SEMIoTICS

Deliverable D3.8
Network Functions Virtualization
for IoT (final)

Deliverable release date	29.02.2020
Authors	1. Jordi Serra, Luis Sanabria-Russo, David Pubill, Angelos Antonopoulos, Christos Verikoukis (CTTC) 2. Nikolaos Petroulakis (FORTH) 3. Ermin Sakic (SAG) 4. Philip Wright, Domenico Presenza (ENG) 5. Tobias Marktscheffel, Felix Klement, Korbinian Spielvogel and Henrich C. Pöhls (UP)
Responsible person	Jordi Serra, Luis Sanabria-Russo (CTTC)
Reviewed by	Ermin Sakic (SAG), Nikolaos Petroulakis (FORTH)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Verikoukis Christos) PCC Members (Vivek Kulkarni, Ioannis Askoxylakis, Verikoukis Christos, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Ulrich Hansen)
Status of the Document	Final
Version	1.0
Dissemination level	Public

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

EXECUTIVE SUMMARY

This deliverable D3.8 is the final deliverable of the SEMIoTICS' task 3.2, in work package 3. Moreover, this deliverable consolidates the contents of deliverable D3.2. This task deals with the design, development and incorporation of the Network Function Virtualization (NFV) component within the context of the SEMIoTICS project.

In general terms, NFV relies on a new concept for networks' management, which is inspired by the cloud computing paradigm. Namely, in NFV generic servers are incorporated in the network with the aim of substituting to a great extent vendor specific special-purpose nodes. The computing, storage and networking resources of these generic purpose servers are virtualized yielding an NFV Infrastructure (NFVI). Thereby, network functionalities can be deployed on top of this NFVI in the form of the so-called Virtual Network Functions (VNF). The global management of the pool of virtual resources along with the lifecycle of VNFs is responsibility of the so-called NFV Management and Orchestration entity (NFV MANO). Therefore, NFV yields a flexible, dynamic and programmable network, which is the perfect match for the needs of SEMIoTICS.

Namely, SEMIoTICS considers a scenario that it is driven by the nature of Internet of Things (IoT), which requires to address the next technical hurdles: dynamicity, scalability, heterogeneity, end-to-end security and privacy. Thereby, from the networking perspective, NFV is a key ingredient to face those challenges. In effect, thanks to the virtualization approach and the global management of the virtual resources, the NFV MANO is able to scale dynamically resources to support the VNFs according to the heterogeneous quality of service requirements of the IoT applications at hand. As an example, a VNF with a given functionality can be deployed either at the network edge or at the backend cloud. The former guarantees lower latency, the latter more resources for computationally intensive tasks. Moreover, end-to-end security or privacy are easier to be delivered within the NFV approach, as the set of security functionalities are mapped to a set of VNFs and are managed globally by the centralized NFV MANO.

Next, we present the organization of this deliverable, whose aim is to face the challenges posed by SEMIoTICS, from the perspective of NFV. In section 1, NFV is introduced and it is motivated within the SEMIoTICS framework. Then, in section 2, the links to the SEMIoTICS architecture, the requirements and the Key Performance Indicators (KPI), where NFV is relevant, are highlighted. Also, the task objectives are discussed. In section 3, we present relevant VNFs, and chains of VNFs, to address the security, privacy and dependability functionalities of SEMIoTICS, along with their challenges. Section 4 deals with the global management of the NFV resources and VNFs for SEMIoTICS, i.e. it treats the NFV MANO for SEMIoTICS. To this end, we describe the practical implementation of the NFV MANO for SEMIoTICS by means of open source tools such as Open Source MANO (OSM) and OpenStack. In this regard, we also describe how we enable chains of VNFs. i.e. Service Function Chains (SFC) and how we enable measurements of the NFV resources. This is of paramount importance for the dynamic management of NFV resources, VNFs and SFC. Then, we present the interaction between the NFV component and other SEMIoTICS components such as the Pattern Engine, the Pattern Orchestrator or SEMIoTICS' components embedded in VNFs. To this end, we explain dynamic sequence diagrams to exemplify the interaction between the NFV component and those other SEMIoTICS components. We also present and develop northbound interfaces between the NFV MANO, and external entities known as Operations Support Systems (OSS), e.g. the Pattern Orchestrator/Pattern Engine. These interfaces are based on Representational state transfer (REST) Application Programming Interfaces (API). Also, in section 4 we explain the dynamic resource management in NFV. More specifically, we present experiments on dynamic scaling the processing rate in VNFs and we treat the load balancing among VNF instances, which arise during the aforementioned scaling out operations of VNFs. Afterwards, section 5 describes the interfaces in the NFV ecosystem, i.e. between the NFV sub blocks as well as the interfaces between the NFV component and other SEMIoTICS components such as the SDN controller and the Pattern Engine. Finally, section 6 presents the conclusions. Also, in this section we present the technical choices of the previous sections and whether they are state-of-the-art (SoA) or beyond SoA.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

Table of Contents

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

ACRONYMS TABLE

Acronym	Definition
ACL	Access Control List
ASIC	Application-Specific Integrated Circuit
API	Application Programming Interfaces
CAPEX	CAPital EXpenditure
CPU	Central Processing Unit
DoS	Denial of Service
DPI	Deep Packet Inspection
EM	Element Management
FPGA	Field-Programmable Gate Array
FW	Firewall
GW	Gateway
HP	Honeypots
IDS	Intrusion Detection System
IoT	Internet of Things
IIoT	Industrial Internet of Things
IPS	Intrusion Prevention System
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LXD	Linux Containers
JSON	JavaScript Object Notation[1]
FW	Firmware
MANO	Management and Orchestration
M2M	Machine-to-Machine
MQTT	Message Queuing Telemetry Transport[2]
NBI	Northbound Interface
NETCONF	Network Configuration Protocol
NFV	Network Functions Virtualization
NFVI	Network Functions Virtualization Infrastructure
NFVI-RA	Network Functions Virtualization Infrastructure Resource Allocation
NFVO	NFV Orchestrator
NS	Network Services
NSd	Network Service descriptor
OFCONF	OpenFlow Configuration
OPEX	OPerational EXpenditure
OSM	Open Source MANO
OUC	OpenStack User Configuration
OVSDB	Open vSwitch Database Management Protocol
PNF	Physical Network Function
POP	Point of Presence
QoS	Quality of Service
REST	Representational state transfer
RO	Resource Orchestrator
SBI	Southbound Interface
SDN	Software-Defined Networking
SEMIoTICS	Smart End-to-end Massive IoT Interoperability, Connectivity and Security
SFC	Service Function Chaining
SoA	State-of-the-Art

---

[1] https://en.wikipedia.org/wiki/JSON
[2] https://en.wikipedia.org/wiki/MQTT

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

SPDI	Security, Privacy, Dependability, and Interoperability
SSC	SEMIoTICS SDN Controller
SSD	Solid State Disk
UC	Use Case
VIM	Virtualized Infrastructure Manager
VLAN	Virtual Local Area Network
VLd	Virtual Link descriptor
VM	Virtual Machine
VNF	Virtual Network Function
VNFd	Virtual Network Function descriptor
VNF-FG	VNF Forwarding Graph
VNFFGd	VNF Forwarding Graph descriptor
vSwitch	Virtual Switch
VTN	Virtual Tenant Networks
VXLAN	Virtual eXtensible Local Area Network
WoT	Web of Things
WP	Work Package

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

1 INTRODUCTION

Network Functions Virtualization (NFV) is the cornerstone behind new networking frameworks, whose aim is to increase the flexibility, programmability, scalability and efficiency of communications networks by leveraging the cloud computing philosophy. That is, by using general purpose equipment, rather than vendor-specific hardware, which allow the virtualization of their computing, storage and communications resources. Thus, network services are deployed in a flexible manner on top of this virtualized infrastructure. Therefore, the aim of this deliverable is to discuss the benefits of the NFV technology for the SEMIoTICS project: support network services with heterogeneous Quality of Service (QoS) guarantees such as low latency, reliability, security and privacy; provide scalable, adaptable and dynamic network services to client IoT applications. This document is the final deliverable of task 3.2. Moreover, it is worth mentioning that this task has interplays with the other work packages (WPs) of the SEMIoTICS project. Namely, WP 2 is an input for task 3.2, and task 3.2 produces outputs for WP 5 and 6. Finally, it interacts with WP 4 to leverage the pattern-driven approach of WP 4 and with the tasks of WP 3 to provide a coherent networking approach.

1.1 Motivation behind NFV

The traditional approach in networking has relied on vendor specific special-purpose network nodes, where hardware and software are tightly coupled. Thereby, the configuration of those nodes is rather costly and leads to a rather rigid network. However, this traditional approach hardly holds nowadays. There are several reasons for this issue. First, the number of devices requiring network connectivity and the data rate have increased dramatically, mostly due to a huge number of IoT devices and mobile terminals. Second, the appearance of IoT demand services with dynamic and heterogeneous (QoS) requirements. In this scenario, the traditional networking approach, based on vendor specific special-purpose nodes, leads to dramatic increases in Capital (CAPEX) and Operational (OPEX) Expenditures [1]. These issues are circumvented thanks to a new networking approach based on NFV.

The aim of NFV is to provide a network that is dynamic, flexible, scalable, programmable and easy to reconfigure. This paves the way to support a huge number of IoT devices and novel IoT services with heterogeneous QoS requirements by allocating the necessary resources. All these features are desirable in the SEMIoTICS project, where a massive amount of IoT devices must be connected with the IoT gateways and the Backend cloud with different QoS constraints, e.g. in terms of latency, reliability, security and privacy. These QoS measures must be guaranteed despite the impairments posed by the network, i.e. data flow paths that guarantee the required QoS must be established dynamically. Moreover, due to latency, computational and communication constraints the IoT data analytics is carried out either at the IoT Gateway or at the backend cloud. Thereby, the network must be flexible and programmable so as to setup and to release the computational and communication resources to convey the information to the appropriate computing resources, and to allow a computation that guarantees the required QoS. For all these reasons, it is mandatory to have a global view of the network state and a global control of the network resources. In NFV this is accomplished thanks to a centralized orchestration in the so-called NFV Management and Orchestration (NFV MANO).

To this end, NFV relies on the following pillars. First, it considers general-purpose hardware nodes, rather than vendor-specific special-purpose ones, in specific parts of the network. Second, the compute, storage and communication resources of the network nodes are virtualized and exposed as a Network Function Virtualization Infrastructure (NFVI). Third, network services are envisaged as a chain of Virtual Network Functions (VNFs) deployed on top of the NFVI by dynamically allocating the required resources demanded by the service so as to guarantee the specified QoS. The coordination and control of the VNF, as well as the NFVI to deploy them, require a specific functional block, the so-called NFV MANO. Further insights are given in the upcoming sections of the deliverable. Namely, section 3 deals with the VNFs and chains of VNFs, so-called SFC, that guarantee the deployment of network services with the QoS desired in SEMIoTICS. Section 4 deals with the NFV MANO. Thus, it discusses its functional architecture, it describes how it controls the VNF deployment and the required NFVI resources as well as the relation between the NFV MANO and the SEMIoTICS' Pattern Orchestrator. Section 5 explains the interfaces between all the functional blocks of sections 3 and 4. Finally, section 6 concludes the deliverable.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

1.2   Functional blocks of an NFV platform

In Figure 1 the functional blocks of an NFV platform are displayed. This architecture is compliant with the one proposed by ETSI in [2]. In this section each of the blocks are overviewed and thorough details on how they are considered and implemented within the SEMIoTICS framework are given in the upcoming sections, namely in sections 3, 4, and 5.

As it has been mentioned in the previous sections NFV offers flexible, programmable, dynamic, scalable and easy ways to reconfigure network resources to provide the QoS demanded by SEMIoTICS IoT Use Cases (UC). To this end, NFV follows the next approach. First, general purpose hardware devices are considered in different parts of the network. In the SEMIoTICS architecture this corresponds to computing and storage nodes within the IoT Gateway and the backend cloud. Moreover, it is assumed that these machines allow the virtualization of their resources in terms of e.g. Virtual Machines (VM) or containers yielding a pool of virtual computing, storage and communication resources available to deploy the network services. The virtualization of the hardware resources is managed by a so-called virtualization layer. As it can be seen in Figure 1, the set of physical hardware resources, the virtualization layer and the virtualized computing storage and networking resources is so-called NFVI. Thereby, NFVI contains all the resources available in the network. NFVI paves the way to obtain a flexible, programmable, dynamic and scalable network, as the virtual network resources exposed to the network services can be dynamically assigned or released in different parts of the infrastructure to meet the required QoS requirements.

FIGURE 1 FUNCTIONAL BLOCKS OF AN NFV PLATFORM

In an NFV, the network services are implemented as a chain of functional blocks, which are called VNF. A VNF is a virtualization of a network function in a legacy non-virtualized network, referred to as Physical Network Functions (PNF). Moreover, the chain of VNFs that implements a network service is called a SFC. As it is displayed in Figure 1, each VNF is deployed on top of the NFVI. Namely, virtual computing, storage

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

and network resources are assigned to run the VNF. This allocation of virtual resources is exemplified in Figure 2. This figure highlights the flexibility, scalability and support for heterogeneous QoS requirements that is provided by an NFV, as virtual resources can be assigned or released easily according to the QoS required by the VNFs and the SFC. It is also worth mentioning that the Element Management (EM) in Figure 1 is the responsible for the VNF management.
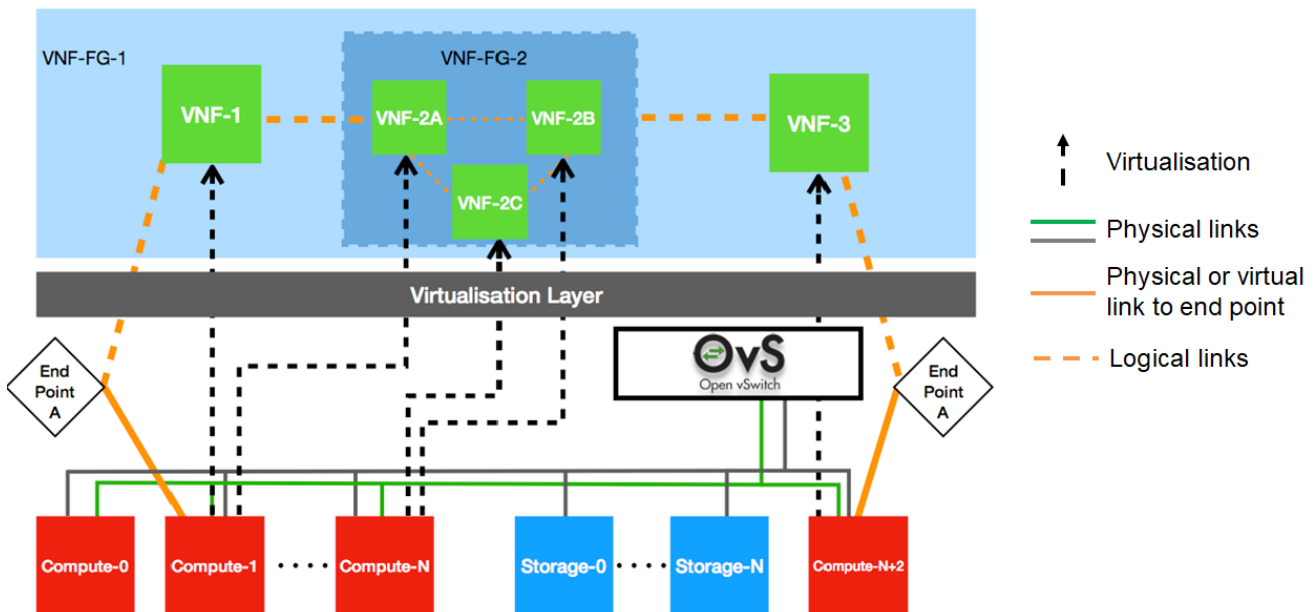
FIGURE 2 EXAMPLE OF VIRTUAL RESOURCES ALLOCATION TO A CHAIN OF VNFS

At this point, we have seen how network services are implemented in an NFV in terms of VNFs, or more in general, SFC. Additionally, it is observed that each VNF requires a set of virtual resources from the NFVI. Obviously, all these blocks, i.e. the network services and the network resources, require a global management. In an NFV architecture, the responsible for this management is the NFV MANO, which is introduced now.

The NFV MANO is composed of three main blocks:
- The NFV Orchestrator (NFVO).
- The VNF manager.
- The Virtualized Infrastructure Manager (VIM).

In fact, these NFV MANO blocks are organized in a hierarchical manner in terms of management responsibility. That is, the orchestrator is the responsible of managing the overall NFV. Thus, it manages the communication with the network service providers by exposing proper northbound interfaces. For instance, the OSM, which is an ETSI-compliant open source implementation of the NFVO and VNF manager blocks, provides open standard-based APIs such as NETCONF and REST, see section 4. Through these interfaces the network service providers can specify the features of their services. Namely, in OSM they use the so-called Network Service Descriptors (NSd) [3]. These NSd in turn refers to a set of VNF descriptors (VNFd), which characterize the VNFs that the network service requires. The VNFs are connected through virtual links that are defined properly through Virtual Link descriptors (VLd). Moreover, the VNF-FG descriptors (VNFFGd) determine the traffic flows between the VNFs in the service chain associated to the network service. Thereby, the NFVO is the responsible of the network service management lifecycle, which implies the next responsibilities:

- Manage the global computing, storage and communication virtual resources.
- Authorize NFVI resources requests.

9

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

- Policy management related to scalability, to reliability, to high availability related to network services instances.
- Manage the catalog of network services templates.
- Onboarding of new network services and VNFs packages.

As it is shown in Figure 1, the NFVO has southbound interfaces (specified via reference points [4]) with the VNF manager and the VIM. Thereby, for a given network service request, the NFVO delegates the management of the VNFs and virtual resources involved in the network service, to the VNF manager and to the VIM, respectively. Moreover, the VIM and the VNF manager use these interfaces in a northbound direction e.g. to send state information on their management and the state of the configurations requested by the NFVO.

