Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2019

SANDRA INÊS FERREIRA MOREIRA

Integração Contínua no 5GinFIRE

5GinFIRE Continuous Integration

Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2019

SANDRA INÊS FERREIRA MOREIRA

Integração Contínua no 5GinFIRE

5GinFIRE Continuous Integration

“*It takes a lot of hard work to create something simple.*”

— Steve Jobs

Universidade de Aveiro Departamento de Eletrónica,
2019 Telecomunicações e Informática

SANDRA INÊS FERREIRA MOREIRA

Integração Contínua no 5GinFIRE

5GinFIRE Continuous Integration

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Diogo Nuno Pereira Gomes, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Rui Luís Andrade Aguiar, Professor catedrático do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

agradecimentos / acknowledgements

Em primeiro lugar, gostaria de agradecer ao Prof. Doutor Diogo Gomes e ao Prof. Doutor Rui Aguiar pela oportunidade de desenvolver este trabalho e por todo o apoio prestado ao longo da evolução do mesmo. Aproveito também para agradecer ao IT (UID/EEA/50008/2019) por fornecer os recursos e condições para a realização desta Dissertação.
Um grande agradecimento aos meus pais e irmãos, a quem dedico este trabalho, por serem sempre os primeiros a acreditarem em mim, estarem sempre disponíveis e fornecerem todas as condições para que fosse possível percorrer este caminho.
Por último, agradeço a todos os meus amigos pelo companheirismo e amizade, em particular ao Jaime, ao Ricardo e ao Marco por estarem sempre disponíveis para me ajudar e por me acompanharem neste longo percurso.
A todos, muito obrigado.

Palavras Chave

nfv, osm, integração contínua, validação de descritores do osm

Resumo

Com a evolução dos equipamentos com capacidade de se ligar à rede, as exigências de tráfego tornam-se muito altas. Os operadores precisam de garantir que oferecem os seus serviços rapidamente, com a mesma qualidade, mas mantendo os custos baixos. Dada a arquitetura tradicional de redes, isso não é possível uma vez que para alcançar essas necessidades é fundamental a aquisição de novos equipamentos, sendo que a sua substituição é cara e pouco flexível. Com a proposta de separação de funções de rede do seu *hardware* específico, NFV é a tecnologia que permite aos operadores alcançar o pretendido. No entanto, esta abordagem traz consigo problemas relacionados com a fiabilidade do código produzido, uma vez que é imperativo assegurar que as funções de rede implementadas (VNFs) se comportam como esperado. O 5GinFIRE é um projeto que tem como objetivo manter uma plataforma de experimentação de 5G-NFV. Como este projeto lida com múltiplas VNFs de vários colaboradores, é necessário haver um mecanismo automatizado que valida as mesmas. Esta dissertação aborda a solução referenciada tendo em si descrito um sistema que valida a sintaxe, semântica e referências de uma VNF de uma forma totalmente automatizada e sem qualquer necessidade de intervenção humana. Assim, o 5GinFIRE contém já uma plataforma de testes totalmente integrada no seu sistema e os seus resultados são analisados neste Documento.

Keywords nfv, osm, continuous integration, osm descriptors validation

Abstract

With the current evolution of network connectable devices, traffic demands are becoming very high. Network operators need to ensure that they can provide new services faster but with the same quality while keeping the costs low. Given the traditional network architecture, that is not possible because the high demands require new hardware, and its substitution is costly and not flexible. By introducing the decoupling of network functions from traditional hardware, NFV is the technology that enables the step that network operators are trying to take. However, this approach also brings reliability concerns since it is mandatory to ensure that the virtual network functions (VNFs) behave as expected. 5GinFIRE is a project that aims to provide a 5G-NFV enabled experimental testbed. As this project handles multiple VNFs from the various experimenters, it is necessary to have an automated mechanism to validate VNFs. This dissertation provides a solution for the stated problem by having a system that verifies the syntax, semantics, and references of a VNF in an automated way without needing any further human interaction. As a result, a fully integrated testing platform is deployed in the 5GinFIRE infrastructure, and the results of the tests are issued in this Document.

Contents

List of Figures

List of Tables

Acronyms

5G	Fifth Generation of Cellular Mobile Communications
API	Application Programming Interface
AWS	Amazon Web Services
CD	Continuous Delivery
CI	Continuous Integration
CLI	Command Line Interface
CM	Continuous Monitoring
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
CT	Continuous Testing
ETSI	European Telecommunications Standards Institute
EVI	Experimental Instances of Verticals
GUI	Graphical User Interface
IM	Information Model
IoT	Internet of Things
ISG	Industry Specification Group
JSON	JavaScript Object Notation
LCM	Lifecycle Manager
MANO	Management and Orchestration
MON	Monitoring Module
NBI	Northbound Interface
NFVI	Network Function Virtualization Infrastructure
NFVO	Network Function Virtualization Orchestrator
NFV	Network Function Virtualization
NF	Network Function
NSD	Network Service Descriptor
NSR	Network Service Record
NST	Network Slice Template
NS	Network Service
NVI	Network Virtualized Infrastructure
ONAP	Open Network Automation Platform
OSM	Open Source Mano
OSS	Operational Support System
POL	Policy Manager
REST	Representational State Transfer
RO	Resource Orchestrator
SCM	Source Code Mananagement
SDK	Software Development Kit
SDN	Software-Defined Network
SFC	Service Function Chaining
SNMP	Simple Network Management Protocol
UI	User Interface
VCA	VNF Configuration and Abstraction
vCPE	Virtual Customer Premises Equipment
VDU	Virtual Deployment Unit
VIM	Virtual Infrastructure Manager
VLD	Virtual Link Descriptor
VL	Virtual Link
VM	Virtual Machine
VNFD	Virtual Network Function Descriptor
VNFFG	Virtual Network Function Forwarding Graph
VNFM	Virtual Network Function Manager
VNFR	Virtual Network Function Record
VNF	Virtual Network Function
VoLTE	Voice over Long Term Evolution
YAML	YAML Ain't Markup Language
YANG	Yet Another Next Generation

CHAPTER 1

Introduction

In 2017, mobile connectivity rose by 71 percent [1]. By 2022, worldwide mobile traffic will reach 77 exabytes per month. That means nearly one zettabyte per year. These numbers are anticipated since society has evolved in a way where one person uses multiple internet-connected devices. Furthermore, the rise of Internet of Things (IoT) also contributes to the increase in traffic since devices and services that used to be offline are now online, such as cars, sensors, robots, and services like immersive media application, smart-manufacturing, surveillance, among others. These services, which nowadays include a lot of heavy traffic operations like video streaming, rely a lot on the network infrastructure for their connectivity needs [2]. This necessity makes the verticals the key drivers of Fifth Generation of Cellular Mobile Communications (5G) networks adoption [3]. 5G networks need to have high bandwidth, low latency, low power consumption and be cost-effective to meet the traffic and vertical demands [4]. In order to reach these requirements, it is necessary to change the network paradigm and start not relying on the manufacturers' hardware.

NFV is of interest to network operators and providers because it facilitates the development of new mechanisms for the delivery and maintenance of network and infrastructure services [5] [6]. NFV provides the decoupling of Network Function (NF) from manufacturers' hardware by providing the service through software running on Commercial off-the-shelf (COTS) devices. The services are then deployed through VNFs, which can be tailored and adjusted to any demands. However, the migration of hardware-based functions to software raises concerns about software reliability [4], since it is necessary to ensure that the VNFs behave as expected. Consequently, one of the biggest challenges is the validation of VNFs and NSs [7].

Projects like 5GTango[1] try to tackle this problem by having a full Software De-

[1]https://www.5gtango.eu/

velopment Kit (SDK) that validates VNFs and NSs. However, the platform requires high customization, and its integration with other projects is not straightforward. It is, therefore, possible to make improvements.

1.1 Motivation

5GinFIRE[2] is a project funded by the European Horizon 2020 Programme with several partners from all around the world. Its main objective is to provide a 5G NFV enabled experimental framework able to instantiate and support vertical industries while using leading and open source technologies [8]. Identifying itself as a "forerunner experimental playground" [9], 5GinFIRE relies on its experiments to validate the infrastructure. Making sure everything runs evenly is therefore mandatory to have a reliable unified testbed. The experiments are activities that are conducted over the 5Ginfire environment and make use of NSs. These NSs are composed of VNFs, which have to be previously submitted on the 5GinFIRE portal.

A crucial step to take is to guarantee that the packages submitted are well built and ready to be deployed on the orchestrator. Currently, the validation is being carried out manually by the 5GinFIRE portal administrator. However, such a solution is neither sustainable, scalable, or practical. These constraints lead to the necessity of having a fully automated validation [10].

1.2 Goals

The focus of this Dissertation is to provide a tool that allows validation of VNFDs and NSDs. The tool should grant quick debugging by providing explicit and direct logs with enough information to identify errors easily. Moreover, it should not require much configuration and should be lightweight.

A CI server should be deployed and integrated with the 5GinFIRE infrastructure. Subsequently, CI pipelines should be configured on the framework in order to call the validation tool and automate its usage. In the end, it is expected that the pipeline is triggered whenever a developer submits a new VNF in the portal, and the produced tests are performed over the VNFD.

Nevertheless, the developed solution must be independent of the CI server so that it can be used in other situations, for example, without automation associated or with another server.

[2]https://5ginfire.eu/

1.3 Dissertation structure

To make the reader acquainted with the most relevant concepts of this document, Chapter 2 describes the necessary background as well as related work. Chapter 3 provides the requirements needed in order to build a solution as well as a full description of its architecture. Afterward, Chapter 4 gives a comprehensive overview of the system implementation, which includes either the package development as well as the integration with the automation platform. Next, Chapter 5 is presented, which lays out the analysis of the results gathered. Finally, Chapter 6 provides the work conclusion. Lastly, the references are presented.

CHAPTER 2

Background concepts

This chapter gives an overview of all the concepts that are important for understanding this Document. With the primary goal of this Dissertation being the creation of a mechanism to validate OSM VNFs and NSs and integrate it with 5GinFIRE, it was first necessary to understand the 5GinFIRE scope and then learn about automation techniques.
5GinFIRE is a project that aims to provide an open and extensible testing platform for NFV related experiments. Therefore, the first section of this chapter addresses NFV by exposing the reasons for its development and a description of its architecture.
One of the main components of NFV is MANO. Many entities have made efforts do develop their MANO implementation. In particular, SONATA, Open Network Automation Platform (ONAP), Open Baton and OSM are the most popular projects. The first three have a dedicated section with a brief description of their characteristics. A more detailed study is provided for OSM since it is the orchestrator used by 5GinFIRE. With the clarification of these concepts, the next section describes the 5GinFIRE architecture and the project workflow. The last two sections address the DevOps and CI concepts, which are fundamental understanding the automation concept followed on this Dissertation.

2.1 Traditional networks architecture

The traditional network system has always been approached as a physical equipment world. Throughout the years, networks have become quicker, more capable, and resilient; however, they are still struggling to meet the evolving market requirements [11]. For network operators, deploying a new service involves purchasing new physical equipment for each of its features. This approach leads to a set of problems for service providers. [11] gives an in-depth overview of those difficulties, which are described in section 2.1.1.

2.1.1 Current architecture problems

Flexibility

Network operators rely on proprietary equipment. This equipment is usually bundled as one - hardware and software - and limited to the vendors' implementation. This approach leads to a limitation of flexibility and customization of such devices.

Scalability

Being dependent on physical network equipment raises problems on space availability and power consumption. Software-wise, these devices are designed to handle limited data. Once that cap is reached, operators have no options rather than upgrading the device.

Time-to-Market

With the evolution of applications, new services often grow on requirements. So, to implement new services and meet their demands, buying new networking equipment and redesigning the network are challenges that operators have to go through. Service providers are, therefore, delaying the launch of new services, resulting in company and revenue losses.

Manageability

Although networks implement standardized monitoring protocols such as Simple Network Management Protocol (SNMP), Netflow[1], or Syslog[2], vendor-specific parameters are usually monitored using non-standard tools. Thus, with the different variety of devices and vendors, monitoring and controlling logs may become too overwhelming.

High Operational Costs

As previously stated, buying new equipment to sustain new services is expensive. Besides, manufacturers require highly trained staff to deploy and maintain their devices, contributing to the raise of additional costs.

Migration

Considering the situations when no new services are to be launched, after some time, networks and devices have to be upgraded and reoptimized. This update includes on-site physical access and workers to deploy new equipment, reconfigure connectivity, and enhance site infrastructure.

[1]https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html
[2]https://tools.ietf.org/html/rfc5424

Operators need to find a way of providing their services with the same quality while keeping the costs low. NFV was proposed to mitigate the identified challenges. Section 2.2 presents a more in-depth description of this technology.

2.2 Network Function Virtualization

NFV's main idea is to isolate physical network equipment from the service running on it. With this approach, it is possible to centralize network devices on COTS hardware. COTS are equipment for general use that do not require the adoption of proprietary hardware or software. These types of devices are, for example, servers, switches, or storage. Network services are then developed in software that is compatible with the referred equipment [5]. The resources are sufficiently abstracted for network services to make use of them without knowing their location and organization.

This cloud model paradigm is about improving how to implement and control network services; therefore, NFV promises to deliver agile operations, quicker role creation, and efficient use of resources.

In the end, NFV differs from the current network practices mainly in three aspects [12], [13]: (i) it decouples software from hardware, which allows the network service to not be a combination of interconnected hardware and software, making these components independent of one another; (ii) enables flexible network function deployment because, as software is detached from hardware, there is more room to combine these components and performing different network functions faster over the same physical platform; and, as a consequence of the described characteristics, (iii) it allows dynamic scaling because as the network function is instantiable software, it is then easier to scale its performance more dynamically. Figure 2.1 pictures the changes on the migration from traditional networks to NFV.

Currently, the entity responsible for the standardization and development of NFV in Europe is ETSI. To guide the research on this field, ETSI created an Industry Specification Group (ISG) for NFV. ETSI NFV ISG was created in 2012 by seven leading telecommunication operators [14]. Since then, the group released over eighty different documents that cover all the architecture specifications, requirements as well as functional components, their interfaces, Application Programming Interfaces (APIs), and protocols [15].

NFV is intended to address the demands of flexibility, agility, and scalability [2].

The goals of NFV proposed by ETSI are [12]:

- Use COTS hardware to deploy NFs through software virtualization. These NFs will then be called VNFs.

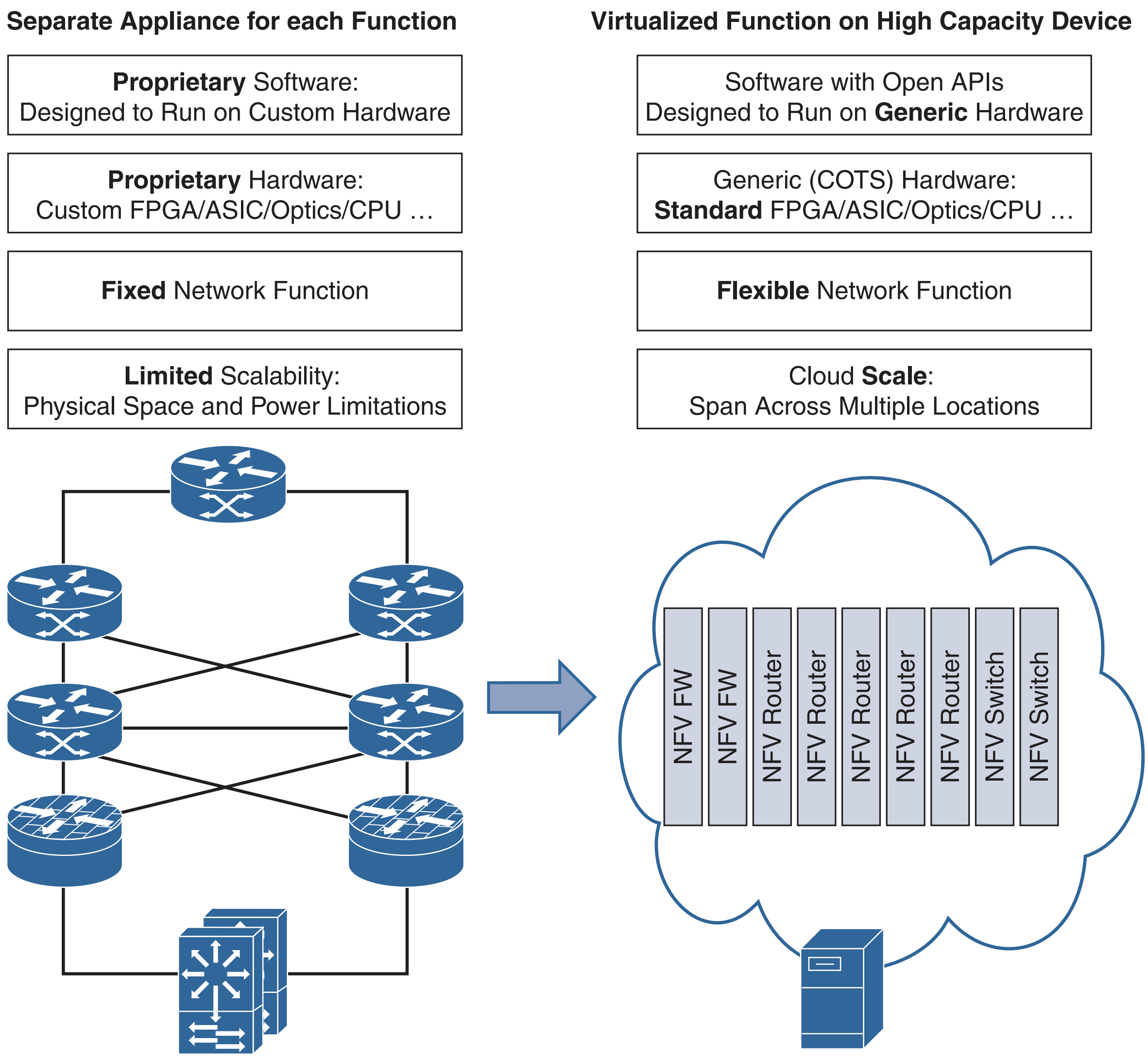

Separate Appliance for each Function
Proprietary Software: Designed to Run on Custom Hardware
Proprietary Hardware: Custom FPGA/ASIC/Optics/CPU …
Fixed Network Function
Limited Scalability: Physical Space and Power Limitations
Virtualized Function on High Capacity Device
Software with Open APIs Designed to Run on Generic Hardware
Generic (COTS) Hardware: Standard FPGA/ASIC/Optics/CPU …
Flexible Network Function
Cloud Scale: Span Across Multiple Locations
NFV FW
NFV FW
NFV Router
NFV Router
NFV Router
NFV Router
NFV Router
NFV Switch
NFV Switch

Figure 2.1: Transition to NFV[11].

- Improve scalability and decouple functionality from location by boosting flexibility in assigning VNFs to hardware. This approach makes it possible to store software at the most appropriate locations, such as data centers.
- Fast service upgrades through software-based service deployment.
- Reduced power consumption accomplished by moving workloads and shutting unused hardware down.
- Standardized and open interfaces between the VNFs, the infrastructure, and the management entities so that different vendors can supply these decoupled components.

In order to achieve the stated goals, ETSI built NFV architecture displayed on figure 2.2.

The referenced diagram shows three main components: the Virtual Network Function, the Network Function Virtualization Infrastructure and the Network Function

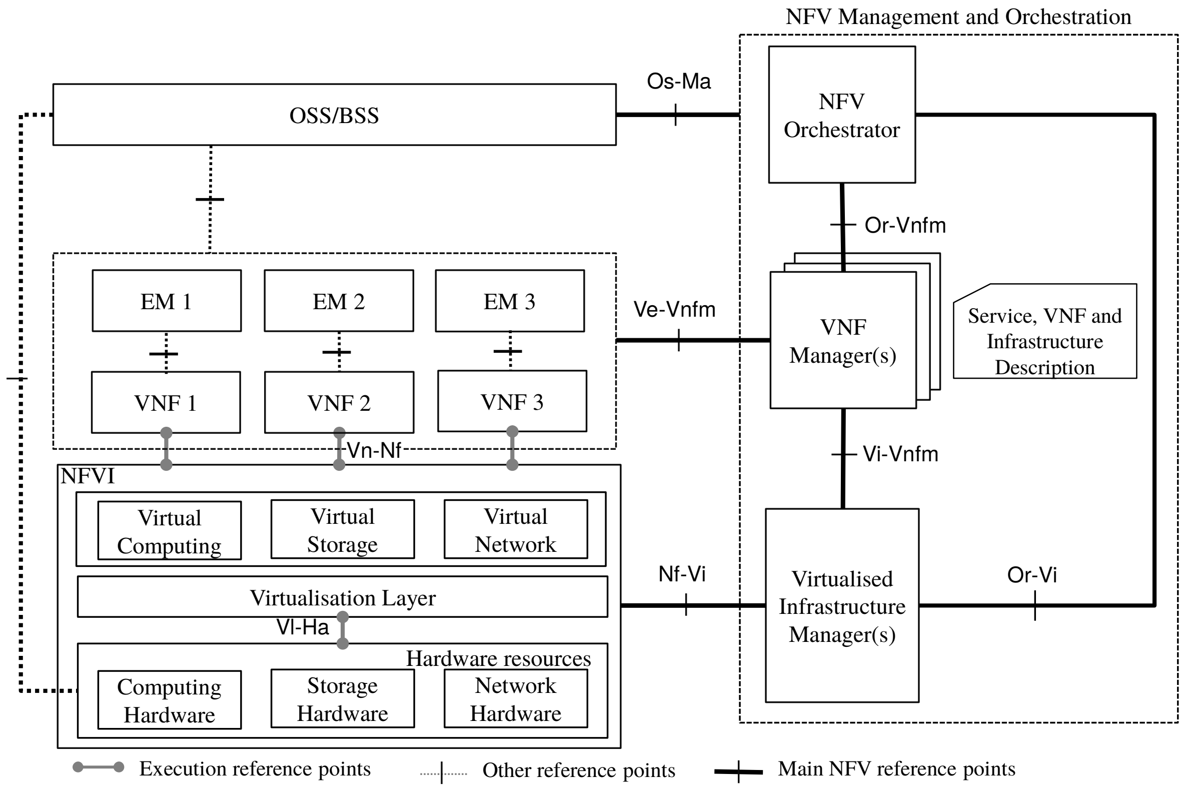


Figure 2.2: ETSI NFV architectural reference[16].

Virtualization Management and Orchestration. A full description about these components based on [11] and [12] is provided in the next following sections.

2.2.1 Virtual Network Function

A VNF is the software virtualization of a NF that can be deployed in a Network Function Virtualization Infrastructure (NFVI) [17]. This NFV component aims to perform the actions of a network device such as routers, switches, firewalls, among others, through software while operating on generic hardware. VNFs use Virtual Machines (VMs) to deploy their software. The VMs are provided by the NFVI. When two or mone VNFs are connected they form a NS.

2.2.2 Network Service

A NS is a group of VNFs described by their functional and behavioral characteristics [17]. NS' objective is to describe the relationship between its constituent VNFs and the links that connect them in the NFVI network. Such connections link VNFs to connection points that provide an interface to the existing network [18].

2.2.3 Network Function Virtualization Infrastructure

NFVI is the combination of hardware and software components that build up the environment where VNFs are deployed [17]. Given that description and figure 2.2, NFVI is composed by COTS, a virtualization layer (which may be, for example, an hypervisor) and virtual resources.

For ETSI, hardware resources, which, as referred above, are COTS, are of three types: computing, storage, and network. Computing resources include both Central Processing Unit (CPU) and memory; storage may be network-attached or local storage; network hardware includes the network interface cards and ports.

The virtualization layer is responsible for abstracting the hardware resources as well as isolating the VNF software from them. Furthermore, this layer communicates directly with the hardware resources making them accessible for the VNF as a VM. In the end, this layer is the component that decouples the software from the hardware.

Figure 2.3 displays the communication provided by the virtualization layer between the physical hardware and the VNFs.

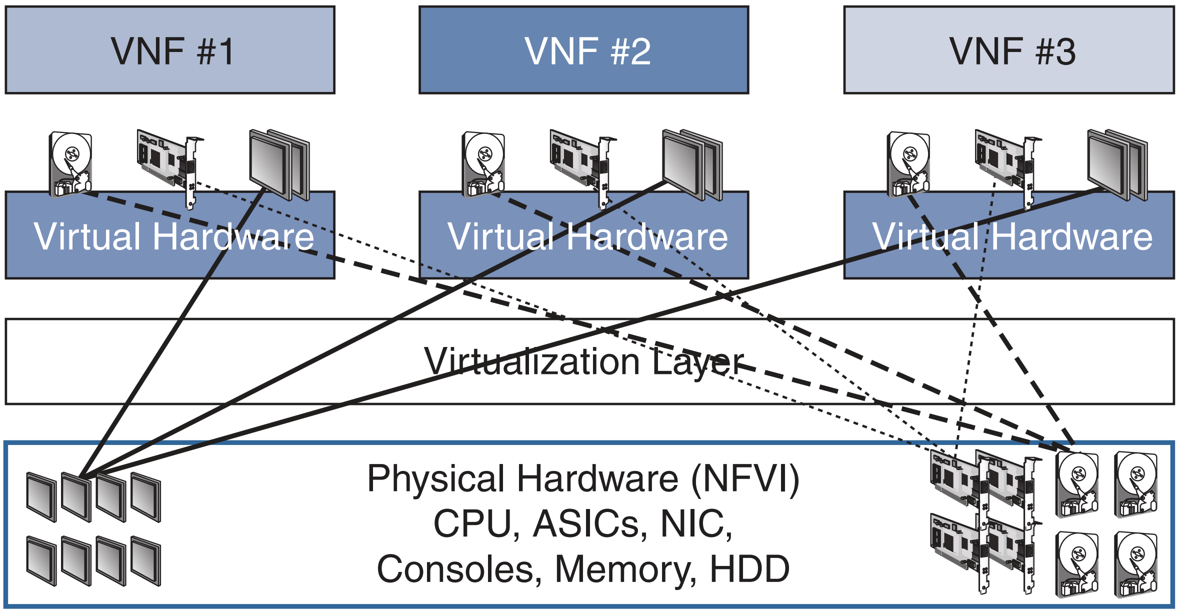


Figure 2.3: Virtualized resources provided to VNFs [11].

