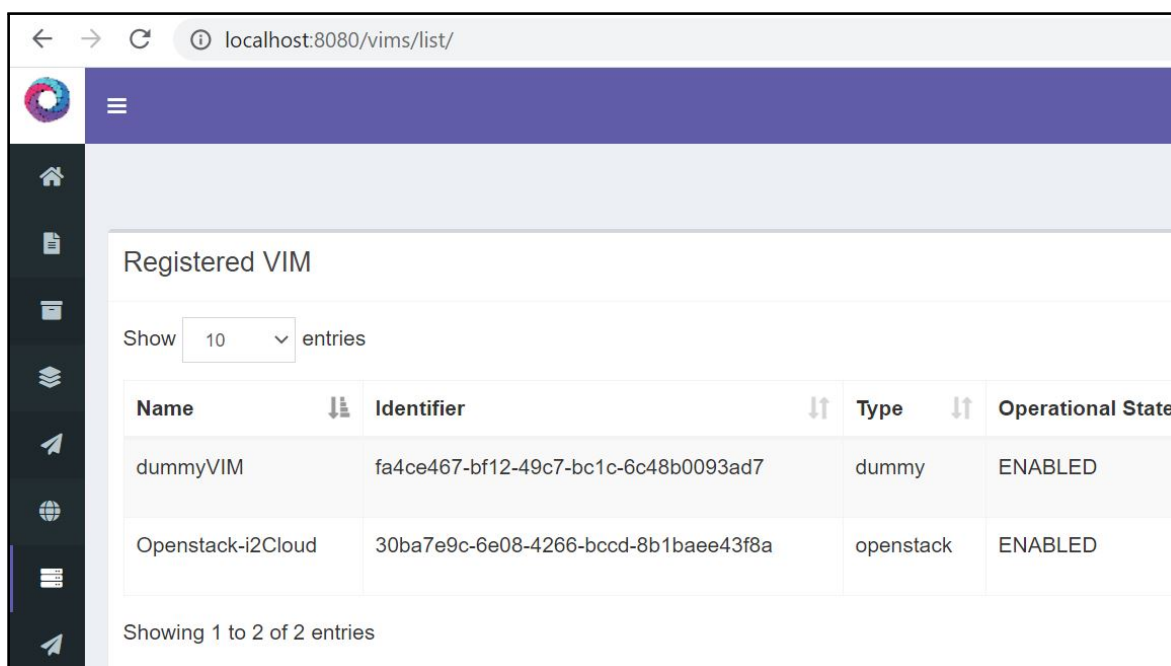


Figure 23: OSM Dashboard: Overview

To continue with, we are going to show the section where we can see the VIMs already registered to OSM. As we can see in Figure 24, we have already two VIMs added, the one used in this testbed (Openstack-i2Cloud), and the one used in the second one (which as we will explain later needs a special VIM with dummy values).



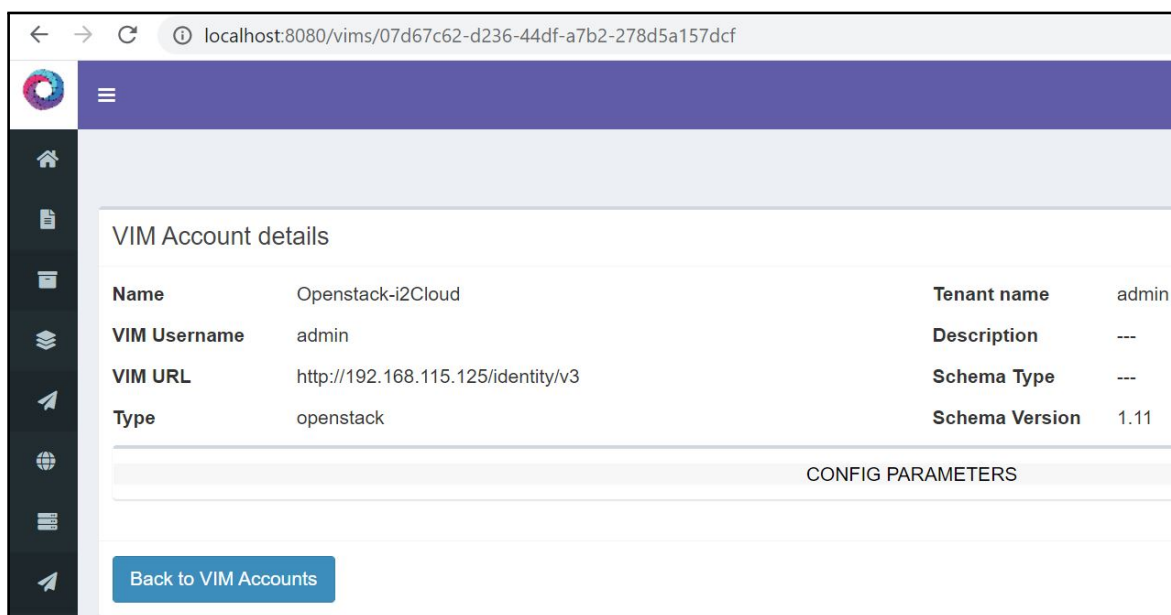
The screenshot shows the OSM Dashboard interface. The browser address bar displays 'localhost:8080/vims/list/'. The dashboard has a dark sidebar with navigation icons. The main content area is titled 'Registered VIM' and includes a 'Show 10 entries' dropdown. Below this is a table with the following data:

Name	Identifier	Type	Operational State
dummyVIM	fa4ce467-bf12-49c7-bc1c-6c48b0093ad7	dummy	ENABLED
Openstack-i2Cloud	30ba7e9c-6e08-4266-bccd-8b1baee43f8a	openstack	ENABLED

Showing 1 to 2 of 2 entries

Figure 24: OSM Dashboard: Registered VIM's section

To add a VIM to OSM, its type, username, password, tenant, and URL needs to be specified (more optional values can be used). The VIM definition to our Openstack-all-in-one cluster is provided in Figure 25. OSM uses these credentials to communicate with OpenStack's API, which performs the operations needed to instantiate the NS (networks creation, virtual links creation, VMs instantiation, etc.) using the information provided in the NSD and VNFD.



The screenshot shows the 'VIM Account details' page in the OSM Dashboard. The browser address bar displays 'localhost:8080/vims/07d67c62-d236-44df-a7b2-278d5a157dcf'. The page displays the following details for the 'Openstack-i2Cloud' VIM:

Name	Openstack-i2Cloud	Tenant name	admin
VIM Username	admin	Description	---
VIM URL	http://192.168.115.125/identity/v3	Schema Type	---
Type	openstack	Schema Version	1.11

Below the details is a section for 'CONFIG PARAMETERS' and a 'Back to VIM Accounts' button.

Figure 25: OSM Dashboard: Openstack account details

Once our VIM (OpenStack) has been added to OSM, and as long as it's API endpoint is reachable from OSM, to perform the communication no more steps need to be done, the testbed is ready.

Next, to test OSM-OpenStack's NFV infrastructure, we have onboarded some NSDs (NS Descriptor) and some VNFDs (VNF Descriptor), and we have instantiated a NS based on 4 VNFs (Each VNF contains a Virtual Deployment Unit (VDU)) which use a cirrOs images. This image must be previously stored in OpenStack. In Fig. 26, we have a screenshot from OSM where we are instantiating a NS, using our OpenStack-all-in-one cluster, as VIM.