Another important block of the NFV MANO is the VNF manager. It is the responsible of the VNF lifecycle management. This includes the next responsibilities:

- VNF instantiation or start, given its associated VNF descriptor.
- VNF monitoring by collecting parameters that determine the VNF health, e.g. CPU load or memory usage.
- VNF scaling. That is, Key Performance Indicators (KPI) of the VNF are monitored and if they are above a given threshold a scaling process is started, which implies the creation of new VM to deploy the VNF.
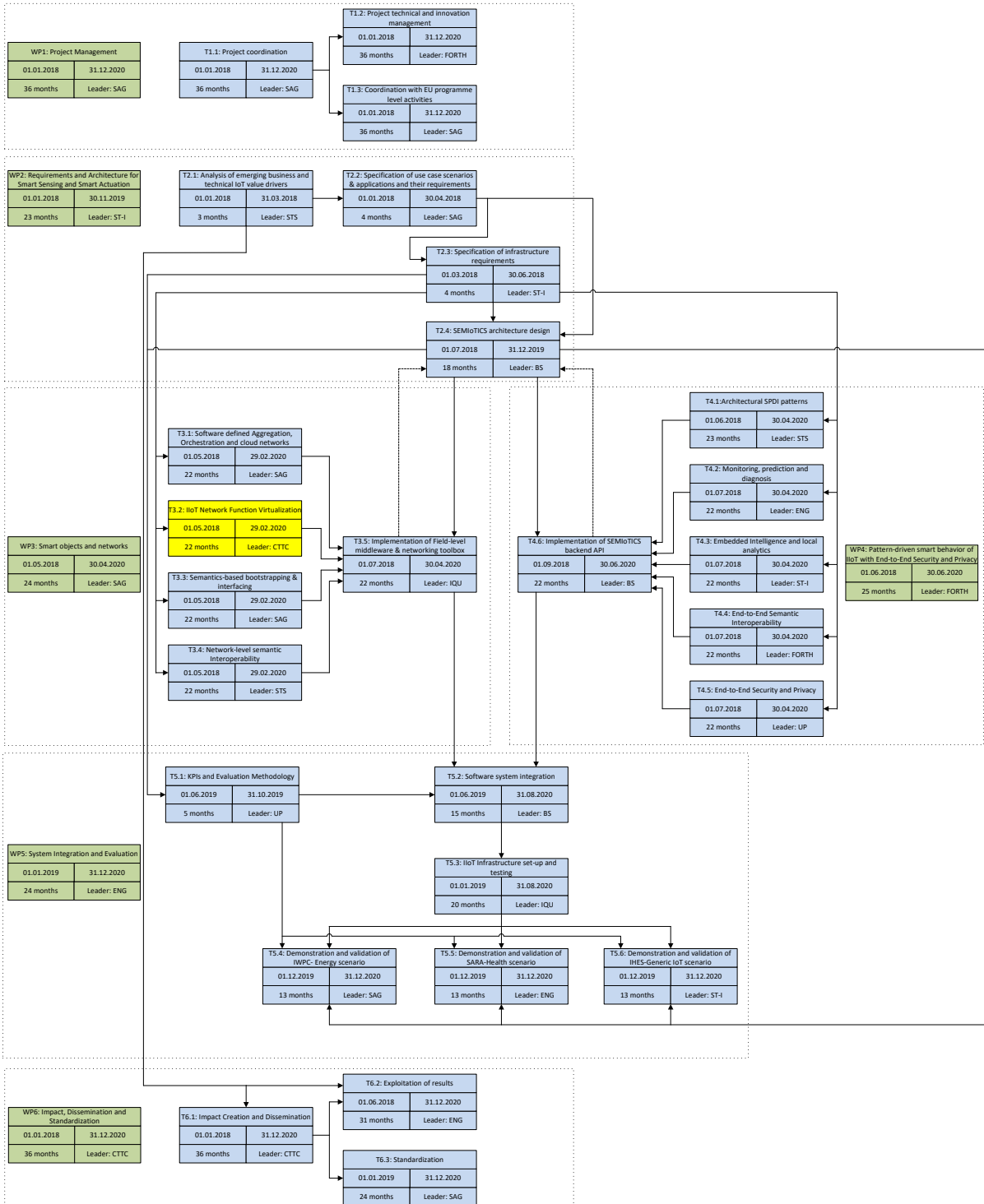- VNF termination.

The third sub-block that builds an NFV MANO is the VIM, which is the responsible for managing the overall NFVI. Thereby, upon request of the VNF manager and the NFVO, the VIM must assign the necessary virtual compute, storage and network resources to run properly the VNFs that form an SFC. The VIM has also the next roles:

- It manages the inventory of the computing, storage and network resources related to the NFVI.
- The VIM manages the NFVI resources allocation. Thus, it facilitates the assignment, increase or release of resources to VMs to run or to terminate a given VNF.
- It provides logs related to performance issues that arise in the NFVI.
- It has information on the infrastructure faults.
- It collects information to monitor the state of the NFVI and to allow the optimization of its resources.

At this point, it is worth mentioning that there are several alternatives to implement an NFV MANO. Herein, on the one hand, the Open Source MANO (OSM release FIVE or newer) is considered. It provides an open source ETSI compliant implementation of the NFV orchestrator and VNF manager. On the other hand, herein OpenStack is considered as the VIM, as it is open source, compatible with OSM, and it is regarded as a stable and consolidated VIM. Further details about these choices and their use for SEMIoTICS are given in section 4.

Last, but not least, observe that the interfaces between functional blocks are named according to the ETSI NFV nomenclature [2], and further details on their role within the SEMIoTICS context are given in section 5.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

SEMIoTICS

1.3   PERT chart of SEMIoTICS

WP1: Project Management
01.01.2018 — 31.12.2020
36 months | Leader: SAG

T1.1: Project coordination
01.01.2018 — 31.12.2020
36 months | Leader: SAG

T1.2: Project technical and innovation management
01.01.2018 — 31.12.2020
36 months | Leader: FORTH

T1.3: Coordination with EU programme level activities
01.01.2018 — 31.12.2020
36 months | Leader: SAG

WP2: Requirements and Architecture for Smart Sensing and Smart Actuation
01.01.2018 — 30.11.2019
23 months | Leader: ST-I

T2.1: Analysis of emerging business and technical IoT value drivers
01.01.2018 — 31.03.2018
3 months | Leader: STS

T2.2: Specification of use case scenarios & applications and their requirements
01.01.2018 — 30.04.2018
4 months | Leader: SAG

T2.3: Specification of infrastructure requirements
01.03.2018 — 30.06.2018
4 months | Leader: ST-I

T2.4: SEMIoTICS architecture design
01.07.2018 — 31.12.2019
18 months | Leader: BS

WP3: Smart objects and networks
01.05.2018 — 30.04.2020
24 months | Leader: SAG

T3.1: Software defined Aggregation, Orchestration and cloud networks
01.05.2018 — 29.02.2020
22 months | Leader: SAG

T3.2: IIoT Network Function Virtualization
01.05.2018 — 29.02.2020
22 months | Leader: CTTC

T3.3: Semantics-based bootstrapping & interfacing
01.05.2018 — 29.02.2020
22 months | Leader: SAG

T3.4: Network-level semantic Interoperability
01.05.2018 — 29.02.2020
22 months | Leader: STS

T3.5: Implementation of Field-level middleware & networking toolbox
01.07.2018 — 30.04.2020
22 months | Leader: IQU

T4.1: Architectural SPDI patterns
01.06.2018 — 30.04.2020
23 months | Leader: STS

T4.2: Monitoring, prediction and diagnosis
01.07.2018 — 30.04.2020
22 months | Leader: ENG

T4.3: Embedded Intelligence and local analytics
01.07.2018 — 30.04.2020
22 months | Leader: ST-I

T4.4: End-to-End Semantic Interoperability
01.07.2018 — 30.04.2020
22 months | Leader: FORTH

T4.5: End-to-End Security and Privacy
01.07.2018 — 30.04.2020
22 months | Leader: UP

T4.6: Implementation of SEMIoTICS backend API
01.09.2018 — 30.06.2020
22 months | Leader: BS

WP4: Pattern-driven smart behavior of IIoT with End-to-End Security and Privacy
01.06.2018 — 30.06.2020
25 months | Leader: FORTH

WP5: System Integration and Evaluation
01.01.2019 — 31.12.2020
24 months | Leader: ENG

T5.1: KPIs and Evaluation Methodology
01.06.2019 — 31.10.2019
5 months | Leader: UP

T5.2: Software system integration
01.06.2019 — 31.08.2020
15 months | Leader: BS

T5.3: IIoT Infrastructure set-up and testing
01.01.2019 — 31.08.2020
20 months | Leader: IQU

T5.4: Demonstration and validation of IWPC- Energy scenario
01.12.2019 — 31.12.2020
13 months | Leader: SAG

T5.5: Demonstration and validation of SARA-Health scenario
01.12.2019 — 31.12.2020
13 months | Leader: ENG

T5.6: Demonstration and validation of IHES-Generic IoT scenario
01.12.2019 — 31.12.2020
13 months | Leader: ST-I

WP6: Impact, Dissemination and Standardization
01.01.2018 — 31.12.2020
36 months | Leader: CTTC

T6.1: Impact Creation and Dissemination
01.01.2018 — 31.12.2020
36 months | Leader: CTTC

T6.2: Exploitation of results
01.06.2018 — 31.12.2020
31 months | Leader: ENG

T6.3: Standardization
01.01.2019 — 31.12.2020
24 months | Leader: SAG

Please note that the PERT chart is kept on task level for better readability.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

1.4   D3.8 updates respect to D3.2

The present document D3.8 is the final deliverable of task 3.2. D3.8 builds upon D3.2, which is a draft deliverable that has been previously delivered in the context of task 3.2. Therefore, it is important to highlight the differences between D3.8 and D3.2. That is the aim of this section. Next, we provide a bullet list with those differences:

- The executive summary has been updated to describe the new content and sections in D3.8.

- The table of acronyms has been updated.

- Section 2 has been updated, as in D3.2 was in a draft state.

- Sections 3.2.4 and 3.2.5 have been included in D3.8. These sections are new and deal with the dynamic instantiation of VNFs upon requests of SFC. This dynamic VNF instantiation is put in the context of the SEMIoTICS use case 2, i.e. ambient assisting living.

- A new section dealing with the NFV MANO implementation for SEMIoTICS has been included, this is section 4.2. More specifically, we explain how the NFVO and VNF manager have been implemented by means of the OSM. Also, how we implement the VIM by means of OpenStack. Moreover, we describe how we enable the functionality of building SFC, leveraging the OpenStack ecosystem. Finally, we explain how we gather metrics of the NFV resources. This requires telemetry services in OpenStack and its integration with OSM. Observe that telemetry services are of paramount importance to manage the NFV resources dynamically.

- We have added a new section regarding the interaction between the NFV MANO and external SEMIoTICS blocks such as the Pattern Orchestrator/Pattern Engine. This is section 4.3.3, and it describes the implementation of a northbound interface for the NFV MANO based on the REST API specified by ETSI in [5]. This northbound interface is of paramount importance, as permits the Pattern Orchestrator to control remotely NFV operations such as the instantiation of VNFs. Thereby, the Pattern Orchestrator can enforce patterns remotely.

- We have added a new section, which provides a real experiment on how to manage dynamically the NFV resources. Namely, this is section 4.5.2 and deals with the dynamic scale out of VNF instances.

- Regarding the previous bullet, when a scale out process happens, multiple instances of a VNF are available. Then, we need an entity that distributes the incoming traffic among the VNF instances. This is the role of the load balancer and it is described in the new section 4.5.3.

- Section 6 was updated, as in D3.2 it contained draft material and references to the future work in D3.8.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

2 TASK OBJECTIVES AND LINKS TO SEMIOTICS' REQUIREMENTS, KPIS AND ARCHITECTURE

2.1 Link with T2.3: SEMIoTICS' requirements in NFV

SEMIoTICS deliverable D2.3 [6] describes a set of requirements associated to the main SEMIoTICS's use cases (UC), to the generic SEMIoTICS platform and its layers. The aim of this section is to depict the role that an NFV platform has on achieving those requirements. Thereby, on the one hand in Table 1 we list the requirements from D2.3 where NFV is relevant. On the other hand, we add a column describing the NFV role associated to the corresponding requirements. Also, we add a column that specifies where these requirements are covered in the present deliverable.

TABLE 1 LINK TO SEMIOTICS' REQUIREMENTES DEFINED IN DELIVERABLE D2.3 [6]

Req-ID	Description	NFV role	Status of the Implementation	Where in D3.8 is covered
General Platform Requirements				
R.GP.2	Scalable infrastructure due to the fast-paced growth of IoT devices	Via VNF scaling out operations, precise placement and network slicing, NFV is able to provide flexibility in face of such requirements.	**Completed.**	Sections 1, 4 and 5.
R.GP.3	High adaptation capability to accommodate different QoS connectivity needs (e.g. low latency, reliable communication).	Via precise placement operations and allocation of virtual resources, NFV provides considerably lower delays or reliable communications e.g. by placing computation agents closer to where they are needed.	**Completed.**	Section 4.
Network layer and Backend/Cloud Layer Requirements				
R.NL.1/R.BC.1	Controller Node requirement: At least 6 CPU cores and 32 GB RAM	These nodes satisfy the hardware requirements of the NFV components.	**Completed.**	Section 6.1.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

R.NL.2/R.BC.2	Controller Node requirement: At least 2 Network interfaces	These nodes satisfy the hardware requirements of the NFV components.	**Completed.**	Section 6.1.
R.NL.3/R.BC.3	Controller Node Requirement: Linux OS	These nodes satisfy the hardware requirements of the NFV components.	**Completed.**	Section 6.1.
R.NL.4/R.BC.4	Controller Node Requirement: Solid State Disk (SSD) of at least 1 TB	These nodes satisfy the hardware requirements of the NFV components.	**Completed.**	Section 6.1.
R.NL.5/R.BC.5/ R.BC.6/ R.BC.7	Data paths / Hypervisor Nodes Requirement: At least 4 CPU cores and 8 GB RAM, at least 2, 1Gbps Network interfaces, Virtualization Extensions (Intel VT-x/AMD-V) must be supported by the Hypervisor CPU for hardware acceleration of VMs.	These nodes satisfy the hardware requirements of the NFV components.	**Completed.**	Section 6.1.
R.NL.6/R.BC.8/ R.BC.9	Data paths / Hypervisor Nodes: KVM and Linux Containers (LXD) must be supported by the Hypervisor Linux OS	These nodes satisfy the hardware requirements of the NFV components.	**Completed.**	Section 6.1.
R.NL.8/ R.BC.12	The VIM and Virtual Network frameworks must support Interfaces that enable VM tenant networking	Delegation of networking functions to SSC is possible through the corresponding interfaces SEMIoTICS' VIM (i.e. OpenStack), which has full multitenancy support.	**Completed**.	Section 5, 6.
R.NL.9/ R.BC.13	Interface between the VIM and the	Interfacing and delegation of	**Completed.**	Section 4 and 5.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

	SDN controller to allow VTN	virtual networking operations to external SDN Controllers is supported by SEMIoTICS VIM via the ML2 plugin interface.		
R.NL.10/ R.BC.14	Interfaces among the MANO and the VIM must ensure seamless interoperability among different entities of the Backend Cloud	MANO and VIM provide well-documented REST APIs, which agents trigger and make fully interoperable (e.g. parse the returned values) with entities in the Backend.	**Completed.**	Section 4.3.3, 5.
R.NL.11/ R.BC.15	Secure communication with the various Backend Cloud components (e.g., use of dedicated management network, appropriate Firewall rules), as well as the communication between VIM, SDN Controller, and MANO, with data paths acting as computing nodes for VNF spinoff.	Distributed compute nodes are used for VNF spinoff that enable data paths throughout the platform.	**Completed.**	Section 3.
		IoT Security and Privacy Requirements		
R.S.4	All components from gateway, via SDN Controller, to cloud platforms and their users MUST authenticate mutually.	Any interaction with the NFV Component must be done by an authorized party. Tokens/creden tials can be	**Completed.**	Section 5.1.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

		distributed to other relevant components (e.g. Global Pattern Orchestrator) for this purpose.		
UC2 SARA				
R.UC2.12	The SEMIoTICS platform SHOULD allow SARA components to delegate to the platform the computation of complex functions over the data received by field devices. These computations may result either in the generation of higher-level observation events (e.g. significant Patient events abstracted form sensor data) towards the ACS or in sensors configuration parameters (including actuators command). The SARA components MAY specify computations either as Dataflow or as Finite State Machine.	SEMIoTICS NFV platform is able to instantiate VNFs at precise locations of the SEMIoTICS infrastructure. Such VNFs represent the computation resources needed by this UC.	**Completed.**	Section 3 and 4.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

UC3 Sensing				
R.UC3.9	IoT Sensing gateway shall support 1 to many standard IP based (i.e. TCP transport) M2M communication protocol to interface a number N of connecting Sensing units (e.g. broadcast type).	The IoT Sensing gateway is either a VM, or a Docker container. Regardless, SEMIoTICS NFV component is able to orchestrate/build such V/C-NF (for Virtual or Container Network Function, respectively) with the specified communication requirements assuming network connectivity to the respective IoT Gateway is available from the NFV layer.	**Completed.**	Section 4 and 5.
R.UC3.12	IoT Sensing gateway shall be capable to run Linux (e.g. Ubuntu OS) and standard graphics and browser libraries.	Similar to R.UC3.9, NFVO is able to orchestrate a VNF with such requirements.	**Completed.**	Section 4.
R.UC3.13	IoT Sensing gateway should be able to support http and standard protocols for cloud interfacing.	The same explanation for R.UC3.12.	**Completed.**	Section 4.
R.UC3.14	The specific M2M protocol adopted on UC3 is based on MQTT. A MQTT broker service will be available to dispatch messages between the coordinating	SEMIoTICS NFV component is able to orchestrate a MQTT broker on demand at a precise	**Completed.**	Section 4.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

	Sensing gateway and its associated Sensing units.	location in the infrastructure.		

2.2 Link to project KPIs

KPI ID	Project KPI	NFV role
KPI-4.6	Development of new security mechanisms/controls	From an NFV perspective, SFC is leveraged to guarantee security procedures for each kind of traffic in UC2. This is done by concatenating different security enforcers (firewalls, Intrusion Detection Systems, Honeypots) and forcing traffic to travers them. As each element is configured with specific security rules according to the expected traffic, only authorized packets are expected to go through to the services' endpoints.
KPI-5.2	Service Function Chaining (SFC) of a minimum 3 VNFs	This KPI aims at the orchestration of SFC able to provide security by the chaining of at least 3 VNFs. That is, from a centralized position in the SEMIoTICS architecture, the SDN Controller and the NFV components should be able to build and configure the SFC for each kind of traffic. Evaluation is reflected in the ability to provide different QoS measures per tenant network (i.e. traffic type) in UC2.
KPI-6.1	Reduce manual interventions required for bootstrapping of smart object in each use case domain by at least 80%	The bootstrapping service involves e.g. computation agents, MQTT brokers, databases. An implementation of these functional blocks in the form of VNFs automates the service bootstrapping process. For UC3, the reduction on manual interventions is reflected in scaling operations. That is, when a specific VNF is overloaded with tasks, the NFV Orchestrator automatically triggers the scaling out of such component automatically. Therefore, such operations are expected to eliminate user intervention completely.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

2.3   Link with T2.4: SEMIoTICS' architecture

Task T2.4 deals with the design of the overall SEMIoTICS architecture. This task describes all the SEMIoTICS functional components as well as the interaction between them. Of course, the NFV platform, treated herein, is among the SEMIoTICS components. Therefore, it is important to highlight what is the role of NFV within the context of the SEMIoTICS architecture and what is the relation with other SEMIoTICS components.