2.2.4 Network Function Virtualization Management and Orchestration

NFV MANO covers all the orchestration and lifecycle management operations of the physical and software resources as well as the VNF's. There are three functional blocks on the NFV MANO framework and four data repositories. As it is portrayed on figure 2.4, the functional blocks are the Network Function Virtualization Orchestrator (NFVO), the Virtual Network Function Manager (VNFM) and the Virtual Infrastructure Manager (VIM) and the repositories are the NS catalog, VNF instances, NS catalog and NFVI resources [19], [20].

The mentioned repositories handle all data from the ETSI NFV framework. Each database has a specific function and targets other framework components [20].

The NS catalog is a collection of predefined models that describe how to build and deploy services as well as its functions and their connectivity. On another hand, the VNF catalog is a compilation of models detailing the deployment and functional features of the VNFs available. The NFVI resources repository holds information on the availability of the NFVI resources. Last but not least, the NFV instances repository stores lifetime data on all function and service instances.

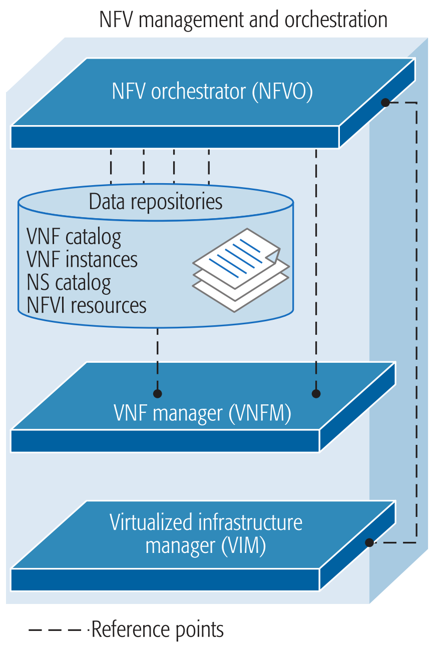


Figure 2.4: ETSI NFV MANO main components [20].

Network Function Virtualization Orchestrator

NFVO has two main roles separated into two distinct categories: resource orchestration and service orchestration. The former includes tasks related to the orchestration of NFVI resources, such as the management of the VNF instances that share resources with the NFVI and providing services that support NFVI access isolated from the VIM. The latter contains duties linked to the lifecycle management of the NSs [19], [20]. The NFVO interacts with all the referred databases [2].

Virtual Network Function Manager

The VNFM is responsible for managing the VNFs' lifecycle [2], [19], [20]. It is possible for a vendor to create their own VNFM which means that one VNFM does not have to be responsible for managing all the VNFs [11]. This functional block has access only to the VNF catalog.

Virtual Infrastructure Manager

A VIM is the component that manages NFVI physical and virtual resources. There may be more than one VIM. Besides the management task, they are also capable of monitoring the performance and status of the hardware resources and, among others, providing network connectivity between VNFs at the VM level. If there are many VIMs, they do not need to be physically located in the same place [2], [11], [19].

With the conceptualization of NFV, several open-source and commercial projects have made efforts to implement a MANO framework (the orchestrator) that meets the described requirements. The biggest projects are: ONAP, OpenBaton, SONATA, and OSM.

2.3 Open Network Automation Platform

ONAP is an open-source project currently backed by Linux Foundation and founded by AT&T and China Mobile, which aims to allow physical and virtual networks orchestration in order to deliver reliable network services faster while keeping the costs low by providing a multi-site and multi-VIM platform [21].

Its architecture relies on two significant frameworks denominated by design-time environment and run-time environment. The former consists of the environment with all the tools needed for the development and improvement of existing network capabilities as well as the management of policies and rules for proper orchestration, and, on another hand, the latter is responsible for the execution of the design-time defined rules and policies [22]. By giving answers for the increasing demands of networks and having proven use cases in real-world situations such as Voice over Long Term Evolution (VoLTE) and Virtual Customer Premises Equipment (vCPE), ONAP has the advantage of being used by big network operators such has AT&T leading to more reliability on the market.

2.4 Open Baton

Open Baton is an open-source project aligned with ETSI NFV that defines its primary goal to build an extensible architecture to orchestrate network services through NFV environments [23].

This orchestrator distinguishes itself from the others because of its features, which are: (i) openness and extensibility because it was developed so it is integratable within heterogenous NFs and cloud infrastructures; (ii) provides VNFM interoperability since not only supports multiple VNFM solutions (by providing mechanisms for new VNFM integrations) as well as a generic VNFM and multiple VIMs; (iii) it provides multi-site deployments and, lastly, (iv) allows different VNF IMs, supporting different VNF deployments [23].

Although Open Baton emerged on the community before other projects like OSM, its infrastructure is not used as much. However, projects like SoftFIRE[3] make use of its features.

2.5 SONATA

SONATA is a vendor-agnostic MANO platform, aligned with ETSI NFV, that provides a virtualization infrastructure for the management and orchestration of network elements in NFV environments [24], [25]. This project was under development from 2015 until 2017 and it is currently being extended by 5GTango[4].

This orchestrator defines that a NS lifecycle is divided into three phases: (i) development, (ii) testing, and (iii) operations. The described stages are taken into account on SONATA's architecture by the SDK, verification, and validation, and service platform blocks [24].

The SDK provides the tools to help developers to build the softwarized network services. The service platform supplies a fully adaptable design of the MANO framework offering service customization in two levels: the orchestration platform may be modified in order to offer support to desired business models and, on another hand, the service developer can influence the management and orchestration operations of their network services platform by configuring, for example, scaling operations [25]. The platform for verification and validation offers advanced mechanisms to qualify VNFs and NSs [24].

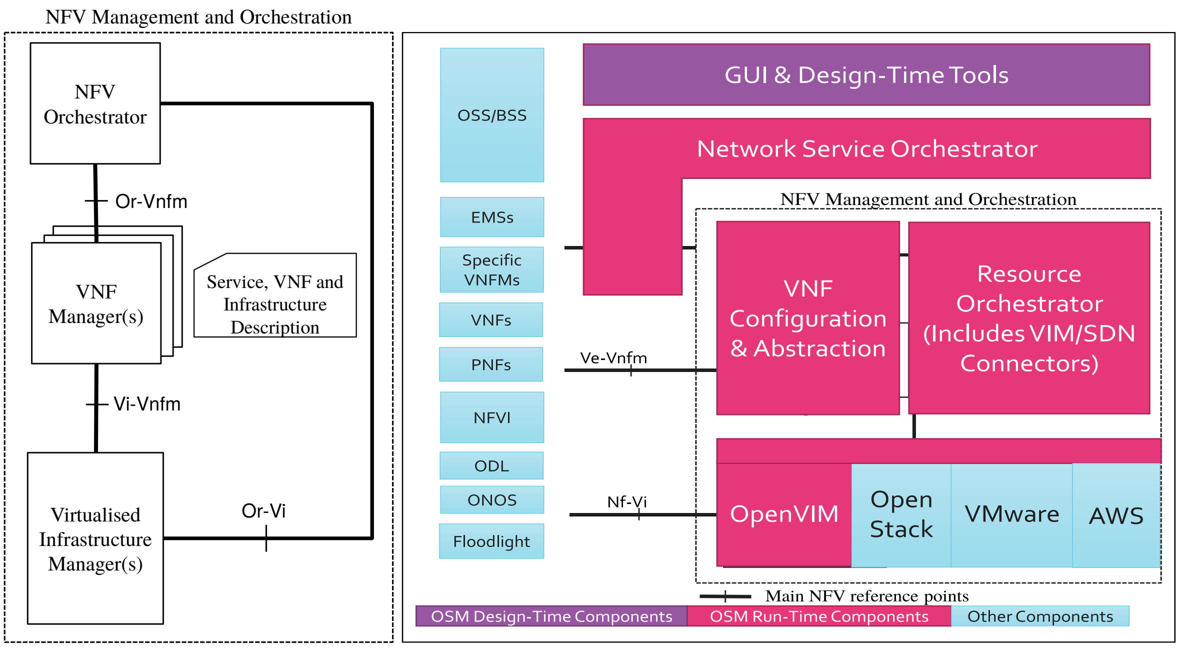
The architectural aspects of the verification and validation component of SONATA are described in section 2.9.1 as they are considered related work on the scope of this Dissertation.

2.6 Open Source Mano

OSM is, as the name suggests, an open source solution for the MANO component of NFV that aims to manage lifecycle, configuration and in-life aspects of the hosted functions [26]. This project is supported by ETSI, therefore, its development is aligned with the ETSI NFV [27]. Figure 2.5 represents that alignment by showing ETSI NFV MANO architectural specification on the left and OSM components over the diagram on the right. On the OSM approach, the NFV orchestrator is represented by the resource

[3]https://www.softfire.eu/
[4]https://www.5gtango.eu/

orchestrator, the VNF manager by the VNF configuration & abstraction and the VIM by OpenVIM[5], OpenStack[6], VMware[7] or Amazon Web Services (AWS)[8].



NFV Management and Orchestration
NFV Orchestrator
Or-Vnfm
VNF Manager(s)
Service, VNF and Infrastructure Description
Vi-Vnfm
Virtualised Infrastructure Manager(s)
Or-Vi
OSS/BSS
EMSs
Specific VNFMs
VNFs
PNFs
NFVI
ODL
ONOS
Floodlight
GUI & Design-Time Tools
Network Service Orchestrator
NFV Management and Orchestration
VNF Configuration & Abstraction
Resource Orchestrator (Includes VIM/SDN Connectors)
Ve-Vnfm
Nf-Vi
OpenVIM
Open Stack
VMware
AWS
Main NFV reference points
OSM Design-Time Components
OSM Run-Time Components
Other Components

Figure 2.5: Comparison between ETSI NFV MANO architectural specification [16] (left) and OSM (right) [28].

Besides, a Graphical User Interface (GUI) is also provided for the interaction between the users and the framework.

When planning OSM, four principles were taken into account that should be followed throughout its development: layering, abstraction, modularity, and simplicity.

First of all, it is crucial to keep the architecture modular and layered so that new features can be quickly introduced and previously implemented ones modified. Nevertheless, the abstraction provided between these layers should be sufficient to make it easy to interact between them and higher-level features. Last but not least, the interaction with the users should be smooth and straightforward. Although it would be interesting to understand the architectural aspects of all of the OSM modules, for this Document, it is more important to understand how they operate and what is their role in the lifecycle of VNFs and NSs. However, to accomplish that, it is imperative to understand what VNF and NS packages are in this context. Each MANO implementation has its specifications (IM) for these packages. In fact, OSM Information Model covers the specifications of VNFD, NSD, Network Slice Template (NST), Virtual Network Function Record (VNFR), and Network Service Record (NSR). Nevertheless,

[5]https://www.openvim.com/
[6]https://www.openstack.org/
[7]https://www.vmware.com/
[8]https://aws.amazon.com/

for this Document, the focus is on the NSD and VNFD since the others are out of the scope of this Document.

Currently, OSM is on release six; however, for the scope of this Document, the focus will be on release five. The main reason for this approach is because 5GinFIRE is aiming to support release five. Another reason is the fact that, currently, the release six white paper is not published yet, so there is no access to the new specifications.

2.6.1 OSM VNF

In OSM, the VNF has the same role as described in the section 2.2.1. In order to onboard the virtualized network function in OSM, the VNFs are distributed as packages which hold information regarding its capacity and functionality aspects. The directory tree presented on figure 2.6 the structure of a VNF package.

```
VNF package/
├── README
├── vnfdescriptor.yaml
├── checksums.txt
├── images/
├── icons/
├── cloud_init/
└── charms/
    └── <charm name>/
```

Figure 2.6: VNF package structure tree.

The package components are (i) the VNFD, which is displayed on the tree as vnfdescriptor.yaml, (ii) the charms, which are inside the charms directory, (iii) additional configurations components such as the cloud_init folder, and (iv) some additional metadata like the icons and the README file.

A VNFD is a YAML Ain't Markup Language (YAML) template file that describes everything related to the topology of a VNF, as well as the resources required by the Resource Orchestrator to deploy the VMs defined in the descriptor. Furthermore, it also contains the primitives that are available for that VNF to execute. It ultimately describes a VNF deployment and operational behavior, and is used in both the onboarding and management of a VNF lifecycle [17].

The charms are optional elements of a VNF, and its specifications are described in subsection 2.6.4.

2.6.2 OSM NS

Just like the VNFs, the NS goal has been already described in section 2.2.2. The NSs are also distributed in packages and contain information about the connection between the VNFs that form the described virtualized Network Service. The structure of a NS package appears on figure 2.7.

```
NS package/
├── README
├── nsdescriptor.yaml
├── checksums.txt
├── icons/
├── ns_config/
├── vnf_config/
└── scripts/
```

Figure 2.7: NS package structure tree.

Similarly to the VNF, the NS package is comprised of (i) multiple configuration files, (ii) metadata, and (iii) a descriptor file. For this Document, however, the emphasis is on the descriptor file defined as nsdescriptor.yaml on figure 2.7.

The NSD is a YAML template that describes a NS by defining the desired topology by providing configurations for the constituent VNFs as well as the relationships between them through Virtual Links (VLs), the Virtual Network Function Forwarding Graph (VNFFG), and all the necessary characteristics for the onboarding and lifecycle management of its instances [17].

2.6.3 OSM IM

In order to achieve consistency in the development of VNFs and NSs, OSM has an infrastructure agnostic Information Model, aligned with ETSI NFV that is able to describe and automate the full lifecycle of VNFs and NSs [29]. The explanation and description of the IM is heavily based on the official IM documentation [18].

The IM may be seen as a tree composed of a group of elements. The elements of each descriptor have two different designations: leaf or container.

A leaf element is a single piece of information that defines a value in the context of the descriptor. The value datatype may be a string, integer, boolean, reference, or enum. On another hand, a container is a component that defines another level in the tree; therefore, it does not have any datatype associated.

In the end, in order to facilitate, all the different elements are called tags. Figure 2.8 portrays the differences between the elements designations.

Since the OSM IM covers many configurations, presenting the full trees would be very extensive. Thus, only the first level of elements is shown.

The first tree level of the VNFD is provided on figure 2.9.

The sub-elements are displayed in bold. In this case, apart from the sub-elements, the rest are only leafs since there are no reference elements.

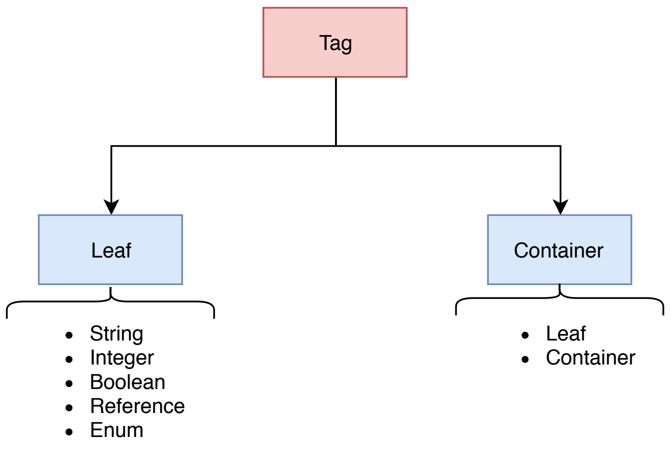


Figure 2.8: Difference between the elements designations in OSM IM.

```
vnfd-catalog/
 ├── schema-version
 └── vnfd/
      ├── id
      ├── name
      ├── short-name
      ├── vendor
      ├── logo
      ├── description
      ├── version
      ├── operational-status
      ├── service-function-chain
      ├── service-function-type
      ├── vnf-configuration/
      ├── mgmt-interface/
      ├── internal-vld/
      ├── ip-profiles/
      ├── connection-point/
      ├── vdu/
      ├── vdu-dependency/
      ├── http-endpoint/
      ├── scaling-group-descriptor/
      ├── monitoring-param/
      └── placement-groups/
```

Figure 2.9: VNFD structure tree.

In terms of leafs, only the operational-status, service-function-chain, and service-function-type do not represent metadata of the VNF. The operational-status, as the name suggests, refers to the operational status of the VNF (init, running, upgrading, terminate, terminated, failed). The service-function-chain and service-function-type are both related to Service Function Chaining (SFC), with the former being the type of node in the SFC architecture and the latter being the type of service function. The ID, schema-version, name, short-name, vendor, logo, description, and version are all metadata elements that are used to give the package a unique identity. Regarding the sub-elements in this first level, they define most of the configurations that are possible to apply to a VNF.

Table 2.1 discriminates the utility of each sub-element presented on the first level of the VNFD tree by describing each one of them.

Sub-element	**Description**
vnf-configuration/	Tag that provides the elements for configuring the VNF. It allows to define if the VNF is configured via a script or a JuJu charm. If the choice is the latter then the primitives have also to be specified. More information regarding JuJu Charms is provided in section 2.6.4.
connection-point	Contains the list of the VNF external connection-points.
mgmt-interface/	Interface over which the VNFM manages the VNF.
vdu	Sub-element on which the configurations of the Virtual Deployment Units (VDUs) that compose the VNF are provided.
vdu-dependency	Informs the orchestrator on which order the configured VDUs should start.
internal-vld/	Provides information regarding the network topology between the VNF internal components such as the VDU.
ip-profiles	List of ip-profiles that describe ip characteristics for a VLs.
placement-groups	Catalog of placement-groups at VNF level. Describe the strategy for the placement of computer resources in a cloud environment, therefore, the VDUs that are within the placemenht-group have to be specified.
scaling-group-descriptor	Defines a group and ratio of VDUs in the VNF that are used as a target for scalling actions.
monitoring-param	Contains monitorable VDU or VNF parameters.

Continues on next page

Sub-element	Description
http-endpoint	The http-endpoint contains a group of endpoints to be used by the monitoring-params.

Table 2.1: VNF possible configurations overview.

Further details on the elements of the VNFD descriptor are present on [30] and [18]. A VNFD example is presented on code block 1.

```
vnfd:vnfd-catalog:
    vnfd:
    -   id: hackfest_basic-vnf
        name: hackfest_basic-vnf
        short-name: hackfest_basic-vnf
        version: '1.0'
        description: A basic VNF descriptor w/ one VDU
        logo: osm.png
        connection-point:
        -   name: vnf-cp0
            type: VPORT
        vdu:
        -   id: hackfest_basic-VM
            name: hackfest_basic-VM
            image: ubuntu1604
            alternative-images:
            -   vim-type: aws
                image: ubuntu-artful-17.10-amd64-server-20180509
            count: '1'
            vm-flavor:
                vcpu-count: '1'
                memory-mb: '1024'
                storage-gb: '10'
            interface:
            -   name: vdu-eth0
                type: EXTERNAL
                virtual-interface:
                    type: PARAVIRT
                external-connection-point-ref: vnf-cp0
        mgmt-interface:
            cp: vnf-cp0
```

Code block 1: Hackfest Basic VNFD [31].

This example, provided on the 6th OSM hackfest, describes a VNF with one VDU. This VNFD starts with a description and definition of basic configurations like the name, ID, description among others. Then, a connection-point named *vnf-cp0* of the type *VPORT* is described.

The VDU defines, in the field count, that one VM based on the image *ubuntu1604*, should be instantiated. The VM should have only one VCPU with 1024MB of memory and 10GB of storage. This VDU should have one external interface named *vdu-eth0*. This interface will be the point of connection between the VNF capsule and the VM. That link is described in the external-connection-point-ref field, which has the VNF connection-point associated. As there is only one VDU, there is no need to specify an internal interface. The management interface is a connection-point and, therefore, the *vnf-cp0*.

This VNF does not handle any cloud-init file nor any JuJu charm. Hence, no service primitive is configured. An example using primitives is shown in section 2.6.4.

Figure 2.10 displays a graphical representation of the VNFD described.

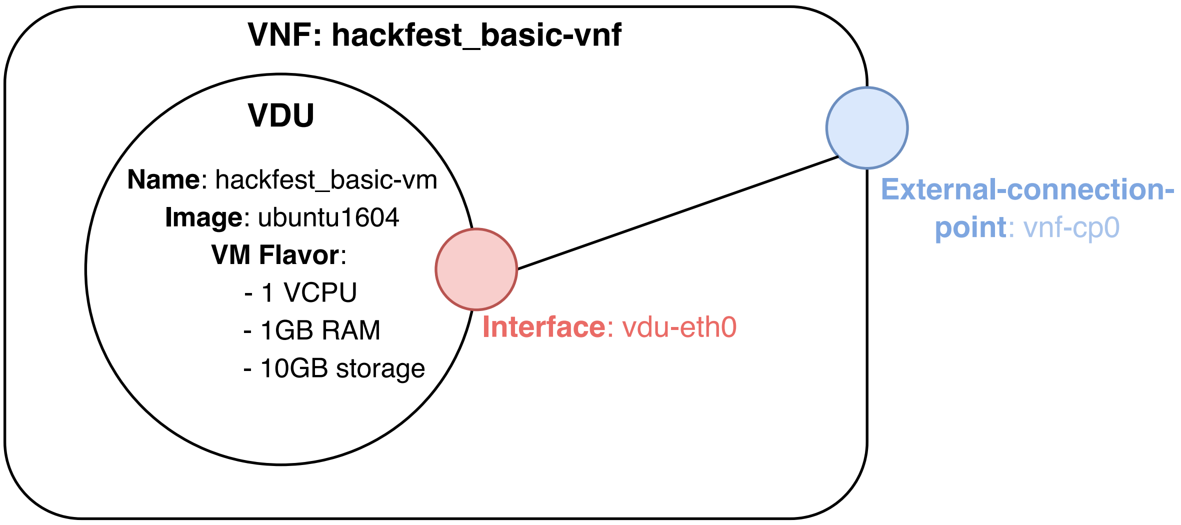


Figure 2.10: Hackfest basic VNF diagram.

As previously stated for the VNFD, in the NSD tree presented on figure 2.11, the sub-elements are displayed in bold. Besides that, the leafs are all metadata. Some sub-elements are common to both of the descriptors, such as the connection-point, scaling-group-descriptor, ip-profiles, placement-groups, and the monitoring-param. The differences in specifications from the VNF and the NS in these elements are minimal. They mostly differ because the VNFs operate with the VDUs and the NSs work directly with the VNFs. Thus, in, for example, the monitoring-param, instead of monitoring the VDU, the NSD refers to monitoring VNFs. Another example is the sub-element vnf-dependency, which has the same role as the vdu-dependency. However, the dependencies should be VNFs from the constituent-vnfds and not, again, VDUs. Due to these reasons, for the NSD, the description of these sub-elements will not be provided. Table 2.2.2 provides the description of the elements that are unique to the NSD.

```
nsd-catalog/
├── schema-version
└── nsd/
    ├── id
    ├── name
    ├── short-name
    ├── vendor
    ├── logo
    ├── description
    ├── version
    ├── connection-point/
    ├── scaling-group-descriptor/
    ├── vnffgd/
    ├── ip-profiles/
    ├── initial-service-primitive/
    ├── terminate-service-primitive/
    ├── input-parameter-xpath/
    ├── parameter-pool/
    ├── key-pair/
    ├── user/
    ├── vld/
    ├── constituent-vnfd/
    ├── placement-groups/
    ├── vnf-dependency/
    ├── monitoring-param/
    └── service-primitive/
```

Figure 2.11: NSD structure tree.

Sub-element	**Description**
vnffgd/	Graph specified by a network service provider that connects network function nodes in a bi-directional way where at least one node is a VNF through which the traffic is directed.
user/	The user is composed of the name and key pair of the person using the NS.
key-pair/	List of public keys to be injected on the NS.
constituent-vnfd/	List of the VNFs that form the NS.

Continues on next page

Sub-element	Description
initial/terminate-service-primitive/	The initial and terminate service primitive have the same configuration. Both of them deal with the definition of primitives to be executed on the NS level. The difference is that the initial service primitive executes the primitives on the initialization of the NS and the terminate service primitive executes on the termination of the NS.
parameter-pool/	Specifies a range of values to use during the configuration of NS.
input-parameter-xpath/	Sub-element which handles a list of XPaths[9] that point to parameters inside the NSD that can be customized during instantiation.
vld/	Deployment model that describes the connection requirements between the VNFs and the NS endpoints. This sub-element resembles the internal-vld element from the VNF in terms of configurations.
service-primitive/	The service primitive works similarly to the initial/terminate-service-primitive; however, this one is not associated with any time frame on the lifecycle of the NS. The primitive is expected to be executed whenever necessary as it is available on service-level for the NS.

Table 2.2: NS possible configurations overview.

Further details on the elements of the NSD descriptor are present on [32] and [18].

A NSD example using the VNF presented on codeblock 1 is provided on code block 2.

This NSD, which was bundled with the previous VNF on the 6th OSM hackfest, provides the description of a simple NS using one VNF and a single VL.

Firstly, all the metadata is set. Then, the constituent-vnfd is specified. Since this NS only holds one VNF, only one vnfd-id-ref is filled and it contains the ID of the VNF previously specified: *hackfest_basic-vnf*.