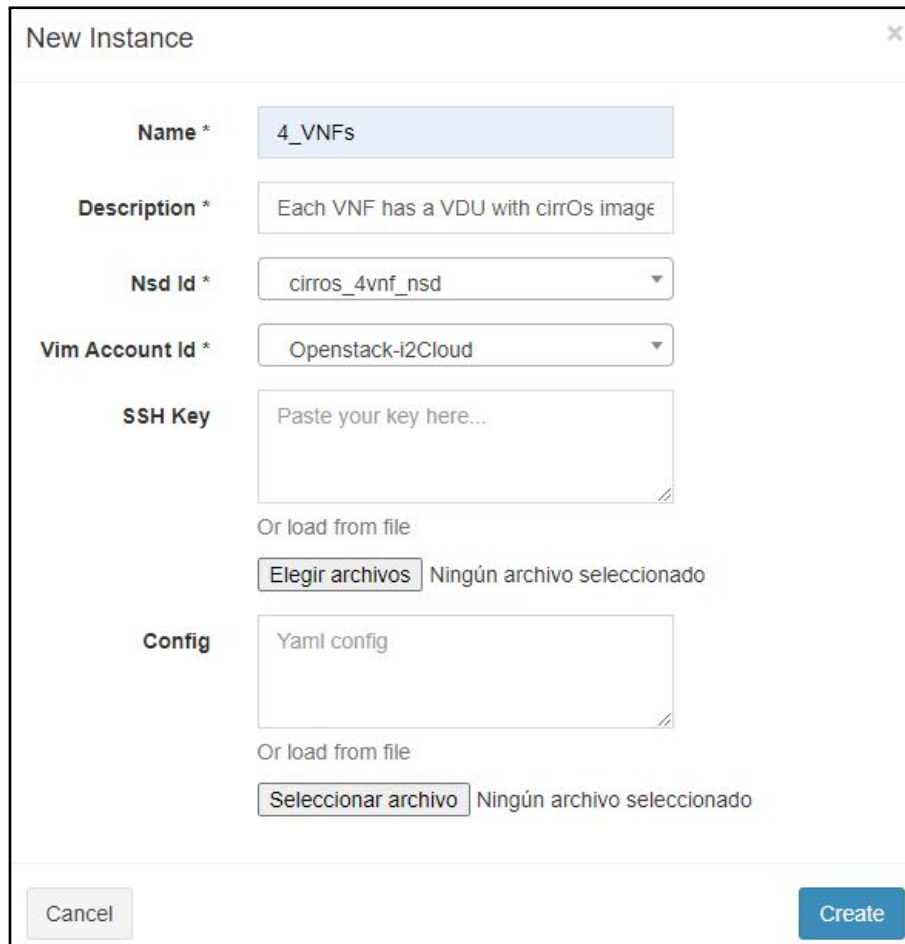
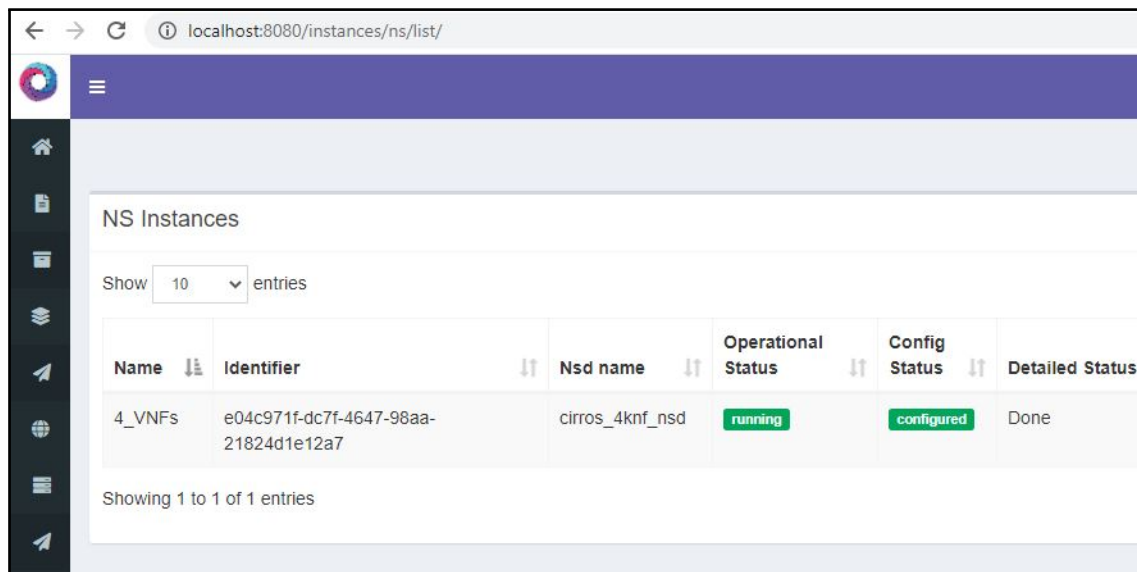


Figure 26: OSM Dashboard: Instantiating a NS based on VNFs

If we go to "NS Instances" section, in OSM Dashboard, Figure 27, we can see the status of the NS already instantiated. In our case, we can see that the NS with the configuration defined in Fig. 26, has been successfully deployed. This means that the NS is up and running, and therefore, no issues either in OSM or in OpenStack have been found. If we go to Openstack we should have 4 VMs instantiated and connected to the same network.



Name	Identifier	Nsd name	Operational Status	Config Status	Detailed Status
4_VNFs	e04c971f-dc7f-4647-98aa-21824d1e12a7	cirros_4knf_nsd	running	configured	Done

Figure 27: OSM dashboard: NS status

From Openstack's Dashboard, we confirm that the 4 VNFs have been successfully instantiated (one in each VM) and they are connected to the same network. The topology described in the NSD, and VNFD has been mapped and implemented in Openstack.

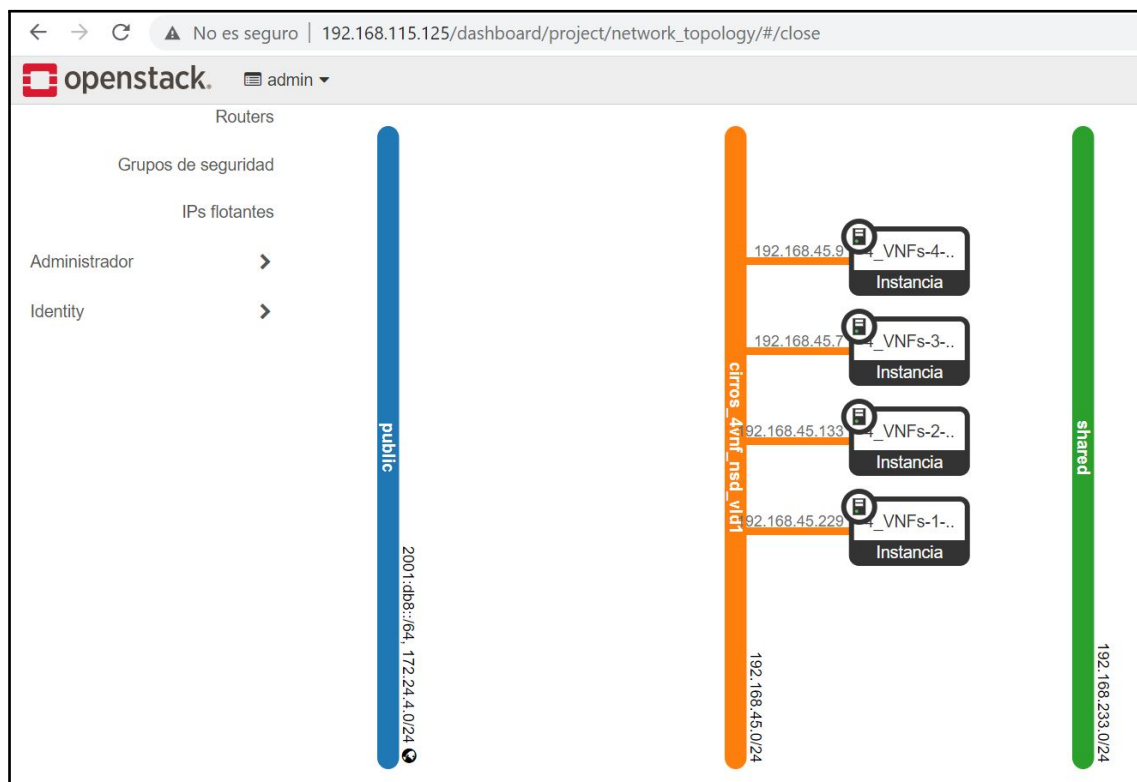


Figure 28: Network Topology in Openstack of the NS with 4 VNFs

To conclude this testbed, again in Openstack dashboard, in the section "instances", Fig. 29, we can have more details about what images are using the VMs, the IP address, the flavour and the status.

<input type="checkbox"/>	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status
<input type="checkbox"/>	4_VNFs-4-cirros_vnfd-VM-1	cirros-0.5.1-x86_64-disk	192.168.45.9	cirros_vnfd-VM-flv	-	Activo
<input type="checkbox"/>	4_VNFs-3-cirros_vnfd-VM-1	cirros-0.5.1-x86_64-disk	192.168.45.7	cirros_vnfd-VM-flv	-	Activo
<input type="checkbox"/>	4_VNFs-2-cirros_vnfd-VM-1	cirros-0.5.1-x86_64-disk	192.168.45.133	cirros_vnfd-VM-flv	-	Activo
<input type="checkbox"/>	4_VNFs-1-cirros_vnfd-VM-1	cirros-0.5.1-x86_64-disk	192.168.45.229	cirros_vnfd-VM-flv	-	Activo

Figure 29: Openstack instances section

5.2 OSM and K8s' Testbed

This testbed has been the main focus of this research, where we have spent most of the time. As stated in section 4.3, **it is important to remark that OSM at the moment of this research, had not any example where they show/test "OSM-K8s isolated cluster's" approach.** That is to say, in this research, among other things, **we have performed a validation task.**

Again, on the left, we have OSM v8, with same specifications. On the other hand, in the Cloud domain, we find two blocks, the K8s-all-in-one cluster (which can be extended seamlessly, adding nodes to the current cluster), and a Helm Repository.