The NFV component belongs, along with SDN component, to the so-called SDN/NFV orchestration layer. In general terms, NFV and SDN, provide SEMIoTICS with a flexible, dynamical, programmable and reconfigurable network. In terms of architecture, NFV is a vertical component that spans almost the whole SEMIoTICS platform. Namely, recall that NFV is composed of two main blocks, the NFV MANO that is the orchestrator of the whole NFV and the NFVI, which is the virtualized infrastructure that supports the virtualized functions, i.e. VNFs, by providing virtual computing, storage and networking resources.
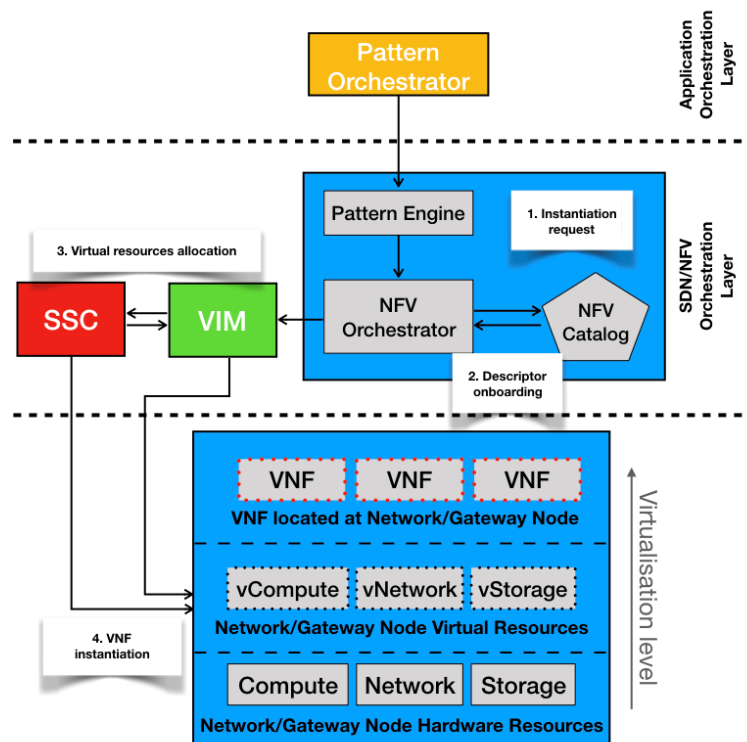
FIGURE 3 INTERACTION BETWEEN NFV AND THE SEMIoTICS COMPONENTS: VNF, SSC AND THE PATTERN ORCHESTRATOR.

The NFV MANO is typically deployed at the backend cloud, and the virtualized infrastructure, i.e. the NFVI, are compute nodes at the IoT GW, at the backend cloud or at the network that connects the IoT GW and the backend cloud. To be more specific, the virtualization of the IoT GW resources allow to deploy in a flexible and dynamical manner the functionalities of the SEMIoTICS IoT GW. E.g. VNFs can implement the "monitoring", "local thing directory" or "patter engine" at the IoT GW level as VNFs orchestrated by the NFV

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

MANO. These VNFs obviously, receive data from the field devices through standard network interfaces. The VNFs at the network between the IoT GW and the backend cloud can be virtual switches. Moreover, VNFs can be deployed at the backend cloud level. For instance, the SEMIoTICS component "GUI" is a VNF in use case 3. Therefore, NFV has an interface northbound with SEMIoTICS components, at the SEMIoTICS application orchestration layer, which is implemented via the *Os-Ma-Nfvo* endpoint (refer to Figure 1). In fact, in section 4.3.3 we provide the implementation of this northbound bound interface by means of REST APIs. It permits the interaction between the NFV MANO and the Pattern Orchestrator. Last, but not least, NFV has an interface westbound with the SEMIoTICS SDN Controller (SSC) through the "VIM connector" component. This interaction allows to provide data flow paths with the required networking resources among different points of presence, e.g. between the IoT GW, the backend cloud and the virtual switches that connect the IoT GW and the backend cloud.

In order to exemplify more clearly the interaction, that we described above, between the NFV MANO and the rest of NFV components we present Figure 3. Namely, on the one hand, we exemplify the interaction between the SEMIoTICS Pattern Orchestrator and the NFV MANO. The former requests the latter to instantiate VNFs in the NFVI with given KPIs. Moreover, we show how the NFV Orchestrator triggers the VIM to allocate virtual resources for the VNF instantiation at the possible points of presence of the NFVI, e.g. at the IoT GW or at a network node. In this regard, the VIM and SSC interact to allocate networking resources with the necessary QoS requirements. Finally, the VNFs embedding SEMIoTICS components are deployed on top of the NFVI leveraging virtual resources of it.

Also related to the last paragraph, in section 4.3 we provide a dynamic sequence diagram that exemplifies how any of the above mentioned VNFs, consisting of SEMIoTICS components, are instantiated in the NFVI upon the request of the Pattern Orchestrator. Moreover, the Pattern Orchestrator specifies, to the NFV MANO, the KPIs associated to the VNFs.

2.4   Validation: Task objectives, KPIs and D3.8

The following Table describes SEMIoTICS' objectives related to Task 3.2 and maps them to different sections on this deliverable.

TABLE 2 TASK OBJECTIVES

T3.2 Objectives	D3.8 Chapters and Observations
Orchestration Platform • MANO platform guaranteeing low latency, high reliability, security and privacy properties to NS.	1,3,4,5,6.1
Allowing dynamic reconfiguration of services • Appropriate interfaces to allow dynamic adaptation of network services, compatible with ETSI-NFV architecture.	4, 5
Implementation • Deployment of NFV infrastructure. • Realize virtual monitoring, caching, security and privacy as network functions.	3, 4, 6.1

Via this task, this deliverable D3.8 contributes to the satisfaction of Objective 4 (Development of core mechanisms for multi-layered embedded intelligence, IoT application adaptation, learning and evolution, and end-to-end security, privacy, accountability and user control), 5 (Development of IoT-aware programmable networking capabilities, based on adaptation and SDN orchestration), and 6 (Development of a reference prototype of the SEMIoTICS open architecture, demonstrated and evaluated in both IIoT (renewable energy) and IoT (healthcare), as well as in a horizontal use case bridging the two landscapes (smart sensing), and delivery of the respective open API). More precisely, through KPI-4.6, 5.2 and 6.1.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

3 VNFs AND SFCs FOR SECURITY, PRIVACY AND DEPENDABILITY IN SEMIOTICS

One of the main purposes of SEMIoTICS is to provide a secure networking infrastructure, via the associated proactive and reactive security mechanisms, such as the deployment of network security services and the continuous monitoring and intrusion detection. In order to achieve this objective, SEMIoTICS contains network monitoring functions, intrusion detection mechanisms for the identification of attacks, and run-time network adaptation for attack response and mitigation mechanisms.

Security in SEMIoTICS propose the creation of a reactive security framework. The framework includes the combination of various Security Functions, employing the flexibility of SDN/NFV and SFC. This reactive security framework can offer continuous monitoring of incoming traffic and detecting and adapting to different types of attacks.

Via this framework, security network functions such as Firewalls (FW), Intrusion Detection Systems (IDS), Deep Packet Inspection Systems (DPI), Honeypots (HP), HoneyNets and Load Balancers (LB), can create a number of function chains to forward traffic based on the type of running application. For the reason that these security functions could be dynamically instantiated, automatically deployed, and transparently inserted into the traffic flow, different security needs can be addressed for different profile types such as per tenant, per traffic and per application. More details appear in the subsections below.

3.1 VNFs for Security, Privacy and Dependability Mechanisms

Security, privacy and dependability mechanisms implemented as security services themselves are typically being deployed as monolithic platforms (often hardware-based), installed at fixed locations inside and/or at the edge of trust domains, and being rigid and static, often lacking automatic reconfiguration and customization capabilities. This approach, combined with the typical networks' architectural restrictions mentioned above, increase operational complexity, prohibit dynamic updates and impose significant (and often unnecessary) performance overheads, as each network packet must be processed by a series of predefined service functions, even when these are redundant.

The use of SDN/NFV and IoT in cross-domain setups can introduce new security, privacy and dependability (SPD) risks since the increased openness of IoT infrastructures makes SPD considerations more critical than ever before. The use of VNFs for SPD is an important aspect in SEMIoTICS, although the maintenance of SPD is necessary to meet regulatory and compliance requirements. This is increasingly challenging and potentially expensive in virtualized networks that can span across several locations, from data centers, remote points of presence, mobile base stations to customer premise locations. However, not all virtual networks are suitable to be centrally hosted for a variety of reasons, which can include latency, bandwidth and performance. The resulting framework could be very effective and practical for hosting various types of VNFs and changes the convention definition of a security perimeter.

3.1.1 SECURITY, PRIVACY AND DEPENDABILITY VNFs

VNFs can be used for various tasks related to security and privacy in secure industrial infrastructures, such as the SEMIoTICS use cases, and deployed as virtualized network service functions as proactive mechanisms able to provide SPD monitoring management.

A list of VNFs for proactive SPD property monitoring includes the following functions:
- **Intrusion Detection System (IDS) / Intrusion Prevention System (IPS)** - a service able to monitor traffic/system activities for suspicious activities or attack violations and prevent malicious attacks
- **Firewall** - a service or appliance running within a virtualized environment providing packet filtering
- **Deep Packet Inspection (DPI)** - a function for advanced packet filtering (data and header) running at the application layer of OSI reference model.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

- **Network Virtualization –** services that can use of Virtual eXtensible Local Area Network (VXLAN) to encapsulate MAC-based OSI layer 2 Ethernet frames within layer 4 UDP packets, brings the scalability and isolation benefits needed in virtualised computing environments.
- **Access Control Lists** – are services to route traffic to the appropriate isolated virtual networks and the corresponding security service functions.
- **Packet inspectors** – a service to detect malformed packets or malicious activity (IPFiX, DDoS).
- **Load-balancer** – is a service or a device that can distribute network or application traffic across several paths or servers. It is used to increase the capacity, reliability and efficient of the applications.
- **HoneyNet** – a set of functions (HoneyPots) emulating a production network deployment, able to attract and detect attacks, acting as a decoy or dummy target.

In addition to the inspection of packets using the DPI's function, VNFs can also modify data packets. For example, for protection of confidentiality of data, a VNF can implement an IPSec tunnel. The remote endpoint of the tunnel can either be another VNF in the network, or the security manager in the backend (cf. also the description of security manager in D2.4). Key distribution for IPSec is also facilitated by the security manager in the network and backend layers.

VNFs can also be used for privacy purposes: A VNF can anonymise or pseudonymise a data stream coming from a sensor. Thus, such VNFs requires information on the structure of the data stream to identify and to label the sensors' data. Based on this information, a VNF can replace the identifiers with pseudonyms. In addition, a dedicated VNF can reduce the granularity of sensor data to avoid traffic analysis from passive listening that can retrieve critical and private information structures. For instance, when a grid of sensors provides temperature values every 10ms over a large area, the VNF could reduce this granularity to one average hourly value over the whole area. Finally, other than the ones mentioned above, other SFCs could be included in a real deployment, such as load balancers, HTTP header enrichment functions, TCP optimisers, Resource Signalling, etc.

While most or all of these functionalities could also be achieved using traditional approaches towards network architectures and software development, using VNFs has the following particular advantages in the context of SEMIoTICS. Thus, VNFs in SDN/NFV, as important parts of 5G networking, provide promising combination leading to programmable connectivity, rapid service provisioning and service chaining and thus can help to reduce the CAPEX/OPEX in the control network infrastructure. Furthermore, by appropriately leveraging the flexibility of SDN/NFV-enabled networks in the context of the adopted security mechanisms, industrial infrastructures can not only match but also improve their security posture compared to the existing, traditional networking environments[3]. More specifically, for the pattern language, as described in D4.1, it is essential that properties for security and privacy can be monitored and enforced. In order to classify the SPD properties that each service function chain can satisfy, Table 3 depicts this correlation properties and functions. Thus, a pattern can check whether an information flow includes, e.g. a required VNF for anonymization. In addition, if a pattern determines that a certain property needs to be enforced, it can add a VNF for this purpose to the respective information flow.

---

[3] N. Petroulakis, T. Mahmoodi, V. Kulkarni, A. Roos, P. Vizarreta, K. Abbasik, X. Vilajosana, S. Spirou, A. Matsiuk, and E. Sakic. Virtuwind: Virtual and programmable industrial network prototype deployed in operational wind park, 2016.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

TABLE 3 SPD PROPERTIES IN SERVICE FUNCTIONS

Functions	Privacy		Security		Dependability
	Access Control	Confidentiality	Integrity	Availability	Reliability
Firewall	o			o	
IDS/IPS		o		o	o
DPI			o	o	
IPSec	o	o	o		
Load-balancer				o	o
HoneyPot/Net	o	o		o	

3.1.2 PROACTIVE MONITORING, INCIDENT DETECTION AND MITIGATION MECHANISMS

The preparation of an incident detection and response for the SEMIoTICS infrastructure contains a generic incident handling of a security framework for cyber-physical system. Additionally, the incident response, vulnerability and artefact handling include analysis, support and coordination. In the same way, the protection detection and response are a combination of monitoring and incident detection, mitigation and trace-back and audit mechanisms. Based on that, within the SEMIoTICS context, we investigate the insertion of specific VNFs for proactive monitoring, incident detection and mitigation. The SEMIoTICS security mechanisms can include continuous network monitoring and intrusion detection for identification of attacks and run-time network adaptation for attack response and mitigation mechanisms. That includes the implementation of the following proactive service functions:

- **Firewall** as a service or appliance runs within a virtualised environment providing packet filtering. Legacy firewalls (e.g. actual hardware appliances) can be also supported and can easily be integrated into the architecture. A software or hardware firewall (legacy firewall appliance already present in the industrial network) instance can be deployed on the SEMIoTICS framework to implement network perimeter security. The type of firewall, as well as its placement, is irrelevant in the context as it allows the use of any type of firewall, and for its placement in any place on an SDN network deployment.
- **IDS/IPS** can monitor traffic or system activities for suspicious activities or attack violations, also able to prevent malicious attacks if needed (in the case of IPS). More specifically, IDS/IPS instances should ensure that the most up-to-date rules are constantly active. A database for event monitoring is presented, while provisions are made to allow for future extensions to transmit relevant information to security backend (e.g. for more sophisticated pattern matching), complex configuration and scaling-out (a consequence of topological dependencies, especially when trying to ensure consistent ordering of service functions and/or when symmetric traffic flows are needed; this complexity also hinders scaling out the infrastructure).
- **DPI** can match the packet payloads against a set of predefined patterns. Extracting the DPI functionality and providing it as a common service function to various applications (combining and matching DPI patterns from different sources) can result in significant performance gains. SEMIoTICS employs the DPI function for monitor the unknown incoming traffic and assign it to the (sub-)set of security service functions intended for the corresponding traffic type.
- **HoneyPot** can react as a service to attract and detect attacks acting as a decoy or dummy target. Network-based honeypots can be used to detect attacks and malware because they can decoy deployment that can fool attackers into thinking they are hitting a real network whereas in the same time it is used to collect information about the attacker and attack method. **HoneyNet** can deploy a set of functions (Honeypots), emulating a production network deployment, able to attract and to detect attacks, acting as a decoy or dummy target.

23

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

The placement of exemplary security VNFs in the NFV architecture is depicted in Figure 4.
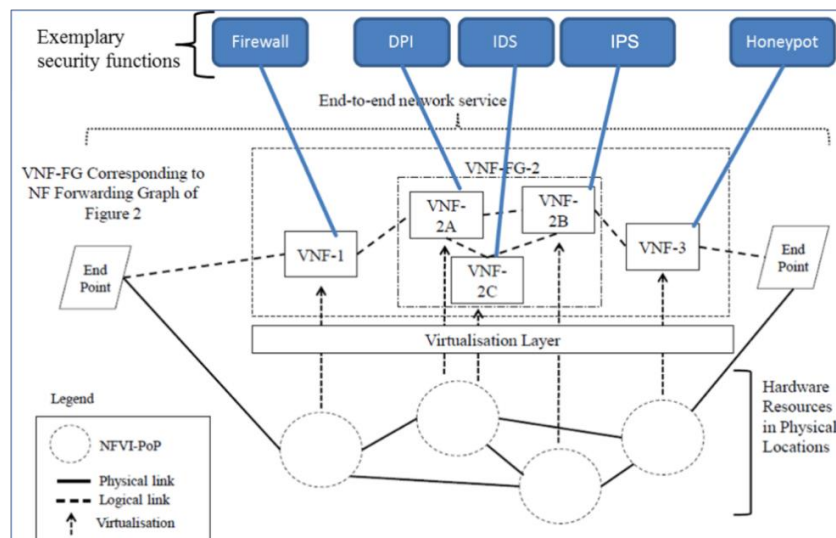
FIGURE 4 EXEMPLARY SECURITY FUNCTIONS IN NFV ARCHITECTURE

3.2 SFC for Security, Privacy and Dependability Mechanisms

In typical network deployments, the end-to-end traffic of various applications typically should go through several network services (e.g. firewalls, load-balancers, WAN accelerators). Furthermore, it can be referred to as Service Functions (SF) (or L4-L7 Services, or Network Functions, depending on the source/organisation) which are placed along its path. This traditional networking concept and the associated service deployments are characterised by a number of constraints and inefficiencies [7]:

- Topology constraints (network services are highly dependent on a specific network topology, which is hard to update).
- Complex configuration and scaling-out (a consequence of topological dependencies, especially when trying to ensure consistent ordering of service functions and/or when symmetric traffic flows are needed; this complexity also hinders scaling out the infrastructure).
- Constrained high availability (as alternative and/or redundant service functions must typically be placed on the same network location as the primary one).
- Inconsistent or inelastic service chains (network administrators have no consistent way to impose and verify the ordering of individual service functions, other than using strict topologies - on the other hand, these topology constraints necessitate that traffic goes through a rigid set of services functions, often imposing unnecessary capacity and latency costs, while changes to this service chain can introduce a significant administrative burden).
- Coarse policy enforcement (classification capabilities and the associated policy enforcements mechanisms are of coarse nature, e.g. using topology information).
- Coarse traffic selection criteria (as all traffic in a particular network segment typically has to traverse all the service functions along its path).