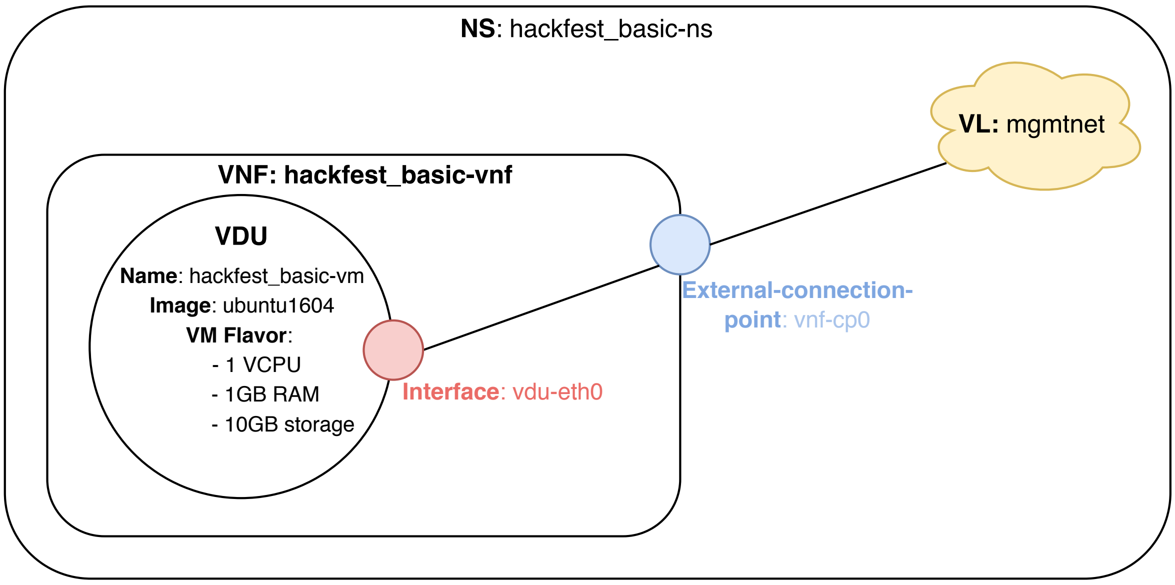
Lastly, the Virtual Link Descriptor (VLD) is created. It is named *mgmtnet*, is a management network and it describes which connection-points should be interconnected. Given the constituent-vnfd, the *hackfest_basic-vnf*'s connection-point

[9]https://www.w3.org/TR/1999/REC-xpath-19991116/

```
nsd:nsd-catalog:
    nsd:
    -   id: hackfest_basic-ns
        name: hackfest_basic-ns
        short-name: hackfest_basic-ns
        description: Simple NS with a single VNF and a single VL
        version: '1.0'
        logo: osm.png
        constituent-vnfd:
        -   vnfd-id-ref: hackfest_basic-vnf
            member-vnf-index: '1'
        vld:
        -   id: mgmtnet
            name: mgmtnet
            short-name: mgmtnet
            type: ELAN
            mgmt-network: 'true'
            vnfd-connection-point-ref:
            -   vnfd-id-ref: hackfest_basic-vnf
                member-vnf-index-ref: '1'
                vnfd-connection-point-ref: vnf-cp0
```

Code block 2: Hackfest Basic NS [33].

should be connected to this VLD.Figure 3.1 displays the connections between the VNF and the NS's VLD.



NS: hackfest_basic-ns
VL: mgmtnet
VNF: hackfest_basic-vnf
VDU
Name: hackfest_basic-vm
Image: ubuntu1604
VM Flavor:
- 1 VCPU
- 1GB RAM
- 10GB storage
Interface: vdu-eth0
External-connection-point: vnf-cp0

Figure 2.12: Hackfest basic NS diagram.

2.6.4 JuJu Charms

Juju is an open-source modeling platform for cloud software service. It allows its users to quickly and efficiently deploy, configure, manage, maintain, and scale cloud applications [34].

There are three main concepts in charms: actions, hooks, and layers. Actions are the programs that the user needs to be executed, hooks are signals that may or may not occur, and layers are an aggregation of actions and hooks.

Charms are built in layers, which means that a charm is a collection of actions and hooks. In addition to those, a layer may import other layers resulting in a new set of functionalities. This approach is right because it allows charms to be reusable and easily modifiable.

Given the layered architecture, when developing a new charm, what is being created is a new layer.

Figure 2.13 and 2.14 present two distinct tree directories. The former presents the structure of a new layer, and the latter presents the final charm after being built, this is, combining the imported layers and the newly created. Starting by figure 2.13, five main files need to be configured: metadata.yaml, actions.yaml, layer.yaml, reactive/action.py, and actions/action.

The metadata.yaml includes all the high-level information of the charm, such as the creator name, description, among others. The layer.yaml file states all the layers on which the new layer is based. Two layers are mandatory when developing a new charm in the OSM context: the basic and the VNF proxy layer. The former is required for all the charms because it contains the necessary configuration for making a charm work. The latter is necessary to set configurations to make the charm a proxy charm. In the OSM context, proxy charms are used because they are the bridge of communication between the OSM infrastructure and the deployed VMs. The actions.yaml contain the high-level description of the actions implemented on the charm. However, to perform the actions, there are two extra steps: the actions folder has to be created, and for each necessary action, a new script, which has to be an executable file, has to be added to that folder. In the tree, the script is presented as actions/action, and it is the connection point between the signal to perform the action and the reactive platform. Therefore, a Python[10] script containing the actual implementation of the action must be created and stored on the reactive folder. The reactive programming pattern allows the charm to respond to changes in state, including lifecycle events, in an asynchronous way [35], [36]. In the tree, this is mapped as reactive/action.py

After these steps, the charm has to be built. In this process, the configurations of the layers described in the layer.yaml and the code of the new layer are joined and form the brand new charm, which will be ready to use. The final tree after this process is presented on figure 2.14.

[10]https://www.python.org/

```
$JUJU_REPOSITORY/layers/
└─ <charm name>/
   ├─ README
   ├─ config.yaml
   ├─ icon.svg
   ├─ layer.yaml
   ├─ metadata.yaml
   ├─ actions.yaml
   ├─ actions/
   │  └─ action
   ├─ reactive/
   │  └─ action.py
   └─ tests/
```

Figure 2.13: Charm new layer structure.

```
$JUJU_REPOSITORY/builds/<charm
name>/
│
├─ requirements.txt
├─ README
├─ icon.svg
├─ copyright
├─ tox.ini
├─ config.yaml
├─ actions.yaml
├─ Makefile
├─ actions/
│  └─ action
├─ reactive/
│  └─ action.py
├─ deps/
├─ bin/
├─ hooks/
└─ tests/
```

Figure 2.14: Charm new layer after building.

Having charms by themselves does not add anything to OSM. They are only useful if there is a way of mapping the charms actions to the descriptors. As described in section 2.6.1, the VNFD holds a set of operations to define primitives. The definition of these primitives provides the necessary connection between the descriptors and the charms since they are responsible for calling the actions to be run on the VNF. The role of these primitives in the OSM ecosystem is described in section 2.6.5.

2.6.5 Open Source Mano architecture

Looking at figure 2.15, the modules that compose OSM's architecture are easily identified: the OSM client as well as the lightweight User Interface (UI), the Northbound Interface (NBI), the Lifecycle Manager (LCM), the VNF Configuration and Abstraction (VCA), the Resource Orchestrator (RO), the Policy Manager (POL) and the Monitoring Module (MON) components. There are also common databases. The communication between these modules occurs via a Kafka[11] bus. One multi present component is the OSM IM.

According to figure 2.15, the OSM access point to the users occurs via the OSM client or the lightweight UI. As it was stated before, OSM was envisioned as a simple system. Therefore, having a transparent interaction with the users is mandatory.

Lightweight UI and the OSM client were developed to provide a fluid user experience by offering straightforward management of the VNFs' and NSs' lifecycle. It also provides

[11]https://kafka.apache.org/

real-time data of the virtualized services and functions as well as a full description of network topologies. The communication between this module and the other OSM components takes place via the Northbound Representational State Transfer (REST) API (NBI). The OSM client, is a Command Line Interface (CLI) tool that replicates most of the functionalities of the lightweight UI.

OSM's NBI, which is based on NFV SOL005 [37], is the hidden access point between the user and OSM's functionalities.

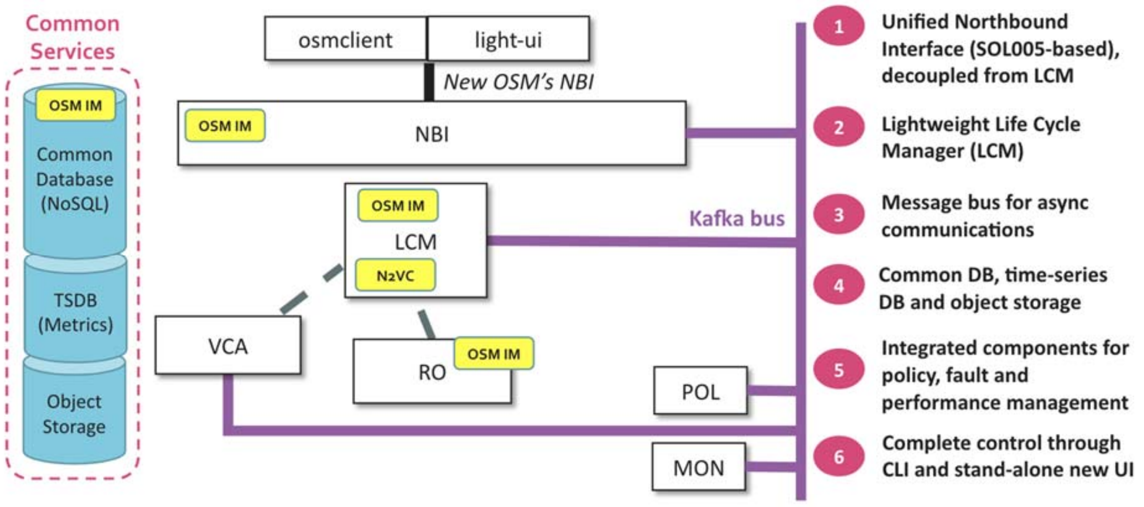

Common Services
OSM IM
Common Database (NoSQL)
TSDB (Metrics)
Object Storage
osmclient
light-ui
New OSM's NBI
OSM IM
NBI
OSM IM
LCM
N2VC
Kafka bus
VCA
OSM IM
RO
POL
MON
1 Unified Northbound Interface (SOL005-based), decoupled from LCM
2 Lightweight Life Cycle Manager (LCM)
3 Message bus for async communications
4 Common DB, time-series DB and object storage
5 Integrated components for policy, fault and performance management
6 Complete control through CLI and stand-alone new UI

Figure 2.15: OSM architecture [38].

Having now the context of the VNF and NS packages and given the architecture presented in figure 2.15, the interaction between these components and modules can now be described. Figure 2.16 presents a high-level overview of the operations on the OSM environment.

Considering the information contained in the VNF and NS packages, their content can be split into two groups: in the case of the VNFs there are the resource descriptions and the management procedures. On another hand, the NS packages have information about topology and management procedures, as well.

When a new NS is onboarded into OSM, the resource descriptions and the topology information go through the NBI to the RO.

The RO module orchestrates the resources available for the OSM environment. It manages and controls the allocation of resources through multiple geo-distributed VIMs and multiple Software-Defined Networks (SDNs) controllers. Therefore, the RO deploys the necessary VMs in order to match the resources and topology manifested in the descriptors using the VIM connectors. In figure 2.16, the purple lines illustrate the described workflow.

The other group of content, the management procedures, are also driven by the

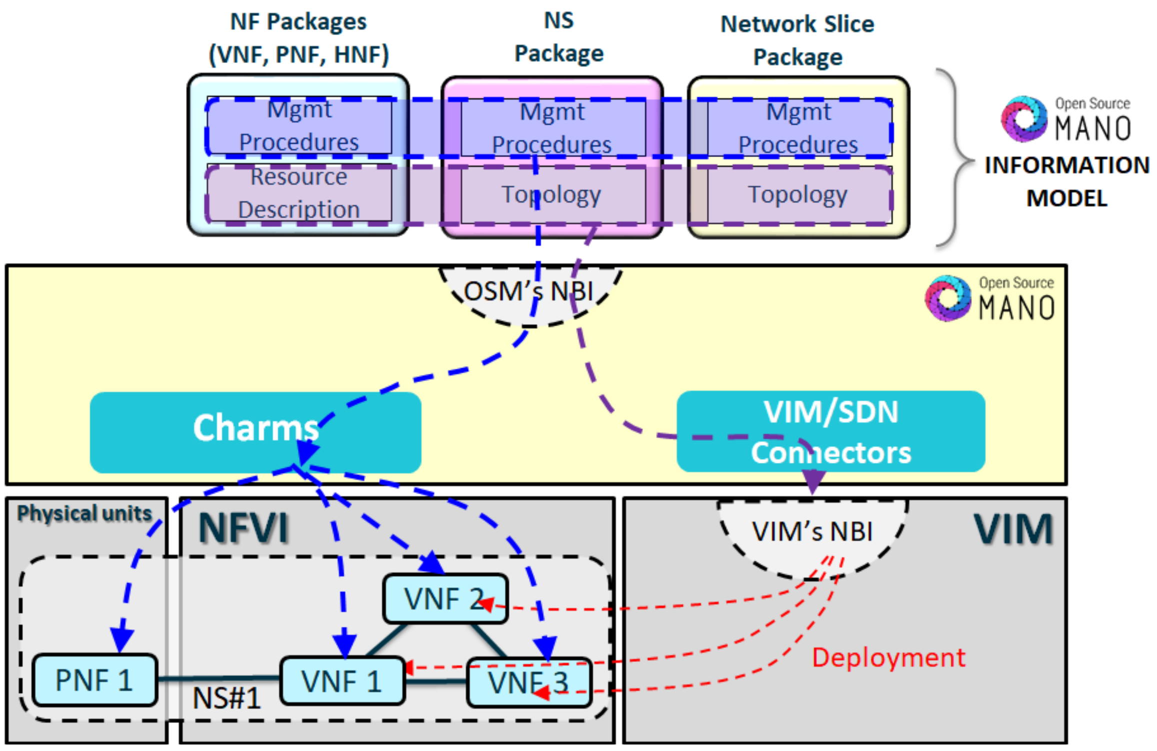


Figure 2.16: Interaction between OSM modules over the deployment of NSs and VNFs [29].

NBI. Nevertheless, they follow a different path since this content is forwarded to the VCA. The VCA works as a VNFM. Consequently, it is the component responsible for enabling configurations to/from the VNF. When allied with JuJu, this module is in charge of signalizing the VNF to perform a specific action. The actions are encapsulated in charms. The blue lines highlight this workflow in figure 2.16.

In conventional networks, day zero involves the configuration of all the physical equipment; this is, connecting all the cables and ensuring connectivity between them. Day 1 is related to making the network ready to work. To do this, the Operational Support System (OSS) extends to all hardware its configurations, including the final network and its neighbors. Licenses injection is also performed at this stage. Finally, day 2 includes all the configurations to keep day-to-day operations running.

When it comes to NFV, the significant differences between the traditional network configurations happen on day zero and day one configurations. However, day two is the same for both approaches. As there is no need for specific hardware configuration in NFV the NSs and VNFs deployments are handled by the orchestrator (MANO component in NFV). Due to this reason, and since the configuration of the components is done basically at the same time as the deployment, day zero and day one co-occur. As OSM implements the ETSI MANO specifications, the same path is followed on its lifecycle.

In OSM, the day zero configuration is done via the cloud-init file. When a NS is instantiated and the request arrives to the RO, this module starts the process of deploying VNFs. To do so, the RO contacts the VIM to request the necessary VMs with the requirements specified on the VNFD. With the VMs deployment, the cloud-init file present on the VNF package is injected and its configuration takes place. These configurations are not related to the VNF itself but only to the configuration of the Virtual Machine.

Happening basically at the same time, when the RO completes the deployment of the VMs, day one configuration takes place. The initial-config-primitive is run for each VNF, and the charm actions that are associated with this primitive are executed.

Regarding day two configuration, the primitives that can be executed are the config-primitive or the service-primitive. If the NBI receives a request for a NS primitive, then the sequence of charm actions associated with the service-primitive are executed. On another hand, if the request is for a VNF primitive, then the config-primitive is triggered, and its sequence of charm actions is run. The request may also be about a scaling operation. A scaling operation may require the deployment of new VMs. If that is the case, the RO needs to take care of that deployment, and the additional configuration should be provided. Apart from that, the primitive process is the same as before. However, the primitive triggered is the pre/post-scaling-primitive, which executes the sequence of charm actions described in the VNFD.

2.7 5GinFIRE

5GinFIRE is a project that started in 2017, and it is expected to end in 2019, funded by the European Horizon 2020 Programme, which aims to build an extensible 5G-NFV based ecosystem focused mainly on vertical industries. This project is identified as a "forerunner experimental playground" since it intends to be the platform where new components, architectures, or APIs are tried before being deployed into production [9].

The described motivation relies on two principles: (i) being driven by ETSI standards and open source code and (ii) focusing on automotive and smart cities verticals [39]. The former principle is crucial since it is one of the aspects on which 5GinFIRE stands out, given that it is the first platform to have leading standardization practices at its core [9].

Figure 2.17 presents the 5GinFIRE architecture. Although the overall concept is, as previously stated, aligned with the ETSI NFV architecture, the project aims to extend it, so it becomes suitable for every vertical's requirements and specifications [9].

From figure 2.17, is is possible to identify four main architectural blocks: (i) the Experimental Instances of Verticalss (EVIs), (ii) the 5GinFIRE Network Virtualized

Infrastructures (NVIs), (iii) the automated MANO, and (iv) the 5GinFIRE design and architecture framework.

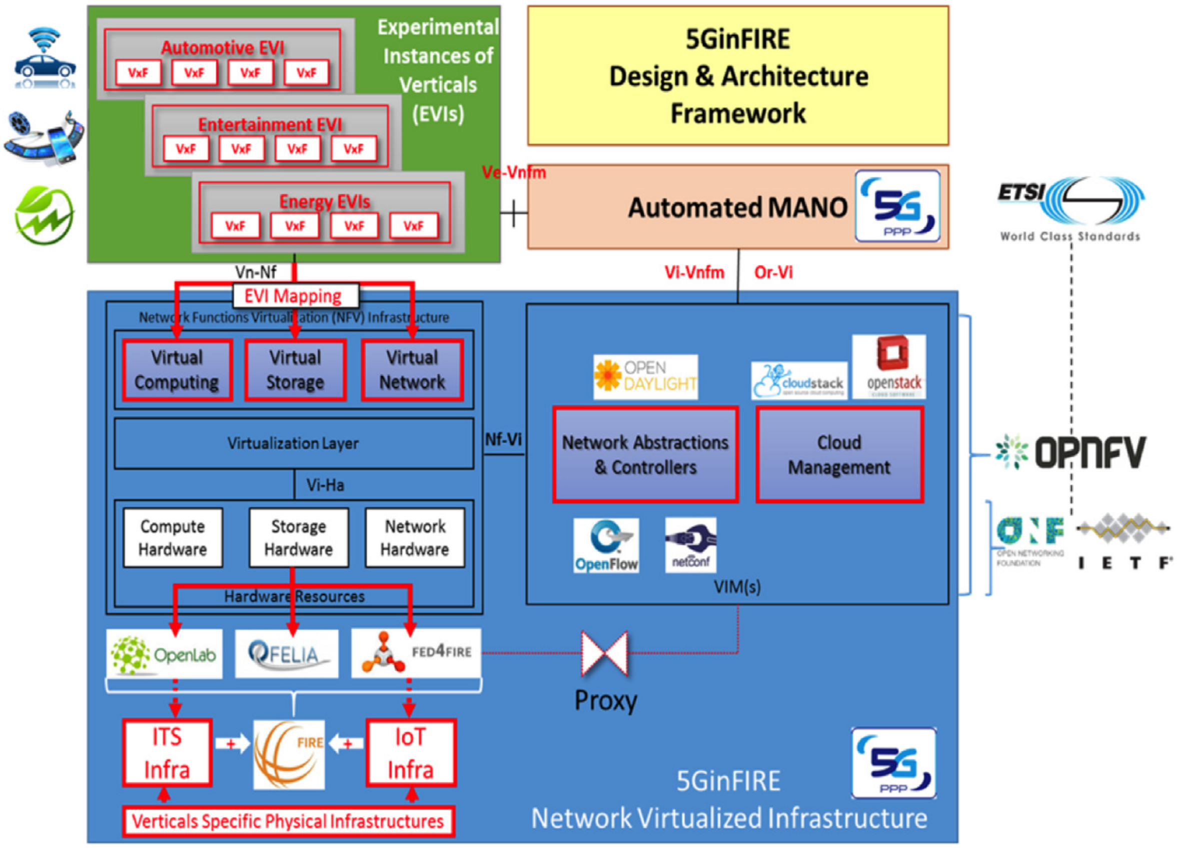


Figure 2.17: 5GinFIRE architecture [8].

The EVIs is the agglomeration of the multiple virtual functions from each vertical [9].

The 5GinFIRE NVI is comprised of the NFVI, which, as previously described in section 2.2.3, offers the resources for the VNFs execution by providing the virtualization layer to abstract the physical hardware from the VNFs. This block also holds the VIMs, which are responsible for the management of the infrastructures' operation. 5GinFIRE is a geographically distributed multi-VIM environment. This approach raises resources placement challenges for each EVI. This issue is addressed by the automated MANO, which, for this project, is OSM. Therefore, the auto-MANO block is responsible for the orchestration and lifecycle management [9].

Last but not least, the 5GinFIRE design and architecture framework contains all the APIs and platforms to provide facilities for experiences and integration of new services [9].

There are multiple designations for each user on the scope of the project. In order to understand each user's interaction and role within the project, the definition of each stakeholder is described in table 2.3.

Actor	Description
Experimenter	User that takes advantage of the 5GinFIRE environment to deploy an experiment.
VNF Developer	User responsible for uploading the VNF and NS on the 5GinFIRE portal.
Testbed Provider	User that provides the testbed and is responsible for its administration, configuration, integration, among others.
Experimenter Mentor	User responsible for keeping track of the experiences status and resource usage.
Services Administrator	User that mantains the 5GinFIRE infrastructure.

Table 2.3: Possible roles on the 5GinFIRE environment [9], [40].

Figure 2.18 shows the different collaborators of 5GinFIRE and their role in the project. University Carlos III of Madrid provides the centralized OSM deployment that orchestrates all the testbeds. The testbeds give different test platforms for different verticals applications such as automotive, media, health, among others, geo-distributed in many different locations. Finally, the 5GinFIRE portal, developed and supported by the University of Patras, offers the connection point between the experimenters and VNF developers and the OSM and testbeds.

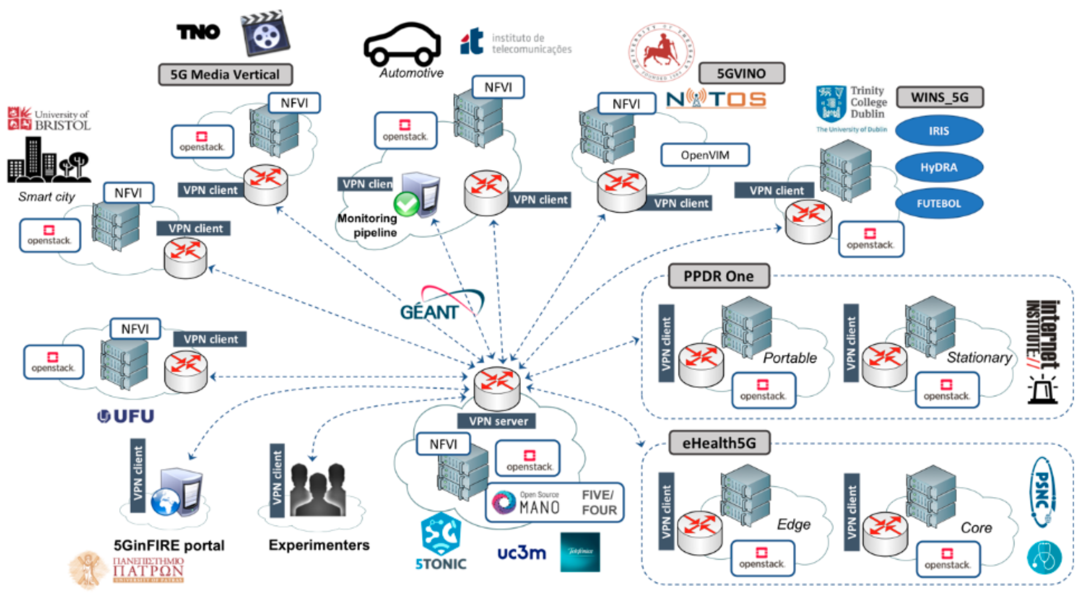


Figure 2.18: 5GinFIRE colaborators and their purposes (adapted from [41]).

The experimentation workflow incorporates each one of the actors and components. Firstly, the VNF developers construct the experiment VNF by providing an OSM compliant VNFD and uploading it into the 5GinFIRE portal. The experimenters are then able to develop the NS with the submitted VNF. Once finished, they can select the testbed in which they want to deploy the experiment and define some other details such as metadata or scheduling. The experiment is then submitted to approval. The experimenter mentor is responsible for taking care of the submitted experiment and has to ensure that the testbed owners and the experimenters are on the same page regarding scheduling, amount of resources, among others. At this point, it is also necessary to ensure that the VNFs utilized by the NS can be deployed. As a consequence, the service administrator has to validate the VNF manually to ensure that the package is ready to be on-boarded. When everything is set, the mentor approves the experiment submission, and the onboarding occurs.

On this workflow, one critical step is the manual validation of the VNFs since it consumes much time, and it is not scalable. Therefore, the need for an automated process arises [10]. This Dissertation aims to provide an automated solution for this step and integrate it on the 5GinFIRE design and architecture framework.

Once the package is validated, the experiment starts. The centralized OSM receives the request to deploy the NS utilizing its constituent VNFs and which VIM (testbed) should utilize resources from. Then, it proceeds to instantiate the necessary VMs, and the required configurations are applied. Once everything is set, the experiment is ready to operate, and the process is completed.

2.8 DevOps

Market requirements are not a problem just for networks. Software also suffers from constant customers' demands since users expect fast delivery of their unceasing new features. Frequent releases are not only essential for fulfilling such requisites but also to create an advantage in the market [42]. Although the usage of Agile [43] methodology brought optimization related to software development, operation tasks are not included in these procedures [44], since the goals of each sector are misaligned: the development teams aim for change whether the operations teams strive for stability [45].

DevOps, which was firstly introduced in 2009 by Patrick Debois, acknowledges the need to incorporate software development and operational deployment [46] continually. Derived by the combination of Development (Dev) and Operations (Ops) [47]–[50], it composes a paradigm that enables the collaboration between developers and operation teams, resulting in more efficient teamwork [49]–[51]. By providing such interconnection, DevOps extends Agile [44], [47], [52].