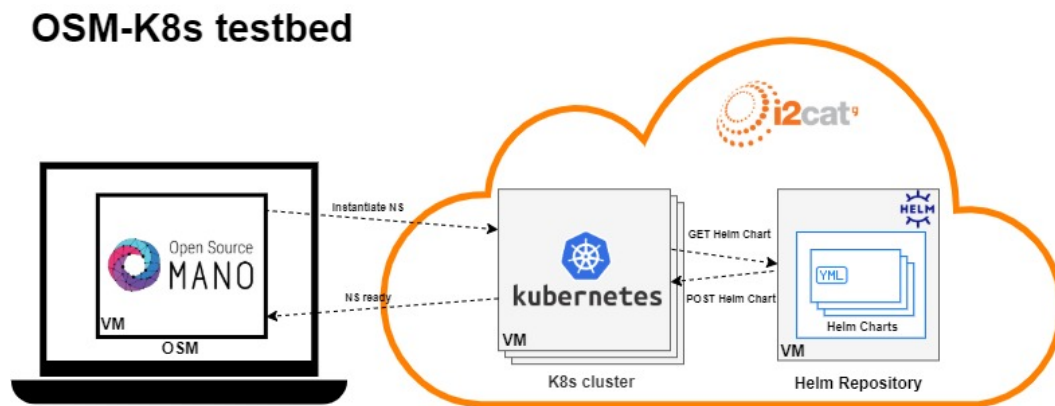


Figure 30: K8s testbed

Component	Installation details and resources needed
OSM	OSM v8.0.1, installation based on K8s. Installed in a VM running Ubuntu 18.04 with 2 vCPUS, 6GB of RAM and 60GB of storage
K8s	Devstack, v5.3.1. Installed in a VM running Ubuntu 18.04 with 2 vCPUS, 8GB of RAM and 60GB of storage
Helm Repository	Helm v2.20. VM running Ubuntu 18.04 with 2 vCPUS, 8GB of RAM and 60GB of storage)

As just mentioned, we present an all-in-one K8s cluster which is up and running. That means that the same node acts as master node, and worker node. Adding more nodes to the cluster would be as easy as installing kubelet in a new machine and adding it to the cluster, using a token coming from the master node.

By default, K8s' cluster does not schedule pods on the control-plane node for security reasons. To allow it, it is necessary to untaint the master node, [18].

Apart from creating the K8s cluster, more steps must be performed in the K8s cluster to ensure a successful connection and communication between OSM and K8s. In a nutshell, some OSM add-ons need to be added; metallb (a load balancer), a storage class, and finally, Tyler permissions need to be set, [19].

Once the K8s cluster has all the necessary OSM's addons running and well configured, we are ready to connect both blocks; OSM and K8s. First of all, even in our case that we are

working with a OSM-K8s isolated cluster (NO VIM), to make OSM and K8s compatible we must add a "dummy VIM". A dummy VIM could be understood as a simply OSM's VIM definition, which contains dummy information in the different fields. An illustrative explanation is provided within Figure 31.

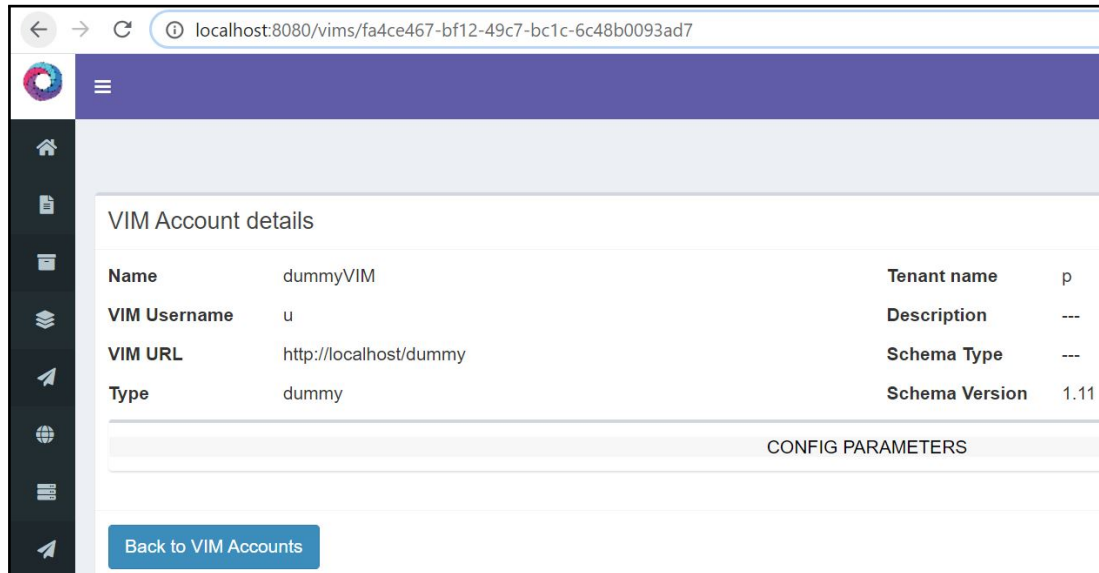


Figure 31: Adding a dummy VIM to OSM

Being performed this "dummy step", we are ready to add the K8s cluster to OSM. In few words, in this step we tell to OSM where it can find the K8s cluster, how to access it (K8s credentials) and what is its dummy VIM. As we can imagine, this dummy VIM does not perform any task, but still need to be specified to satisfy OSM. (We guess that this will change in future versions). In Figure 37, we present the OSM's section called "Registered K8s cluster", where we can find that our K8s cluster is "enabled".

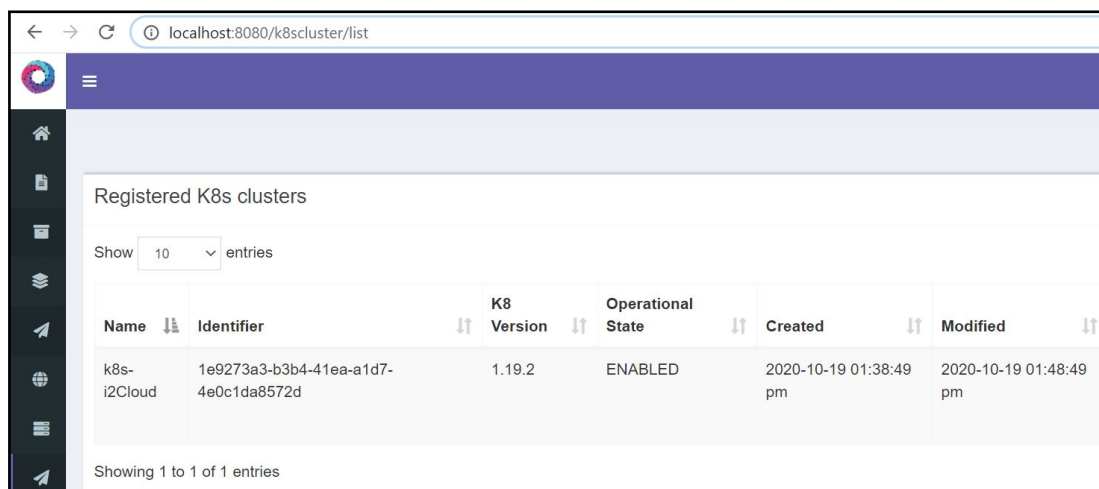
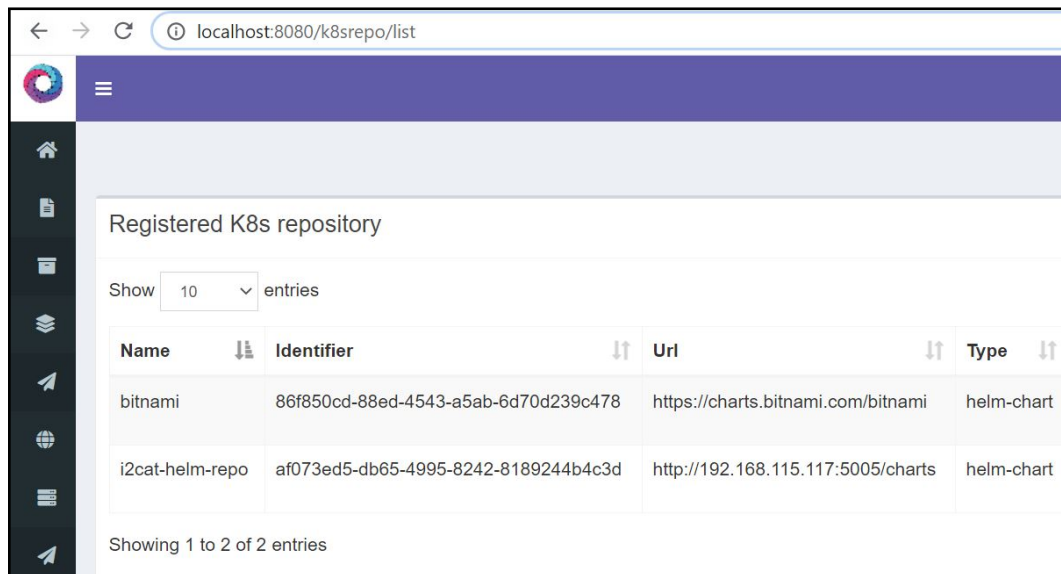


Figure 32: Registered K8s Clusters in OSM

When exploiting this testbed, OSM (in its official doc) only provided instructions to test KNFs based on public helm-charts. To do so, we must add the helm repository where we can find the helm-chart defined in the VNFD, before instantiating any KNF. In Figure 33, we can observe the added helm-charts repositories. We have a public one, such as bitnami, and our own corporative helm-chart repository, called i2cat-helm-repo.