All the previous are exacerbated nowadays, with the ubiquitous use of virtual platforms, which necessitates the use of dynamic and flexible service environments. This is even more pronounced in service providers and/or cloud environments, with infrastructures spanning different domains and serving numerous tenants, each with their own requirements. Said tenants share a subset of the providers' service functions and require dynamic changes to traffic and service function routing, to follow updates to their policies (e.g. security) or Service Level Agreements.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

SFC aims to address these issues via a service-specific overlay that creates a service-oriented topology, on top of the existing network topology, thus providing service function interoperability [8]. An SDN-based SFC Architecture, such as the one defined by the Open Networking Foundation [9], can extend this concept, exploiting the flexibility and advanced capabilities of software defined networks, to provide innovative and comprehensive solutions for the above-stated presented weaknesses of the legacy networks.

3.2.1   SFC BACKGROUND

3.2.1.1   TERMS AND DEFINITIONS

The definitions of SFC terms are described in IETF [10]. Based on these descriptions, the used terms and definitions are listed below:

Network Service Function: A function that is responsible for specific treatment of received packets.

Service Function Chaining: A service function chain defines an ordered set of abstract service functions and ordering constraints that must be applied to packets and/or frames and/or flows selected as a result of classification.

Service Function Forwarder: A service function forwarder is responsible for forwarding traffic to one or more connected service functions according to information carried in the SFC encapsulation, as well as handling traffic coming back from the service function (legacy or virtual).

Service Function Path: The service function path is a constrained specification of where packets assigned to a certain route must go. Any overlay or underlay technology can be used to create service paths (VLAN, ECMP, GRE, VXLAN, etc.).

Service Function Classifier: An entity that classifies traffic flows for service chaining according to classification rules. The Classifier is responsible to classify and mark packets, based on the predefined ACL, with the corresponding SF Chain Identifier. It can be placed on a data path or run as an application on top of a network controller.

SFC Header: A header that is embedded into the flow packet by the SFC Classifier to facilitate the forwarding of flow packets along the service function chain path. This header also allows the transport of metadata to support various service chain related functionality.

Tenant: A tenant is one organization that is using SFC. A tenant uses SFC on one's own private infrastructure or on an infrastructure shared with other tenants.

Tenant's User Data Plane: The tenant uses SFC to provide service to its customers or users.

3.2.1.2   CONTROLLER COMPONENTS AND TEMPLATES

OpenDaylight (ODL) supports SFC[4] via the use of suitable templates (Service Functions, Service Function Forwarders, Service Function Classifiers, Service Function Chains and Access Control Lists) in the controller in JSON formats. The templates of SFC components as provide in the controller is defined in Table 4:

TABLE 4 SFC COMPONENTS AND JSON TEMPLATES

Service-nodes	Syntax
Service Function (SF)	"service-function": [{"name","ip-mgmt-address", "rest-uri","type", "nsh-aware", "sf-data-plane-locator": [{"name","port","ip","transport", "service-function-forwarder"}] }]
Service Function Forwarder (SFF)	"service-function-forwarder": [{"name","service-node", "service-function-forwarder-ovs:ovs-bridge": {"bridge-name"}, "sf-data-plane-locator": [{"name","port","ip","transport", "service-function-forwarder"}] }], "service-function-dictionary": [

---

[4] https://docs.opendaylight.org/en/stable-fluorine/user-guide/service-function-chaining.html

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

	{"name", "sff-sf-data-plane-locator": {"sf-dpl-name", "sff-dpl-name" }}]
Classifier	"service-function-classifier": [{"name","scl-service-function-forwarder": [{"name", "interface"}], "acl":{"name","type"}]
Service Function Chain	"service-function-chain": [{"name", "symmetric", "sfc-service-function": [{"name", "type"}, {"name", "type"}]
Service Function Path	"service-function-path": [{"name","service-chain-name","starting-index", "symmetric","context-metadata", "service-path-hop": [{"hop-number", "service-function-name" }]

3.2.2 SFC FOR LOW LATENCY, HIGH RELIABILITY, SECURITY AND PRIVACY

Security services are a prime example of traditional network service functions that can benefit from the adoption of SFC, especially in the context of SDN networks. Indeed, security functions such as Access Control List (ACL), Segment, Edge and Application Firewalls, Intrusion Detection and/or Intrusion Prevention systems IDS/IPS and DPI are some of the principal service functions considered by IETF when presenting SFC use cases pertaining to Data Centers [10] and Mobile Networks [11]. Said IETF studies consider several SFC use cases and highlight the numerous drawbacks of using traditional service provision methods when applying, among others, the security functions. The security services themselves are typically been deployed as monolithic platforms (often hardware-based), installed at fixed locations inside and/or at the edge of trust domains, and being rigid and static, often lacking automatic reconfiguration and customization capabilities. This approach, combined with the typical networks' architectural restrictions mentioned above, increase operational complexity, prohibit dynamic updates and impose significant (and often unnecessary) performance overheads, as each network packet must be processed by a series of predefined service functions, even when these are redundant [12].

A typical example of an important, and also ubiquitous, security-related function is DPI, whereby packet payloads are matched against a set of predefined patterns. DPI imposes a significant performance overhead, because of the pattern matching mechanisms that are at the core its operation, and thus largely unavoidable (motivating a wealth of research efforts focusing on improving their performance [13] [14]). Nevertheless, DPI, in one form or another, is part of many network (hardware or software) appliances and middleboxes; some examples can be seen in [15]. Thus, leveraging the benefits of SDN-based SFC deployments involves reversing this trend for monolithic, "all- in-one", security services, which are now commonplace. This is an approach, brought forward in part because of the advancements in hardware performance, which meant that a single, relatively affordable, hardware platform had enough resources to accomplish multiple tasks simultaneously. Instead, in the context of SFC, the focus is on breaking-up these complex services into dedicated service functions, each providing a single task.

Another SFC example, which is interesting for the SEMIoTICS purposes, is the one where low latency and reliability is needed. In this case, in the SARA UC, the humanoid robot (Pepper), from the SARA UC, needs to send a reliable live video stream with low latency to the SARA Web App located at the backend cloud. Thereby, the NFV network must support a chain of VNFs that forward the data flow from end-to-end with low latency regardless of the network impairments. To this end, the NFV MANO allocates the necessary communication and computing resources to guarantee the required QoS, i.e. it provides a network slice with low latency and reliability guarantees. This is possible thanks to the programmability and flexibility provided by the NFV framework.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
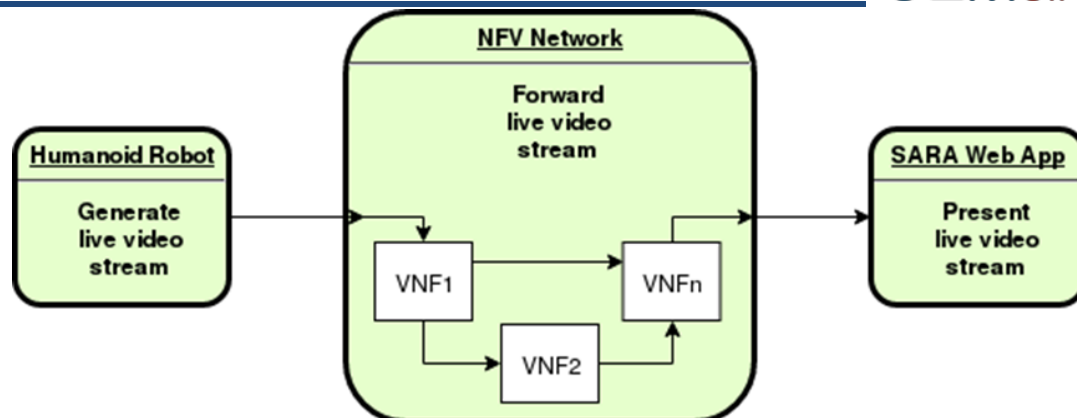
FIGURE 5 SFC EXAMPLE INVOLVING LIVE VIDEO STREAMING FOR SARA UC.

3.2.3 REACTIVE MONITORING AND NETWORK SECURITY INCIDENT MECHANISMS

Different from the proactive deployment of specific security mechanisms, that are setup and deployed before an attack takes place (typically at the network's design phase), the reactive mechanisms employed are able to react in real time to changes in the network as well as the traffic traversing said network, e.g. to automatically mitigate attacks, block malicious entities, route them to specific, dummy network components to allow for enhanced monitoring of their actions or even trigger the deployment of new security functions to help alleviate the effects of an ongoing attack.

The core part of the reactive security monitoring is based on the SFC framework and the previous described components and templates. That includes the definition of the service functions, the placement of functions in the forwarders and the classifiers that can classify the traffic (Figure 6). Moreover, the final stage of the definition includes the creation of the service function chains related also with the predefined ACLs (Figure 7).
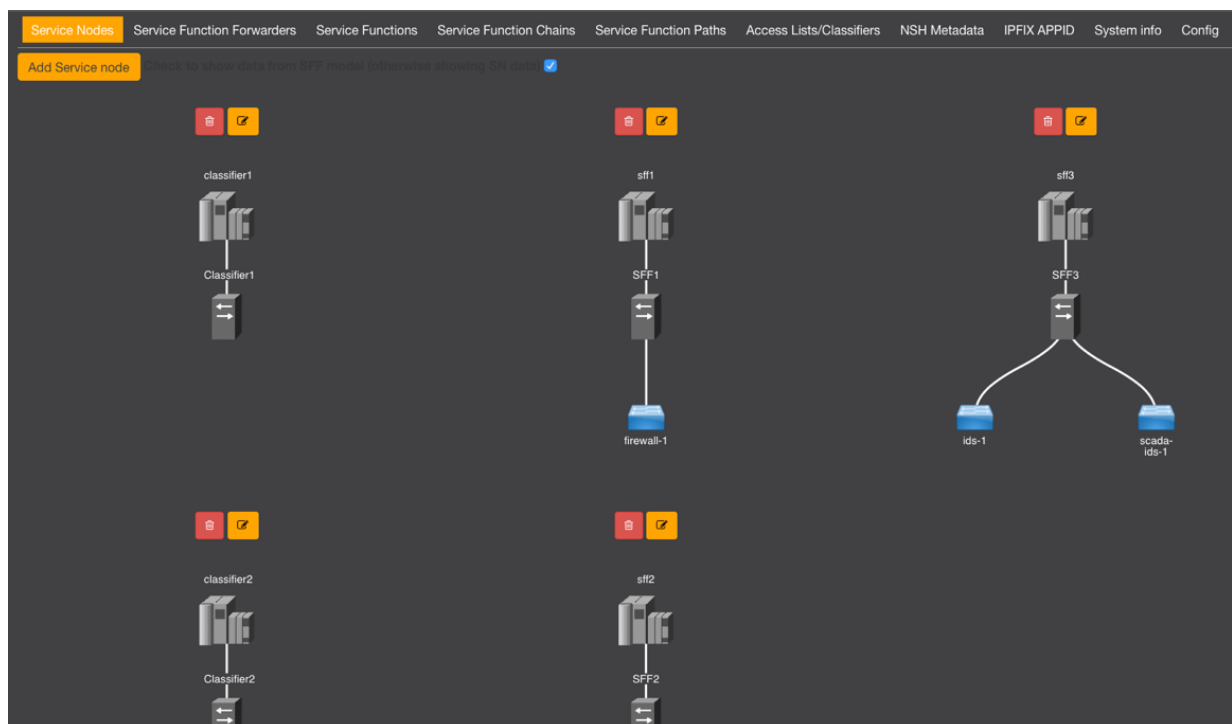
FIGURE 6 SFC SERVICE NODES

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
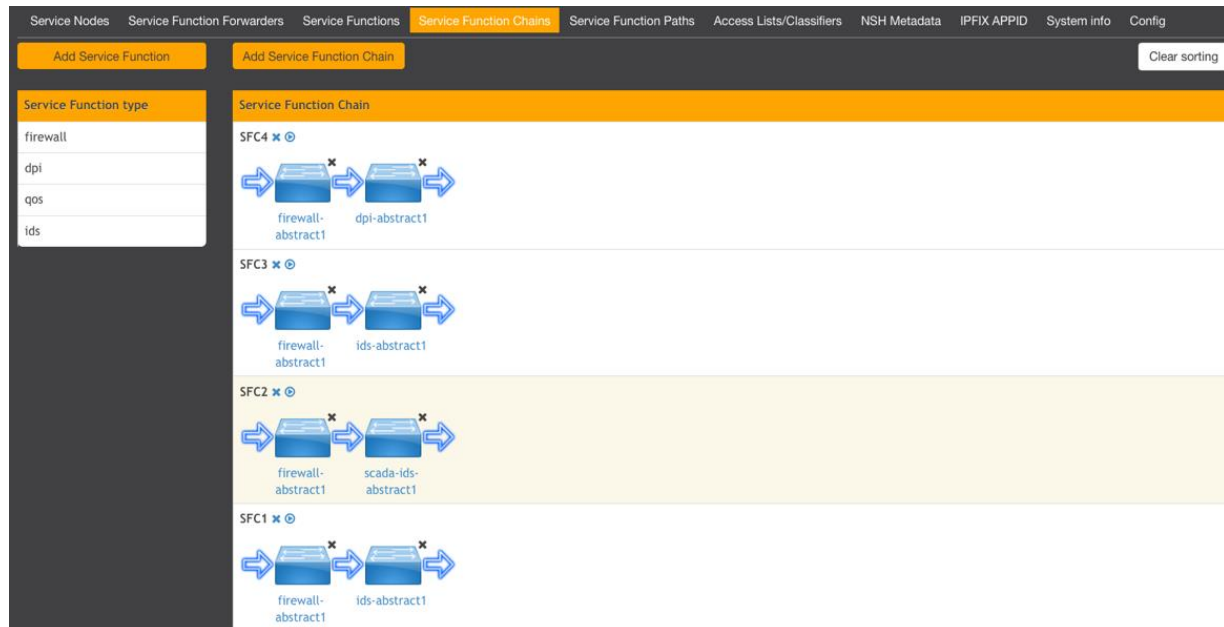
FIGURE 7 PREDEFINED SERVICE CHAINS OF SERVICE FUNCTIONS

By leveraging the flexibility of SDN-based deployments and the concept of SFC, a service-specific overlay creates a service-oriented topology, on top of the existing network topology, thus providing service function interoperability. The SFC provides the ability to define an ordered list of network services. The framework's SFs include the security functions proactively deployed. Whether the underlying network and the service functions are virtualised or not, is irrelevant from the perspective of the SFC. These services are then "stitched" together in the network to create a service chain allowing us to route unknown/suspicious traffic via the IDS and the DPI SFs, to classify it (as either legitimate or malicious) and to forward it accordingly. With this mechanism, malicious traffic can be isolated in the honeypot, allowing us to track the attacker, identify her purpose and keep her occupied. Using this scheme, the honeynet's effectiveness is enhanced, taking advantage of the SDN capabilities of dynamic network reconfigurations and traffic forwarding, and this is something that is exploited in the context of SEMIoTICS reactive security framework, to reroute malicious traffic to honeypots/honeynets instances. A typical example of the reactive security framework for the Wind Park use case is depicted in Figure 8. In this example, three different SFCs are defined to classify traffic in three different types, a legitimate, malicious and unknown one.
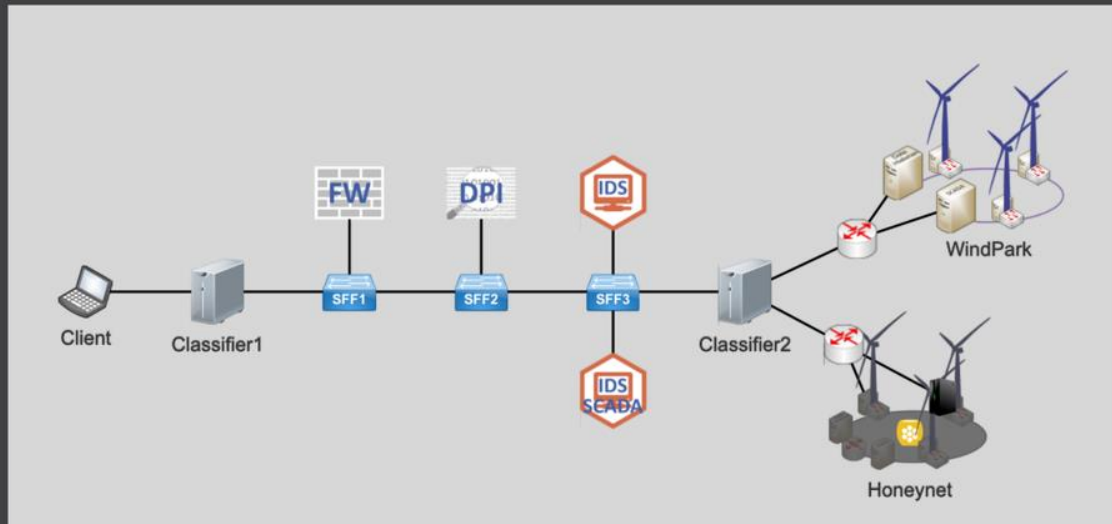
780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
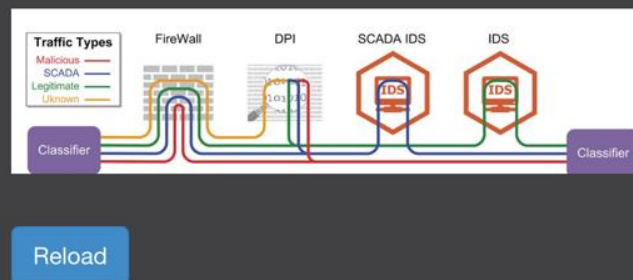
FIGURE 8 SFC EXAMPLE INVOLVING WIND PARK TRAFFIC CLASSIFICATION

3.2.4 DYNAMIC INSTANTIATION OF VNFS BASED ON SFC REQUESTS

In order to enhance the dynamic SFC instantiation in a network, the principles of NFV MANO can be combined with the framework presented here. The expanded architecture can be aligned to the approach described in ETSI GS NFV 002 [2]. This enhances the proposed framework with flexible deployment and instantiation of network functions and the automated preparation of service functions chains. For that reason, the SFC Manager can be enhanced to handle the interactions between the SDN controller and the MANO, to receive networking information about instantiated VNFs, as well as to provide information about possible service function chains.