There are four key enablers of DevOps: culture, automation, measurement, and sharing. Regarding culture, both operations and development teams must make an effort to participate in each other's tasks in order to be aware of what is happening at each end. Build, deployment, and testing automation allows faster feedback contributing to the referred gain of efficiency. Measurement embraces the monitoring by collecting metrics not only from the deployment itself but also from the developers. Furthermore, collecting system logs should be another collaborative measurement task between both teams. Finally, sharing is about spreading knowledge, specifically by providing information about development tools or, on another hand, techniques for managing the infrastructure [53].

The main goals of DevOps are to continuously deliver high-quality services while emphasizing simplicity and agility as well as blend the development and operations tasks by encouraging collaboration and trust [52].

DevOps main components are pictured on figure 2.19, which is based on [54]. On the referred image, it is possible to acknowledge four stages within DevOps: CI, Continuous Testing (CT), Continuous Monitoring (CM), and Continuous Delivery (CD).

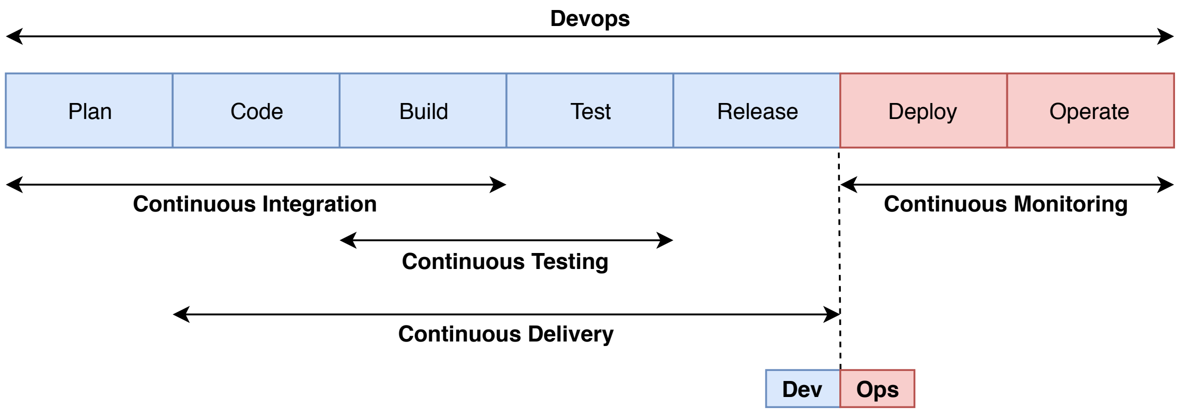


Figure 2.19: DevOps different components.

Continuous Testing is the component in which tests are configured to run automatically as soon as the code is committed to the repository. This approach reduces the time between the introductions of errors and their detections. Continuous Monitoring provides the monitoring of the hardware and software after deployment. Continuous Delivery is the methodology of continuously ensuring that the software is ready to be deployed in the production environment. Lastly, Continuous Integration is an automatically triggered process that provides code testing and validation and its subsequent packaging in order to be deployed later. There should be multiple code submissions for CI during the day, and failure details should also be provided [54], [55].

The work described in this Document is inserted on the DevOps approach, most precisely on the CI step. Thus, the next subsection, 2.8.1, provides a more detailed overview of this process.

2.8.1 Continuous Integration

In software engineering, Continuous Integration was firstly introduced by Grady Booch in 1991 [56] as the practice of interactively building software once a day. Nowadays, CI is, as previously stated, a software development practice where whenever a new code change is committed, and automated build is triggered. The submission is verified in order to detect and solve integration errors quickly [57].

The CI definition is associated with the term "build". In this context, building is more than compiling. It defines a combination of processes such as compilation, testing, inspection, deployment, among others [58]. In the end, it is the operation that holds the necessary processes that the code should be put through in order to reach the final stage. The definition of what is meant to execute in each build is done via a configuration script.

The code produced by developers is usually committed to a version control repository. The repository is used to keep track of the software changes between users. This approach also provides a centralized access point for the source code. By applying CI to the repository, it is possible to verify whenever a change occurs and perform the desired tasks through building [58].

A CI server is a tool that allows the automation of the development process by being able to test, build, and deploy code changes submitted in a version control repository. It is also responsible for holding the configuration script that composes the defined build.

Although not necessary, a CI server is useful since it provides a set of helpful features such as a convenient dashboard for results visualization as well as scheduling features [58].

Multiple tools provide CI frameworks. A detailed overview of the most popular ones is described further in this Document in section 3.2.2.

2.9 Related work

2.9.1 5GTango

With SONATA, 5GTango introduced a verification and validation SDK that aims to validate VNFs in two different stages: the first stage contains three different steps, which are: (i) structural validation of the descriptor, (ii) functional testing and (iii) performance evaluation; the second stage refers to a deeper level of testing. This next level involves a fully test of the virtualized function by using test plans according the function of the VNF [59].

The first phase starts by validating the structure of the VNF. For that, there are four tests associated: syntax, integrity, topology, and custom rules. The syntax test validates the submitted descriptors regarding the 5GTango schemas; the integrity test checks if all the fields have correct values; the topology test verifies the sub-networks formed inside the VNFD and NSD; last but not least, the custom rules are an open concept for users to validate whatever they need. To do so, the developer must provide another YAML file with the rules that wants to verify. The functional testing at this stage happens by deploying the NSD on a NFV emulation platform and verifying if the deployment works as expected. Lastly, the last evaluation measures the performance of the network services when instantiated in multiple scenarios [59].

For the second phase, 5GTango introduced the concept of test plans. This concept is defined by the association of tests with the NS using a new descriptor. For this, the user has to define which functional tests wants to perform on the service and write them using a YAML file. The test plan is then packaged with the NS and deployed. After the execution, it is possible to get results and metrics in order to understand the performance of the NS towards the test [59].

Although it is a complete solution, there are a few reasons why it is not feasible to implement 5GTango's validation and verification system on 5GinFIRE. The first reason is that despite being possible to implement just the validation and verification module without the whole SDK, it still demands many configurations. Another reason is that 5GinFIRE needs a fully automated service, as previously described. The second validation phase of 5GTango requires human configuration of the test plans, which does not seem possible to automate yet since it requires detection of the type of service it is described on the packages.

2.10 Summary

This chapter started by addressing the traditional network problems, with section 2.1 being reserved for this matter. It was possible to understand that network operators have trouble with releasing new services without having to deal with problems such as high costs, lack of flexibility, among others. Such problems lead to delays in the delivery of new services, resulting in market losses.

NFV is exposed as the solution for these problems in section 2.2. With the decoupling of software from hardware, NFV brings flexibility, agility, and scalability solutions for network operators. For this reason, it is one of the most promising technologies of 5G networks.

Sections 2.3, 2.4, 2.5 and, 2.6 have descriptions of the main features of each orchestrator. All of the orchestrators are alike, although Open Baton is the less used.

SONATA, on another hand, provides the most features outside orchestration, such as VNF and NS validation, which constitutes related work to this Dissertation. OSM has the most in-depth description. The main component described is the IM, as it is the focus for the development of the validation tests. The IM is very wide; it offers many different configurations and handles multiple datatypes, providing a huge range of possible descriptors set.

5GinFIRE architecture and workflow are described in section 2.7. Its architecture is aligned with ETSI NFV using OSM as the orchestration engine and JuJu Charms (2.6.4) as the VNFM. A simple usage of the platform is by uploading a VNF package into the portal, designing its corresponding NS, waiting for the approval of the experience, and then deploy it on the desired VIM. It is at the submission stage that the solution proposed by this Dissertation should be applied.

Sections 2.8 and 2.8.1 issue the overview of both concepts, and it described their main components. CI is part of the DevOps methodology. Therefore, DevOps is presented as the theoretical background of the CI concept. The CI server is the framework that has to be integrated with 5GinFIRE in order to produce automation of the validation.

Finally, section 2.9.1 addresses the related work regarding this Document. 5GTango project is the maintainer of SONATA. As part of its development, an SDK that validates VNFs and NS was developed. The tool provides structural and functional validation. None of the approaches is automated, although the former could be. Nevertheless, the latter relies on the creation of a template file that describes the tests to be performed, which can not be automated. The installation and integration of this framework with 5GinFIRE was also not straightforward. For these reasons, this work is not a viable solution for the problem stated.

CHAPTER 3

Architecture and Specifications

To achieve a final architecture, it is necessary to make choices in terms of what tests to make, which automation tool fits the requirements and how to integrate all components into one final architecture. This chapter provides an overview of all the specified challenges.
Section 3.1 starts by addressing the problem statement. Then, in section 3.2, the type of tests to be developed are defined. Since syntactic, semantic, and reference validation is intended to be performed, it is necessary to understand the range of values and tags that it is necessary to deal with and their interconnections. The second challenge focused on this section is the choice of the CI server. Subsection 3.2.2 starts by outline the requirements of a CI tool in order to be feasible for the final architecture. Then, the most common automation servers are analyzed, their performance over the defined requirements is provided, and the tool is chosen. Finally, section 3.3 describes the proposed solution. Subsequently, two architectures are presented: firstly, the validation tool's and then the 5GinFIRE automated platform's.

3.1 Problem statement

Given the 5GinFIRE scope, described previously in section 2.7, 5GinFIRE developers can submit VNFs through the portal. However, these submissions are not controlled, since there is no validation on whether the package submitted contains a valid VNFD. While manual validation was already carried out, it is time-consuming as it is usually a trial-error task. The aim is, thus, to create an automated process so that when a new VNF is submitted through the portal, the validation process is activated, the tests are conducted, and the results are returned to the experimenters. Nevertheless, the tests applied should be platform-independent so they can be used in other scenarios.

To this end, it is necessary to develop validation scripts, configure a CI server, and integrate this system with the 5GinFIRE architecture.

3.2 Requirements and specifications

3.2.1 Validation scripts

When thinking about validating VNFs and NSs, there are two possible paths to follow: functional validation or structural validation. Given that the main scenario is to apply validation in the scope of 5GinFIRE and, currently, the validation is carried out manually, one of the main goals is to automate the process. Functional validation implies the necessity of knowing what type of operations the VNFs are meant to perform. Therefore, the process automation would always be dependent on a description by the user regarding which functionalities the VNF has. Another problem is that VNFs rely on VMs to run. The names of the images are specified in the descriptors, but most of the time, they are not available for the 5GinFIRE portal. Instead, the experimenters would talk directly with the testbed providers for them to upload the images directly on their VIMs. Given that the images sometimes contain pre-packaged configurations for the VNF execution, it becomes impossible to test the functionalities without them. Consequently, only structural validation will be performed.

Given the IM specifications displayed in section 2.6.3, three structural tests were defined: a syntactical test, a semantic test, and a reference test. The reference test can also be extended to cross validate VNF with NS. OSM has Yet Another Next Generation (YANG) models provided by Rift.io [60], [61] which define the rules to develop VNFDs and NSDs correctly. These models have to be the base for each one of the tests.

The syntactic test should perform an evaluation of all the tags, evaluating if their name is correct and if they are defined correctly according to the OSM IM trees [30], [32].

The semantic test validates the content of each tag. This test approaches two things: it verifies if the datatype of the tag content is correct and, for tags with pre-defined values, it verifies if the content is within the possible values for that tag. When analyzing the datatypes provided on the OSM YANG models, it was noted that apart from the normal datatypes, there were also custom datatypes, which are defined to specific configurations of the descriptors. These datatypes are the ones that have only a range of selected possibilities that may be assigned to the tags. To facilitate the process, these types have *enum* assigned to their datatype and then a list of possible options associated.

References are used to enumerate dependencies between descriptor elements; therefore, they are a possible tag datatype. As a consequence, a tag can contain a reference to another tag. Nevertheless, that can only happen if the reference value was declared somewhere else on the descriptor, leading to the necessity of verifying if those values are well referenced. Moreover, if both a VNF and NS are provided, it should be possible to validate the NS over the VNF to understand if the dependencies between both descriptors are correct.

Last but not least, the outputs should be straightforward and contain as much detail as possible so that the developer can quickly identify its mistakes.

The aim is to develop the scripts in Python 3.6[1]. Although they can be directly developed adopting the CI server language, this would lead to a restriction of its usage to the CI platform. The intent is to build a solution that is independent of other frameworks so it can be executed anywhere without many requirements.

3.2.2 Selection of the CI server

Regarding the CI server, the first step is to choose between the various options available.

There are four requirements that the CI server needs to meet in order to fulfill the 5GinFIRE demands: (i) it has to be highly customizable, so any operation could be performed from the server, (ii) has to have a REST API so that the integration with the portal is straightforward, (iii) has to be free or at least offer free functionalities that can cope with the projects demands a lastly (iv) has to offer the possibility of the creation of a CI project non-based on Source Code Mananagement (SCM) repositories.

This last requirement is vital for the integration with the 5GinFIRE project. As previously stated in section 2.8.1, the traditional CI approach is to keep listening to a SCM repository, and whenever new changes are committed, the automated scripts are executed. In this case, there is no repository involved as the VNFs arrives directly from the portal. However, the process is fundamentally the same. The difference is that instead of having a code submission, the request from a VNF submission via the portal is what triggers the tests. If none of the CI servers supports configurations outside SCM environments, then another layer will have to be added to the portal, and whenever a package is submitted, it is committed to a repository and the tests are executed from there.

There are several CI tools available. Table 3.1 offers an overview of how diverse technologies accomplish the specifications defined.

[1]https://www.python.org/

	Requirements			
Tools	Customization	Configuration outside SCM systems	Has API	Cost
Jenkins [62]	Wide plugin repository	Yes	Yes	Free
TravisCI [63]	Custom builds with restricted options	No	Yes	Free for open source projects but limited to three concurrent jobs.
BitBucketCI [64]	Good plugin repository	No	No	Free 50 minutes of builds per month and up to five users if working on cloud. No free options for self-hosted service.
CircleCI [65]	Interface integrated options; possibility to configure new features via bash scripts	No	Yes	Free for cloud deployments but limited builds and user.
GitlabCI [66]	Customization via plugins	No	Yes	Free with build time restrictions on cloud; free self-host with features restriction.
Bamboo [67]	Good apps repository	No	Yes	No free options.

Continues on next page

Codeship [68]	Low flexibility	No	Yes	100 builds per month for free.
TeamCity [69]	Wide range of plugins	Yes	Yes	100 builds per month for free and three build agents.
Buddy [70]	Low flexibility	No	Yes	Free for open-source projects on s cloud; self-hosted service without free options.
DroneCI [71]	Wide plugin repository	Yes	Yes	Free on cloud; Paid enterprise version.

Table 3.1: Performance of the different CI servers over the established requirements.

TravisCI and Bamboo are not good choices because none offers free services.

For the development of this work, the possibility of pipelines configuration without relying on SCM systems is essential. Due to this reason, except for Jenkins and DroneCI, the other technologies are not viable for the implementation needed.

Jenkins and DroneCI are a lot similar in terms of features since they both provide high customization, a REST API, and the possibility of configuring a project without having to associate it to a repository. The difference point is on the cost. Jenkins is free without any limitations. DroneCI has two approaches: they offer a free service on the cloud, and an enterprise paid service that has to be deployed by the user. The former is not a viable option for the pretended use case since it only works with SCM repositories. The latter offers a free option with a limitation of five thousand builds per year. Although this number is high enough, between a completely free option and one with limitations, the decision relied on the free option, Jenkins. Furthermore, Jenkins community is more prominent, which solidifies the choice even more.

3.2.3 Jenkins

Jenkins is the chosen CI server, therefore, its configuration is mandatory. This automation tool provides a dashboard, and all the configurations can be done through it. Nevertheless, before configuring the Jenkins environment itself, it is necessary to understand some key concepts of the framework, which are (i) jenkinsfiles, (ii) pipeline, (iii) jobs, and (iv) builds.

A Jenkinsfile is a file that configures a pipeline. To do so, it is usually placed on the SCM repository. Whenever the pipeline connected to it is triggered, it performs the rules associated.

A pipeline is a set of operations that are meant to be performed on the CI workflow. In Jenkins, there are two types of pipelines: declarative and scripted.

The scripted pipeline is usually written directly on the Jenkins dashboard, and it offers more flexibility since the configurations are directly done using the Jenkins base language, Groovy [2]. On another hand, the declarative pipeline limits the user to a more structured approach, and it is more approximate of the traditional CI.

Given the requirements and since the traditional approach will not be followed, the configuration will rely on a scripted pipeline. Therefore, in the dashboard, a job of the type pipeline must be created and then configured.

The builds are the execution of the jobs. Consequently, the process of triggering the code defined on the scripted pipeline is referred to as a build. Whenever a job is built, Jenkins creates a workspace for that job on the machine where it is running. This workspace is used to store possible elements that can be resultant of the build or to provide other necessary elements for the pipeline execution.

In order to configure the pipelines, it is necessary to prepare the environment. The first step to take is to deploy Jenkins on the virtual machine assigned for running the CI server. The server is hosted outside the 5GinFIRE domain; therefore, Jenkins API must be used to ensure the communication with the 5GinFIRE Portal. The CI server must be configured in order to have a scripted pipeline that holds a set of configurations, which allows the execution of the VNFD tests automatically. The portal should be responsible for triggering the pipeline job that runs the tests. The pipeline, when ready, must send a response back to the portal with the results of the tests. All of these tasks, except for the tests, must be configured directly using Groovy.

3.3 Proposed solution

Given the goals described on chapter 1, the aim is to build a fast, lightweight, easy-to-use tool for validating NSD and VNFD. It is expected that this tool provides extensive and straightforward logs that indicate the state of the descriptors submitted for analysis. Although the work is expected to be integrated with 5GinFIRE, the solution proposed does not aim only to serve the project, it aims to serve all the VNF and NS developers that want to assure that their descriptors are well built.

To fulfill the 5GinFIRE demands, Jenkins should be installed, a pipeline must be created, and the configurations to install the validation tool should be performed. The

[2]http://groovy-lang.org/

connection between the CI server and the 5GinFIRE portal should also be made. The pipeline should be triggered every time a package is submitted to the portal. After the tests are performed, the logs should be sent back to the portal. Although the scripts have the features to validate VNF and NS, the solution will be deployed using only VNFDs since it is what the 5GinFIRE project demands.

3.3.1 Architecture

In order to meet the proposed solution, two steps have to be taken: develop the scripts and integrate them into the CI framework, which is itself integrated with the 5GinFIRE portal. As previously referred, the aim is not to build a solution for 5GinFIRE only, and, therefore, the architecture of the scripts is independent of the CI server.

Starting by the validation tool, its name is *osm-descriptor-validator* and its high-level architecture is pictured on figure 3.1.

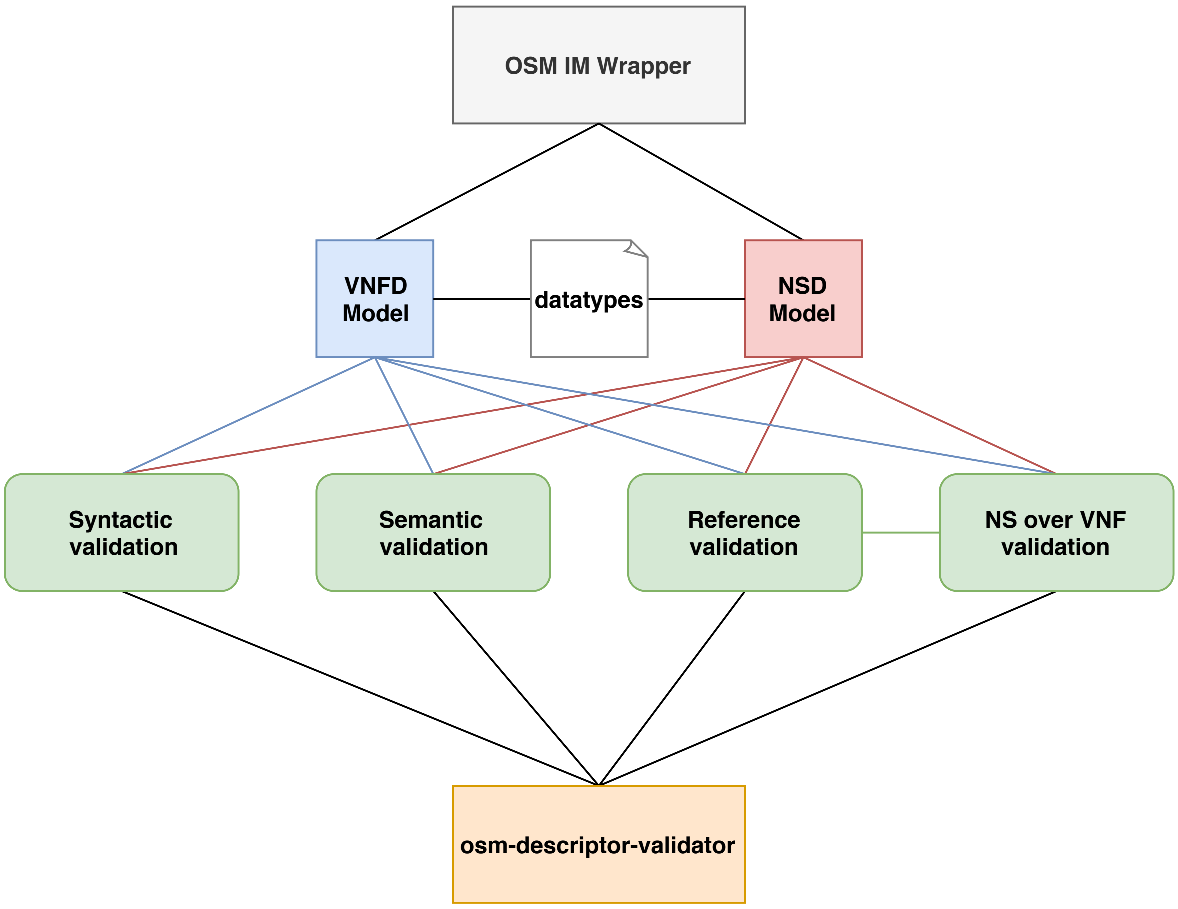


Figure 3.1: osm-descriptor-validator architecure.

The first step is to create a connection point between the YANG models and the package. That is the purpose of the block OSM IM Wrapper. This module handles all the necessary data structures to create the VNFD and NSD model. In this case, the

YANG models provide the skeleton of both the VNF and NS descriptor, all the tags, and relationships between them.

The VNFD and NSD models components are responsible for filling all the necessary information for each tag and container. Both models use the OSM IM Wrapper to retrieve constituent tags of each container. They also use the datatypes module to get each tag datatype as well as their possible values. It is also at this stage that the path of each tag and container is added as well as the reference paths, if applicable.

The four tests pictured make use of both models to get all the necessary details to perform the validation. The syntactic test retrieves the names and paths of each tag and container; the semantic test accesses the datatypes and possible values, the reference test as well as the NS over VNF validation test gather the reference paths. The reference test and the NS over VNF test are connected because the latter reuses some of the former functions to perform its validation.

Finally, the osm-descriptor-validator is the higher-level module that provides the necessary functions for the user to call the tests quickly.

The tests are independent of the models that they are using. Therefore, if it is necessary to adopt any other VNF or NS information model, it is just necessary to substitute the Wrapper, both the VNFD and NSD models and the datatypes, making the solution easily integrable with other IMs.

To achieve an automated solution for 5GinFIRE, the validator should be integrated with Jenkins, which should be able to communicate with the portal. The architecture is pictured on figure 3.2.

The portal communicates with Jenkins via REST API. Whenever a new package is submitted, the portal sends a request to Jenkins to trigger the validation pipeline.

The validation pipeline, which was previously configured in Jenkins, goes through two stages when triggered for the first time: its environment is configured, and then the actual tests run. When the pipeline is created, a Dockerfile[3] is added to its workspace. The Dockerfile contains configurations that are run the first time that the pipeline is triggered, and it sets up the environment with all the tools that are necessary to run the tests and communicate the results successfully. The environment configuration only happens the first time the pipeline is triggered. Jenkins then saves the Docker container and runs all the jobs there.

Five stages should compose the pipeline. When Jenkins receives a request from the portal, it triggers the pipeline. The first stage to run is the VNF fetching stage. At this point, the pipeline, with the information received from the portal request, fetches the VNF package from the portal's VNF repository. Then, in the next stage, the package is

[3]https://www.docker.com/

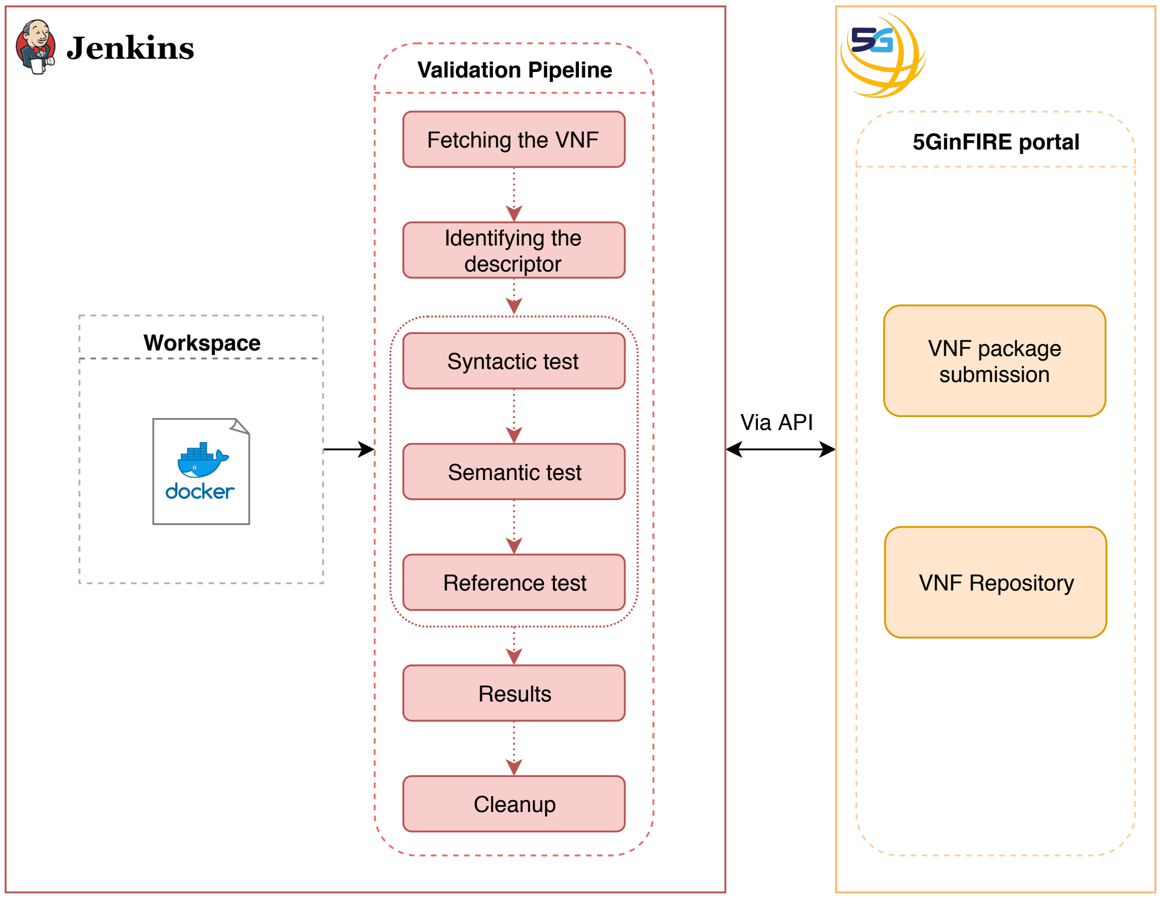


Figure 3.2: Deployment architecture of the automated solution.

extracted, the descriptor file identified, and the tests start. The tests run independently from the pipeline; therefore, they only occupy one stage. When the tests are completed, at the results stage, the results of the tests are sent to the portal. Lastly, Jenkins' workspace is cleaned, eliminating the resulting package and other files that may have been created during the execution.