Name	Identifier	Url	Type
bitnami	86f850cd-88ed-4543-a5ab-6d70d239c478	https://charts.bitnami.com/bitnami	helm-chart
i2cat-helm-repo	af073ed5-db65-4995-8242-8189244b4c3d	http://192.168.115.117:5005/charts	helm-chart

Figure 33: Registered K8s repositories in OSM

Nevertheless, if we want to customize KNFs, in order to use our own helm-charts (they can be of course based on public images, or private ones), a piece of the puzzle was missing. Our solution is to use a private/corporative helm repository deployed in another VM in our cloud domain (called i2Cloud). Deploying our own Helm repository under our corporate network, is an advantage in terms of security, as we avoid perform connections with external entities, outside of our network. Basically, what this machine contains is Helm(v2) along with the helm-chart(s). In this helm-charts we have our applications packaged, ready to be downloaded and be used. In Figure 34, we present the structure of a helm-chart, as we can see, using .yaml files the applications can be easy defined.

```
i2cat-cirros
├── Chart.yaml
├── charts
├── templates
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── ingress.yaml
│   ├── service.yaml
│   └── serviceaccount.yaml
├── tests
│   └── test-connection.yaml
└── values.yaml
```

Figure 34: Structure of the helm-chart used in the KNF

To be able to receive incoming connections from a K8s cluster we expose our helm-chart repository in a desired port. The information of this testbed must be provided to OSM, which will then inform the K8s cluster where it can find the repository.

It is important to highlight that **the container image specified in the helm-chart, is not stored in the helm chart repository**. Helm charts are at the same level as K8s, but container images are at a lower one. (We could make an analogy between a helm chart and a large .yaml file that fully describes our deployment from the point of view of K8s). Therefore, in an approach where we want to have a private/local docker registry/repository, **it could be an interesting idea to use the same machine that we are using as helm chart repository as docker registry**.

Once all the steps explained above have been performed, **we are ready to instantiate a KNF based on a personalized Helm-chart in a pure Cloud Native environment**, without any VIM (such as Openstack) between OSM and K8s.

To test our OSM-K8s standalone NFV infrastructure, we have onboarded some NSDs and KNFDs. At Figure 35, we present the definition of a KNF which uses the helm chart called "i2cat-cirros", stored in our helm-chart repository (i2cat-helm-repo). As we can see, KNF definitions use the variable "kdu" (kubernetes deployment unit) instead of using "vdu". Each helm-chart defined in the KNFD is considered a kdu. Therefore, KNFs can contain more than one helm-chart (and helm-charts of course can contain more than one deployment).

```
1  vnfd-catalog:
2    schema-version: '3.0'
3    vnfd:
4      - id: cirros_knfd
5        name: cirros_knfd
6        short-name: cirros_knfd
7        description: Generated by i2CAT for testing purpose.
8                    KNF with one K8s-deployment running a cirros image.
9        vendor: i2CAT
10       version: '1.0'
11       mgmt-interface:
12         cp: mgmt
13       connection-point:
14         - name: mgmt
15       k8s-cluster:
16         nets:
17           - id: mgmtnet
18             external-connection-point-ref: mgmt
19       kdu:
20         - name: i2cat-cirros
21           helm-chart: i2cat-helm-repo/i2cat-cirros
```

Figure 35: Example of a KNFD, written in .yaml

To fully test our testbed (OSM-K8s-Helm Chart Repository), we can instantiate a NS based on

KNFs, which use helm-charts located in our helm-chart repository. In Figure 36, we show how to do it from OSM's dashboard.

It is worth noting two points regarding this version of OSM (v8). One the one hand, even if we want to work with KNFs in a K8s-standalone scenario, we still have to select one VIM in the section "VIM Accounts". Here is where our dummy VIM comes into place. On the other hand, we can neither select in which K8s cluster we want to instantiate a NS based on KNFs. In fact, if we add more than one K8s cluster to OSM, OSM prevent us to instantiating any NS (OSM is not able to select in which we want to instantiate them).

New Instance

Name *

4_KNFs

Description *

Each KNF has a helm chart with 1 cont:

Nsd Id *

cirros_4knf_nsd

Vim Account Id *

dummyVIM

SSH Key

Paste your key here...

Or load from file

Elegir archivos Ningún archivo seleccionado

Config

Yaml config

Or load from file

Seleccionar archivo Ningún archivo seleccionado

Cancel

Create

Figure 36: OSM Dashboard: Instantiating a NS based on KNFs

If we instantiate the NS described in the previous figure, in "NS Instances" section, we can see that we successfully have instantiated a NS based on KNF in a pure CN-scenario (OSM-K8s standalone).

Name	Identifier	Nsd name	Operational Status	Config Status	Detailed Status
4_KNFs	ef6969a8-21c1-498c-9c94-155b4ea53bab	cirros_4knf_nsd	running	configured	Done
4_VNFs	ac8ece78-2cd5-4e93-bec5-2a401a79198a	cirros_4vnf_nsd	running	configured	Done

Showing 1 to 2 of 2 entries

Figure 37: OSM Dashboard: NS status

Furthermore, we are showing at the same time that both testbeds are up and running, one based on VNFs and deployed in OpenStack, and the other based on KNFs and deployed in K8s-standalone. OSM is able to handle both without any issue.

In Fig 38, we show the NS instantiated from K8s point of view. Each deployment (pod + rs as stated in Figure 10) corresponds to the KNFs defined in the NS. It is worth to mention that a namespace is created to handle the OSM objects.

```
ubuntu@sn-onofre2:~$ kubectl get deploy,pods,rs -n 94ce4f5f-c34c-4477-ac4a-210398bbd177
NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/i2cat-helm-repo-i2cat-cirros-0007346164  1/1     1             1           6h33m
deployment.apps/i2cat-helm-repo-i2cat-cirros-0012299582  1/1     1             1           6h33m
deployment.apps/i2cat-helm-repo-i2cat-cirros-0013073644  1/1     1             1           6h33m
deployment.apps/i2cat-helm-repo-i2cat-cirros-0016703723  1/1     1             1           6h33m

NAME                                     READY   STATUS    RESTARTS   AGE
pod/i2cat-helm-repo-i2cat-cirros-0007346164-665d7ff9d4-hhdkx  1/1     Running   0           6h33m
pod/i2cat-helm-repo-i2cat-cirros-0012299582-79dd74b457-q7fzc  1/1     Running   0           6h33m
pod/i2cat-helm-repo-i2cat-cirros-0013073644-857f4448dc-c5vxx  1/1     Running   0           6h33m
pod/i2cat-helm-repo-i2cat-cirros-0016703723-9c8cf88-w4f4z     1/1     Running   0           6h33m

NAME                                     DESIRED   CURRENT   READY   AGE
replicaset.apps/i2cat-helm-repo-i2cat-cirros-0007346164-665d7ff9d4  1          1          1       6h33m
replicaset.apps/i2cat-helm-repo-i2cat-cirros-0012299582-79dd74b457  1          1          1       6h33m
replicaset.apps/i2cat-helm-repo-i2cat-cirros-0013073644-857f4448dc  1          1          1       6h33m
replicaset.apps/i2cat-helm-repo-i2cat-cirros-0016703723-9c8cf88     1          1          1       6h33m
ubuntu@sn-onofre2:~$
```

Figure 38: NS based on 4 KNFs from K8s point of view

To conclude and to better understand the KNF instantiation process, in Figure 39, we have presented in a sequence diagram, the different messages that take place. We have included the preparation of OSM as well, hence the first three steps only need to be performed once.

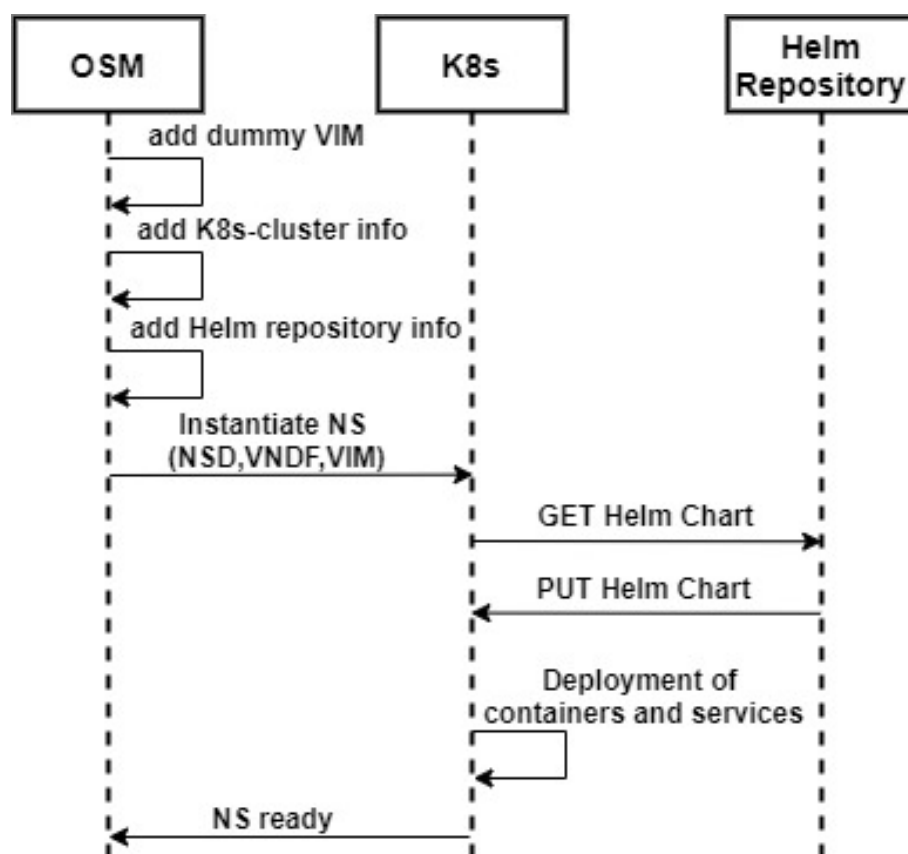


Figure 39: Preparation of the scenario and Instantiation of a NS based on KNF Sequence diagram