One of the innovative approaches supported by this work, is the dynamic instantiation of SFCs based on the predefined SFC patterns. This can be applied based on the patterns as will be presented in D4.8. When there is a request for an SFC instantiation containing service functions, the depicted in Figure 9 procedure should be followed. If the SFC does not exist, the instantiation of the respective SFC is deployed through the identification of the requested VNFs. If the VNFs exist in the service nodes, the SFC is updated including these VNFs. If the VNFs do not exist, the service node with the available resources is requested to instantiate the respective VNFs. The procedure is ended when all the requested VNFs are included in the SFC.

29

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
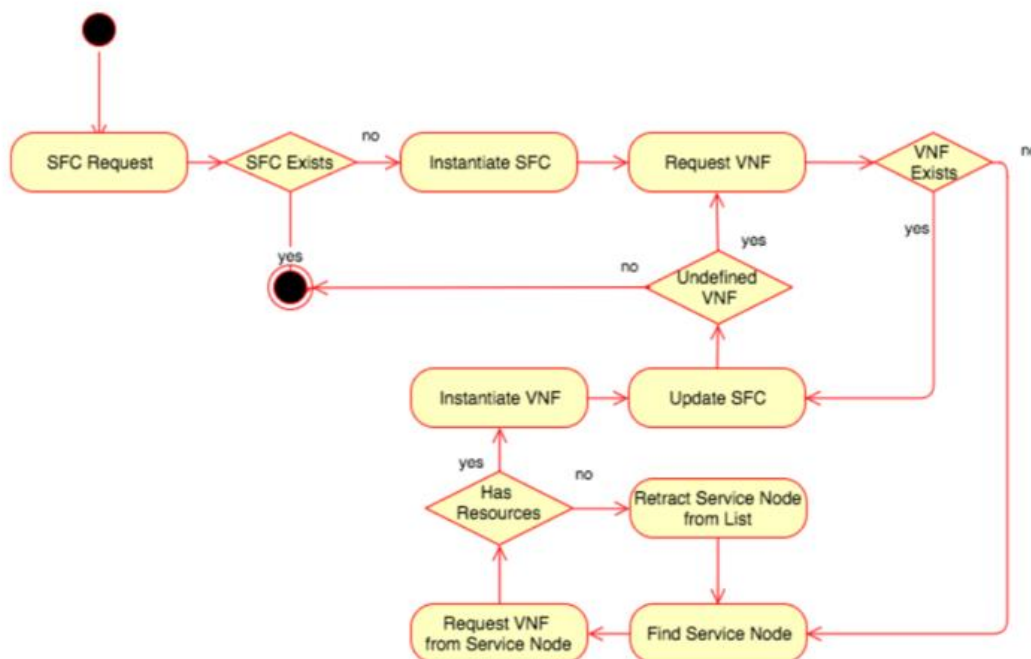
FIGURE 9 VNF INSTANTIATION BASED ON SFC REQUEST

The procedure of instantiation and the identification of the respective SFCs and the VNFs based on the patterns is depicted in the Figure 9. That can be based on the actual interaction between the components of the SEMIoTICS architecture. Pattern Orchestrator forwards a specific chain request to the Pattern Engine for forwarding the traffic between entities through a specific chain of functions. Pattern Engine forwards this request to the SFC manager which is located in the SDN controller responding to the Pattern Engine whether the chain exists or not. If the chain exists, then a respond of the chain satisfaction is returned to the Pattern Orchestrator. If the chain does not exist, then a requested is forwarded from the MANO requesting whether the service functions exist or not. If functions exist in the VIM, then the chain can be instantiated in the SFC Manager and a respond of the chain satisfaction is returned to the Pattern Orchestrator. If functions do not exist in the VIM then, a function instantiation request is forwarded to the NFV Orchestrator, which is responsible to instantiate them in the VIM. Then, the chain can be instantiated in the SFC Manager and a respond of the chain satisfaction is returned to the Pattern Orchestrator.

As initially described in D2.5 and in D5.2, in order to fulfil SFC request (`chain{vnf1,vnf2,..}`) the following procedures and abstract rest calls are presented. It is a four steps approach get or post `chain` and get or post `vnfs` as presented below.

Request: `chain{vnf1,vnf2,..}`

1) **Get service** `chains` **stored in the SDN controller.**

```
chains: GET(SDN_CONTROLLER_IP, 8181, "/restconf/config/service-function-
                chain:service-function-chains/")
```

2) **Verify if service** `chain(vnf1,vnf2,…)` **is included in the instantiated** `chains`.
   This can be done by use of the SFC pattern rules as enforced through the Pattern Orchestrator and the Pattern Engine at the backend. The following rules can be extended to patterns to support such verification.

30

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

a) **When** the `chain` exists,
   then the SFC request is satisfied.

b) **When** the `chain` is not included in the `chains`,
   then the existing `vnfs` should be gathered (next steps).

3) **Get network service instances (`vnfs`) that are running in the VIM.**
   NFV Mano can get the required information regarding the up and running network service instance.

```
vnfs: GET(MANO_IP, 9999, "/osm/nslcm/v1/ns_instances/")
```

4) **Verify if network services** {`vnf1,vnf2,…`} **of the** `chain` **are included in the** `vnfs`.

   a) **When** {`vnf1, vnf2,…`} are included in the `vnfs`,
      then a new `chain` can be instantiated and inserted in the SDN controller

```
chain: POST(SDN_CONTROLLER_IP, 8181, "/restconf/config/service-function-
              chain:service-function-chains/", chain)
```

   b)
      When {`vnf1, vnf2,..`} *are* not included in the `vnfs`,
      then all the requested {`vnf1, vnf2,..`} should be instantiated based on the existence of the `vnf_descriptions` {nsdId,nsName, nsDescriptions,vimAccountId}

   i) **Get Network Service Descriptors (`nsds`)**
      The first step includes the search on the available service descriptors.

```
nsds: GET(MANO_IP, 9999, "/osm/nslcm/v1/ns_ descriptors/")
```

   ii) **Create network service resource (`nsdID`)**
       The second step includes the creation of the network service.
```
nsdID: POST(MANO_IP, 9999, "/osm/nslcm/v1/ns_ instances/", vnf_instance )
```

   iii) **Instantiate network service instance (`nsi`)**
        The third step includes the actual instantiation of the network service as a `vnf` to be included in the `chain`.
```
nsi: POST(MANO_IP, 9999, "/osm/nslcm/v1/ns_instances/nsdId/instantiate/",
                          vnf_description)
```

And since the required {`vnf1, vnf2,..`} will exist, the chain can be instantiated (step 4a).

3.2.5 DYNAMIC SFC INSTANTIATION IN THE AMBIENT ASSISTING LIVING USE CASE

The second use case of SEMIoTICS focuses on an ambient-assisted living scenario in a smart home environment, for the well-being and independent living of the elderly. In this context, a full implementation of the SFC-based security framework is presented featuring various services and operational security service functions in the following:

i. *Body Area Network (BAN)*. Short-range network of wearables (e.g. sensors and identification tags carried or worn on the patient's person) for fall detection, fall risk assessment, and other mobility-related data.
ii. *Robotic Rollator*. A powered, wheeled walking frame, primarily used for physical support, but also equipped with various sensors and computational units, and capable of identifying a patient (the user of the rollator) and monitoring their behavior (e.g. gait & posture).

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

iii. *Mobile Phone*. User's mobile phone that acts a gateway for the BAN devices, as well as the Robotic Rollator devices (but only in case of outdoors use).
iv. *Smart Home Infrastructure*. Sensors, actuators, lighting, climate control, and other smart devices, as well as the corresponding gateway(s), that comprise a smart living environment.
v. *Robotic Assistant*. A robotic component (in our case a Humanoid Robot) for monitoring a patient's activities (ADL data), health status, and treatment/training progress, as well as for supporting cognitive skills training, notifying/reminding the patient of upcoming treatments (e.g. medication & training schedules) and visits.
vi. *Backend*. The backend system providing an assortment of assistance services for the elderly and being monitored by caregivers and healthcare professionals.

Considering the above, there is significant motivation to leverage the flexibility provided by SFC to define specific service chains for each type of traffic. By applying the previous described procedure of chain instantiation, the legacy SARA use case can be extended to support traffic forwarding through specific service functions. That includes the traffic forwarding for the different type of traffic exchanged between the different actors as following:

▪ *Chain 1 – Mobile Phone:* Firewall -> DPI -> IDS -> Output.
▪ *Chain 2 – Robotic Rolator:* Firewall -> IDS -> Load Balancer -> Output.
▪ *Chain 3 – Smart Home:* Firewall -> IDS -> Output.
▪ *Chain 4 – Robotic Assistant:* Firewall -> Load Balancer -> Output.
▪ *Chain 5 - Malicious:* Firewall -> Honeypot.

The above scenario sketches a complex environment, requiring support for integration of heterogeneous devices and communication protocols, high degrees of interoperability, and support for distributed services and applications (each with its own set of intrinsic requirements), while guaranteeing the safety of the patient and the security and privacy of her patient data. This use case is visualized in Figure 10, which depicts the various types of devices, their interactions, and the involved communication technologies.
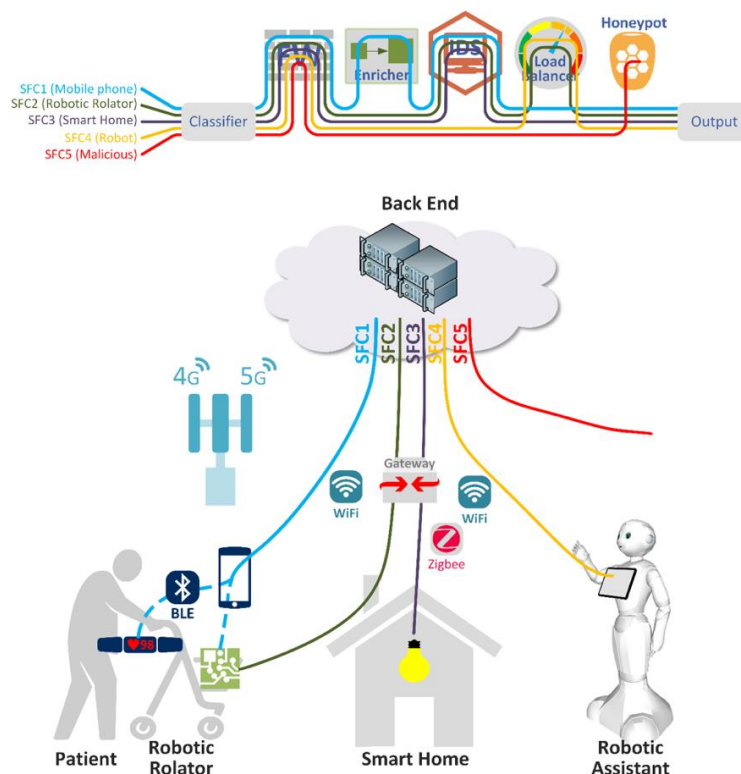
FIGURE 10 AMBIENT-ASSISTED LIVIND SCENARIO AND TRAFFIC CLASSIFICATION

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

The instantiation of a sequence of functions can constitute a service chain. Similar to the insertion of service functions in the SFC manager through the exposed service function REST interface, service chains can be inserted. In the list of the service functions, the firewall, the DPI, the IDS and the Load Balancer have been defined as the most crucial ones to enable the SPDI properties required by each chain to guarantee. Each VNF has a unique IP address which is required for the configuration and integration with the other functions interacting also with the use case devices and apps. Pattern Orchestrator is responsible to forward the SFC request to the Pattern Engine in order to verify or instantiate SFC requests and insert them in the SFC Manager as presented in Figure 11.
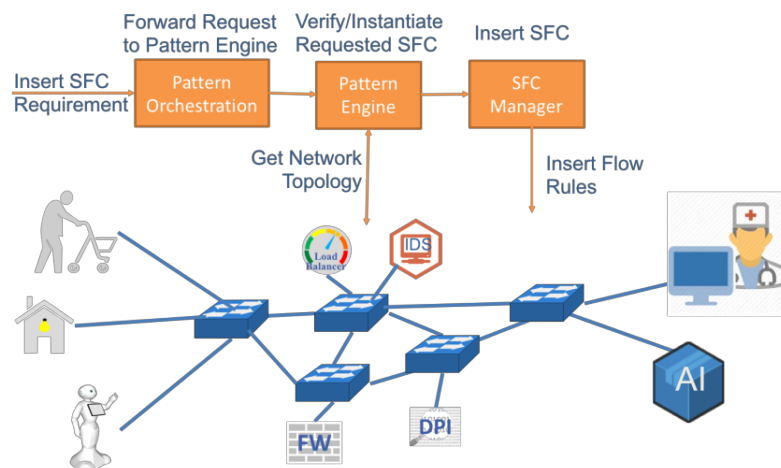
FIGURE 11 INSTANTIATION OF VNFS AND SFC

Following the procedure presented in the previous subsection, the request for the `SFC1=chain{firewall, dpi, ids)` includes the instantiation of the `chains` and the respective `VNFs` in the SFC Manager by the use of the NFV Orchestrator. The JSON output for the insertion of the chain in the SFC manager is presented in Figure 12.

```
 1 ▼ {
 2 ▼     "service-function-chain": [
 3 ▼         {
 4             "name": "SFC1",
 5             "symmetric": "true",
 6 ▼         "sfc-service-function": [
 7 ▼             {
 8                 "name": "firewall-abstract1",
 9                 "type": "firewall"
10             },
11 ▼             {
12                 "name": "dpi-abstract1",
13                 "type": "dpi"
14             },
15 ▼             {
16                 "name": "ids-abstract1",
17                 "type": "ids"
18             }
19         ]
20         }
21     ]
22 }
```

FIGURE 12 JSON OF THE SERVICE FUNCTION CHAIN CONTAINING FIREWALL, DPI AND IDS

Following the procedure and SFC requests (presented in Figure 9), the topology and the configuration of the assisting living use case can be based on the dynamic instantiation and insertion of service functions and chains. Compared to the statically configuration of wind park use case presented in Figure 8, the instantiation of the ambient assisting living aims to be completed dynamically as presented in Figure 13.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
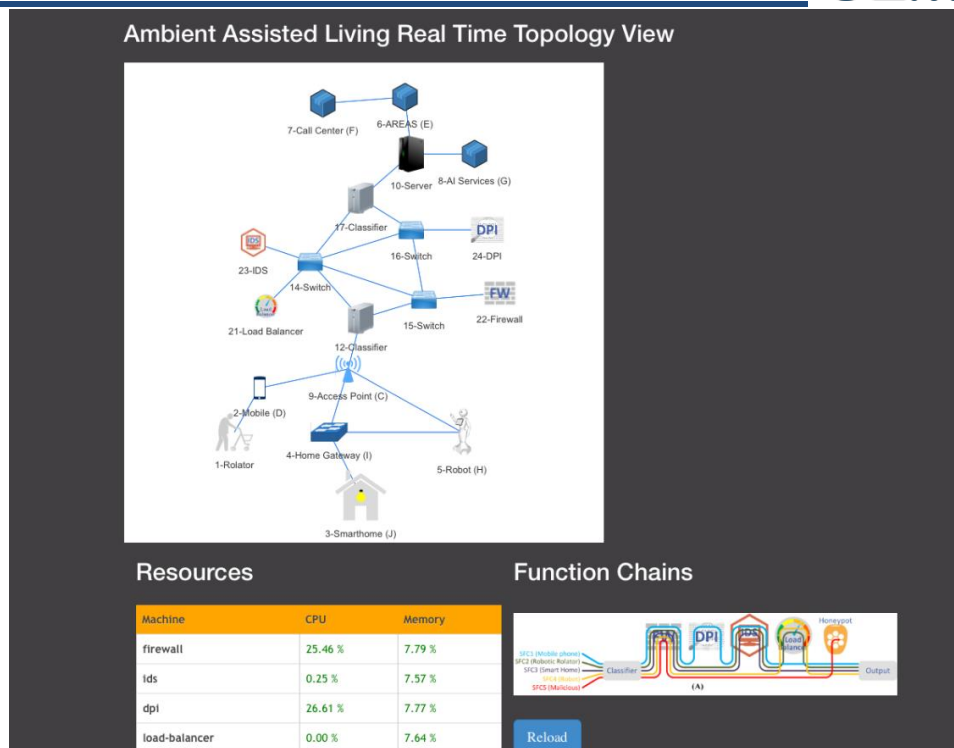
FIGURE 13 SFC EXAMPLE INVOLVING SARA USE CASE TRAFFIC CLASSIFICATION

A more detailed technical description of the network descriptor instantiation will be presented in the next section.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

4 NFV MANAGEMENT AND ORCHESTRATION FOR SEMIoTICS

In legacy networks, Network Functions (NF) implementations are tightly related to the hardware they run on top of it. That is, NF such as routing or switching are implemented by routers or switches, respectively, and so forth. NFV breaks such coupling via virtualization technologies [4]. That is, by implementing NF as software on top of a pool of fairly generic hardware resources (i.e. data center), it is possible to provide VNF which could be effectively re-instantiated, scaled or replaced in a very short time and reduce CAPEX/OPEX when compared with PNF.

Virtualization carries new challenges to the traditional network management as well as new entities and relationships among them. For instance, a virtual Network Service (NS) is usually composed of various VNF/PNF connected together in what is referred to as an SFC, specified in Virtual Network Function Forwarding Graphs (VNFFG) descriptors. The emergence of these new software elements, namely, VNFs, Virtual Links (VL), VNFFG, and their relationship with PNFs in a decoupled NFVI is handled by the NFV MANO framework.

4.1 NFV MANO functional blocks

The creation, instantiation, updating, and termination of NS is a new concept in networking, requiring the definition of new reference points (e.g. interfaces), functionality and entities. Moreover, the management of existing physical resources for virtualization, assignment of virtual resources to VNFs, lifecycle management of each VNF, and the realization of NS across a distributed set of physical resources impose new challenges to traditional networking. Efforts towards standardization in this regard have yielded ETSI's NFVI, which include the VIM and the NFV MANO framework (see Figure 1).

The aforementioned components of the NFVI are to be described here, as well as the interaction among them to orchestrate NS and the role they play within the SEMIoTICS framework.

4.1.1 VIRTUALIZED INFRASTRUCTURE MANAGER

NFVI defines two Administrative Domains [4] namely the Infrastructure and Tenant domains. The former contemplates the physical infrastructure upon which virtualization is performed, and therefore application agnostic; while the latter makes use of virtualized resources to spawn VNFs and create NS. Unlike resource allocation in other virtualized environments, in NFVI requests simultaneously ask for compute, storage and network resources. Moreover, NS could be composed of VNFs with hardware affinity/anti-affinity or require specific latency/bandwidth constraints in virtual links connecting VNFs. Such demands occur dynamically, allocating or freeing resources that could then be used for other NS, e.g. scaling up VNF's computing rate.