3.4 Summary

In this chapter, three essential steps in the development of this Dissertation have been addressed: the identification of the problem, the study of the requirements and specifications for encountering solutions, and lastly, the found solution and its architecture.

In section 3.1, it was possible to understand that 5GinFIRE has trouble with its VNF submissions since they are manually validated, which is time consuming and neither scalable or practical. Therefore, the need for an automated validation tool arises. To do so, it is necessary to define which tests should be performed, which tool should be used for automation, and how to integrate everything with 5GinFIRE.

Section 3.2 starts by addressing which types of tests should be performed. Although it would be essential to perform functional and structural tests, the decision relies on just performing structural tests since the functional rise problems of lack of automation possibilities and limitations on the necessary resources to perform them. Therefore, it is decided to validate the VNFs via syntactic, semantic, and reference tests.

With the type of tests decided, it is necessary to choose between the available automation tools. Subsection 3.1 defines three requirements that the CI server needs to fulfill in order to fit the projects demands: high customization, existence of a REST API and allowing configurations outside SCM systems. After analyzing multiple automation tools, the chosen one was Jenkins since it was the CI server that met the most requirements without any cost associated. Afterward, on subsection 3.2.3, a description of Jenkins was provided, defining its main components and features, in particular, the type of pipelines and jobs. It was then decided to advance with a scripted pipeline job because it offered higher flexibility.

Having all the details sorted, section 3.3 provides the proposed solution, which is to have the Jenkins scripted pipeline integrated with the OSM VNFD and NSD validation scripts connected to the 5GinFIRE portal, so that whenever a VNF is submitted the tests are performed. The results are then sent back to the portal.

Lastly, the subsection 3.3.1 presents two architectures: one for the validation tool and the other with the full integration of the CI server, the scripts, and the 5GinFIRE portal.

CHAPTER 4

Implementation

With the architectures defined, the next step is to implement the necessary modules. Sections 4.1, 4.2, and 4.3 address the in-depth description of the strategies that were followed in order to develop the osm-descriptor-validator tool by describing the methods and structures created as well as the algorithms developed and their workflows. For each test developed, examples of errors and their outputs are also shown, so it is more understandable the type of output that the user will be dealing with.
Regarding the full architecture, section 4.4 provides the solutions for integrating the osm-descriptor-validator with Jenkins and the subsequent integration of Jenkins with the 5GinFIRE infrastructure. The steps performed on the validation pipeline are provided, and the techniques used are described.

4.1 OSM IM Wrapper

When developing the OSM IM wrapper, the main idea is to have well-defined structures that is able to provide the needed information in a structured way. Consequently, this module holds three classes: the wrapper class, the tag class, and the datatype class.

The wrapper class is composed of a set of endpoints that retrieve all the tags of each container from the YANG model. An example of such endpoint is provided on code block 3.

```
def get_vdu_tags(self):
    return get_tags(vnfd_model.yc_vdu_vnfd__vnfd_catalog_vnfd_vdu)
```

Code block 3: Endpoint to retrieve the VDU possible tags.

The *get_tags* is a helper function that produces the expected output, since the YAML returns the tags in their specific class.

The tag class is constituted by the crucial elements that define a tag: the name, path, datatype, child, possible values, and reference path. As not all the tags have child nodes, possible values, or reference path, these elements are characterized as optional for the definition of a tag.

The child, possible values, and reference path are all optional because they are not part of every tag denotation. The class definition is exposed on code block 4.

```python
class Tag:
    def __init__(self, name, path, datatype, child=None, possible_values=None,
    reference_path=None):
        self.name = name
        self.path = path
        self.datatype = datatype
        self.child = child
        self.possible_values = possible_values
        self.reference_path = reference_path
```

Code block 4: Tag class.

The other class defined in this module is the datatype class. The datatype class contains two methods, one for retrieving the datatype and another to retrieve the possible values for a certain datatype. These two functions get the information from a JavaScript Object Notation (JSON) file named *datatypes.json*, that was developed concerning the values provided by the IM. A snippet of the datatypes file is presented on the code block 5, and the datatype class is presented on the code block 6.

```json
{
"ALARM_VALUE": {
    "type": "decimal64",
    "options": [
        ""
        ]
    },
"PARAMETER_DATA_TYPE": {
    "type": "enum",
    "options": [
        "STRING",
        "INTEGER",
        "BOOLEAN"
        ]
    }
}
```

Code block 5: Snippet of the file datatypes.json.

The Wrapper is the base for all the validation process. Without it, there is no way to know the rules that the descriptors should follow. Therefore, as seen on the package architecture in figure 3.3.1, all the other modules are dependent on this file.

```
class Datatype:
    def get_datatype(self, value):
        return datatypes[value]['type']

    def get_datatype_options(self, value):
        return datatypes[value]['options']
```

Code block 6: Datatype class.

4.2 VNFD and NSD models class

In order to define all of the VNFD and NSD tags, each of the possible tags must be filled into the models by making use of the tag class defined on the OSM IM wrapper. A container is defined by the name, path, and the tags that it holds. A property, on another hand, is defined by the name, path, datatype, possible values, and reference path. As previously referenced, the last two are optional.

Although the tag class is generic for both containers and properties, the calls differ given the elements that define each one of them. Therefore, two functions were created, one to add containers and another to add tags to the class. They are displayed on code block 7.

```
def add_container(tags_list, name, path, child):
    tags_list.append(Tag(name, path, "container", child=child))

def add_property(tags_list, name, path, datatype, possible_values=None,
    reference_path=None):
    tags_list.append(Tag(name, path, datatype, possible_values=possible_values,
        reference_path=reference_path))
```

Code block 7: Functions to add containers and properties to the models.

The containers datatype is always the same; on another hand, a property does not have child properties; therefore, that parameter is always None. Consequently, both of these values are not defined on the functions in order to facilitate the calls and avoid repetition.

The VNFD and NSD models are both composed by the invocation of the described functions for every element of their data model. Code block 8 provides examples of the function calls.

```
utils.add_container(all_tags, "placement-groups", "vnfd/",
    child=wrapper.get_placement_groups_tags())

utils.add_property(all_tags, "vdu-id", "vnfd/mgmt-interface",
    datatype.get_datatype("STRING"), reference_path="vnfd/vdu/id")
```

Code block 8: Functions for filling the VNFD and NSD classes.

The function assigned to the element *child* on the *add_container* function named *get_placement_groups_tags()* returns all the tags of the container placement-groups, being similar to the example provided on code block 3.

4.3 Tests

The proposed tests have mostly the same structure, with the variations occurring just on the type of validation, which is being performed. The first common step is to verify if the file inputted for testing is valid. Firstly, it is verified if the file is a YAML or JSON since these are the two supported file formats by OSM. The file content is also loaded at this stage, which makes it available for the rest of the test execution. Then, it is identified whether a VNFD or a NSD was submitted for validation. Although the user is obligated to define which type of file is trying to validate, this ensures that there are no mistakes. The next step is to perform the tests; however, in order to validate all the tags, it is necessary to navigate through the descriptor structure. The approach is simple: a list of the container tags from the root path is retrieved. Then, the necessary tests are performed over that list of tags. When the tests are over, the list is analyzed, and it is verified if there is any container in that list. If the condition is false, the function ends. However, if it is true, the path is updated to the path of the new container that is going to be validated, and the function is called recursively using the new parameters (new path and new container). This is the basic workflow, although due to the requirements of each test, some need more steps to be executed. The navigation workflow is pictured in figure 4.1. The next subsections provide details about the performed tests and the workflow changes, accordingly.

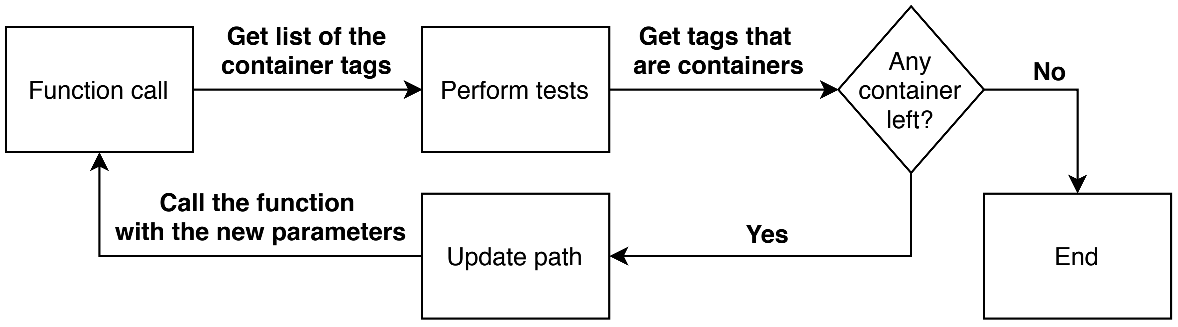

Function call
Get list of the container tags
Perform tests
Get tags that are containers
Any container left?
No
End
Yes
Update path
Call the function with the new parameters

Figure 4.1: Tests workflow.

4.3.1 Syntactic validation

The syntactical validation aims to ensure that all the tags described exist in the OSM IM as well as that they are correctly placed in the VNFD or NSD tree. The syntactical validation should be the first to be done since it prevents further errors, for example, in references.

After ensuring the file and descriptor type, the navigation function is invoked. The function performed is called *verify_bad_tags* and its workflow is shown in figure 4.2. The function's signature is presented on code block 9: The arguments are the container_tags,

```python
def verify_bad_tags(container_tags, im_all_tags, path, logger):
```

Code block 9: Function to verify invalid tags in a descriptor.

which are the tags of the container that is being validated, im_all_tags which is a list of all the tags of the VNFD or NSD model, the path which is the path in the model tree of the container and the logger which is an auxiliary function to provide better structured logs.

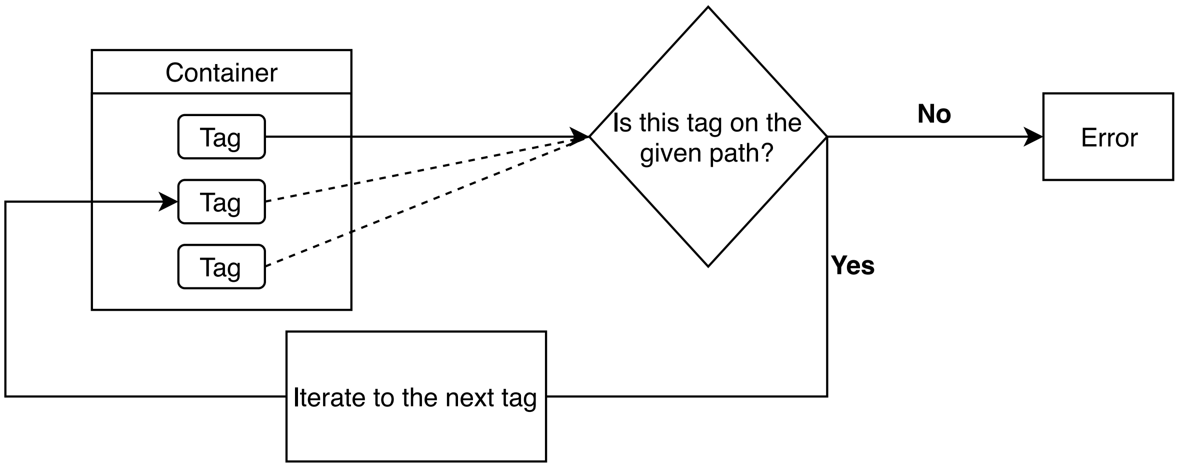


Figure 4.2: Syntactic test workflow.

The *get_path_tags* is an auxiliary function that returns all the tags from a given path according to the IM. Thereby, for each tag in the container provided, it is verified whether they are in the list of tags of their path (retrieved from the auxiliary function). This allows the verification of the correct tag path as well as its syntactical aspect.

This function can only provide errors and not warnings since every failure at this point is considered critical and has to be resolved. An example of the output is presented here:

```
ERROR  Invalid tag named storge-gb on path vnfd/vdu/vm-flavor/.
```

where the storage-gb tag was written as storge-gb.

4.3.2 Semantic validation

Semantic validation has the goal of verifying if a tag content is described with the correct datatype and, in case of a tag having a restricted set of possible values, assuring that its content is within the list.

This test does not follow completely the workflow described in figure 4.1, as it needs the list of containers inside a container for the test execution. Its workflow is described on figure 4.3. The first step is to verify the tag values; therefore, the *verify_tag_values* is invoked. The signature of the tag is presented on code block 10.

```
def verify_tag_values(container_data, containers_list, im_all_tags,
    path, logger):
```

Code block 10: Function to verify invalid tags in a descriptor.

The container_data holds the tags and values of an individual container, and the containers_list is the list of the tags inside the container_data that are defined as containers. The im_all_tags, path, and logger are the same as described before.

In this function, firstly, a loop is created, which iterates over the tags and values of the current container. Then, it is verified whether the tag is a container or not. If it is, it means that no datatype analysis is needed, and therefore, the next tag should be tested. On another hand, if the tag is not a container, its datatype and possible values should be retrieved. If the tag holds the "enum" datatype, it means that it is a choice parameter, and its possible values must be checked. If its value is not within the possible values, an error is thrown. However, if it is, then the *verify_datatype* function is invoked. Further details on this function will be provided later. Nevertheless, if the tag's datatype is not "enum", then the *verify_datatype* function should be called right away.

An error at this stage would have the following appearance:

```
ERROR  Invalid value "ESTERNAL" assigned to type in
    vnfd/vdu/interface/.The possible values for this parameter are:
    ['INTERNAL', 'EXTERNAL'].
```

In this case, the tag interface can only support INTERNAL or EXTERNAL as configuration values. The user, by mistake, typed "ESTERNAL" on the descriptor, triggering an error on this test.

The function *verify_datatype* is responsible for verifying the datatypes. At this stage, it is verified if the tag value is expressed in the correct datatype and if its within the datatype's possible range.

Although the OSM IM has well-defined datatypes for each tag, it accepts that the numeric tags are expressed in strings as well. Therefore, the output of the *verify_datatype* is an error when the value datatype does not correspond to a string or the real datatype, or a warning message whenever a value is expected to be numeric, but it is assigned as a string. However, the string value is still tested to ensure that, even though it is

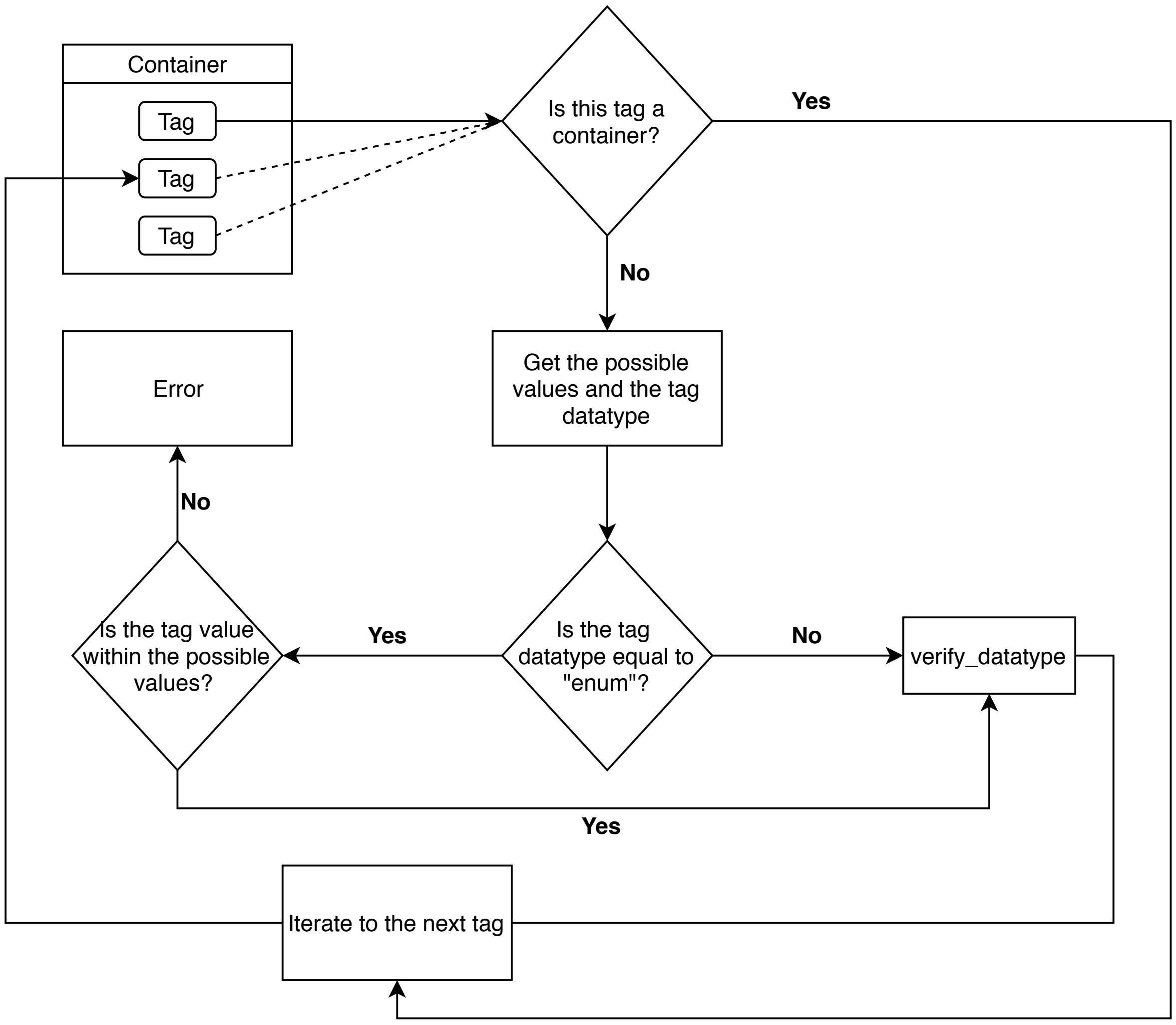

Container
Tag
Tag
Tag
Is this tag a container?
Yes
No
Get the possible values and the tag datatype
Error
No
Is the tag value within the possible values?
Yes
Is the tag datatype equal to "enum"?
No
verify_datatype
Yes
Iterate to the next tag

Figure 4.3: Semantic test workflow.

specified as a string, it is convertible to the expected datatype. If it is not, then an error is thrown. Consequently, an example output is

```
WARNING  Value "0" assigned to bandwidth in
    vnfd/vdu/interface/virtual-interface/ is specified as a string
    but should be an uint64.

ERROR  Invalid value "two" assigned to vcpu-count in
    vnfd/vdu/vm-flavor/. This value should be specified as an
    uint16.
```

Looking at the provided example, the warning is thrown exactly because the tag bandwidth expects a uint64, but a string was provided. However, the value is convertible for the specified datatype, so it is not necessary to throw an error. On another hand, on the second example, the error, albeit the value assigned to a vcpu-count tag, was a string; it has still thrown an error instead of a warning because the specified value is not convertible to the expected datatype, uint16.

4.3.3 Reference validation

The reference validation workflow is significantly more different when comparing to the other tests, and the one described in figure 4.1. Whereas on the other tests, the second step is to perform the test; at this test, the tests are only performed after the descriptor navigation is completed. This happens because the only way to know the value of all the references is to analyze all the descriptors first. The reference validation workflow is presented on figure 4.4.

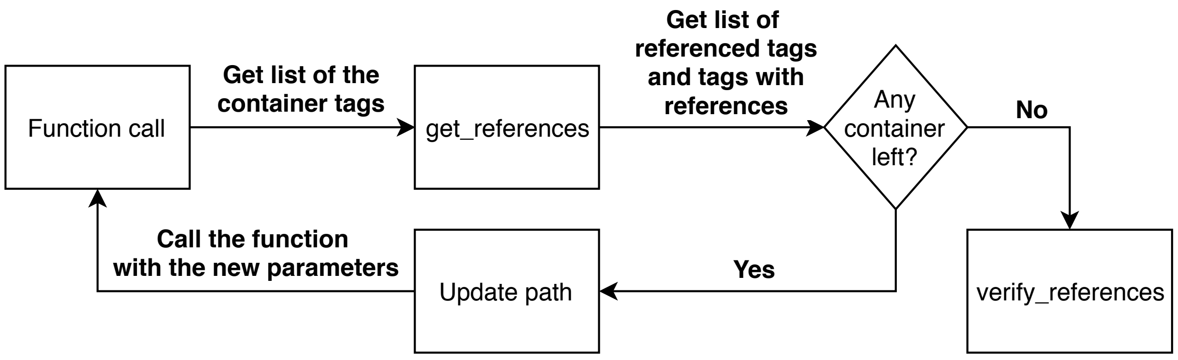

Function call
Get list of the container tags
get_references
Get list of referenced tags and tags with references
Any container left?
No
verify_references
Yes
Update path
Call the function with the new parameters

Figure 4.4: Reference validation workflow.

The *get_references* function is responsible for handling four lists: (i) the tags_referenced, which is the list that contains all the tags from the IM that may be referenced by other tags; (ii) the tags_with_reference, that contains all the tags from the IM that expect references for other tags as their values; (iii) the reference_values list that contains the values of all the existent tags_referenced in the descriptor and, lastly, (iv) the referenced_tag_values which contains the values that are expected to be references for other tags. The function as the signature presented on code block 11.

```
def get_references(container_content, im_all_tags, path):
```

Code block 11: Function to verify invalid tags in a descriptor.

The container_content and the path field are necessary for retrieving the values of the tags. On another hand, the im_all_tags are necessary to get the model tags and their information.

To gather all this information, three independent loops are declared. The first one goes through a list of all the tags with references and obtains the tags_referenced and the tags_with_reference directly. The former list tags are appended in a tuple format of (name, path). The latter are also appended in a tuple format but with (name, path, reference_path).

The second loop is made in order to get the values of the tags referenced. For this, it is verified whether, for each tuple defining a tag in the tags_referenced list, the tag appears in the containter_content. If this condition is true, then a tuple with the structure (name, path, value) is appended to the reference_values list.

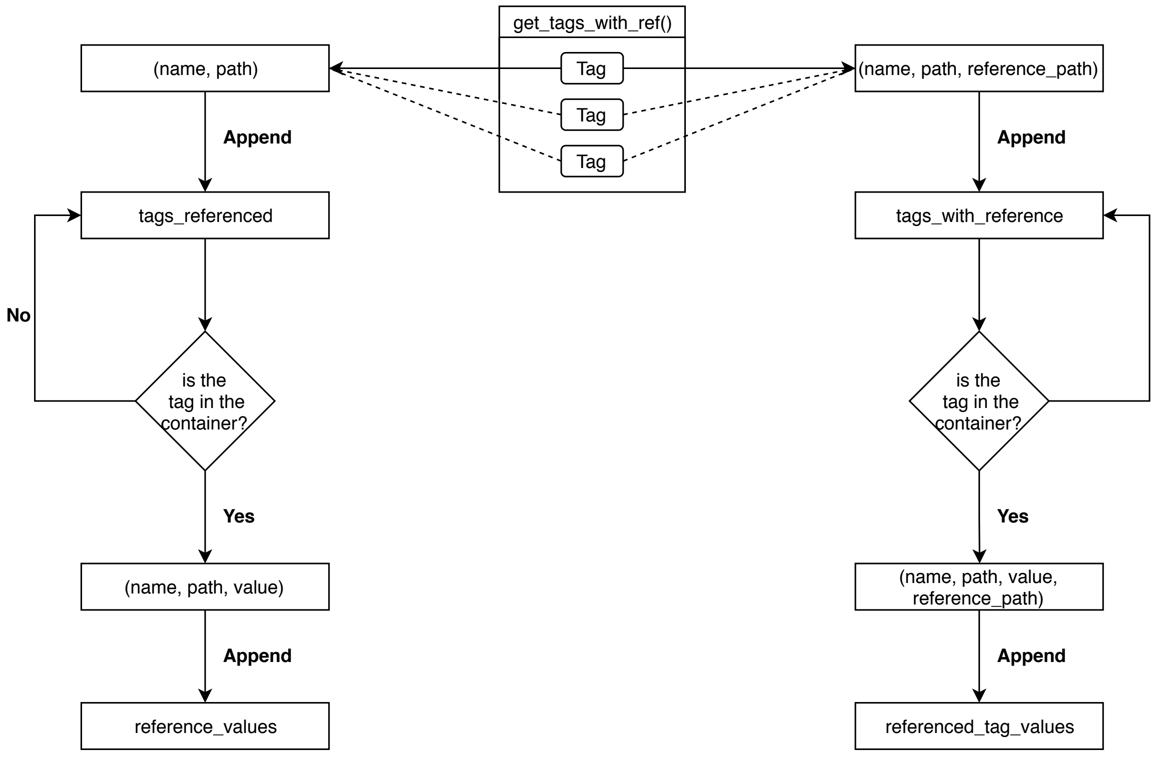


Figure 4.5: Second loop workflow.