5.3 OSM-K8s Lessons learnt

To conclude this section, some of the issues faced during the implementation of OSM-K8s' testbed are going to be listed.

- **Need of a dummy VIM in an OSM-K8s standalone cluster.**

This was the very first problem encountered. We did not know that a dummy VIM was needed in this scenario. We ask for support via mail to OSM's team as could not instantiate any VIM after adding the K8s cluster to OSM. Weeks later, the official documentation was updated explaining the need of this VIM and giving an example in how to add it to OSM.

- **Not able to select what K8s cluster to use.**

Besides, we notice that after adding more than one K8s cluster to OSM, we could not instantiate any NS. OSMr8 does not support more than a single K8s cluster added to OSM.

- **No information regarding how to proceed using personalized Helms.**

OSM's official doc. only provides information to instantiate some test-KNFs running helm-charts stored in public helm-charts repositories such as bitnami. To use our helm-charts we had to deploy a local helm-chart repository (helm server), add our helm-charts

there, put to listen the repository and give its details to OSM in the section "K8s repositories".

- **Lack of information in errors related to Helm-charts.**

If any error related the helm-chart specified in the VNFD occurred, OSMr8 does not provide much details about the error, (A message such as: "Error: failed to download Helm-Repo/Helm-Chart"), as showed in Figure 40, making debugging a hard task.

We could neither access to the pod where the the communication OSM-Helm is handled and perform helm commands (with a debugging objective) in the pod due to permission issues.

To solve it, OSM should include the flag "--debug" in all its helm-related commands. With it, a more complete error-reporting is provided, specifying the cause of the error.

Name	Identifier	Nsd name	Operational Status	Config Status	Detailed Status
test2	5d9dec65-2fb1-4b31-ac67-984302983aa1	cirros_1knf_nsd	failed	configured	Operation: INSTANTIATING.c3bbc105-7274-44b8-a439-32549bfdc73d, Stage 2/5: deployment of KDUs, VMs and execution environments. Detail: Deploying at VIM: datacenter 'f481dc78-0e02-11eb-b328-7efaf4d6e731' not found. Deploying KDU i2cat-cirros: Error executing command: /usr/local/bin/helm install --atomic --output yaml --kubeconfig=/app/storage/1e9273a3-b3b4-41ea-a1d7-4e0c1da8572d/.kube/config --home=/app/storage/1e9273a3-b3b4-41ea-a1d7-4e0c1da8572d/.helm --timeout 600 -name=i2cat-helm-repo-i2cat-cirros-0086534972 --namespace 94ce4f5f-c34c-4477-ac4a-210398bbd177 i2cat-helm-repo/i2cat-cirros Output: Error: failed to download "i2cat-helm-repo/i2cat-cirros" (hint: running 'helm repo update' may help)

Figure 40: OSM Dashboard: helm-related error

6 Perform validation and benchmarking

The purpose of this section is to experiment, evaluate and benchmark the behaviour of KNFs (Kubernetes based Virtual Network Functions) and compare it with the classical solution based on VMs, the conventional VNFs.

To do so, different KPIs are used. In the simulations, different Network Services (NS) varying the number of KNFs/VNFs have been onboarded in OSM, and deployed in the two testbeds; the one based on K8s, and the one based on OpenStack. From this simulations we can study different values as we will show next. Besides, a comparison of the footprint required by the two VIMs is provided.

6.1 Related Works in the literature

Some papers in the literature study both virtualization technologies in a MEC context, such as [31]. Regarding NFV, recently some papers with similar scope than this work have been published. On the one hand, in [32], two of the most important MANO platforms are studied and compared in detail; OSM and ONAP. On the other hand, in [33], some KNF-use cases in a smart manufacturing domain, are showcased.

Nevertheless, the literature is still missing a benchmark of the KNFs. In our work, we have fixed OSM as the MANO platform (following ETSI NFV), and we have studied the behaviour of KNFs in a pure Cloud Native NFV Architecture; OSM-K8s standalone (Some people consider that K8s is not a pure VIM itself, and suggest to use it along with OpenStack, in specific inside OpenStack's platform), but in our study its functionality is like that of a VIM). Both OSM, K8s and NS (via KNFs) run in containers. We have compared this architecture with one of the most important and adopted ones: OSM-Openstack, which runs NS (via VNFs) in VMs.

6.2 Images used in the simulations

In order to properly compare the behaviour of VNFs and KNFs, and hence compare both testbeds fairly and equitably, it is necessary to fix an image with similar size. In fact, it is not easy to find an image with similar size both formats; cloud format (VMs) and container format. Containers images, due to its nature, use to be way lighter weight.

For instance, Ubuntu's 18.04 image in cloud format [34] does not weight less than 300Mb. On the contrary, Ubuntu's 18.04 docker image [35] takes not more that 30Mb, [35]. Therefore, in the context of a fairly benchmarking, is not a good candidate due to huge difference in terms of size.

On the other hand, CirroS images [36] which are minimal Linux distributions designed for testing, are available in both formats and have similar size (12.6MB in docker image format and 15.58MB in cloud format). Hence, if we use these images in our different KNFs/VNFs, we guarantee a fairness evaluation and benchmark. Of course, it should be noted that the results and graphs obtained in our simulations are subject to images used and the resources of our scenarios.



Figure 41: CirrOs' docker image, [36]

6.3 KPI 1: Deployment Process Delay (DPD)

Deployment Process Delay (DPD), defined in [32], is the time needed to deploy and instantiate the VNF(s) of a NS. This process involves the provision, instantiation and configuration of the VM(s)/Containers. The creation of the VLs (if necessary) is considered as well. All these actions follow the specifications provided in the VNFD(s)/KNFD(s) and NSD.

This time gives us an insight of the time necessary to scale a NS, or even to re-instantiate it due to any issue. Other works in the literature use similar concepts, such as network service instantiation, mentioned in [37].

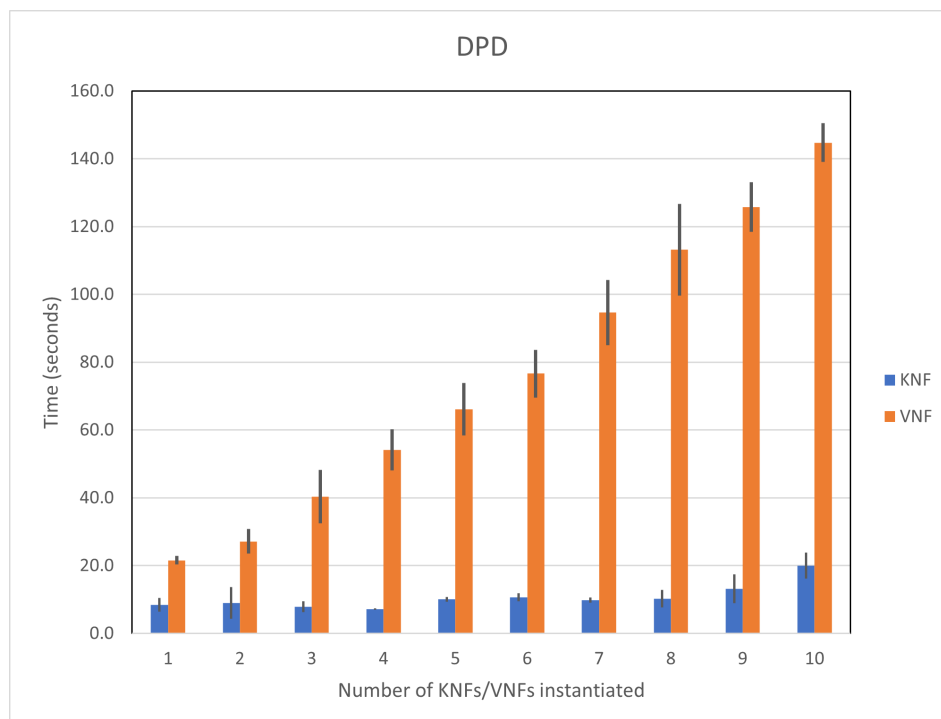


Figure 42: Deployment Process Delay (DPD) of a NS varying the number of VNFs/KNFs

In Figure 42 we observe the behaviour of the DPD needed by different NS in terms of number of VNF/KNF. Each VNF/KNF contains a single VDU/KDU. We observe the huge difference between VNFs (running inside VMs) and KNFs (running inside containers). Only in the first simulation, comparing the NS based on 1 KNF/VNF, the VNF takes more than 2.5 times to be ready to use. This difference gets greater at each iteration, and in the last one, with 10 KNFs/VNFs, the NS based on VNFs take more than 7 times to be up and running than the one based on containers.