A VIM lies in the Infrastructure Domain. It takes care of abstracting the physical resources of the NFVI and making them available as virtual resources for VNFs. This is achieved through the reference point *Nf-Vi,* which interconnects the VIM and NFVI (see Figure 1). It allows the VIM to acknowledge the physical infrastructure (compute, storage) as well as enabling communication with network controllers (e.g. SDN Controllers) to provide virtual network resources to NS. Even-though VIMs could well control all resources of the NFVI (compute, storage and network), they could also be specialized in handling only a certain type of NFVI resource (e.g. compute-only, storage-only, network-only) [4].

Beyond the already-mentioned, functions carried on by the VIM are the following:
- Orchestrate requests made to the NFVI from higher layers (NFVO), e.g. allocation/update/release/reclamation of resources.
- Keep an inventory of allocated virtual resources to physical resources.
- Ensure network/traffic control by maintaining virtual network assets, e.g. virtual links, networks, subnets, ports.
- Management of VNF-FG by guaranteeing their compute, storage and network requirements.
- Management and reporting of virtualized resources utilization, capacity, and density (e.g. virtualized to physical resources ratio).

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

- Management of software resources (such as hypervisors and images), as well as discovery of capabilities of such resources.

As detailed in [4] other relevant VIM responsibilities within the NFVI network are:
- Provide "Network as a Service" northbound interface to the NFVO (realized via the ***Or-Vi*** reference point, see Figure 1).
- Abstract the various southbound interfaces (SBI) and network overlays mechanisms exposed by the NFVI network.
- Invoke SBI mechanisms of the underlying NFVI network.
- Establish connectivity by directly configuring forwarding instructions to network VNFs (e.g. vSwitches), or other VNFs not in the domain of an external network controller.

These compose the network controller part of the VIM. Nevertheless, and as mentioned previously, the required network abstractions mechanisms and management can be left to an external network controller, which feeds of NFVI information via the defined reference points (***Nf-Vi***, see Figure 1). It is reasonable to assume the VIM as key part of the NFVI. Being the only NFV component interfacing with the physical infrastructure it exposes open and comprehensible APIs to higher layers, i.e. NFVO, so functions could trigger them to get relevant information from the physical as well as the virtualized infrastructure, and trigger actions upon such information, e.g. create a NS with the necessary resources.

In the SEMIoTICS framework, the physical NFVI is able to support virtualization as realised by the VIM. This allows the NFVO to instantiate VNFs subject to the available compute and storage resources, as well as interconnect such VNFs together via an external SEMIoTICS SDN controller. The following subsections describe relevant Northbound Interfaces (NBI) or APIs usually exposed by VIMs, i.e. OpenStack, which are used by the Resource Orchestration (RO) function in the NFVO in order to assist the creation of NS by satisfying the requirements of the SEMIoTICS use cases (UC).

4.1.1.1 COMPUTE

Compute services at the VIM not only are in charge of creating virtual servers (or containers) on top of physical machines, but also to provision bare metal nodes. In the case of OpenStack this is achieved by means of projects such as Ironic [16]. The compute API for OpenStack is provided through the project Nova [17]. It provides "*scalable, on demand, self-service access to compute resources*" through RESTful HTTP endpoints that can be triggered by any authorized entity. All content sent or received from the Compute API endpoints are in JavaScript Object Notation (JSON) format. As it is a text-based type, it allows developers to employ a wide range of tools to reach such APIs, easing automation.

The following is a non-exhaustive list of concepts related to the Compute service as well as the information they provide or actions they are able to execute through the corresponding API for SEMIoTICS [17]:

- **Hosts**: physical machines that provide enough resources to spawn a Server. In SEMIoTICS, hosts conform the set of field level, network, and backend devices that together compose the NFVI. For instance, IoT Gateways at field level are assumed to provide enough compute resources to host VNFs realising local smart behaviour. Similarly, network level devices support VNFs for forwarding/routing/firewalling data to and from upper layers; and finally, backend/cloud servers have enough resources to host a wide variety of VNFs, e.g.: SCADA, Web applications and servers.
- **Server**: a virtual machine (VM) instance. In NFV it is often assumed that VNFs reside inside VMs or other type of virtualization container, such as LXC [18]. Some of the server status and actions reachable through the Compute API [19]:
  - Status: ACTIVE, BUILD, DELETED, ERROR, SHUTOFF, SUSPENDED, among others.
  - Actions: Start/Stop, Reboot, Resize, Pause/Unpause, Suspend/Resume, Snapshot, Delete/Restore, Migrate/Live Migrate, among others.
    - Migration and live migration relate to moving the Server to another Host. Live Migration performs this action without powering off the Server, avoiding downtime.

The ability to read the current status of Server and modify it, opens the way for dynamic (re)allocation of resources, specifically relevant as performance metrics from the underlying NFVI change in time.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

For SEMIoTICS this is of paramount importance, as it paves the way to optimize the end-to-end performance of network services in terms of e.g. latency or reliability.

- **Hypervisor**: the piece of computer software that creates and runs VMs. Hosts in each layer of the SEMIoTICS framework run a Hypervisor, which can be queried via the Compute API in order to obtain information regarding the Server, e.g. CPU, memory or other configuration.
- **Flavour**: virtual hardware configuration requested for a given Server, i.e. disk space, memory, vCPUs. Such configurations are onboarded prior to deployment, quantising the scaling factor of Servers e.g.: flavour small (1 vCPU), flavour medium (2 vCPUs), flavour big (4 vCPUs).
- **Image**: a collection of files used to create a Server, i.e. OS images. For SEMIoTICS, each UC component is assumed to run a preconfigured image tailored to its role, i.e. VNF. Such images are uploaded to the VIM for instantiation or passed as parameters to NFVO at orchestration time.
- **Volume**: a block storage device the Compute service could use as a permanent storage for a given Server.
- **Quotas and Limits**: upper bound on the resources a tenant could consume for the creation of Servers. SEMIoTICS employs such functionality to enforce an efficient sharing of the NFVI resources among the different UC.
- **Availability zones**: a grouping of host machines that can be used to control where a new server is created. As different SEMIoTICS UC require the placement of Servers at specific Hosts, this VIM capability allows the NFVO to orchestrate VNFs at precisely the right physical locations in the NFVI.

4.1.1.2 NETWORKING

VIMs are responsible for building virtual network overlays connecting VNFs, but also should expose or relay such information to other components. For instance, if an external network controller is assigned the task of managing connectivity between virtual endpoints, as in the case with the SEMIoTICS SDN Controller (SSC), the VIM should expose API endpoints where the necessary network information can be retrieved or modified. Furthermore, in the presence of a NFVO, Network as a Service (NaaS) APIs are expected.

OpenStack Neutron Networking [20] provides the virtual networking resources commonly expected in NFVI, such as L2/L3 networking, security, resource management, QoS, virtual private networks (VPN), virtual tenant networks (VTN), among others [21]. To configure such functionality or to retrieve logging information, functions are exposed through a set of RESTful HTTP APIs in JSON format. The following shows a non-exhaustive list providing a description of the functionality exposed through the Networking API (as shown in [21]).

- **L2 Networking**
  - Networks: list, shows details for, creates, updates and deletes networks. It provides a wide range of extensions capable of configuring several aspects of L2 networking, such as: network availability zones, port security, definition of QoS policies, VLAN trunks, among others.
  - Ports: list, shows details for, creates, updates and deletes ports. Ports are associated with Servers (VMs). They expose a similar set of extensions than the "Networks" mentioned above.
- **L3 Networking**
  - Addresses: list, shows details for, updates and deletes address scopes. Deals with the reservation of IPv4 addresses for Servers (Floating IPs), port forwarding, among others.
  - Routers: when enabled, it allows the forwarding of packets across internal subnets and applying NAT, so they can reach external networks through the appropriate gateway. Routers can be realized in a distributed manner (spanning all compute nodes of the NFVI) or using Router availability zones.
  - Subnets: lists, creates, shows details for, updates, and deletes subnet or subnet pools.
- **Security**
  - Firewall as a Service (FWaaS): applies firewall rules to ingoing or outgoing traffic, creates and manages an ordered collections of firewall rules.
  - Security groups: lists, creates, shows information for, updates and deletes security groups. Such groups are used to classify types of traffic, allowing or prohibiting certain kind of network traffic through a set of predefined, but also user-defined rules.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

- o <u>VPN as a Service (VPNaaS)</u>: enables tenants to extend their private networks across the public network infrastructure. Provided functionality includes:
  - ▪ Site-to-Site VPN.
  - ▪ IPSec using several types of encryption algorithms.
  - ▪ Tunnel or transport mode encapsulation.
  - ▪ Dead Peer Detection (DPD).
- **Others**
  - o QoS bandwidth limiting rules.
    - ▪ With the ability to distinguish between egress or ingress traffic.
  - o QoS Minimum bandwidth rules.
  - o QoS Differentiated Service Code Point (DSCP).
  - o Logging resources.
  - o DHCP servers.

SEMIoTICS falls within the particular case where the delegation of NFVI networking control may be relayed to an external SEMIoTICS SDN Controller. For such cases, Neutron exposes control tools via the Modular Layer 2 (ML2) north-bound plug-in [22]. This way, external controllers could manage the network flows traversing the NFVI via southbound interfaces, such as OVSDB.

4.1.1.3 STORAGE

Block storage is common place in virtual environments. Such type of storage can be though similar to USB drives: you can attach one to a compute Server (VM), and then detach it when turning the Server off or destroying it. Particularly interesting is the fact that in a NFVI the storage and compute Hosts are separate. Despite such separation of physical hardware, VMs are exposed to users as if they were running on top of a single Node thanks to the virtual networking resources used by the VIM; allowing the NFVI to grow to massive scales, e.g. server farms.

VIMs such as OpenStack manage block storage through the Cinder project. As concisely put in [23]: "*It virtualizes the management of block storage devices and provides end users with a self-service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device*". A non-exhaustive list of functionalities realised through the Storage API is shown below:

- Create, list, update, or delete volumes.
- Read volumes statuses:
  - o Among such statuses are: creating, available, reserved, attaching, detaching, in-use, maintenance, deleting, error, backing-up, among others [23].
- Modify a volume:
  - o Extend size, reset statuses, set metadata, attach/detach.
- Management of volumes: create or list volumes.
- Volume snapshots: creates point-in-time copies of the data.
- Volume transfer: transfer a volume from one user to another.
- Backups: full copy of a volume to an external service, as well as the restoration from such backup.
- Snapshots and Group Snapshots.
- Quotas and Limits: per tenant quotas and limits on storage resource allocation.

In general, the SEMIoTICS UC require an NS, as the data generated by the field devices is transmitted to the IoT Gateways or the backend cloud, where they are consumed by the IoT applications. Each NS is the composition of a set of VNFs, which run within VMs with specific compute and storage resources and are connected in a predefined manner with network resources. Thereby, the proper allocation of computing, communication and storage resources, to run the chain of VNFs at the corresponding VMs is fundamental to guarantee the desired performance of SEMIoTICS use cases. Namely, these performance metrics are related to e.g. latency or reliability. Therefore, compute, networking and storage resources are allocated by the VIM to deploy the chain of VNFs that compose the NS according to the requests made through the corresponding APIs.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

All in all, SEMIoTICS UC can be considered complex NS, mostly due to their specific requirements, e.g. Host affinity/anti-affinity (e.g. smart behaviour VNFs at specific IoT gateways), specific bandwidth/delay requirements between VNF links, firewalls at the backend/cloud, and/or others. Such specifications are collected in NS descriptors (NSd), which in turn are composed of VNF descriptors (VNFd), and VNFFG descriptors (VNFFGd) that realize Service Function Chains (SFC) according to the specifications contained in their respective descriptors. It is then the task of the NFVO to store/maintain such descriptors and interface with the VIM to realise the NS/VNF/VNF-FG therein.

4.1.2 FUNCTIONAL ARCHITECTURE OF THE NFV ORCHESTRATOR

SEMIoTICS NFV MANO framework is composed of a VIM, VNF Manager (VNFM), and NFVO (see Figure 1). This section deals with the functional description of the NFVO, particularly, the Network Service and Resource Orchestration functions, and the related Information Models (IM) that help spawn NS.

Management and Orchestration of VNF relates to providing each VNF with the NFVI resources they need[5]. But also, other aspects such as registering available VNFs or NS, scaling in/out each VNF according to policies or load, lifecycle management, snapshots, modifying the network interconnection among VNFs, modifying the VNFs in a VNFFG, creation and termination of NS. These are potentially complex tasks, primarily because VNF's NFVI resource requirements and constrains need to be satisfied simultaneously on top of a very dynamic environment (VNFs are instantiated or terminated, changing the pool of available resources). To leverage this, the NFV MANO (VIM+VNFM+NFVO) should expose services that support accessing these resources, preferably using standard APIs [4]. The NFVO performs two main functions, called Network Service and Resource Orchestration functions (NSO and RO, respectively). Capabilities of each function are exposed via standard interfaces consumed by other elements of the NFV MANO.

4.1.2.1 NETWORK SERVICE AND RESOURCE ORCHESTRATION FUNCTIONS

As suggested by its name, NSO function handles the registration (onboarding), creation, modification and termination of network services. The following non-exhaustive list gathers some of the functionality performed by the NFVO employing the NSO function:

- Checks that VNF or NS descriptors include all mandatory information for onboarding.
- Through VIM's exposed services, NSO checks that the software images specified in the descriptors are available at the targeted VIM.
- NS lifecycle management, that is: instantiation, update, scaling, event collection and correlation, and termination.
- Collects performance metrics from NS.
- Management of the instantiation of VNFs (alongside VNFM).
- Validation and authorization of NFVI requests from VNFM.
- Management of the relationship between NS instances and VNF instances.
- NS automation management based on triggers specified in the NS descriptors.

On the other hand, the RSO function interfaces with the NFVI to make sure resources are available for the instantiation of VNF/NS. The following non-exhaustive list gathers some of the services provided by the RSO function:

- Validation and authorization of NFVI requests from VNFM.
- NFVI resource management (distribution, reservation and allocation) by maintaining a NFVI repository.
- Leverages resource utilization information gathered from VIMs to manage the relationship between VNF instances and NFVI resources.
- Policy management and enforcement, e.g.: NFVI resource access control, affinity/anti-affinity rules, resource usage, among others.
- Collects usage information of NFVI resources by VNF instances.

---

[5] NFVI resources are those that can be consumed by virtualization containers, such as compute (CPU, virtual machines, bare metal hosts, memory), storage (volumes of storage), and network (networks, subnets, ports, addresses, forwarding rules, links).

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

4.1.2.2 NFVO DESCRIPTORS, NS ONBOARDING AND INSTANTIATION

Apart from APIs exposed by VIMs (which are triggered through the `Or-Vi` reference point, see Figure 1), descriptors are a main element in the instantiation of NS. In them, administrators specify details about VNFs, as well as VL, VNFFG, and the NS as a whole (even PNFs). All descriptors should be onboarded to the NFVO in order for the NSO function to verify them (e.g.: checking the validity of all fields, checking availability of software images at VIMs, among others). The following is a list of descriptors and a short description of their functionality:

- NS descriptor (NSd): used by the NFVO to instantiate a NS, which would be formed by one or several VNFFG, VNF, PNF, and VL. It also specifies deployment flavors of NS.
- VNF descriptor (VNFd): describes a VNF in terms of deployment and operation behavior. It includes network connectivity, interfaces and KPIs requirements that can be used by NFV-MANO functional blocks to establish appropriate VL within the NFVI.
- VL descriptor (VLd): provides information of each virtual link. It is used by NFVO to determine the appropriate placement of a VNF instance, and by the VIM to select a host with adequate network infrastructure. The VIM or external SDN controller uses this information to establish the appropriate paths and VLANs.
- VNFFG descriptor (VNFFGd): it includes metadata about the VNFFG itself, that is, VL, VNFs, PNFs, and policies (e.g.: MAC forwarding rules, routing entries, firewall rules, etc.).
- PNF descriptor (PNFd): is used by NFVO to create links between VNFs and PNFs. It includes information about connection points exposed by the PNF, and VLs that such physical connection points should be attached to.

4.1.3 VNF LIFECYCLE MANAGEMENT

VNF lifecycle management refers to the creation and lifecycle management of the needed virtualized resources for the VNF [4], as well as the traditional Fault Management, Configuration Management, Accounting Management, Performance Management and Security Management (FCAPS).

By making use of the information stored in a VNFd during onboarding, VNF Management functions make sure such requirements are met at the moment of instantiation. Furthermore, VNFd also contain information relevant for the lifecycle management (e.g.: constrains, KPIs, scale factor, policies, etc.). Such lifecycle management information is used for scaling operations, adding a new virtualized resource, shutting down an instance, or terminating it.

VNF Management maintains the virtualized resources that support the VNF functionality, without interfering with the VNFs' logical functions. Like NFVO, its functions are exposed through APIs as services to other functions. Each VNF instance is assumed to have an associated VNF Manager, and a VNF Manager could handle several VNFs. The following non-exhaustive list gathers the functions implemented by the VNF Manager [4]:

- VNF instantiation (based on onboarded VNFd).
- VNF instantiation feasibility checking.
- Scale VNFs (increase or decrease the resources of a VNF).
- Software Update/Upgrade on VNFs.
- Correlation between NFVI measurement results and faults/events, and the VNF instances.
- VNF instance assisted or automated healing.
- Terminate VNF (releasing the VNF-associated NFVI resources).
- Management of the VNF instance's integrity during its lifecycle.

From the information presented above, it is fair to conclude that any attempt to deploy an NFV NS must count with a NFVI, but also the specification of such NS via descriptors. For SEMIoTICS, VNFs related to networking might be available out-of-the-box, but other network elements such as gateways, smart elements and so forth,

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

must be specified as NFV descriptors for onboarding in the NFVO[6]. Otherwise instantiation would not be possible via NFV MANO, forsaking desired functionality such as dynamic scaling of VNFs, policy management/enforcement, and automation.

4.2 NVF MANO implementation

This section deals with the configuration of an ETSI-based NFV MANO infrastructure. It covers most of the details involved in the configuration of a Virtualized Infrastructure Manager (VIM) (i.e. OpenStack) with an SDN data plane, telemetry services and Service Function Chain (SFC) capability. It also describes the integration of an NFV Orchestrator (i.e. OSM), descriptor onboarding leveraging default clients or triggering NFV SOL-005 APIs [5] from the Os-Ma-nfvo reference point.