The third and last loop does the same as the second, but it gathers the values of the tags in the tags_with_references list instead. The final tuple appended to the referenced_tags_values is also different because it contains one extra element, the reference_path. Therefore, the tuple structure is (name, path, value, reference_path). This procedure is pictured in figure 4.5.

According to figure 4.4, when there is no containers left to analyze, the next step is to call the *verify_references* function. This function is the one that performs the test itself, and its signature is pictured on code block 12.

```
def verify_references(referenced_tags_values, reference_values, logger):
```

Code block 12: Function to verify invalid tags in a descriptor.

The arguments of the verify_references function are both the lists filled on the previous function. The logger is used, as previously mentioned, to provide more explicit logging messages.

At the beginning of this function, another loop is created to go through the tuples on the referenced_tags_values list. For each element and using the reference_path value, a new tuple is created with the same structure of the ones in the reference_values; this is (name, path, value). Then, it is verified whether this tuple is or not on the reference_values list. If it is, the reference is correct. If it is not, an error is thrown. The manipulation of the tuples in order to proceed with the test is demonstrated in figure 4.6.

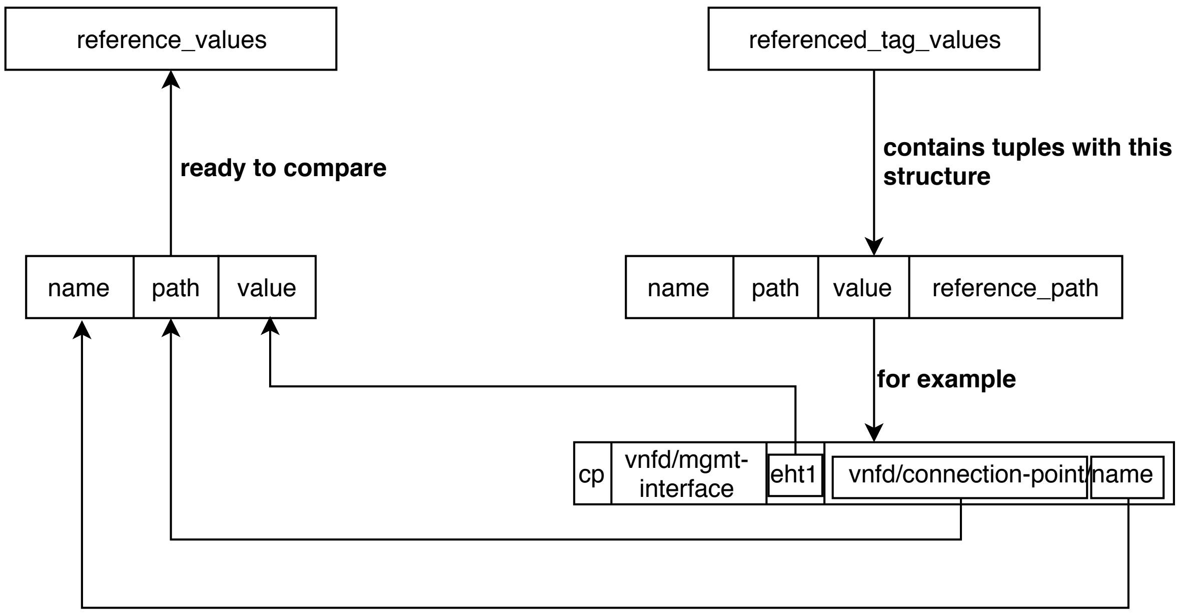


Figure 4.6: Manipulation of the tuples in *verify_references*.

This test just outputs errors, not warnings, and its appearence is the following:

```
ERROR    Invalid value "eth1" assigned to the tag cp in
     vnfd/mgmt-interface/.
     The value should be a reference to the tag name in the path
     vnfd/connection-point/. Possible values are: ['eth0'].
```

In this case, the error occurred because of the tag with the path vnfd/mgmt-interface/cp references values from the tag vnfd/connection-point/name. The latter only has the value "eth0", and the user is trying to reference "eth1", leading to an error.

4.3.4 NS over VNF validation

In the package architecture described on figure 3.1, the only test that shares a connection is the NS over VNF validation with the reference test. This is because this test is an extension of the reference test since it does the same thing but between two different files: the VNFD and the NSD.

Given that there are already developed functions on the reference test that can fetch the referenced values on each file, this test makes direct use of the functions specified on subsection 4.3.3, which leads to the following exactly the workflow described on 4.4.

Knowing that it is the NSD that references elements from the VNFD, a list containing the reference_values should be retrieved from the VNFD and the list with the referenced_values should be, therefore, fetched from the NSD. The first step is to get both of those lists. To do that, the navigation function from the reference test is invoked.

The navigation function on the reference validation script has the signature provided in code block 13.

```
def descriptor_navigation(descriptor_content, im_all_tags,
    im_all_tags_names, dtype, path):
```

Code block 13: Function to verify invalid tags in a descriptor.

The arguments are the descriptor_content, a list with all the tags from the IM, as well as a list of their names, the descriptor type, and the path to start the analysis.

With this in mind, the invocation proceeds as described in code block 14.

```
vnfd_reference_values = reference_validation.descriptor_navigation(
    vnf_descriptor_data, vnfd_im_all_tags, vnfd_im_all_tags_names,
    "vnfd", "vnfd/")

nsd_referenced_values = reference_validation.descriptor_navigation(
    ns_descriptor_data, nsd_im_all_tags, nsd_im_all_tags_names,
    "nsd", "nsd/")
```

Code block 14: Invocation of the navigation functions from the reference validation.

Having both of the lists, the next step is, just as in the reference test, to call the *verify_references* function with the lists that were just obtained. The output structure is the same as in the reference test.

4.3.5 osm-descriptor-validator

This module is the connection point between the user and the tests. It provides a set of options for the user to execute them.

The set of commands that may be executed are presented on code block 15

By executing any of the operations shown, the output will be displayed depending on the type of errors that may exist. The messages are just as described in previous sections. If there are no errors or warnings to report, the program will not output anything.

```
$ osm-descriptor-validator -h

usage: osm-descriptor-validator [-h]
    (--vnfd vnfd.yaml | --nsd nsd.yaml | -a vnfd.yaml nsd.yaml)

Validate a OSM VNF or NS descriptor.

optional arguments:
-h, --help              Shows this help message and exit
--vnfd vnfd.yaml        Performs syntactic, semantic and referential tests to
                        a VNF descriptor. Takes as argument the path to the
                        VNF descriptor file to be tested.
--nsd nsd.yaml          Performs syntactic, semantic and referential tests to
                        a NS descriptor.Takes as argument the path to the NS
                        descriptor file to be tested.
-a vnfd.yaml nsd.yaml, --all vnfd.yaml nsd.yaml
                        Performs syntactic, semantic and referential tests to
                        a VNF and NS descriptor. Also validates the NSD
                        towards the VNF.Takes as argument the path to the VNF
                        and NS descriptor files.
```

Code block 15: osm-descriptor-validator help menu.

4.4 Jenkins, osm-descriptor-validator and 5GinFIRE integration

4.4.1 5GinFIRE integration

In order to integrate the Jenkins deployment with 5GinFIRE, it is necessary to authenticate Jenkins with the portal and vice versa as well as establishing the connection points and messages to be exchanged.

The authentication is done via the exchange of API keys. Those API keys are provided directly on the request headers.

According to the diagram presented in figure 4.7, the portal, and the pipeline have three interactions: the pipeline trigger, the package retrieval, and the results delivery.

To trigger the pipeline, the 5GinFIRE portal just needs to send a request to the pipeline endpoint via its REST API with the VNF ID as a parameter whenever a new package is submitted. The ID is very important since it is through it that the pipeline is capable of retrieving the package from the portal. To do this, the link to the repository is hardcoded on the pipeline configuration, with the ID being the only nonstatic element. Consequently, when the ID is retrieved from the trigger request, the package download starts.

Once the tests are performed, its results are sent back to the portal via a JSON message, which is pictured on code block 16.

This message is then processed back on the 5GinFIRE portal. If the validation result is positive, then the VNF gets verified, and the experimentation process continues. If the result is negative, then a new issue regarding the VNF is posted to the project's

```
{
    "vnfd_id": <integer>,
    "build_id": <integer>,
    "validation_status": <boolean>,
    "jenkins_output_log": {
        "Errors": [<list of errors>],
        "Warnings": [<list of warnings>]
    }
}
```

Code block 16: Results JSON response.

Bugzilla[1] containing the validation logs sent in the response message so that the VNF developers have access to the results of the tests.

4.4.2 Jenkins configuration

Jenkins is available as an external service for the 5GinFIRE project. The service is deployed on a VM, configured, and it is running online[2]. For security purposes, apart from the 5GinFIRE administrator and the pipeline maintainers, no one else has credentials to access the platform. The CI server contains the validation pipeline, which is responsible for running the osm-descriptor-validator over the VNFs.

Validation pipeline

In figure 4.7, one of the interactions is the Docker build request. In order to be able to perform many tasks, the pipeline environment has to be prepared with the necessary tools. The pipeline workspace contains a Dockerfile, as shown in figure 3.2 that runs the first time the pipeline is triggered and installs every requirement specified. The environment set up happens the first time that the pipeline runs or whenever a Dockerfile modification occurs.

The Dockerfile is configured to install tools related to the utilization of JSON and YAML files, the extraction of compressed packages, and the usage of network tools that need to be installed.

The osm-descriptor-validator is available for execution via the pipeline because its files are provided via the validation pipeline workspace.

The pipeline currently has five stages. The first stage being the VNF fetching, sends a request to the portal to download the package by using the static link with the ID received with the trigger request. The workflow is provided in code block 17.

First, the package is retrieved from the 5GinFIRE repository, and the information is saved on a temporary file. The *curl* call is between *set +x* and *set -x* so it is not visible on the Jenkins logs.

[1]https://www.bugzilla.org/
[2]http://ci.5ginfire.eu

```
sh '''
    set +x
        curl -H "X-APIKEY:123456-1233"
        https://portal.5ginfire.eu/5ginfireportal/services/api/repo/admin/vxfs/
        $VNF_ID > tmp.txt
    set -x
'''
(...)
sh "wget -O package $package_url"
```

Code block 17: Snippet for retrieving the VNF package.

Then, the JSON is read from the saved file, and the link for the VNF is retrieved from the *packageLocation* element of the JSON. The link comes without the communication protocol assigned; therefore, it is necessary to append it to it. Lastly, the package is downloaded and saved on the validation pipeline workspace.

The second stage starts and pursuits the identification of the descriptor. All the VNF packages are compressed as tar.gz. Therefore, it is necessary to decompress the package and retrieve the descriptor path. This stage is described on code block 18.

```
sh 'tar -xvzf package'
descriptor = sh(
    script: "find . -maxdepth 1 -name \"*.yaml*\" -print | tail -c +3",
    returnStdout: true
    ).trim()
```

Code block 18: Snippet for identifying the VNFD.

The first command intends to decompress the VNF package. Secondly, a bash script is utilized in order to find the descriptor file. The script defines that a search should be conducted on the current directory but with a max depth of one, which means that should also search on the first level inside the existing folders for files that contain ".yaml" on their name. The max depth is vital in order to not retrieve files that are not descriptors. In the second part of the command, *tail -c +3* is utilized to eliminate unnecessary characters from the name.

Once the path is found, the pipeline moves to the next stage, which is the tests. The tests are performed by following the package specifications described on code block 15. For each test, the output is retrieved and saved for the response. The osm-descriptor-validator execution is very straightforward and is presented on code block 19.

With this approach, all the logs are saved in a JSON format, which is easily accessible on the rest of the pipeline execution.

After all, tests are completed, the fields of the code block 16 are filled. The vnfd_id is the ID of the tested VNFD; the build_id is the number of the Jenkins job, the

```
log = sh(
    script: "python3.6 osm-descriptor-validator.py
        --vnfd ../" + descriptor + "",
    returnStdout: true
    ).trim()
```

Code block 19: Tests execution on the pipeline.

validation_status is a variable that is true if all the tests are run without any errors and the jenkins_output_log is an aggregation of all the errors and warnings of each test. At the results stage, this message is sent to the portal.

The two types of messages that can be sent are provided in code block 20 and 21. The former sends a notification saying that there is nothing to report regarding the submitted VNF. On another hand, the latter sends the osm-descriptor-validator log attached.

```
response = "{\"vnfd_id\": ${VNF_ID},
            \"build_id\": ${BUILD_NUMBER},
            \"validation_status\": ${VALIDATION_STATUS},
            \"jenkins_output_log\": \"Nothing to report.\"}"


sh "curl -v -H \"Content-Type: application/json\"  -H \"X-APIKEY:123456-1233\"
    -X PUT -d '${response}'
    https://portal.5ginfire.eu/5ginfireportal/services/api/repo/admin/
    validationjobs/$VNF_ID"
```

Code block 20: VNF valid response.

```
response = "{\"vnfd_id\": ${VNF_ID},
            \"build_id\": ${BUILD_NUMBER},
            \"validation_status\": ${VALIDATION_STATUS},
            \"jenkins_output_log\":
            \"{ ERRORS: " + log + "\"}}"


sh "curl -v -H \"Content-Type: application/json\" -H \"X-APIKEY:123456-1233\"
    -X PUT -d '${response}'
    https://portal.5ginfire.eu/5ginfireportal/services/api/repo/admin/
    validationjobs/$VNF_ID"
```

Code block 21: VNF invalid response.

The last stage is a clean up so that the package is removed from the workspace, and it is performed by the command provided on code block 22, which deletes all the files with the exception of the Dockerfile and the osm-descriptor-validator.

With all the integration performed, the messages workflow has to be as pictured in figure 4.7.

```
sh "rm -rf !("osm-descriptor-validator"|"Dockerfile")"
```

Code block 22: Clean up of the validation pipeline workspace.

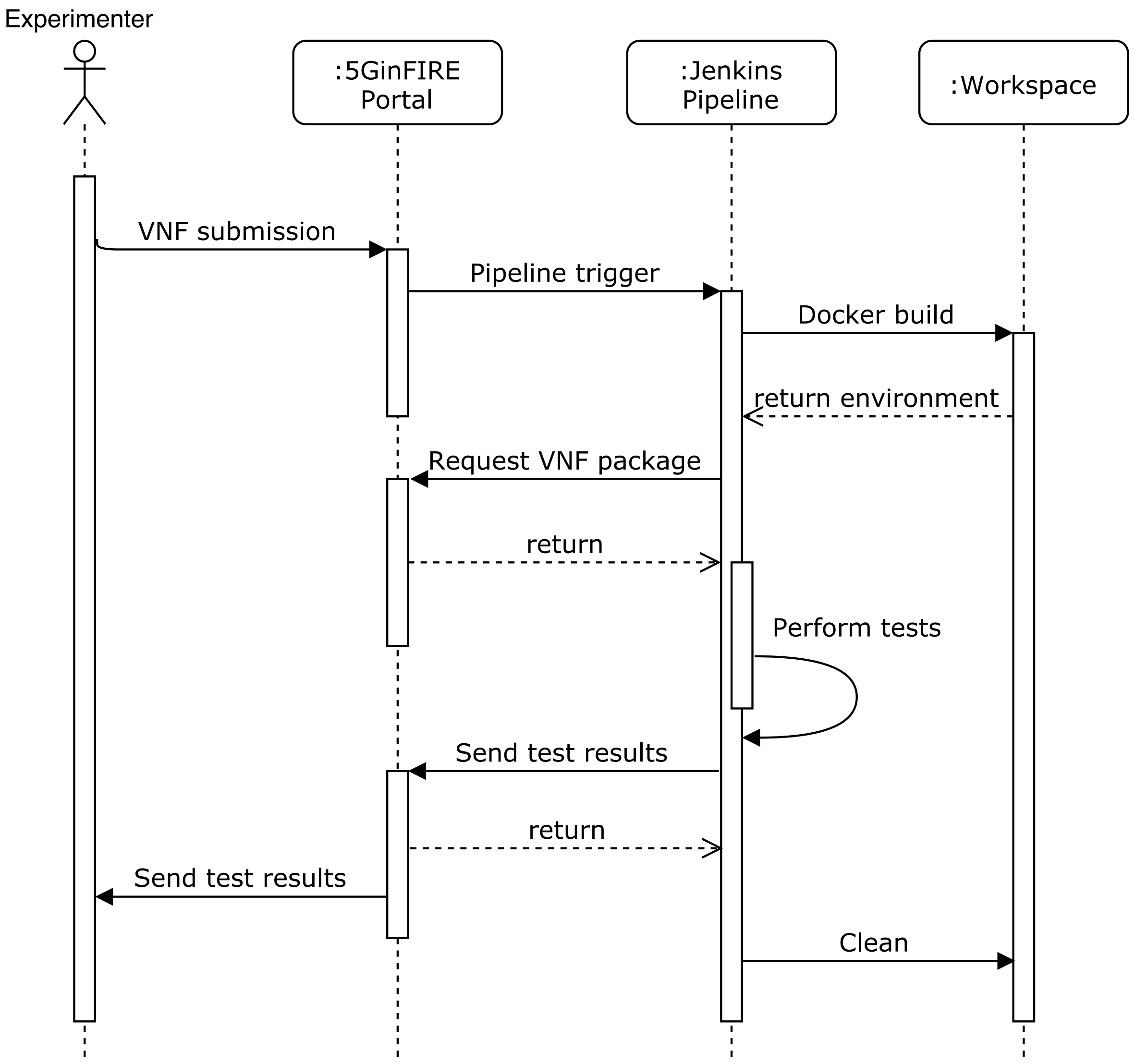


Figure 4.7: Sequence of events between 5GinFIRE and the Jenkins pipeline.

When an experimenter submits a new VNF, the Jenkins pipeline is triggered. Then, requests the package from the portal, extract the VNFD file, and perform the tests on it. Once completed, the results are sent back to the portal, and the environment is cleaned.

The pipeline is configured to support concurrent runs, so another user may trigger the pipeline while it is still running without making it fail.

4.5 Summary

The proposed architecture and modules were developed with success. Starting by the OSM IM Wrapper, which is the base class of the scripts, it was developed using the OSM YANG models so that there are no failures on retrieving the list of tags. Then, in section 4.2, it was possible to understand how the tags are available with the necessary information for the osm-descriptor-validator execution. The process followed is good because if it becomes necessary to add any new tag, it is just necessary to modify one of these classes, and the element will be immediately available for usage in the tests. Section 4.3 provided and in-depth analysis of how the tests are built.

It was possible to verify that the execution workflow is very similar except for the test that is being carried out and, in the case of the reference test, that differs mostly because it needs to run the descriptor twice in order to get the values that are referenced and the references.

Lastly, section 4.4 provided the description of the necessary configurations from the three components: 5GinFIRE, osm-descriptor-validator, and Jenkins in order to perform the desired workflow. It was possible to understand that the configurations either from the 5GinFIRE portal or from Jenkins were minimal and straightforward since for triggering the pipeline and to send the pipeline results, it was just necessary the exchange of API keys and calling an endpoint API. In regards to the integration of the osm-descriptor-validator and Jenkins, it was also straightforward with just the necessity of cloning the validator from its SCM repository to the pipeline workspace, install the minimal list of requirements and executing using Python3.6 directly on the pipeline configuration.

CHAPTER 5

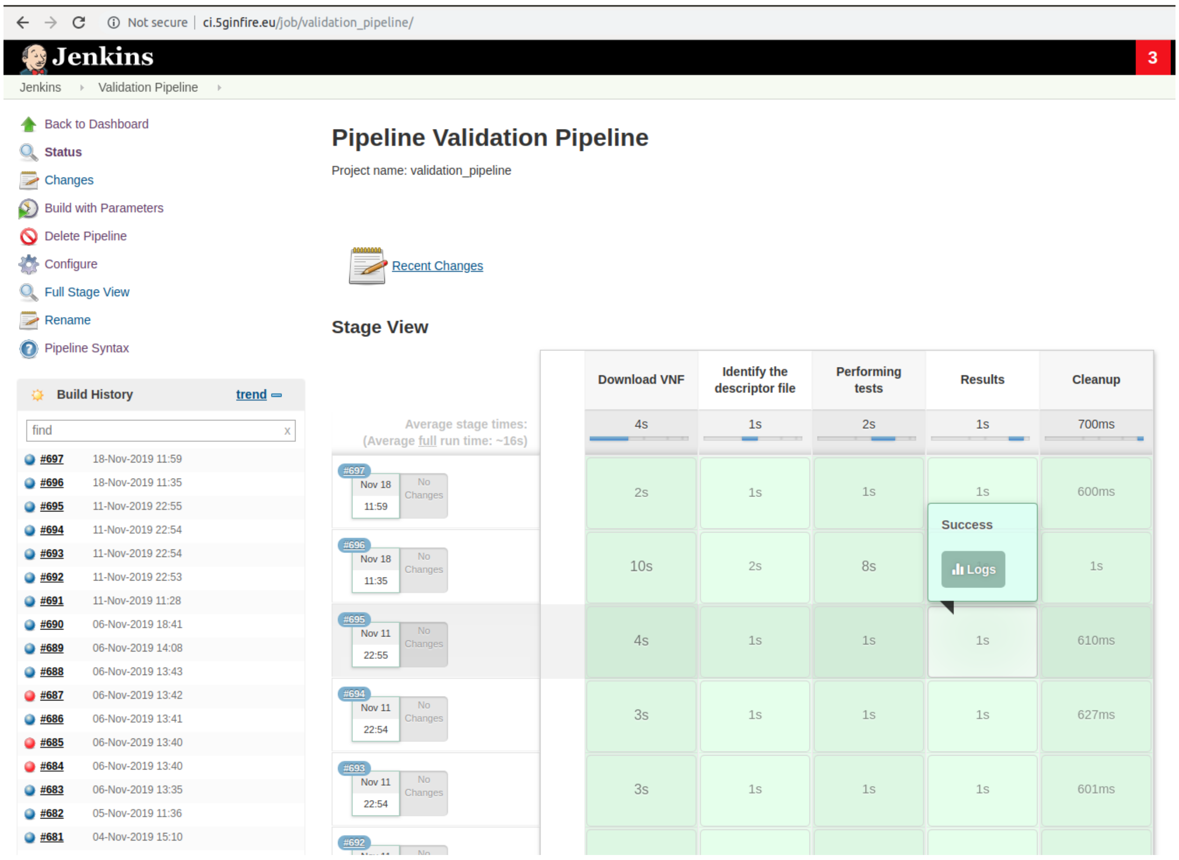
Results

After developing the proposed solution, there are two scenarios where the architecture can be tested, which are presented in section 5.1.
As OSM is a recent technology, there are not many public VNFD implementing its IM for making tests. Therefore, section 5.2 presents the strategy followed in order to collect descriptors to test.
Finally, section 5.3 provides the tests for both scenarios. Then, it is also presented a more in-depth review of all the logs collected the usual errors and warnings gathered from all the descriptors in order to understand which are the tags with the most failures associated.

5.1 Scenarios

The solution implemented for this Dissertation was, as previously stated, developed as part of the 5GinFIRE project. In the end, the CI server was deployed and integrated with the project's portal. The Jenkins dashboard deployed for 5GinFIRE is portrayed in figure 5.1.

Although this was the approach followed, the package developed is not dependent on the CI server. Therefore, it can be used outside the project scope, leading to two possible usage scenarios. The first scenario is within the 5GinFIRE project. A VNF developer starts by producing a VNF. Then, it submits the package in the 5GinFIRE portal from which the CI job is triggered automatically. The osm-descriptor-validator runs over the descriptor and sends the results back to the portal. The VNF developer has then access to the logs generated. Since the deployment to production of the validation pipeline described in section 4.4.2, this scenario was tested in the real 5GinFIRE environment. Figure 5.2 presents the scenario's workflow.


Not secure | ci.5ginfire.eu/job/validation_pipeline/

Jenkins

Jenkins › Validation Pipeline ›

- Back to Dashboard
- Status
- Changes
- Build with Parameters
- Delete Pipeline
- Configure
- Full Stage View
- Rename
- Pipeline Syntax

Build History trend

find

#697	18-Nov-2019 11:59
#696	18-Nov-2019 11:35
#695	11-Nov-2019 22:55
#694	11-Nov-2019 22:54
#693	11-Nov-2019 22:54
#692	11-Nov-2019 22:53
#691	11-Nov-2019 11:28
#690	06-Nov-2019 18:41
#689	06-Nov-2019 14:08
#688	06-Nov-2019 13:43
#687	06-Nov-2019 13:42
#686	06-Nov-2019 13:41
#685	06-Nov-2019 13:40
#684	06-Nov-2019 13:40
#683	06-Nov-2019 13:35
#682	05-Nov-2019 11:36
#681	04-Nov-2019 15:10

Pipeline Validation Pipeline

Project name: validation_pipeline

Recent Changes

Stage View

	Download VNF	Identify the descriptor file	Performing tests	Results	Cleanup
Average stage times: (Average full run time: ~16s)	4s	1s	2s	1s	700ms
#697 Nov 18 11:59 No Changes	2s	1s	1s	1s	600ms
#696 Nov 18 11:35 No Changes	10s	2s	8s	Success Logs	1s
#695 Nov 11 22:55 No Changes	4s	1s	1s	1s	610ms
#694 Nov 11 22:54 No Changes	3s	1s	1s	1s	627ms
#693 Nov 11 22:54 No Changes	3s	1s	1s	1s	601ms
#692					

Figure 5.1: 5GinFIRE's Jenkins dashboard.