It is worth to highlight that in the first iteration when instantiating KNF(s), the docker image needs to be downloaded, thus, it takes an extra time that is not needed in the following iterations.

6.4 KPI 2: CPU and RAM

As we know CPU and RAM are some of the must-do KPIs when analyzing this type of infrastructures. From Figure 43, we present the percentage of CPU usage in both all-in-one clusters. We observe that by default, the K8s cluster consumes more CPU than the Openstack one.

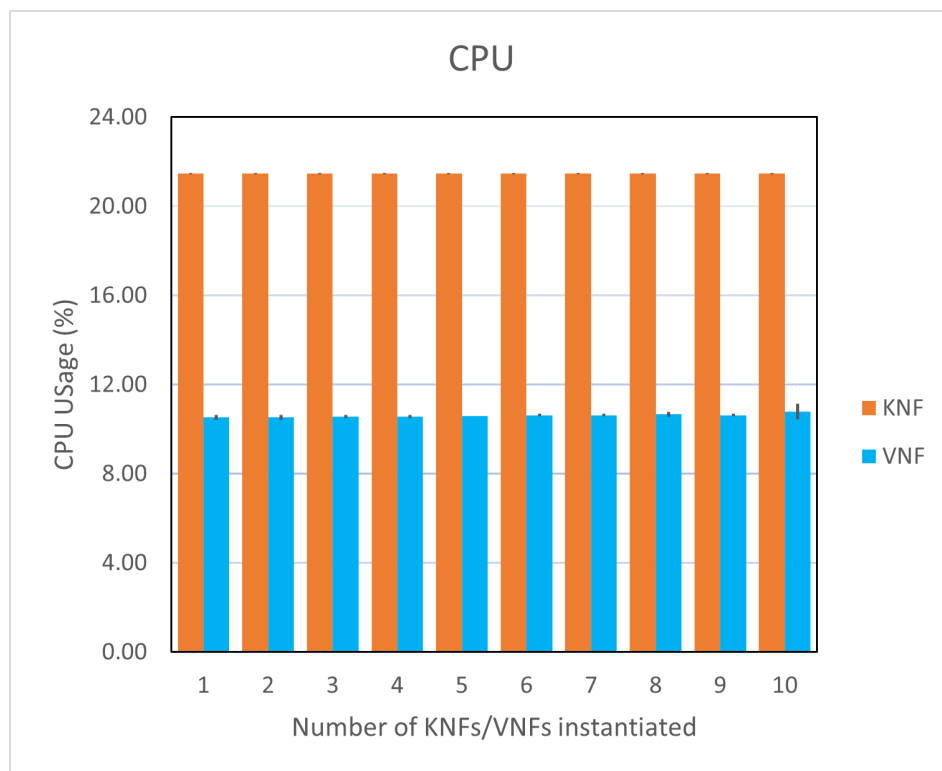


Figure 43: CPU Usage of a NS varying the number of VNFs/KNFs

When instantiating the different NSs, we notice that they have a different behaviour; the VNFs in the VMs consume each iteration more CPU, even though we are not running task with them. This is due to the resource-allocation constraint of VMs.

Conversely, containers by default use the resources that they need, hence if we are not running any task, they will not have any impact in the node resources where they are running. (It should also be noted that it is possible to reserve and limit the minimum and maximum resources allowed to a container).

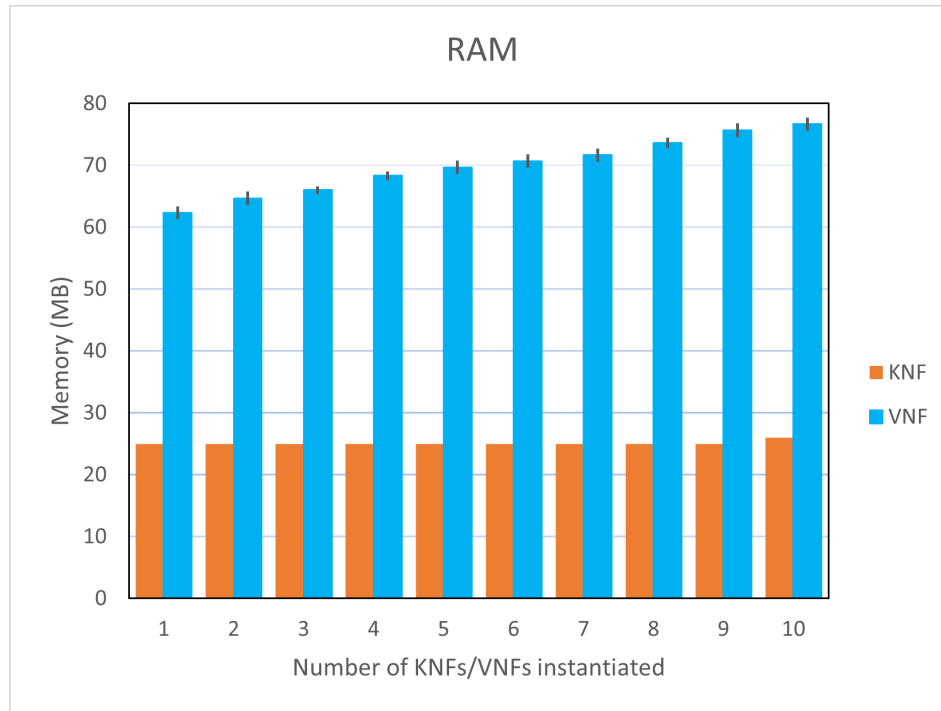


Figure 44: RAM Usage of a NS varying the number of VNFs/KNFs

Considering the RAM (Figure 44), this difference in behaviour is further accentuated, as the OpenStack-all-in-one cluster consumes 60% only to run OpenStack services. We observe that with 10 VNFs running in VMs, we have our OpenStack cluster almost at its limit, whereas the K8s cluster is almost unaffected. It consumes a 25% of RAM to run the containers that implement K8s architecture, and it only increases to 26% running 10 KNFs with CirroS image containers in parallel.

Finally, it can underscored that both graphs can be considered as functional validation of both testbeds, since they present logical results, mostly due to the nature of the type of virtualization used.

6.5 KPI 3: Max number of VNFs/KNFs supported using same resources

As both Openstack and K8s all-in-one cluster's machines have the same resources, and as we are using images with similar size, we can compare the max number of VNFs and KNFs that we can instantiate. As we can imagine, the KNFs that can be instantiated in the K8s cluster will outnumber the VNFs in the Openstack.

Assuming that each KNF runs on a pod and each VNF runs on a VM, we obtain the following table.

Testbed	Max number of VNFs/KNFs running simultaneously
OSM-Standalone K8s	116
OSM-OpenStack	10

As shown, the relation is 116-10; This means that, having equal resources in both testbeds, we can instantiate up to 116 KNFs in the OSM-K8s one, but on the contrary, we can only instantiate 10 VNFs in the Openstack one.

The main factor here in favour of KNFs is that containers can consume less than 1vCPU if needed. (E.g.: a container that needs only 0.6 vCPU to run). VMs, on the other hand, need at least 1 vCPU to be instantiated. This means that, by using KNFs not only we save compute time and system resources, but also we have direct impact on CAPEX and OPEX. For the same resources per server, the number of applications that we can instantiate is 100 times more, allowing us to deploy several NS in parallel, or forming VNF-Forwarding Graphs formed by different NSs.

Nevertheless, cope with scenarios where a high volume of containers are interconnected could be a hard task. To address this, service mesh can be used. Using a service mesh a fast, reliable and secure communication between containers is ensured, facilitating service-to-service communications between microservices [38].

From this results we realise the doors that containerization brings to NFV. Virtual Network Functions through containers could be deployed inside lightweight devices (such as a Raspberry pi located for example in a post lamp), giving to the NFV world a previously unseen flexibility and a crucial role, helping to meet most of the 5G Verticals, such as automotive, smart manufacturing, smart cities, etc., as most of the 5G services will be located in the Edge.

7 Conclusions and future work

In this last section we are going to introduce the conclusions, highlighting the main milestones achieved in this dissertation. Besides, some future lines of work are suggested.

7.1 Conclusions

The overview of many researchers, institutions and the European Commission regarding the current state and the evolution of NFV has been considered in the elaboration of Section 2, where some context regarding both 5G and Cloud Native has been included.