4.2.1 VIM: OPENSTACK

OpenStack is the industry default VIM for NFV leveraging virtual machines as virtualization technology. It provides all required reference points and interfaces detailed by ETSI in the specification of the NFV MANO platform, as well as strong security and High Available (HA) configurations, which makes it suitable for production environments.

In order to enable SEMIoTICS functionality, such as dynamic traffic stirring though software network overlays, or monitoring/management interfaces accessible to SEMIoTICS Global Pattern Orchestrator or responsible Pattern Engine; the default configuration of this VIM needs to be adapted.

4.2.1.1 ENABLING SFC

Service Function Chains (SFC) as the name suggests refers to the interconnection in chain-form of several service functions or VNFs. Traditionally, this technique allowed a sort of networking pipeline by forcing traffic to traverse different services (e.g. firewall, load balancers, IDS/IPS, etc.) before arriving to its final destination (e.g. Web server). The advent of SDN and control plane protocols such as OpenFlow (refer to D3.1 or D3.7 for more details) opened the way to more flexible and agile methods for either stirring traffic or building the SFC. With a compatible network fabric (i.e. OpenFlow-supporting data path), an SDN Controller (e.g. OpenStack Neutron, or SSC) could then reconfigure the data plane instantly. Moreover, NFV help realize network functions as virtual machines, therefore allowing SFC that do not need to be physically close (i.e. could be hosted on different NFVI under the same management domain), or physical at all.

There are three critical configuration changes needed for SEMIoTICS NFV Component to support SFC at the VIM. First, each compute node of the NFVI must reserve a dedicated network interface for connecting VNFs to VLAN provider networks[7]. Second, VXLAN are the only supported network technology for SFC data planes. And thirdly, the data plane interconnecting VNFs in the SFC must support OpenFlow. From the last requirement stems the first one, as Linux bridges do not support OpenFlow and it cannot share physical interfaces with other software switches (i.e. Open vSwitch).

SEMIoTICS VIM is composed of a manager, referred to as Infrastructure node, and several Compute nodes. The Infrastructure node hosts all VIM services managers, while Compute nodes host the virtual services instantiated on top as virtual machines and network overlays. Following the SFC requirements above, all Compute nodes have an OpenFlow-compatible data plane for VLAN and VXLAN networks with a dedicated network interface for VLAN provider networks. The following Configuration 1 shows an example network configuration of a Compute node in the SEMIoTICS NFVI.

In Configuration 1 several bridges and interfaces are defined. These are:
- **eno1:** physical interface leveraged for the creation of sub-interfaces on distinct VLAN segments.
- **eno1.1:** sub-interface in the VLAN id 1. This is used for node management purposes.

---

[6] As these other elements are considered VNFs, software (cloud) images should be created for each one of them.
[7] Networks to reach outside the NFVI are referred to as provider networks.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

- **br-semiotics:** switch connected to the eno1.1 interface, therefore handles all device management traffic. Not used for SEMIoTICS data.
- **eno1.110:** sub-interface in the VLAN id 110. This is the VLAN where the VIM Infrastructure node resides. It is used for OpenStack services' communication.
- **br-mgmt:** Linux bridge for VIM Infrastructure traffic.
- **provider-veth:** one of the extremes of a Virtual Ethernet (veth) pair (the other is eth101). provider-veth is connected to the OpenFlow-ready br-vlan.
- **eth101:** Interface connecting br-vlan to any OpenStack VLAN provider network switch (which could be created dynamically during runtime).
- **enx000ec6c9d385:** physical interface.
- **br-vlan:** VLAN provider networks data plane switch with OpenFlow support. Open vSwitch (OVS) 2.11.0 is used.
- **eno1.130:** sub-interface in VLAN id 130. Used for VXLAN traffic.
- **br-vxlan:** VXLAN networks data plane switch with OpenFlow support. Open vSwitch (OVS) 2.11.0 is used.

Once the above network configuration is done, next is the deployment of the VIM (i.e. OpenStack) with SFC support. There are several methods for deploying OpenStack, such as DevStack [24] or OpenStack Ansible (OSA) [25]. For SEMIoTICS, OpenStack Ansible is used on the basis of ease of use and previous experience.

OSA requires an external Deployment node. This node will be in the same LAN segment and have the required authentication and authorization credentials for managing the future Infrastructure and Compute nodes (see Figure 14). Following the OSA Deployment Guide [26], after applying the network configuration the next step is related to the desired OpenStack configuration (which can be based on different OSA examples [27]).
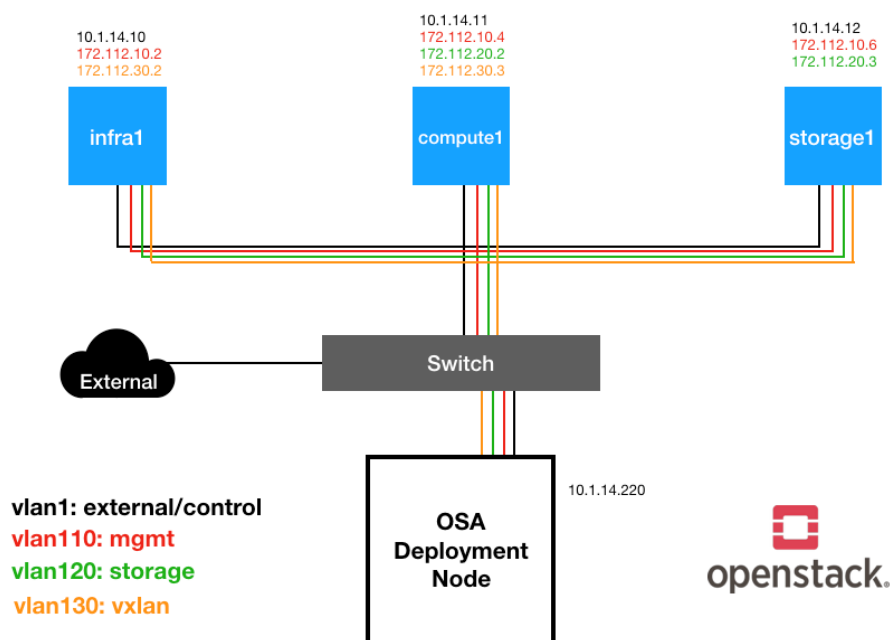
FIGURE 14 EXAMPLE OSA DEPLOYMENT TOPOLOGY

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
# Physical interface
auto eno1
iface eno1 inet manual

# Configuring SEMIoTICS access
auto eno1.1
iface eno1.1 inet manual
    vlan-raw-device eno1

auto br-semiotics
iface br-semiotics inet static
    bridge_stp off
    bridge_waitport 0
    bridge_fd 0
    bridge_ports eno1.1
    address 10.1.14.11
    netmask 255.255.255.0
    gateway 10.1.14.246
    dns-nameserver 8.8.8.8

# Container/Host management VLAN interface
auto eno1.110
iface eno1.110 inet manual
    vlan-raw-device eno1

# Container/Host management bridge
auto br-mgmt
iface br-mgmt inet static
    bridge_stp off
    bridge_waitport 0
    bridge_fd 0
    bridge_ports eno1.110
    address 172.112.10.4
    netmask 255.255.255.0
    dns-nameservers 8.8.8.8 8.8.4.4


# compute1 Network VLAN OVS
#
# Creating veth pair to connect to br-vlan
auto provider-veth
allow-br-vlan provider-veth
iface provider-veth inet manual
    pre-up /sbin/ip link add provider-veth type veth peer name eth101
    pre-up /sbin/ip link set provider-veth up
    pre-up /sbin/ip link set eth101 up

    ovs_bridge br-vlan
    ovs_type OVSPort

    pre-down /sbin/ip link delete provider-veth
    pre-down /sbin/ip link delete eth101


auto enx000ec6c9d385
allow-br-vlan enx000ec6c9d385
iface enx000ec6c9d385 inet manual
    ovs_bridge br-vlan
    ovs_type OVSPort

auto br-vlan
allow-ovs br-vlan
iface br-vlan inet manual
    ovs_type OVSBridge
    ovs_ports enx000ec6c9d385 provider-veth

# OpenStack Networking VXLAN (tunnel) OVS bridge
auto eno1.130
allow-br-vxlan eno1.130
iface eno1.130 inet manual
    vlan-raw-device eno1
    ovs_bridge br-vxlan
    ovs_type OVSPort
    mtu 9000

auto br-vxlan
allow-ovs br-vxlan
iface br-vxlan inet static
    ovs_type OVSBridge
    ovs_ports eno1.130
    address 172.112.30.3/24
```

CONFIGURATION 1 SEMIOTICS NFVI COMPUTE NODE NETWORK CONFIGURATION EXAMPLE

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

The SEMIoTICS OSA OpenStack User Configuration file (OUC) is written in a data serialization language called YAML [28]. The OUC specifies everything related to the OpenStack deployment, including reserved IP addresses, networks, and OpenStack services to be deployed (e.g. Neutron, Nova, Cinder, etc.). Configuration 2 shows an example segment of an OUC that specifies what OpenStack services to install and where.

As can be seen in Configuration 2, all the OpenStack services required by SEMIoTICS are installed, these are: Nova (virtualization), Neutron (networking), Cinder (storage), Ceilometer (telemetry collection), Gnocchi (telemetry storage), Glance (software images), Horizon (GUI), and Keystone (authentication and authorization). With the exception of Nova Hypervisors, Cinder storage host, and Ceilometer compute agents, all services run in the Infrastructure node.

The last step towards configuring SFC is to install the Networking-SFC Neutron plugin [29]. This plugin provides APIs and implementation to support SFC directly from OpenStack Networking (i.e. Neutron). With OSA, this step simply requires the definition of specific variables that tell Ansible to install and configure the required SFC packages for Neutron. Configuration 3 gathers the required variable definitions that tell OSA to download and install the Networking-SFC Neutron plugin.

Figure 16 shows the NFVO's graphical representation of an SFC as a network service. VNFs are represented as purple squares, connection points as small blue squares, while networks are represented by green triangles. In summary, the SFC is composed of three VNFs, referred to as **a)** 1-sfc-generic-endpoint, **b)** 2-sfc-generic-endpoint, and **c)** 3-sfc-mpls. The one-way asymmetrical chain shown in the figure guarantees matched traffic flows in the following direction: **a→c→b**. That is, in this specific example all HTTP traffic with TCP port 80 in the **a→b** direction (i.e. matched traffic) should traverse **c**. The three Terminal windows in Figure 15 show: VNF **b** in the top right, VNF **a** in the bottom right, while the whole left is dedicated to displaying live Tcpdump [30] captures of the matched traffic in VNF **c**. In the latter it is possible to see the flow of HTTP traffic on port 80 traversing **c** on their way from **a** to **b**.

The specification of VNFs composing the SFC, as well as the forwarding instructions and matched traffic are all contained in the SFC NSd. Configuration 5 shows a relevant segment included in the NSd, specifically under the VNF Forwarding Graph (VNFFG) descriptor that defines the traffic characteristics to match and stir into the SFC. In this example, **member-vnf-index: 1** (VNF **a**) data plane switch is instructed to match TCP traffic (**ip-proto: 6**) with destination port 80 and IP 13.0.1.102, and forward it to **member-vnf-index: 3** (VNF **c**)[8].

---

[8] According to the Rendered Service Path (RSP) specification (i.e. rsp1).

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
###
### Infrastructure
###

# galera, memcache, rabbitmq, utility
shared-infra_hosts:
  infra1:
    ip: 172.112.10.10

# repository (apt cache, python packages, etc)
repo-infra_hosts:
  infra1:
    ip: 172.112.10.10

# load balancer
haproxy_hosts:
  infra1:
    ip: 172.112.10.10

###
### OpenStack
###

# keystone
identity_hosts:
  infra1:
    ip: 172.112.10.10

# cinder api services
storage-infra_hosts:
  infra1:
    ip: 172.112.10.10

# glance
image_hosts:
  infra1:
    ip: 172.112.10.10

# nova api, conductor, etc services
compute-infra_hosts:
  infra1:
    ip: 172.112.10.10

# horizon
dashboard_hosts:
  infra1:
    ip: 172.112.10.10

# neutron server, agents (L3, etc)
network_hosts:
  infra1:
    ip: 172.112.10.10

# nova hypervisors
compute_hosts:
  compute1:
    ip: 172.112.10.12

# ceilometer (telemetry data collection)
metering-infra_hosts:
  infra1:
    ip: 172.112.10.10

# ceilometer compute agent (telemetry data collection)
metering-compute_hosts:
  compute1:
    ip: 172.112.10.12

gnocchi (telemetry metrics storage)
metrics_hosts:
  infra1:
    ip: 172.112.10.10

# cinder storage host (LVM-backed)
storage_hosts:
  storage1:
    ip: 172.112.10.7
    container_vars:
      cinder_backends:
        limit_container_types: cinder_volume
        lvm:
          volume_group: cinder-volumes
          volume_driver: cinder.volume.drivers.lvm.LVMVolumeDriver
          volume_backend_name: LVM_iSCSI
          iscsi_ip_address: "172.112.20.3"
```

CONFIGURATION 2 OUC SPECIFYING OPENSTACK SERVICES AND THE CORRESPONDING NODE HOSTING IT

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
# Ensure the openvswitch kernel module is loaded
openstack_host_specific_kernel_modules:
  - name: "openvswitch"
    pattern: "CONFIG_OPENVSWITCH"

### neutron specific config
neutron_plugin_type: ml2.ovs
neutron_ml2_drivers_type: "vlan,vxlan"

neutron_plugin_base:
  - router
  - metering
  - flow_classifier
  - sfc

neutron_ml2_conf_ini_overrides:
    agent:
      extensions: sfc
neutron_vxlan_enabled: true
```

CONFIGURATION 3 OSA OUC USER VARIABLES ENABLING NEUTRON SFC

FIGURE 15 SNIFFING TRAFFIC TRAVERSING THE MIDDLE VNF IN AN SFC

4.2.1.2   TELEMETRY SERVICES FOR MONITORING NFVI
OpenStack telemetry services query and store specific metrics from both NFVI and VNFs. Ceilometer, OpenStack de facto telemetry service, deploys agents at compute nodes that periodically poll components for different metrics. Such metrics are then gathered by a central Ceilometer agent which later transmits them to a telemetry storage service for processing and exposure.

In SEMIoTICS we have adopted the Ceilometer and Gnocchi combination for polling and storage solutions, respectively. This has been the most used configuration by the open source community and enjoys ample support from deployment tools (such as OSA). Gnocchi is a Time Series Database as a Service (TSDaaS) project completely compatible with Ceilometer. Moreover, is provides RESTful APIs for metrics monitoring (e.g. which may be realized via the Nf-Vi, Vi-Vnfm, or Or-Vi reference points of the NFV MANO architecture). One example of Gnocchi API usage is the one performed by the NFVO to trigger dynamic VNF scaling operations.

Configuration 6 shows the definition of OSA user variables enabling the monitoring of Compute nodes' performance, as well as a pointer for Ceilometer to the Gnocchi TSDaaS endpoint.

46

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
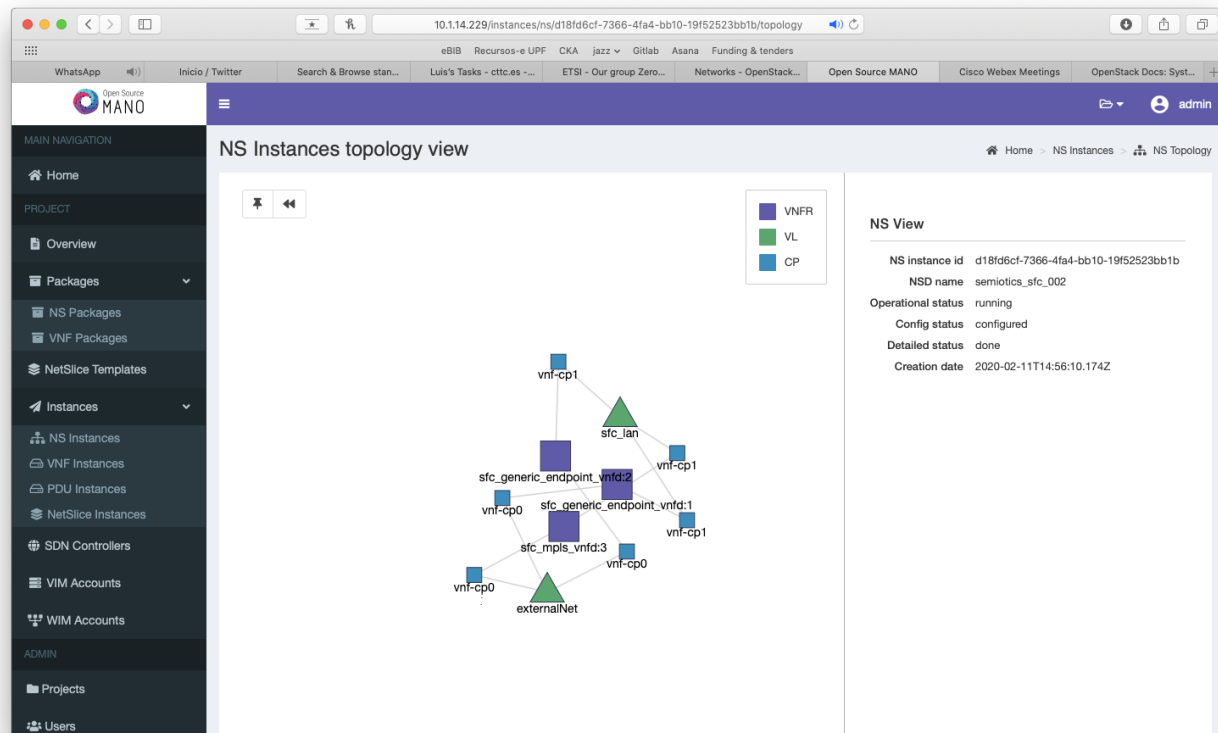
FIGURE 16 NFVO DIAGRAM OF AN SFC

4.2.2 NFVO AND VNF MANAGER: OSM

SEMIoTICS adopts ETSI Open Source MANO (OSM) as its NFVO. OSM is an ETSI NFV compliant NFVO capable of supporting a wide range of VIMs, offers standard northbound REST APIs (i.e. SOL 005) for the onboarding of descriptors, orchestration and termination of network services, among others.

OSM can be deployed as a VNF, PNF or as a set of microservices in a Container Orchestration Engine (COE[9]) cluster. In SEMIoTICS, OSM release 6 is deployed as a Virtual Machine inside the NFVI, but out of the domain of the VIM. Furthermore, OSM requires IP connectivity to the VIM's endpoints (e.g. authentication, instantiation, networking, etc).