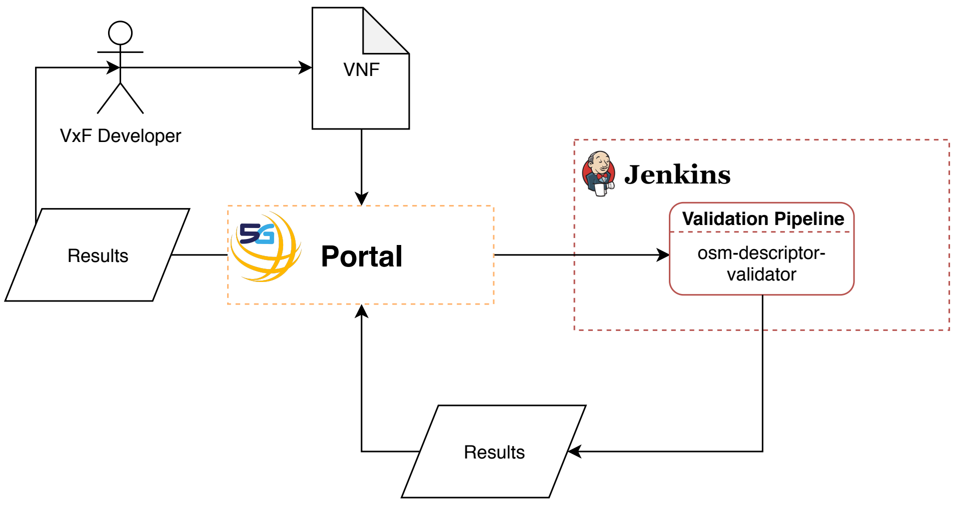


Figure 5.2: 5GinFIRE's VNF developer scenario.

The second scenario is associated with an independent developer who builds a VNF and needs to test the package before deploying it. In this case, the user has the osm-descriptor-validator on its machine and runs the tests directly according to its needs. The logs are immediately displayed. Figure 5.3 presents this scenario workflow. Differently from the previous scenario, it was not possible to collect results from a real approach to this case. In order to simulate this scenario, it was necessary to have VNFs to test.

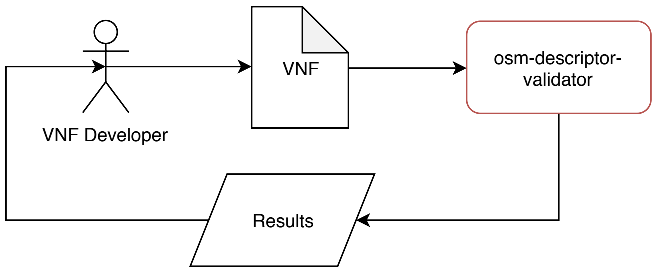


Figure 5.3: Independent VNF developer scenario.

Although to validate this scenario, only one VNF had to be tested, many VNF were gathered during the development of this Dissertation. Therefore, all of the VNFDs were tested. With this, not only the scenario was tested, but also the number of results collected was much more prominent. The process of collecting VNF packages to test is described in the next section.

5.2 Data Gathering

OSM is a very recent technology; therefore, there are not many descriptors available that make use of its IM to perform tests. 5GinFIRE currently holds a great repository of VNFs. Nonetheless, just testing the descriptors on that repository would lead to nowhere because they are only stored when they pass the tests and are instantiated.

Instead of merely deploying the Jenkins platform once the validator was completed, the pipeline was deployed in August 2018. The validation performed was done via the OSM simple validation scripts. However, it allowed storing all the descriptors that were being tested, giving a vast dataset of good and bad descriptors.

With this approach, 460 VNF descriptors were collected. However, for the project scope, it was only possible to apply the pipeline to VNFs. For this reason, there is no set of NSDs.

5.3 Results

Until the end of September 2019, the pipeline had been running the OSM validation tests, which provided an initial syntactical validation of the descriptor. The package developed was, since then, integrated with Jenkins. The results of the pipeline since the new tests are described in table 5.1. They are the results of the first scenario.

The debugging fields on the successful column refer to test builds in order to solve problems with the pipeline configuration. The same can be said about the debugging

Number of successful runs		Number of failures		
Descriptors	Debugging	Descriptor errors	Debugging/ Connection problems	Total number of builds
108	9	1	32	150

Table 5.1: Pipeline results from September to October 2019.

field on the failure columns. However, in this case, it also addresses the failures related to connection problems.

When looking at the table, it is possible to state that the number of failures when comparing with the successful builds of the pipeline is meager, since there is only one failure related to the descriptors. That failure did not happen directly because of the descriptor itself but actually because the VNF package was not well structured.

Looking at those numbers and subtracting the number of debugging runs, there are 109 builds that were dedicated to VNFDs validation. Given that 108 were successful, that leads to 99% of the success rate. One thing to state is that 108 runs do not mean 108 different descriptors. The descriptors may go through the validation process multiple times. Nevertheless, the results are still impressively high.

Although it was possible to gather 460 VNFDs, a lot of the VNFs were resubmissions. Therefore, in order to make the results more reliable, all the descriptors that had no differences between each other were removed, gathering 166 unique descriptors. In order to test the second scenario, the osm-descriptor-validator tested all the 166 descriptors filtered. The results are on table 5.2.

	Syntactic test	Semantic test		Reference test			
Descriptors	Errors	Errors	Warnings	Errors	No errors/ warnings	Pass	Fail
166	26	3	1129	8	80	147	19

Table 5.2: Validation results from osm-descriptor-validator.

As previously stated, when using the osm-descriptor-validator, the warnings are just used to advise the developed in using the correct datatypes. Therefore, having warnings does not mean that the package is not valid. With this in mind, and looking at table 5.2, it is possible to verify that in 166 descriptors, only 19 failed the tests with 26 errors related to syntactical issues, three errors because of lousy semantics and eight errors due to incorrect references. Such results lead to approximately 88.5% of success rate. That is still a high number given the sample size.

In order to have a better understanding of what is failing in the descriptors, it is important to verify the range of errors provided by the osm-descriptor-validator. Figure 5.4 provides an overview of all the different errors detected by the tests. On another hand, table 5.3 presents statistics about the number of errors from each descriptor regarding each test. Lastly, figure 5.5 portraits the distribution of tags with errors.

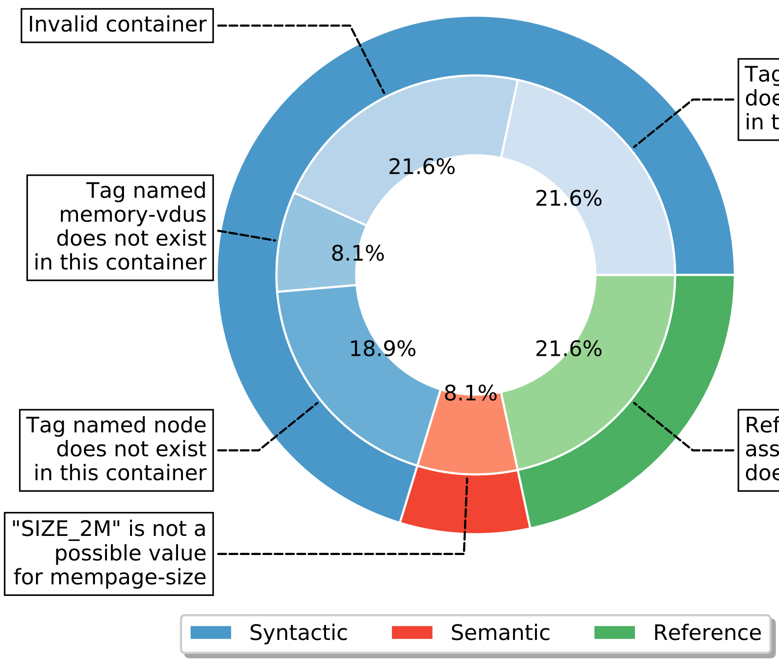


Figure 5.4: Distribution of errors and warnings according the reasons.

		Statistics			Errors distribution				
Test type	Files with errors	Min	Max	Mean (Except 0s)	0	1	2	3	4
Syntactic	8	0	4	3.25	158	0	0	6	2
Semantic	3	0	1	1	163	3	0	0	0
Reference	8	0	1	1	158	8	0	0	0

Table 5.3: Errors per descriptor.

The range of errors collected is, as expected, very low. The syntactic test holds most of the errors. This result was predicted because the syntactic validation refers to the structure of the VNFD, which means that if a tag is poorly defined, it may compromise the other tags. In fact, in figure 5.4, it is possible to verify that the error "Invalid container" has a significant percentage of syntactic errors. The other two errors with a higher percentage are both related to the node and node-cnt tags, which on the OSM IM are defined under the path vnfd/vdu/guest-epa/numa-node-policy.

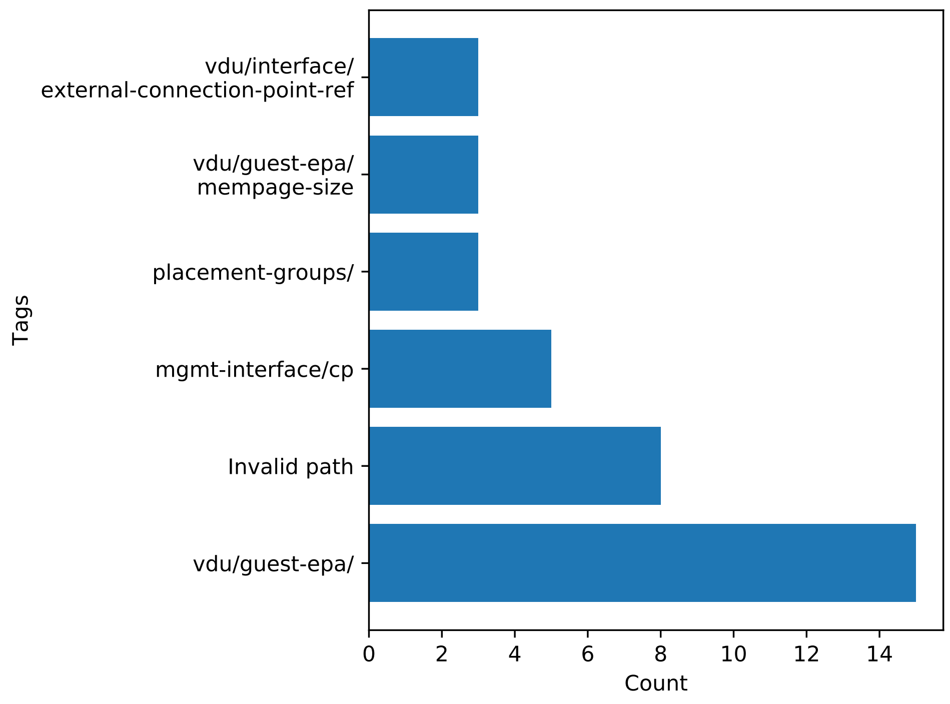


Figure 5.5: Distribution of tags with errors.

Given that in figure 5.5 the path vdu/guest-epa is the most common with errors, it is possible to infer that the numa-node-policy was being poorly defined on many descriptors, contributing for the increasing of the error "Invalid container" and the further failure of the node and node-cnt tags. The error with fewer occurrences was triggered because the tag "memory-vdus" does not exist whatsoever on OSM IM.

The syntactic errors are, according to table 5.3, distributed over eight different descriptors. The number of errors ranges between zero and four, with the preeminent value being zero. Given the low number of errors, the mean values were calculated without taking them into account; otherwise, the mean would always be zero. Therefore, when only considering descriptors with errors, there are approximately three errors per file. This value converges into the analysis presented since it represents the errors concerning badly structured descriptors.

The reference test also does not have a considerable variation. The values are comprised between zero and one with zero being very dominant. This superiority may be higher than the reality because this test is the last to be performed. The osm-descriptor-validator has specified that the developer must first solve the existing problems to progress to the next test whenever the validation fails. This approach was followed in order to make the logs cleaner since, for example, if a tag were poorly defined, it would lead to failure of the semantic test because that tag would not exist and would, therefore, not have an associated datatype. The downside of this approach is that, in this case, the results of the eight descriptors that failed the syntactic test plus the three that failed the semantic test have not had their references tested, contributing to the zero counts.

The reference errors detected were related to the vnfd/vdu/interface/external-connection-point-ref and vnfd/mgmt-interface/cp tags with the latter being more common. Such a statement implies that the references assigned to each tag did not exist in the descriptor. According to table 5.3, the eight descriptors that failed this test had one semantic error each, contributing to a mean of errors per file of one.

The semantic test has the same problem described for the reference test; this is, the eight descriptors that failed the syntactic test did not have their semantics validated. However, the pictured results defined in figure 5.4 reveal only one type of error for the semantic test. The error occurred because the value "SIZE_2M" is not a possible value for the tag "mempage-size". The correct value that could be assigned is "SIZE_2MB". The three semantic errors detected are all the same, which means that the descriptors causing such error were probably resubmitted into the validation process without having the problem solved. The three errors were detected on three different files leading to a mean of errors per file of one. Differently from the other tests, the semantic validation also triggers warnings. Figure 5.6 presents the reasons for the triggered warnings and figure 5.7 provides the distribution of number of warnings per descriptor.

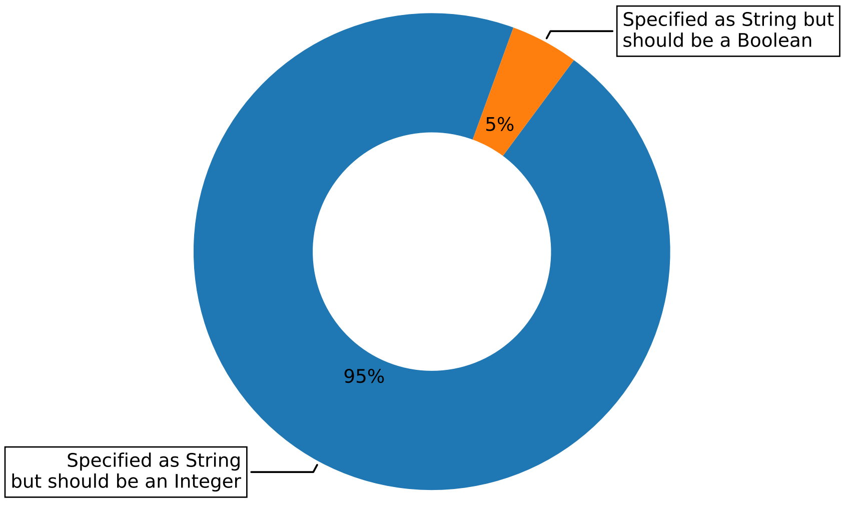


Figure 5.6: Distribution of warnings according the reasons.

The number of warnings is very high when compared with the number of errors. Table 5.2 displays 1129 warnings over 166 descriptors. In figure 5.6, the warning reasons are distributed in two: assignment of values that should be integers or booleans as strings. There is a size and signal separation of Integers within OSM IM; nevertheless, they are all grouped as Integers in order to provide better plot visualization.

The 166 files provided 6073 tags assignments. Given the 1129 warnings in total, there are 18.6% of tags with warnings. Given that the majority of the tags are either strings and some others can only be assigned a strict set of possible values, the result is very high. In fact, in figure 5.7, it is possible to verify that there are two warnings per file on average, with the count ranging between zero and eleven.

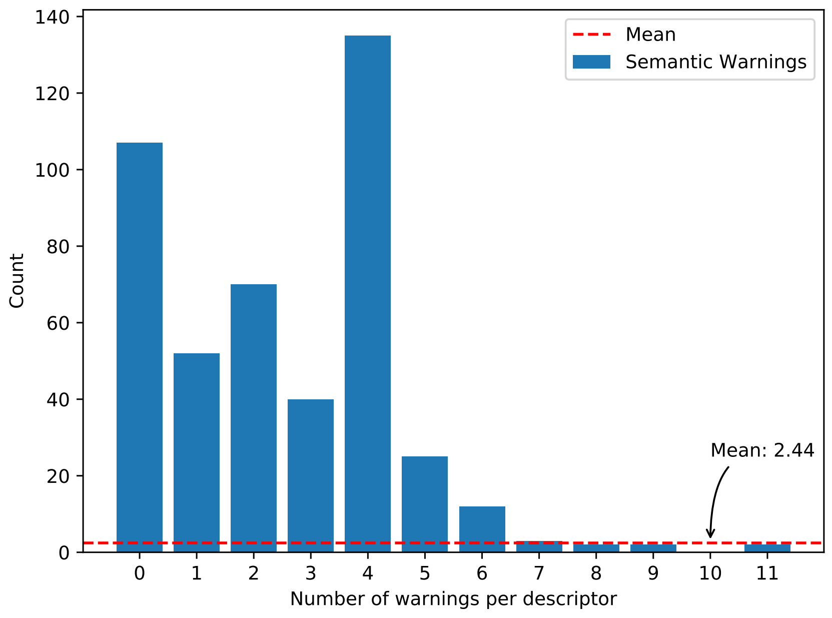


Figure 5.7: Distribution of the number of warnings per file.

When looking at figure 5.8, it is possible to verify that tags on the path, "vnfd/vdu" have the highest number of warnings associated. In fact, three tags - storage-gb, memory-mb, and vpcu-count - trigger warnings with the same frequency. With the representation presented in figure 5.7, it is possible to see that the developers do not pay attention to the IM specifications in terms of the datatypes. The problem is that OSM parses these values, making the IM datatype specification useless. Another interesting point is that, just like the errors, there is not much variation on the tags that trigger warnings.

In fact, OSM provides 313 distinct tags. With a 166 descriptors dataset, only 84 were detected. From those tags, still 82% did not trigger any error or warning as pictured on figure 5.9.

Since there are not many errors and the warnings seem to always occur on a stringent group of tags, it is necessary to understand if there is a problem from the developers in understanding the IM for those tags or if those are the tags that trigger errors and warnings the most because the others are not used.

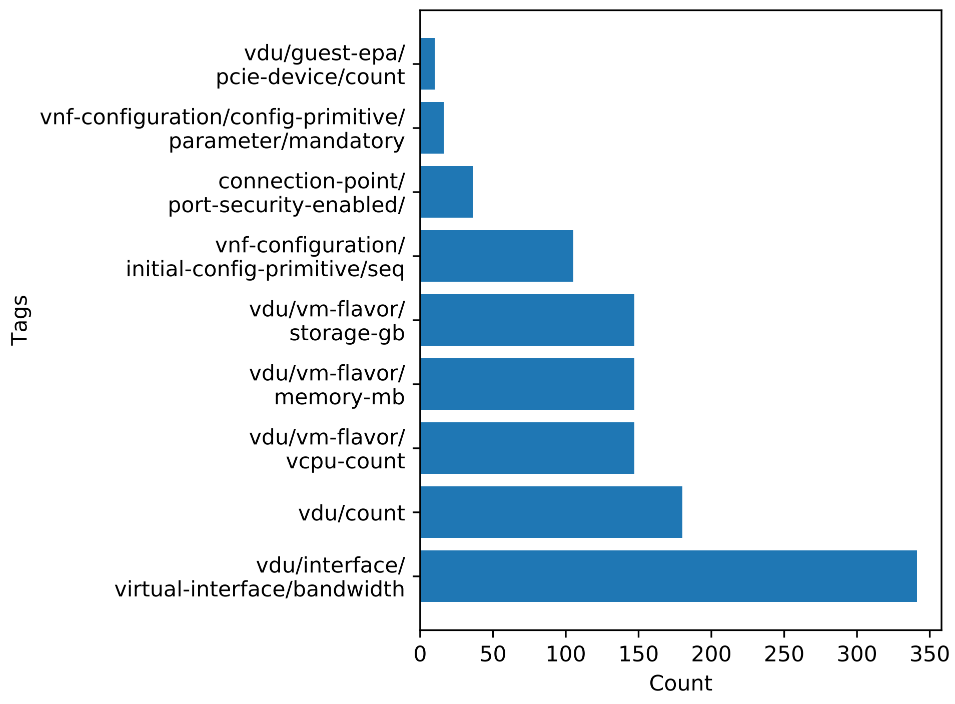


Figure 5.8: Number of errors and warnings from each of the used tags.

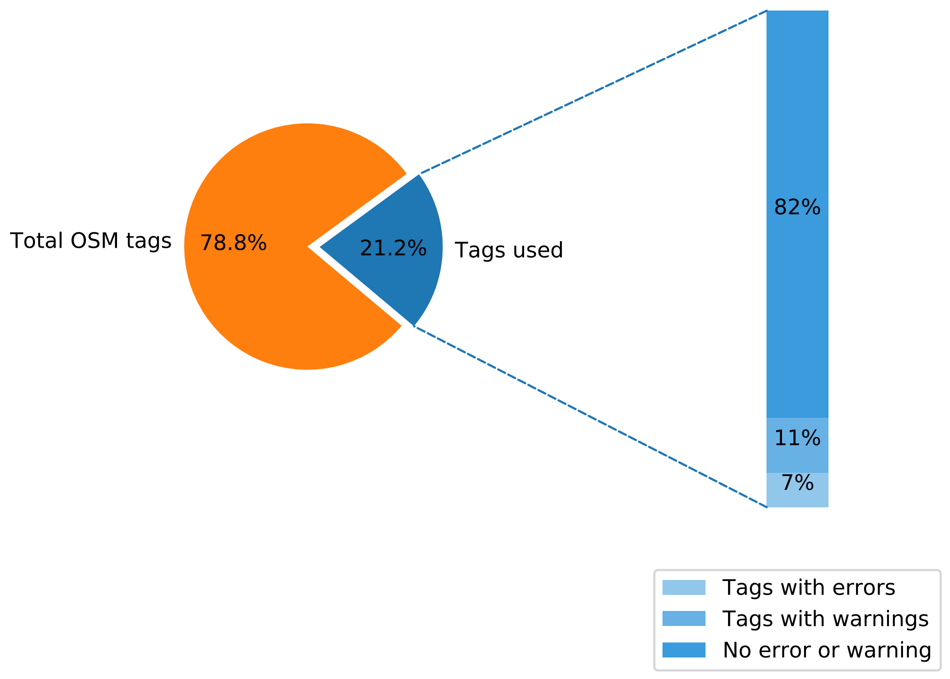


Figure 5.9: Distribution of errors and warnings from each of the used tags.

Figure 5.10, presents the distribution of all the tags under the vnfd/ path used on the dataset gathered. The plot does not represent all the possible tags because it is too much information. Instead, only the tags existent of the first level of the vnfd tree are represented since all the other configurations must be under a container of these. Thus, every other tag can only be declared if one of the represented is also declared.

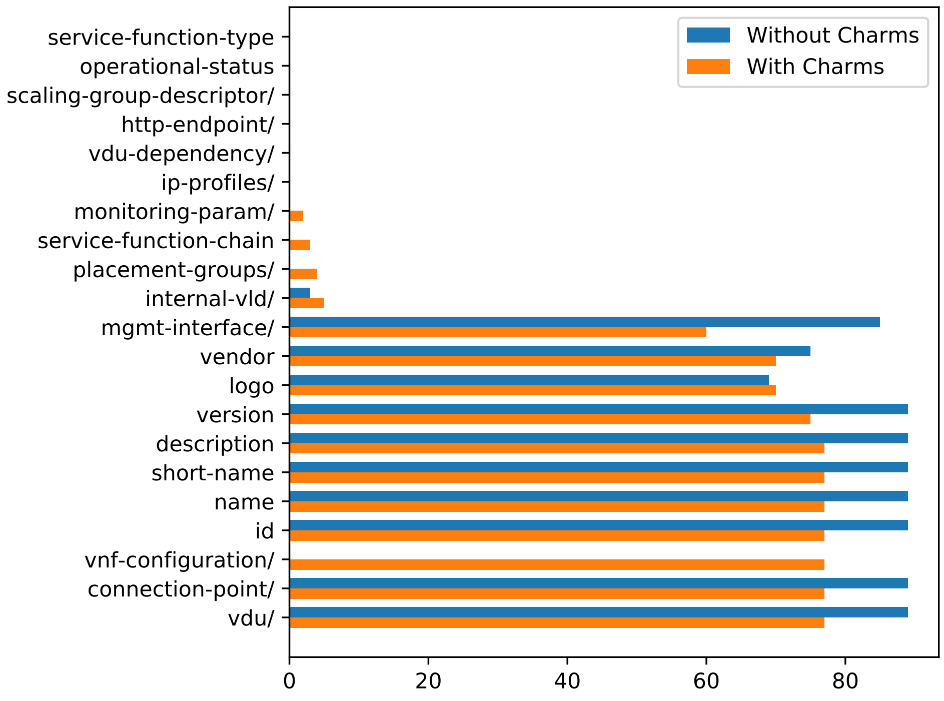


Figure 5.10: Difference between the tags usage between charmed and non-charmed descriptors.

On the referred plot, it is possible to verify that four containers and two tag properties are not designated in any descriptor. Moreover, the tags that have the most occurrences can be divided into two categories: metadata and configuration tags. The metadata tags are the ones that do not add any functionality to the VNF and are only used to give the package identity. These tags are the ID, name, short-name, description, version, vendor, and logo. On another hand, the configuration tags are the tags that customize the VNFs and add functionalities. In this case, these tags are the connection-point, the vdu, the mgmt-interface, and the vnf-configuration.

While the mgmt-interface and connection-point are specified as configuration tags, they do not add much to the VNFs customization. However, the usage of the vdu and vnf-configuration tags may be the reason for the non-utilization of other configuration tags specified in figure 5.10. The vnf-configuration may contribute to this problem because it is the tag where the charms are specified, and the primitives to execute the actions are defined. Since this tag is used so much when contrasted with other configuration tags, the VNF configuration is possibly being done over the charms instead of the available tags.

On another hand, and, even though the vdu definition is mandatory for the VNF description, when analyzing the tags used when there are no charms on the VNF (without the configuration of the vnf-configuration tag) and as pictured on figure 5.10, it is possible to verify that, apart from the mgmt-interface and connection-point, the only extraordinary configuration tag used is the internal-vld and it still has a meager usage rate.

Given that without charms and that the majority of descriptors will not have any further configuration rather than the basic, the produced VNFs would all be the same. However, this is not true; the VNFs are different and serve different purposes and verticals within the 5GinFIRE project. The problem is that in this type of descriptor without charms, the configurations are applied directly to the vdu image that is uploaded to the VIM, and the descriptor is just configured to instantiate VMs with the given image.