Regarding NFV, some users could be reluctant to start working with KNFs, due to the constraint of having an OpenStack cluster acting as an intermediary between OSM and K8s (OSM-OpenStack-K8s scenario). Taking into account the lack of validation in the approach where KNFs are instantiated directly in a K8s standalone cluster, (no OpenStack between; OSM-K8s scenario), we decided to work on this scenario and make the validation from a user point of view.

Therefore, in this work we have validated and studied Kubernetes-based Network Functions (KNFs) in the ETSI-hosted MANO platform which is entirely aligned with ETSI NFV, open source MANO (OSM). The aim of the project is, among other things, to encourage the use and development of KNFs from operators, users and developers.

Apart from this validation, a benchmarking has been performed, where KNFs instantiated in containers in a K8s cluster, are compared to VNFs instantiated in VMs in an OpenStack cluster. With this benchmarking, the advantages of KNFs over VNFs has been tangibly demonstrated in terms of cpu, ram, deployment process delay (DPD), and maximum number of KNFs/VNFs able to instantiate in two machines with equal resources. Apart from using machines with same resources, to compare both testbeds fairly and equitably, we have fixed images in the VNFs and KNFs with similar size and behaviour.

When comparing VNFs and KNFs, we could highlight two main points on which KNFs behaviour is clearly superior than VNFs. On the one hand, the deployment process delay (DPD). To instantiate for instance 5 VNFs, OSM-OpenStack's testbed take 6 times more than OSM-K8s to instantiate 5 KNFs, having the same image in terms of functionality, and similar size. This KPI is very important as it give us an insight of the time necessary to instantiate a NS, to scale it, or even to re-instantiate it due to any issue. Another point in favour of KNFs is that containers can consume less than 1vCPU if needed. VMs running VNFs on the other hand, consume at least 1 vCPU per VM.

To conclude, we hope that this benchmarking can help to further encourage users and operators the use of KNFs and get the most out of containerization in NFV.

7.2 Future work

Due to resources constrains (overall the vCPU), our OpenStack-all-in-one cluster has limited our potential tests in terms of max number of VNFs instanciable. Hence, a good line of future

work could be implement both testbeds in more powerful bare-metal servers. This would allow as immediately to improve the quality of the tests and to be able to observe and analyze the behaviour of both NFV testbeds in a production environment.

Another natural next step could be analyze the behaviour of the OSM-K8s testbed with more complex and heterogeneous images, such as a firewall, a DHCP server or a DNS one while they experiment real traffic.

At the time of writing this document, a new version of OSM just got released, OSM Release 9. Resource orchestration in different scenarios, as well as and improved VNF and CNF lifecycle management supporting Helm v3, are some of the new features. Upgrading both testbeds with OSMr9 and re-run some of the simulations using Helm v3 could be some interesting further steps.

References

- [1] Evolving Cloud Native Landscape. Presentation by Chris Aniszyk. Slide 17. Cloud Native Computing Foundation (CNCF). Container Days. Japan. December 2018. <https://events19.linuxfoundation.org/wp-content/uploads/2018/09/Evolving-Cloud-Native-Landscape-Dec-2018-ContainerDays-Japan.pdf>.
- [2] Network Functions Virtualization – Introductory White Paper. SDN and OpenFlow World Congress (Darmstadt-Germany). https://portal.etsi.org/NFV/NFV_White_Paper.pdf.
- [3] ITU-R. IMT vision, overall objectives of the future deployment of IMT for 2020, and beyond. https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.2083-0-201509-I!!PDF-E.pdf.
- [4] Qualcomm. Making 5g nr a reality. <https://www.qualcomm.com/media/documents/files/making-5g-nr-a-reality.pdf>, 2016. September, 2016.
- [5] NCTA Technical Paper. Bridging the gap between ETSI NFV and Cloud Native architecture. <https://www.nctatechnicalpapers.com/Paper/2017/2017-bridging-the-gap-between-etsi-nfv-and-cloud-native-architecture/download>, October 2017.
- [6] Cloud Native Computing Foundation (CNCF). CNCF Cloud Native Definition v1.0. <https://github.com/cncf/toc/blob/master/DEFINITION.md>, February 2020.
- [7] DevOps life cycle. Explore About Each Phase in DevOps Life cycle. Medium. Edureka. <https://medium.com/edureka/devops-lifecycle-8412a213a654>.
- [8] 5G-PPP White Paper. Cloud Native and 5G Verticals' services. <https://5g-ppp.eu/wp-content/uploads/2020/02/5G-PPP-SN-WG-5G-and-Cloud-Native.pdf>, February 2020.
- [9] Cloud Native Computing Foundation CNCF. <https://www.cncf.io/>.
- [10] CI CD Pipeline: Learn How to Setup a CI CD Pipeline From Scratch. Medium. Edureka. <https://medium.com/edureka/ci-cd-pipeline-5508227b19ca>.
- [11] Software Defined Networking (SDN) Network Function Virtualization (NFV). Redeem systems. <https://www.redeemsystems.com/sdnfv.php>.
- [12] 5G Infrastructure Public Private Partnership (5G PPP) official web site. <https://5g-ppp.eu/>.
- [13] ETSI GS NFV 002 V1.2.1. Network functions virtualisation (nfv); architectural framework. https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf, December 2014.
- [14] 8th OSM Hackfest. Session 0: Introduction to NFV and OSM. https://osm.etsi.org/wikipub/index.php/8th_OSM_Hackfest.

- [15] ETSI GS NFV 003 V1.4.1. Network functions virtualisation (nfv). terminology for main concepts in nfv. https://www.etsi.org/deliver/etsi_gs/NFV/001_099/003/01.04.01_60/gs_nfv003v010401p.pdf, August 2018.
- [16] Cisco. Global mobile data traffic forecast update, 2017–2022 white paper. <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html>, 2019. Accessed June 8, 2019.
- [17] K8s architecture. K8s official documentation. <https://kubernetes.io/docs/concepts/overview/components/>.
- [18] Control plane node isolation. K8s official documentation. <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/#control-plane-node-isolation>.
- [19] OSM official documentation. ANNEX 7: Kubernetes installation and requirements. <https://osm.etsi.org/docs/user-guide/15-k8s-installation.html#method-3-manual-cluster-installation-steps-for-ubuntu>.
- [20] CNCF Cloud Native Interactive Landscape. <https://landscape.cncf.io/>.
- [21] Red Hat OpenShift official documentation. <https://www.openshift.com/>.
- [22] ETSI. OSM definition. <https://www.etsi.org/technologies/open-source-mano>.
- [23] OSM MR9 Hackfest. Hack 0: Introduction to NFV and OSM. https://osm.etsi.org/wikipub/index.php/OSM-MR9_Hackfest.
- [24] OSM Release EIGHT. Release Notes. https://osm.etsi.org/wikipub/images/5/56/OSM_Release_EIGHT_-_Release_Notes.pdf.
- [25] OSM Official doc. Kubernetes Installation. <https://osm.etsi.org/docs/user-guide/05-osm-usage.html#kubernetes-installation>.
- [26] OSM MR9 Hackfest – Hack 4 Running Containerized Network Functions. https://osm.etsi.org/wikipub/index.php/OSM-MR9_Hackfest.
- [27] Adnan Abdulhussein. Bitnami. Kubernetes CI/CD with Helm. Slide 12. <https://www.slideshare.net/AdnanAbdulhussein/kubernetes-cicd-with-helm>.
- [28] Helm official documentation. What is a Helm Chart. <https://helm.sh/docs/topics/charts/>.
- [29] Openstack official doc. What is OpenStack. <https://www.openstack.org/software/>.
- [30] Red Hat documentation. Openstack components. https://access.redhat.com/documentation/en-us/red_hat_opensstack_platform/9/html/architecture_guide/components.
- [31] T. V. Doan, G. T. Nguyen, H. Salah, S. Pandi, M. Jarschel, R. Pries, and F. H. P. Fitzek. Containers vs virtual machines: Choosing the right virtualization technology for mobile edge cloud. In *2019 IEEE 2nd 5G World Forum (5GWF)*, pages 46–52, 2019.

-
- [32] Girma M. Yilma, Zarrar F. Yousaf, Vincenzo Sciancalepore, and Xavier Costa-Perez. Benchmarking open source nfv mano systems: Osm and onap. *Computer Communications*, 161:86 – 98, 2020.
- [33] Stefan Schneider, Manuel Peuster, Kai Hannemann, Daniel Behnke, Marcel Müller, Patrick-Benjamin Bök, and Holger Karl. "producing cloud-native": Smart manufacturing use cases on kubernetes. 11 2019.
- [34] Ubuntu's 18.04 Image. Cloud format. <https://cloud-images.ubuntu.com/bionic/current/>.
- [35] Ubuntu's 18.04 Image. DockerHub. <https://hub.docker.com/layers/ubuntu/library/ubuntu/18.04/images/sha256-9bcfcd2cbadc77f41dd75535f694f2a5809f53eb2f6682398cedc1b6469ef783?context=explore>.
- [36] CirrOs' Image. DockerHub. https://hub.docker.com/_/cirros.
- [37] Project S. Xilouris (NCSRD) G. (Ed.). D6.3 Final Demonstration roadmap and Validation Results (2018). <https://www.sonata-nfv.eu/content/d63-final-demonstration-roadmap-and-validation-results>.
- [38] What is a Service Mesh? Nginx official documentation. <https://www.nginx.com/blog/what-is-a-service-mesh/>.