Looking at the utilization of the tags presented in figure 5.10, it already gives a big hint that the possibility that all the descriptors are very alike and the level of customization is shallow, leading to a reduced number of errors. However, in order to prove this statement, it is essential to check the real similarity between the descriptors. This similarity can be measured by gathering all the tags from each descriptor and measuring the Jaccard Index [72], which calculates the similarity between two finite sets, between, firstly, a reference file and then the existent dataset.

Another critical step was to find a reference point to compare the descriptors. OSM provides a script for generating the basic structure of a working VNF, which is also prepared to support charms or not [73]. This is the perfect reference point because it is the most basic working configuration, so it is interesting to understand how different are the dataset descriptors from this file.

Finally, the comparison was made by just measuring the Jaccard index between the tags and not its values since the goal is not to prove that the descriptors are completely equal but that experimenters do not take advantage of such a descriptive and dense IM.

With all the requirements met, the first test was to compare each descriptor to the reference file. In the first run, the descriptors were compared with a reference file that did not use charms, and the second time, the descriptors were compared with a reference with a charm. The results are presented on figure 5.11.

The Jaccard Index results range from zero to one with the former, meaning that the files are completely different and the latter meaning that the files are exactly equal. By analyzing the information displayed, it is possible to understand that the similarity index ranges between approximately 0.35 and 0.95. Regarding the comparison with a charmed reference, the peak value is 0.6. The reference without charm also scores many occurrences on this value. Looking at figure 5.10, it is possible to understand

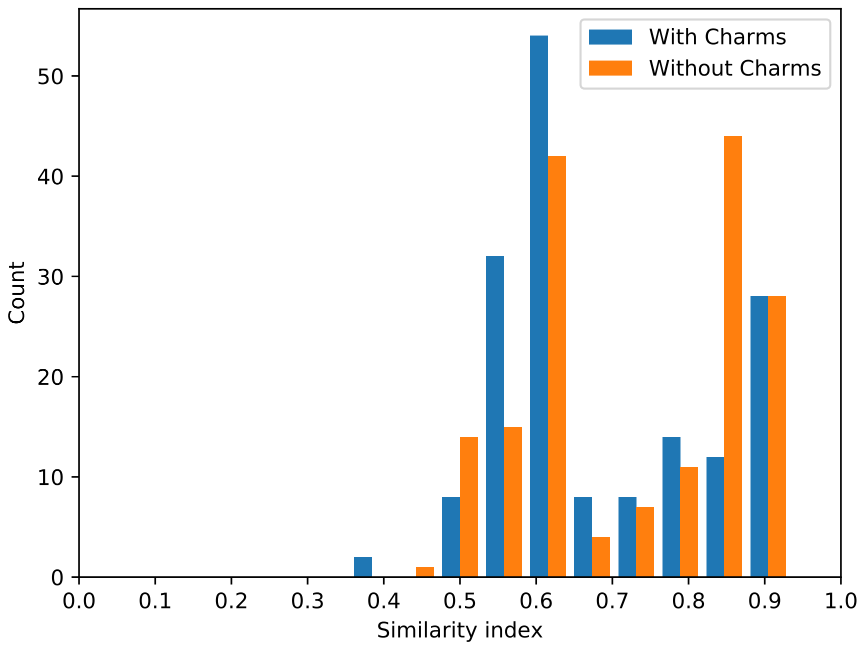


Figure 5.11: Jaccard index distribution when the dataset is compared with the reference VNFD.

why since this refers to all the metadata and containers (vdu, connection-point, and mgmt-interface) that are practically always present on every descriptor. On another hand, the reference without charms has its peak at approximately 0.85. Such value conducts to a very high similarity index that is also explained by the lack of tag variation for non-charmed descriptors on the figure referenced before.

The disparity between the peak values of charmed and non-charmed comparisons is a consequence of the higher variation in tags used for each type of descriptor presented before.

Overall, the similarity index, when compared with the reference files, is very high. The majority of the descriptors scored higher than 0.5, which means that the generality of the dataset is half equal to the reference. It is also true that the metadata definition has a significant role in this similarity; however, the reference descriptor does not have tags that were found on the dataset, such as placement-groups.

In order to have a deeper understanding of how this similarity applies in real scenarios, all the dataset descriptors were compared with one another, and the Jaccard Index was calculated for each interaction. Figure 5.12 portrays the similarity of each dataset descriptor regarding the others.

The referred plot holds a set of relevant information. The first detail that has to be noted is that the range of the Jaccard Index is practically the same as the results when

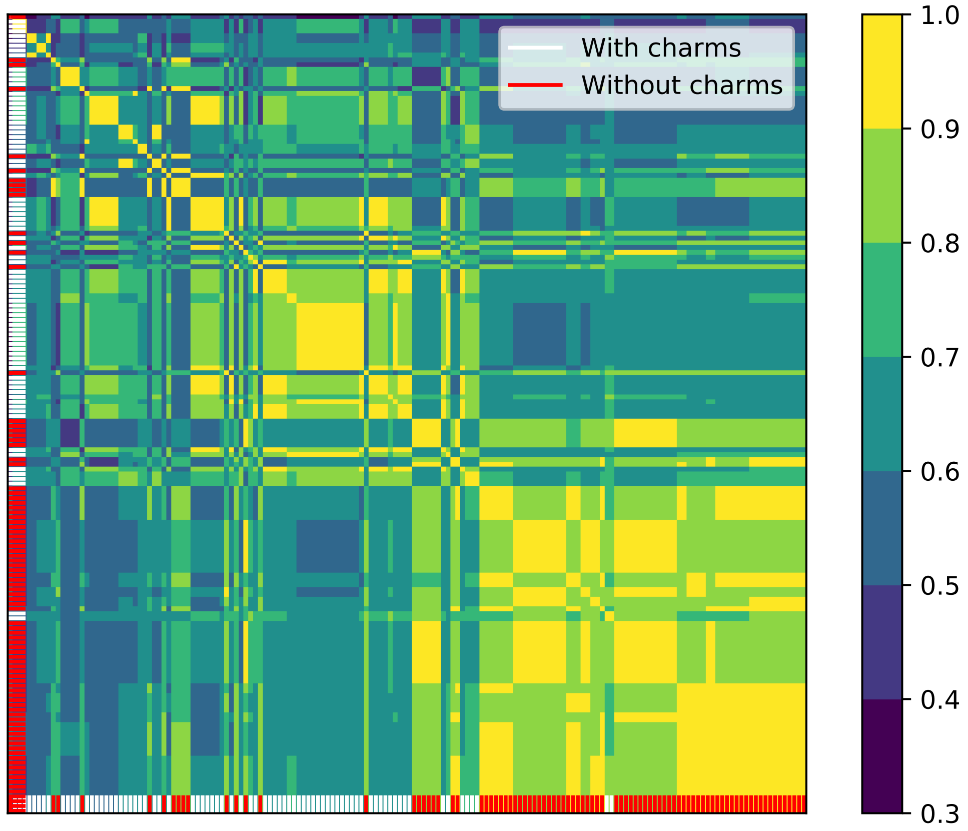


Figure 5.12: Matrix with distribution of the Jaccard Index between all the descriptors.

comparing with the reference. However, in this case, some descriptors score one, which means that they have precisely the same tags of the other descriptor on the dataset.

The red and white lines are references to help to visualize where the descriptors which contain charms and the ones who do not are located in the results matrix.

When looking at the similarities, it is possible to note that the most significant changes on the index occur when comparing charmed with non-charmed descriptors. This is expected since the charmed descriptors require extra configuration. Nevertheless, every time the descriptor types are compared directly, this is charmed with charmed and non-charmed with non-charmed, the similarity index is very high and, sometimes, even one. The index hardly goes lower than 0.6, and the variation is very low.

Taking into account these results, it is possible to state that the higher variations on the index occur when comparing descriptors that are from different types (charmed and non-charmed). When analyzing only the differences between the descriptors from the same group, it is possible to conclude that the differences are minimal, and many descriptors are equal to others.

This conclusion endorses the argument that the VNFD configurations have to be external from the descriptors; otherwise, the 166 descriptors would all have very similar functionalities since they are, in general, very homogeneous.

The similarity of these files has then the repercussions displayed on table 5.2. The configurations on the descriptor files are so low that there are hardly any errors to report.

In the end, it is possible to state that the main reason for the low number of errors in the descriptors is due to the lack of utilization of the capabilities of the OSM IM.

5.4 Summary

This chapter started by, in section 5.1, defining the two testing scenarios of the developed tools: the first was associated with the 5GinFIRE and was intended to test the automated mechanism. On another hand, the second was associated with the independent usage of the osm-descriptor-validator. In order to test the second scenario, it was necessary to gather descriptors from the portal. In order to have a big dataset, the pipeline was firstly deployed and integrated with the 5GinFIRE portal before the osm-descriptor-validator was developed, gathering 460 descriptors in total. This process was described in section 5.2.

The scenarios were then tested, and the results were collected being described in section 5.3. The first scenario was tested in the real world with the integration of Jenkins with 5GinFIRE and the deployment and integration of the validation pipeline with the osm-descriptor-validator. Since the integration, the pipeline ran 150 times, and the descriptors passed the tests 108 times.

The second scenario was tested using 166 unique descriptors and ended up with 88.5% of success rate.

As the high rates sounded odd, a more in-depth study of the results was followed. A lot of different metrics were analyzed, such as the distribution of errors and warnings, the most used tags, the tags with the most warnings and errors associated, and, finally, the Jaccard Index between descriptors.

With the described study performed, it was possible to verify that the majority of the descriptor files were very alike. In fact, when compared with descriptors of the same type (descriptors with charms being compared with descriptors with charms and vice-versa), the similarity index was very high, with sometimes reaching one (maximum similarity). Nevertheless, if the descriptors are very identical, it also means that their functionalities are the same unless the configurations are being done via the JuJu Charms or the VMs. Therefore, it is possible to state that the high success rate is due to the lack of utilization of the OSM IM.

CHAPTER 6

Conclusions

This work presented a way of validating VNFDs and NSD, its integration with a Continuous Integration server, and the deployment on the 5GinFIRE infrastructure. It also provided an in-depth study of the problems with the current VNFD development, as well as the most common errors.

Given the current evolution of networks and the migration to NFV environments, the softwarization of Network Function brings reliability concerns since network operators need to assure that the product being deployed behaves as expected. This quality assurance aspect has to come in an automated way in order to meet other operator requirements like fast deployments.

The problem stated was faced by the 5GinFIRE project. By working with multiple partners trying to do experiments with multiple verticals, the VNFs submitted were being validated manually. Projects like 5GTango have made efforts to produce a full SDK that performs tests according to the VNF submissions. However, the solution is not portable, required multiple configurations, and it is hard to integrate with different projects, like 5GinFIRE.

5GinFIRE needed an easy to integrate platform that would perform the necessary tests automatically. Having this statement as motivation, an OSM descriptor validator was proposed. This validator aimed to have extensive, easy to understand logs while being enough abstracted and lightweight to be deployed anywhere. These features, allied with Continuous Integration infrastructures, provided a fully automated service that met the necessities required by the project.

The solution had two parts: the development of the scripts and the setup and integration of the CI server with both the validator and the 5GinFIRE portal. The validation addresses syntactic, semantic, and referential possible errors and warnings, and the CI server handles the VNF fetching and the communication of the results to the portal.

As a result, the CI server with the validation tool is currently fully deployed in the 5GinFIRE environment. Currently, the VNFD are tested before being deployed into the infrastructure, and the success rate is very high. Given that NFV and its components are such a recent technology, the lack of descriptor errors raised curiosity. Since this deployment enabled the gathering of many VNFDs, details about its configurations have been analyzed in order to understand if there was any correlation between the type of VNFs being deployed in 5GinFIRE and the lack of errors. The Jaccard Index was calculated for the different descriptors, and the similarity between them was very high. In the end, it was possible to understand that developers are using the same descriptor configuration multiple times and doing the extra configuration via the images or the JuJu Charms. Nevertheless, the reason behind the low usage of some tags may be due to the fact that OSM was still not parsing some elements from its IM in the RO such as the scaling-group-descriptor or the monitoring-param, among others.

The solution developed will remain deployed in 5GinFIRE. However, it will not be completely helpful to ensure that a VNF is correct while developers use external procedures to configure it.

6.1 Future work

Given the solution proposed and the problems stated, it would be essential to have functional tests. A connection test was also envisioned for this validator; however, due to the lack of more complex descriptors that enabled the testing of that tool, it was decided not to include it on this document. Another problem was that the connection test relied on the usage of a different image from the one intended by the developer since there is usually no access to the original image. However, this was a problem because, as it was possible to understand, many times, the configuration comes with the VM image itself. Substituting that component would lead to testing a topology that was not defined by the developer. However, if the descriptor images could be available beforehand, it would be a good starting point for the functional testing.

Another problem that 5GinFIRE still faces is that the NS validation is still not implemented. Since the developed tool provides the validation of such descriptors, it is just necessary to configure another pipeline with the NSD validator with the portal endpoints for NSD submission.

References

[1] Cisco, “Cisco visual networking index: Global mobile data traffic forecast update, 2017–2022”, White paper, Cisco Systems Inc., Tech. Rep., Feb. 2019.

[2] F. Z. Yousaf, M. Bredel, S. Schaller, and F. Schneider, “Nfv and sdn—key technology enablers for 5g networks”, *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2468–2478, 2017.

[3] M. Peuster, S. Schneider, D. Behnke, M. Müller, P.-B. Bök, and H. Karl, “Prototyping and demonstrating 5g verticals: The smart manufacturing case”, in *2019 IEEE Conference on Network Softwarization (NetSoft)*, IEEE, 2019, pp. 236–238.

[4] D. Cotroneo, L. De Simone, A. K. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping, “Network function virtualization: Challenges and directions for reliability assurance”, in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, IEEE, 2014, pp. 37–42.

[5] R. Guerzoni *et al.*, “Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action, introductory white paper”, in *SDN and OpenFlow World Congress*, vol. 1, 2012, pp. 1–16.

[6] J. Batalle, J. F. Riera, E. Escalona, and J. A. Garcia-Espin, “On the implementation of nfv over an openflow infrastructure: Routing function virtualization”, in *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, IEEE, 2013, pp. 1–6.

[7] H. Karl, S. Dräxler, M. Peuster, A. Galis, M. Bredel, A. Ramos, J. Martrat, M. S. Siddiqui, S. Van Rossem, W. Tavernier, *et al.*, “Devops for network function virtualisation: An architectural approach”, *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 9, pp. 1206–1215, 2016.

[8] *5GinFIRE project objectives*, Accessed: 2019-10-29. [Online]. Available: `https://5ginfire.eu/project-objectives/`.

[9] A. P. Silva, C. Tranoris, S. Denazis, S. Sargento, J. Pereira, M. Luís, R. Moreira, F. Silva, I. Vidal, B. Nogales, *et al.*, “5ginfire: An end-to-end open5g vertical network function ecosystem”, *Ad Hoc Networks*, vol. 93, p. 101 895, 2019.

[10] Christos Tranoris, Apostolos Palladinos, Diogo Gomes, Eduardo Sousa, Borja Nogales, Iván Vidal, Olivier Tosello, “D3.1 - 5G Experimental portal, tools and middleware”, 2017, 5GinFIRE.

[11] P. S. Rajenda Chayapathi Syed Farrukh Hassan, *Network Functions Virtualizations (NFV) with a touch of SDN.* Pearson Education, Inc, 2017, ISBN: 0134463056.

[12] ETSI ISG NFV, “ETSI GS NFV 002 V1.1.1: Network Function Virtualisation (NFV): Architectural Framework”, Oct. 2013.

[13] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, “Network function virtualization: State-of-the-art and research challenges”, *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[14] *Network Function Virtualization (NFV)*, Accessed: 2019-10-07. [Online]. Available: https://www.etsi.org/technologies/nfv.

[15] *Industry Specification Group (ISG) Network Functions Virtualisation (NFV)*, Accessed: 2019-10-07. [Online]. Available: https://www.etsi.org/committee/nfv.

[16] ETSI ISG NFV, "ETSI GS NFV 002 V1.2.1: Network Function Virtualisation (NFV): Architectural Framework", Dec. 2014.

[17] ——, "ETSI GS NFV 003 V1.4.2: Network Function Virtualisation (NFV): Terminology for main concepts in NFV", Aug. 2018.

[18] ETSI OSM, "OSM Information Model: Release 2", Jul. 2017.

[19] ETSI ISG NFV, "ETSI GS NFV-MAN 001 V1.1.1: Network Function Virtualisation (NFV): Management and Orchestration", Dec. 2014.

[20] R. Mijumbi, J. Serrat, J.-L. Gorricho, S. Latré, M. Charalambides, and D. Lopez, "Management and orchestration challenges in network functions virtualization", *IEEE Communications Magazine*, vol. 54, no. 1, pp. 98–105, 2016.

[21] *ONAP Architecture Overview*, Accessed: 2019-11-26. [Online]. Available: https://www.onap.org/wp-content/uploads/sites/20/2019/07/ONAP_CaseSolution_Architecture_062519.pdf.

[22] *ONAP: Orchestration for Real Results - A Guide to ONAP Architecture and Use Cases*, Accessed: 2019-11-24. [Online]. Available: https://cloudify.co/wp-content/uploads/2018/02/ONAP-Orchestration-Architecture-Use-Cases-WP-Feb-2018.pdf.

[23] *Open Baton*, Accessed: 2019-11-26. [Online]. Available: https://openbaton.github.io.

[24] *SONATA*, Accessed: 2019-11-27. [Online]. Available: https://www.sonata-nfv.eu/.

[25] S. Dräxler, H. Karl, M. Peuster, H. R. Kouchaksaraei, M. Bredel, J. Lessmann, T. Soenen, W. Tavernier, S. Mendel-Brin, and G. Xilouris, "Sonata: Service programming and orchestration for virtualized software networks", in *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, IEEE, 2017, pp. 973–978.

[26] OSM End User Advisory Group, "OSM scope, functionality, operation and integration guidelines", Feb. 2019, White paper.

[27] *ETSI Open Source Mano*, Accessed: 2019-10-07. [Online]. Available: https://www.etsi.org/technologies/nfv/open-source-mano.

[28] ETSI OSM, "OSM Release Three: A Technical Overview", Oct. 2017, White paper.

[29] ——, "OSM Scope, Functionality, Operation and Integration Guidelines", Dec. 2019, White paper.

[30] *OSM VNFD tree*, Accessed: 2019-10-15. [Online]. Available: http://osm-download.etsi.org/repository/osm/debian/ReleaseFIVE/docs/osm-im/osm_im_trees/vnfd.html.

[31] *OSM Hackfest Basic VNF*, Accessed: 2019-10-15. [Online]. Available: https://osm-download.etsi.org/ftp/osm-5.0-five/6th-hackfest/packages/hackfest_basic_vnf.tar.gz.

[32] *OSM NSD tree*, Accessed: 2019-10-15. [Online]. Available: http://osm-download.etsi.org/repository/osm/debian/ReleaseFIVE/docs/osm-im/osm_im_trees/nsd.html.

[33] *OSM Hackfest Basic NS*, Accessed: 2019-10-15. [Online]. Available: https://osm-download.etsi.org/ftp/osm-5.0-five/6th-hackfest/packages/hackfest_basic_ns.tar.gz.

[34] *JuJu Charms*, Accessed: 2019-10-10. [Online]. Available: https://jaas.ai/docs/what-is-juju.

[35] *OSM Hackfest SimpleCharm VNF*, Accessed: 2019-10-15. [Online]. Available: https://osm-download.etsi.org/ftp/osm-5.0-five/6th-hackfest/presentations/6th%20OSM%20Hackfest%20-%20Session%206%20-%203-Building%20a%20VNF%20with%20primitives.pdf.

[36] *OSM Hackfest SimpleCharm VNF*, Accessed: 2019-10-15. [Online]. Available: https://osm-download.etsi.org/ftp/osm-5.0-five/6th-hackfest/presentations/6th%20OSM%20Hackfest%20-%20Session%206%20-%202-Building%20a%20Proxy%20charm.pdf.

[37] ETSI ISG NFV, "ETSI GS NFV-SOL 005 V2.6.1: Network Function Virtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Os-Ma-nfvo Reference Point", Apr. 2019.

[38] ETSI OSM, "OSM Release Five: A Technical Overview", Jan. 2019, White paper.

[39] Anastasius Gavras, C. Tranoris, and S. Denazis, "The 5ginfire platform a testbed for end-to-end 5g experimentation", 2018.

[40] *5GinFIRE Wiki*, Accessed: 2019-11-28. [Online]. Available: http://wiki.5ginfire.eu/.

[41] Diego López, Juan Martínez, Iván Fernández, Luis Blázquez, Diogo Gomes, Eduardo Sousa, Inês Moreira, Aloizio Silva, Flávio Silva, Nikos Makris, Christos Zarafetas, Alexandros Valantasis, Thanasis Korakis, Diarmuid Collins, Maicon Kist, Damian Parniewicz, Bartosz Krakowiak, "D4.2 - Intermediate Report on the MANO Platform", 2018, 5GinFIRE.

[42] J. Wettinger, U. Breitenbücher, and F. Leymann, "Devopslang–bridging the gap between development and operations", in *European Conference on Service-Oriented and Cloud Computing*, Springer, 2014, pp. 108–122.

[43] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, *et al.*, "Manifesto for agile software development", 2001.

[44] M. Virmani, "Understanding devops & bridging the gap from continuous integration to continuous delivery", in *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, IEEE, 2015, pp. 78–82.

[45] M. Hüttermann, *DevOps for developers.* Apress, 2012.

[46] *DevOps Days Ghent*, Accessed: 2019-11-25. [Online]. Available: https://legacy.devopsdays.org/events/2009-ghent/.

[47] R. Jabbari, N. bin Ali, K. Petersen, and B. Tanveer, "What is devops?: A systematic mapping study on definitions and practices", in *Proceedings of the Scientific Workshop Proceedings of XP2016*, ACM, 2016, p. 12.

[48] S. K. Bang, S. Chung, Y. Choh, and M. Dupuis, "A grounded theory analysis of modern web applications: Knowledge, skills, and abilities for devops", in *Proceedings of the 2nd annual conference on Research in information technology*, ACM, 2013, pp. 61–62.

[49] A. Dyck, R. Penners, and H. Lichter, "Towards definitions for release engineering and devops", in *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, IEEE, 2015, pp. 3–3.

[50] S. W. Hussaini, "Strengthening harmonization of development (dev) and operations (ops) silos in it environment through systems approach", in *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, IEEE, 2014, pp. 178–183.

[51] C. A. Cois, J. Yankel, and A. Connell, "Modern devops: Optimizing software development through effective system interactions", in *2014 IEEE International Professional Communication Conference (IPCC)*, IEEE, 2014, pp. 1–7.

[52] B. S. Farroha and D. L. Farroha, “A framework for managing mission needs, compliance, and trust in the devops environment”, in *2014 IEEE Military Communications Conference*, IEEE, 2014, pp. 288–293.

[53] J. Humble and J. Molesky, “Why enterprises must adopt devops to enable continuous delivery”, *Cutter IT Journal*, vol. 24, no. 8, p. 6, 2011.

[54] V. Gupta, P. K. Kapur, and D. Kumar, “Modeling and measuring attributes influencing devops implementation in an enterprise using structural equation modeling”, *Information and Software Technology*, vol. 92, pp. 75–91, 2017.

[55] B. Fitzgerald and K.-J. Stol, “Continuous software engineering and beyond: Trends and challenges”, in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, ACM, 2014, pp. 1–9.

[56] G. Booch, *Object Oriented Design with Applications*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991, ISBN: 0-8053-0091-0.

[57] M. Fowler and M. Foemmel, “Continuous integration”, *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, vol. 122, p. 14, 2006.

[58] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[59] Marios Touloupou, Evgenia Kapassa, Michael Filippakis, Dimitris Dres, Manuel Peuster, Stefan Schneider, Eleni Fotopoulou, Anastasios Zafeiropoulos, Antón Román Portabales, Ana Pol González, Peer Hasselmeyer, Thomas Soenen, Askhat Nuriddinov, Wouter Tavernier, “D4.2 Final release of the service validation SDK toolset”, 2019, 5GTango.

[60] *VNFD YANG model*, Accessed: 2019-10-20. [Online]. Available: `https://open.riftio.com/documentation/riftware/7.1/a/descriptor/yang-models/vnfd-yang-model.htm`.

[61] *NSD YANG model*, Accessed: 2019-10-20. [Online]. Available: `https://open.riftio.com/documentation/riftware/7.1/a/descriptor/yang-models/nsd-yang-model.htm`.

[62] *Jenkins*, Accessed: 2019-11-22. [Online]. Available: `https://jenkins.io/`.

[63] *TravisCI*, Accessed: 2019-11-22. [Online]. Available: `https://travis-ci.org/`.

[64] *BitbucketCI*, Accessed: 2019-11-22. [Online]. Available: `https://bitbucket.org/product/features/pipelines`.

[65] *CircleCI*, Accessed: 2019-11-22. [Online]. Available: `https://circleci.com/`.

[66] *GitlabCI*, Accessed: 2019-11-22. [Online]. Available: `https://docs.gitlab.com/ee/ci/`.

[67] *Bamboo*, Accessed: 2019-11-22. [Online]. Available: `https://www.atlassian.com/software/bamboo`.

[68] *Codeship*, Accessed: 2019-11-22. [Online]. Available: `https://codeship.com/`.

[69] *TeamCity*, Accessed: 2019-11-22. [Online]. Available: `https://www.jetbrains.com/teamcity/`.

[70] *Buddy*, Accessed: 2019-11-22. [Online]. Available: `https://buddy.works/`.

[71] *DroneCI*, Accessed: 2019-11-22. [Online]. Available: `https://drone.io/`.

[72] P. Jaccard, “The distribution of the flora in the alpine zone. 1”, *New phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[73] *Creating your own VNF package*, Accessed: 2019-11-16. [Online]. Available: `https://osm.etsi.org/wikipub/index.php/Creating_your_own_VNF_package`.