Appendices

A OSM Descriptors: NSDs and VNFDs

A.1 VNFD "cirros_knfd" (It contains a kdu (helm-chart) with a cirros docker image).

```
1 vnfd-catalog:
2   schema-version: '3.0'
3   vnfd:
4     - connection-point:
5       - name: mgmt
6       description: Generated by i2CAT for testing purpose. KNF with
7         one container running a cirros image.
8       id: cirros_knfd
9       k8s-cluster:
10        nets:
11          - external-connection-point-ref: mgmt
12            id: mgmtnet
13      kdu:
14        - helm-chart: i2cat-helm-repo/i2cat-cirros
15          name: i2cat-cirros
16      mgmt-interface:
17        cp: mgmt
18        name: cirros_knfd
19        short-name: cirros_knfd
20        vendor: i2CAT
21        version: '1.0'
```

A.2 VNFD "cirros_vnfd" (It contains a vdu with a cirros cloud image).

```
1 vnfd:vnfd-catalog:
2   vnfd:
3     - connection-point:
4       - name: eth0
5       type: VPORT
6       description: Generated by i2CAT for testing purpose. VNF with
7         one instance
8         of cirros.
9       id: cirros_vnfd
10      logo: cirros-64.png
11      mgmt-interface:
12        cp: eth0
13        name: cirros_vnfd
14        short-name: cirros_vnfd
15      vdu:
16        - count: 1
17          description: cirros_vnfd-VM
18          id: cirros_vnfd-VM
19          image: cirros-0.5.1-x86_64-disk
20          interface:
```

```

20         - external-connection-point-ref: eth0
21           name: eth0
22           type: EXTERNAL
23           virtual-interface:
24             bandwidth: '0'
25             type: VIRTIO
26             vpci: 0000:00:0a.0
27         name: cirros_vnfd-VM
28         vm-flavor:
29           memory-mb: 256
30           storage-gb: 2
31           vcpu-count: 1
32         vendor: i2CAT
33         version: '1.0'

```

A.3 NSD "cirros_2knf_nsd" (It contains 2 "cirros knfd" KNFs).

```

1 nsd-catalog:
2   nsd:
3     - constituent-vnfd:
4       - member-vnf-index: 1
5         vnfd-id-ref: cirros_knfd
6       - member-vnf-index: 2
7         vnfd-id-ref: cirros_knfd
8     description: Generated by i2CAT for testing purpose.
9     id: cirros_2knf_nsd
10    name: cirros_2knf_nsd
11    short-name: cirros_2knf_nsd
12    vendor: i2CAT
13    version: '1.0'
14    vld:
15      - id: mgmtnet
16        mgmt-network: 'true'
17        name: mgmtnet
18        type: ELAN
19        vim-network-name: mgmt
20        vnfd-connection-point-ref:
21          - member-vnf-index-ref: 1
22            vnfd-connection-point-ref: mgmt
23            vnfd-id-ref: cirros_knfd
24          - member-vnf-index-ref: 2
25            vnfd-connection-point-ref: mgmt
26            vnfd-id-ref: cirros_knfd

```

A.4 NSD "cirros_2vnf_nsd" (It contains 2 "cirros_vnfd" VNFs).

```

1 nsd:nsd-catalog:
2   nsd:
3     - constituent-vnfd:
4       - member-vnf-index: 1
5         vnfd-id-ref: cirros_vnfd
6       - member-vnf-index: 2
7         vnfd-id-ref: cirros_vnfd

```

```
8      description: Generated by OSM pacakage generator
9      id: cirros_2vnf_nsd
10     logo: osm_2x.png
11     name: cirros_2vnf_nsd
12     short-name: cirros_2vnf_nsd
13     vendor: OSM
14     version: '1.0'
15     vld:
16     -   id: cirros_2vnf_nsd_vld1
17         mgmt-network: 'true'
18         name: cirros_2vnf_nsd_vld1
19         short-name: cirros_2vnf_nsd_vld1
20         type: ELAN
21         vnfd-connection-point-ref:
22         -   member-vnf-index-ref: 1
23             vnfd-connection-point-ref: eth0
24             vnfd-id-ref: cirros_vnfd
25         -   member-vnf-index-ref: 2
26             vnfd-connection-point-ref: eth0
27             vnfd-id-ref: cirros_vnfd
```

B Code Fragments

B.1 KNF Instantiation

```
1 #/bin/bash
2 X=$1
3
4 #Checking if the user has provided an input (Number of replicas)
5 if [ -z "$X" ]
6 then
7     echo "[Error] You have to provide the number of replicas. ---> e
8     .g: ./instantiate-NS_KNF_cirros.sh 5"
9     exit 1
10 fi
11 echo " "
12 echo "***** "
13 echo "Instantiating a NS with 1 KNF that contains $X KDUs"
14 echo "***** "
15 echo "Image: cirros."
16 echo " "
17
18 osm ns-create --ns_name KNF_$X --nsd_name cirros_"$X"knf_nsd --
19     vim_account dummyVIM
20 KNF_id=$(osm ns-show KNF_$X | grep -i _id | sed -n 1,1p | awk '{print $4
21     }' | sed 's/\\/\\/g')
22 echo " "
23 while [ "$(osm ns-list | grep -i $KNF_id | awk '{print $8}')" != 'READY'
24 ]
25 do
26 :
27 done
28 echo "[NS up and running]"
```

B.2 KNF DPD time

```
1 #/bin/bash
2 X=$1
3 KNF_id=$(osm ns-show KNF_$X | grep -i _id | sed -n 1,1p | awk '{print $4
4     }' | sed 's/\\/\\/g')
5 output_file="$HOME/benchmarking/database/KNF-K8s/instantiation_data.txt"
6 echo " "
7 echo "Running ./KNF-instantiation_time.sh "
8
9 if [ -z "$X" ]
10 then
11     echo Error! DO NOT CALL IT FROM HERE
12     exit 1
13 fi
14
15 echo -n "$X " >> $output_file
```



```

16
17 creation_time=$(osm ns-show $KNF_id | grep -i create-time)
18 echo -n "$(echo $creation_time | awk '{print $4}')" >> $output_file
19
20
21 status_time=$(osm ns-show $KNF_id | grep -i status-time)
22 echo "$(echo $status_time | awk '{print $4}' | sed 's/\"//g')" >>
    $output_file
23
24 echo "[Creation time and status time values saved in the file
    $output_file]"
25 echo " "
26
27 echo "Deleting the NS.."
28 osm ns-delete $KNF_id

```

B.3 VNF Instantiation

```

1 #/bin/bash
2 X=$1
3
4 #Checking if the user has provided an input (Number of replicas)
5 if [ -z "$X" ]
6 then
7     echo "[Error] You have to provide the number of VNFs of the NS
    ---> e.g: ./instantiate-NS_VNF_cirros.sh 4"
8     exit 1
9 fi
10
11 echo " "
12 echo "***** "
13 echo "Instantiating a NS with $X VNF(s) that contain 1 VDU per VNF"
14 echo "***** "
15 echo "Image: cirros"
16 echo " "
17
18 osm ns-create --ns_name VNF_$X --nsd_name cirros_"$X"vnf_nsd --
    vim_account Openstack-i2Cloud
19
20 VNF_id=$(osm ns-show VNF_$X | grep -i _id | sed -n 1,1p | awk '{print $4
    }' | sed 's/\"//g')
21
22 echo " "
23 while [ "$(osm ns-list | grep -i $VNF_id | awk '{print $8}')" != 'READY'
    ]
24 do
25     :
26 done
27 echo "[NS up and running]"

```

B.4 VNF DPD time

```
1 #/bin/bash
2 X=$1
3 VNF_id=$(osm ns-show VNF_$X | grep -i _id | sed -n 1,1p | awk '{print $4
   }' | sed 's/\"//g')
4 output_file="$HOME/benchmarking/database/VNF-Openstack/
   instantiation_data.txt"
5
6 echo " "
7 echo "Running ./VNF-instantiation_time.sh "
8
9 if [ -z "$X" ]
10 then
11     echo Error! DO NOT CALL IT FROM HERE
12     exit 1
13 fi
14
15 echo -n "$X " >> $output_file
16
17 #creation time
18 creation_time=$(osm ns-show $VNF_id | grep -i create-time)
19 echo -n "$(echo $creation_time | awk '{print $4}')" " >> $output_file
20
21 #status time
22 status_time=$(osm ns-show $VNF_id | grep -i modified)
23 echo "$(echo $status_time | awk '{print $4}' | sed 's/\"//g')" >>
   $output_file
24
25 echo "[Creation time and status time values saved in the file
   $output_file]"
26 echo " "
27
28 echo "Deleting the NS.."
29 osm ns-delete $VNF_id
```