```
osm vim-create --name semiotics_playground_queens_001 \
        --account_type openstack \
        --auth_url http://172.112.10.1:5000/v3 \
        --user smartech \
        --password smartech \
        --tenant playground \
        --description "SEMIoTICS VIM @ playground" \
        --config='{insecure: 'True', security_groups: 'default', keypair: 'lsanabria',
use_internal_endpoint: 'True', APIversion: 'v3.3', project_domain_id: 'default', project_domain_name:
'Default', user_domain_id: 'default', user_domain_name: 'Default'}'
```
CONFIGURATION 4 ADDING A VIM ACCOUNT TO OSM NFVO

Configuration 4 shows an example of how to add a VIM account to OSM NFVO for the orchestration of VNFs. The example uses the OSM Command Line Interface (CLI) client for performing this task, even though a GUI is also available.

---

[9] Only from OSM Release 7 on.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
...
vld:
- id: sfc_lan
  name: sfc_lan
  short-name: sfc_lan
  ip-profile-ref: sfc_generic_ip_profile
  type: ELAN
  vim-network-name: "sfc"
  vnfd-connection-point-ref:
  -   member-vnf-index-ref: 1
      vnfd-id-ref: sfc_generic_endpoint_vnfd
      vnfd-connection-point-ref: vnf-cp1
      ip-address: 13.0.1.101
  -   member-vnf-index-ref: 2
      vnfd-id-ref: sfc_generic_endpoint_vnfd
      vnfd-connection-point-ref: vnf-cp1
      ip-address: 13.0.1.102
  -   member-vnf-index-ref: 3
      vnfd-id-ref: sfc_mpls_vnfd
      vnfd-connection-point-ref: vnf-cp1

vnffgd:
- id: vnffg1
  name: vnffg1
  description: vnffg1
  short-name: vnffg1
  vendor: vnffg1
  version: '1.0'
  rsp:
  - id: rsp1
    name: rsp1
    vnfd-connection-point-ref:
    - member-vnf-index-ref: 3
      order: 0
      vnfd-egress-connection-point-ref: vnf-cp1
      vnfd-id-ref: sfc_mpls_vnfd
      vnfd-ingress-connection-point-ref: vnf-cp1
  classifier:
  - id: class1
    name: class1
    member-vnf-index-ref: 1
    rsp-id-ref: rsp1
    vnfd-connection-point-ref: vnf-cp1
    vnfd-id-ref: sfc_generic_endpoint_vnfd
    match-attributes:
    - id: HTTP
      destination-ip-address: 13.0.1.102
      destination-port: 80
      ip-proto: 6
      source-ip-address: 13.0.1.101
      source-port: 0
```

CONFIGURATION 5 VNFFG DESCRIPTOR FLOW CLASSIFIER

```
## Nova conf Overrides
nova_ceilometer_enabled: True
nova_nova_conf_overrides:
  DEFAULT:
    compute_monitors: cpu.virt_driver
    force_config_drive: true
    resume_guests_state_on_host_boot: true


## Ceilometer user variables
ceilometer_sample_interval: 10
ceilometer_gnocchi_enabled: True

ceilometer_ceilometer_conf_overrides:
  dispatcher_gnocchi:
    archive_policy: high
    url: http://172.114.10.10:8041
```

CONFIGURATION 6 OSA USER VARIABLES TO CONFIGURE TELEMETRY WITH CEILOMETER AND GNOCCHI

Descriptor onboarding is another procedure that needs to be performed before doing any network service orchestration. Users can either use the GUI, the OSM CLI client, or the NBI REST API for this task. The latter is often preferred when integrating orchestration functions in other software (e.g. Pattern Orchestrator).

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

4.3   NFV MANO interaction with the Pattern Orchestrator

4.3.1   PATTERN ORCHESTRATOR IN THE NFV CONTEXT

The SEMIoTICS project relies on a pattern-driven approach, which allows the network operators to enforce patterns that reflect the requirements of the corresponding network services in terms of e.g. latency, reliability, security or privacy. To this end, they gather metrics of the network to extract the patterns that shed light on such patterns and requirements. In the context of the NFV framework, this pattern-driven approach is contemplated as follows.

On the one hand, a Pattern Engine is considered locally, i.e. this is an entity that has a direct link with the NFV MANO. Moreover, there is the Pattern Orchestrator at the backend cloud. Upon request of the Pattern Orchestrator, the Pattern Engine can ask the NFV MANO (e.g. the VIM), to gather metrics about the state of the virtualized network, i.e. the NFVI. This information can be processed locally, or it can be sent to the Pattern Orchestrator.

After that processing, patterns related to the requirements of the network services are extracted. These patterns are used to specify the descriptors of VNFs and NS. Namely, upon request of the Pattern Orchestrator, the Pattern Engine updates and prepares such descriptors and communicates with the NFV MANO. Recall that, as it was explained above, in the NFV MANO context, all the network services require an associated network service descriptor to be deployed.

An example on the role of the Pattern Orchestrator and the Pattern Engine in the NFV context is given in the next section. Namely, the sequence diagram to instantiate an onboarded VNF is considered and the role of the Pattern Orchestrator and Pattern Engine is illustrated.

4.3.2   SEQUENCE DIAGRAMS

In this section, a dynamic view of the NFV operation is illustrated. To this end, sequence diagrams associated to the NFV MANO procedure are presented. Thereby, in Figure 17 the sequence diagram related to the instantiation of an onboarded VNF is presented, i.e. the instantiation of a VNF that is already in the NFV MANO catalogue. The rest of the sequence diagrams, as well as further insights on the involvement of the Global Patter Orchestrator in the NFV Orchestration process, will be presented in the final deliverable D3.8.

As it is shown in Figure 17, the VNF instantiation starts upon request of a sender, i.e. the entity that wants to deploy the VNF functionality in the NFVI. The sender communicates with the Pattern Orchestrator, as the patterns associated with the VNF must be updated to configure properly the VNF descriptor. Then, the Pattern Orchestrator communicates with the Pattern Engine, which has a direct link with the NFV MANO (VIM) and thereby can ask to gather metrics on the state of the NFVI. Afterwards, with that updated information, the Pattern Orchestrator can extract the patterns related to the VNF requirements or KPIs and asks the local Pattern Engine to configure the corresponding VNF descriptor.

At this point the Pattern Engine communicates with the NFV orchestrator to start the VNF instantiation. Then, owing to the NFV MANO hierarchical architecture, the NFV orchestrator asks the VNF manager to instantiate the VNF. After validation by the VNF manager, a set of resources must be allocated to run properly the VNF. As we can see, this is the responsibility of the VIM. And the instantiation finishes after a set of acknowledgments messages among the different actors.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
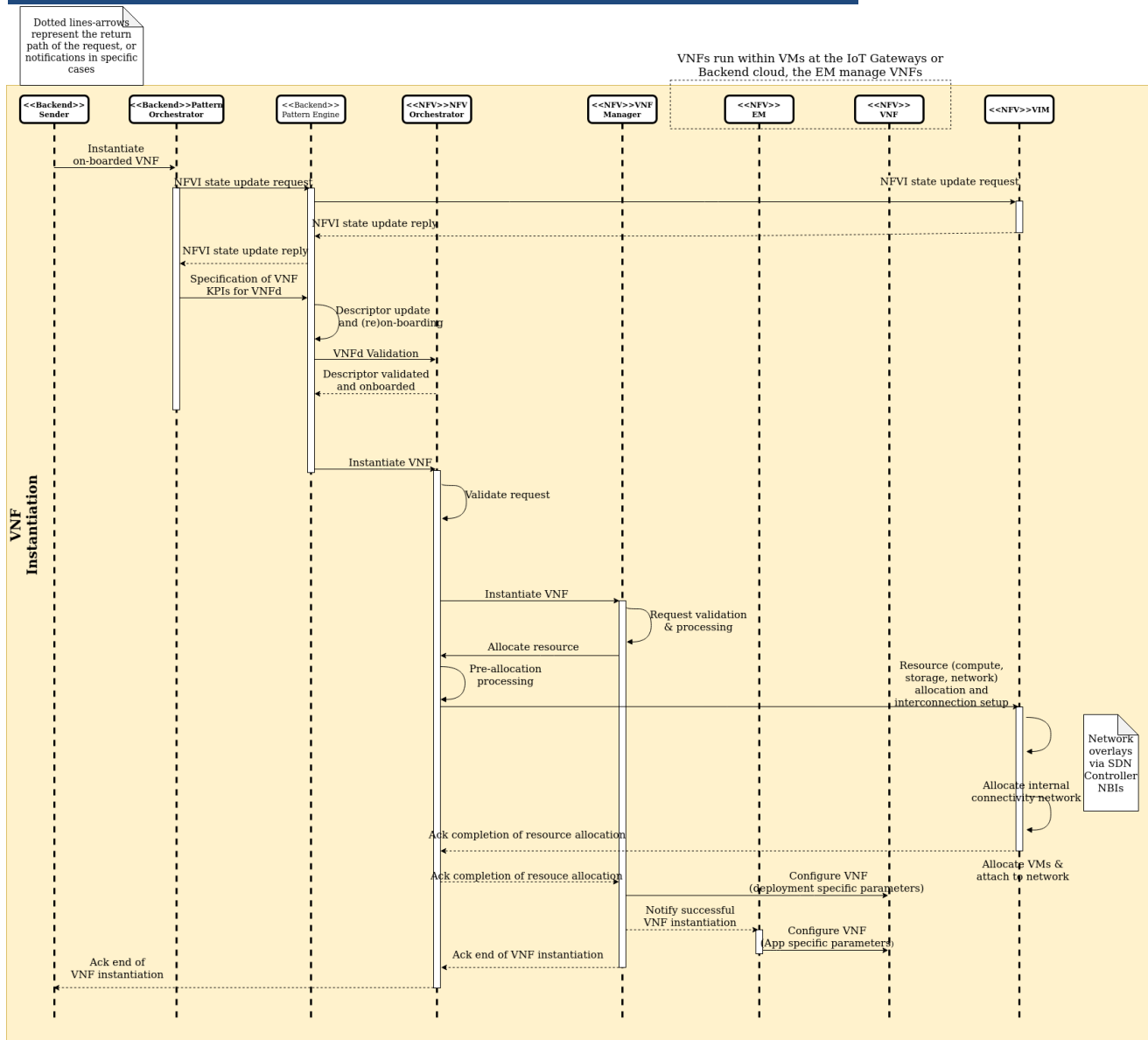
FIGURE 17 SEQUENCE DIAGRAM FOR THE INSTANTIATION OF A VNF IN THE NFV FRAMEWORK.

4.3.3 INTERACTION WITH THE NFV MANO BASED ON RESTFUL NBI.

The previous sections highlight that the Pattern Orchestrator needs to interact with the NFV MANO to enforce patterns in the NS. In fact, from the NFV MANO viewpoint, the Pattern Orchestrator and the pattern engine can be regarded as OSS entities. Thereby, the interface between the NFV MANO and the Pattern Orchestrator/Engine is well defined through the Os-Ma-NFVO reference point, which is specified by the ETSI standards, see [2] [4]. In practical terms, the Os-Ma-NFVO interface can be implemented through RESTful protocols, as suggested by ETSI in the ETSI NFV-SOL specification [5]. This is the approach embraced herein, as RESTful APIs are a widely accepted means of communicating between software applications and computers in the Internet. Therefore, in this section we explain the RESTful protocols that enable the interaction between the NFV MANO and external OSS such as the Pattern Orchestrator. Also, in this regard, we provide practical examples on the RESTful commands needed to onboard NS packages or manage the

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

lifecycle of NS instances. Note that these operations are mandatory to enable the remote control of the Pattern Orchestrator on specific NS.

In general terms, [5] defines the RESTful protocols and the associated data models to implement the Os-Ma-Nfvo reference point. Thereby, we can implement the operations related to the management of the NS descriptor (NSD) or the NS lifecycle management. For instance, the creation of a NSD, upload the content of a NSD, instantiate a NS or terminate a NS. To this end, the RESTful protocol defines:

- The URI resource structure. For instance, the structure that identifies a NSD.
- The HTTP methods that can be applied to the URI resources. For instance, a GET method to consult the information of a given NSD.
- The data structure that we need to specify for a given HTTP method. For instance, a POST method to instantiate a NS requires to specify the identification of the NSD to be instantiated. And this corresponds to a data structure field called "nsdId", which is specified in the body of the HTTP request.

Next, we give more details on these RESTful protocols for the operations involving the interface between the NFV MANO and OSS/BSS such as the Pattern Orchestrator/Pattern Engine. These operations are illustrative examples for the SEMIoTICS purposes.

4.3.3.1   OPERATIONS RELATED TO THE VNF PACKAGE MANAGEMENT INTERFACE

This interface permits the OSS (in our case the Pattern Orchestrator/Pattern Engine) to carry out operations related to the VNF packages such as:
- Create a VNF package resource.
- Upload the content of a VNF package.
- Query the VNF package information, e.g. the VNF descriptor.
- Delete a VNF package.

A simplified structure of the URI resource for the VNF package management interface is displayed in Figure 18. Where {apiRoot} indicates the scheme ("http" or "https"), the host name and optional port, and an optional prefix path. For instance, in the CTTC testbed, the OSM that implements the NFV MANO has the IP 10.1.14.248, which is accessible through the port 9999. Thereby, the {apiRoot} in our case has the following expression:

- https://10.1.14.248:9999/osm
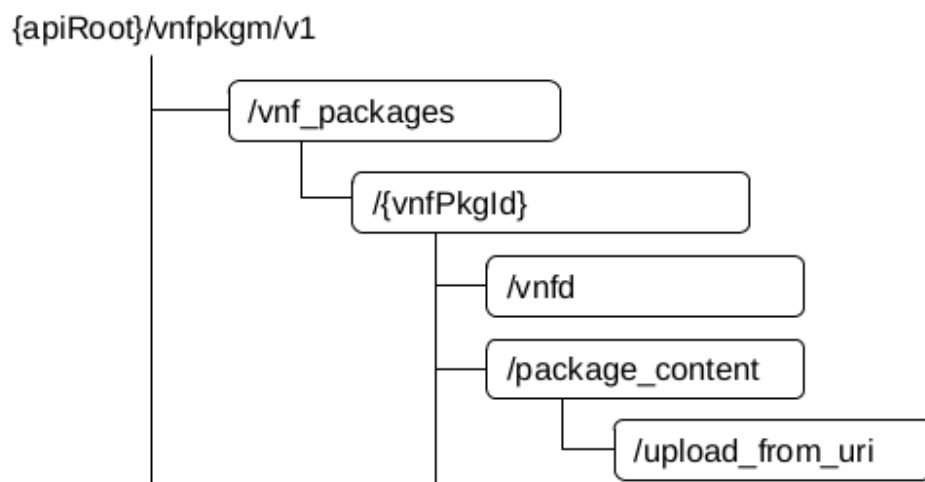
FIGURE 18 SIMPLIFIED VIEW OF THE URI RESOURCE STRUCTURE FOR THE VNF PACKAGE INTERFACE MANAGEMENT [5].

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

Then, the RESTful protocol to perform management operations on the VNF package interface consists of applying HTTP methods to the URI resources depicted in Figure 18. The HTTP methods that can be applied to each individual VNF package resource are thoroughly explained in table 9.2-1 of [5]. In Table 5 we provide a summary of relevant HTTP methods applicable to the URI resource and its meaning.

TABLE 5 RESOURCES AND HTTP METHODS FOR THE VNF PACKAGE MANAGEMENT INTERFACE

Resource URI	HTTP method	Meaning
/vnf_packages	GET	Obtain information on all VNF packages that are onboarded in the NFV MANO, i.e. the OSM.
	POST	Create a VNF package resource. This creates like a placeholder, and afterwards with a PUT method we upload the VNF package content.
/vnf_packages/{vnfPkgId}/package_content	PUT	Upload the content of a VNF package.

We have implemented the RESTful protocols just described above in the CTTC testbed, which implements the NFV MANO by means of OSM and the VIM through OpenStack, as it was described in previous sections. Namely, we emulate an external OSS by means of the postman software tool[10], which allows RESTful API testing. Next, we give details on the experiments that we carry out to implement the RESTful API described in Table 5.

GET method applied to the /vnf_packages resource

We applied a GET method to the /vnf_packages resource to get the information of all the onboarded VNF packages. The RESTful query and the result are displayed in Figure 19. Observe that the complete URI resource reads "https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages", which corresponds to the specification of Figure 18. Also, for the sake of completeness we provide the curl code:

```
curl -X GET \
 https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages \
 -H 'Accept: */*' \
 -H 'Accept-Encoding: gzip, deflate' \
 -H 'Authorization: Bearer dYzzv3pZzYPIXFmnVFGOJxHF1sBDpiG6' \
 -H 'Cache-Control: no-cache' \
 -H 'Connection: keep-alive' \
 -H 'Content-Type: application/x-www-form-urlencoded' \
 -H 'Host: 10.1.14.248:9999' \
 -H 'Postman-Token: 3f2b5a61-dcde-4157-a405-fd83eda5d252,9d361317-9b7b-4995-9bd5-1a7351402adb' \
 -H 'User-Agent: PostmanRuntime/7.20.1' \
 -H 'cache-control: no-cache'
```

---

[10] www.postman.com

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
6        operationalState: ENABLED
7        projects_read:
8        - 7d29ec11-0169-4d31-a2e2-7a0650966336
9        projects_write:
10       - 7d29ec11-0169-4d31-a2e2-7a0650966336
11       storage:
12           descriptor: generic-vnf_vnfd/generic-vnf_vnfd.yaml
13           folder: 7433aafd-b695-4aaa-9d41-b623ef1883bf
14           fs: local
15           path: /app/storage/
16           pkg-dir: generic-vnf_vnfd
17           zipfile: generic-vnf_vnfd.tar.gz
18       type: vnfd
19       usageState: NOT_IN_USE
20       userDefinedData: {}
21       _id: 7433aafd-b695-4aaa-9d41-b623ef1883bf
22   connection-point:
23   -   name: vnf-cp0
24       type: VPORT
25   description: Generated by OSM package generator
26   id: generic-vnf_vnfd
27   mgmt-interface:
28       cp: vnf-cp0
29   name: generic-vnf_vnfd
30   short-name: generic-vnf_vnfd
31   vdu:
32   -   cloud-init-file: generic-user-data
33       count: '1'
34       description: generic-vnf_vnfd-VM
35       id: generic-vnf_vnfd-VM
36       image: ubuntu18-server
37       interface:
38       -   external-connection-point-ref: vnf-cp0
39           name: eth0
40           virtual-interface:
41               type: VIRTIO
42       name: generic-vnf_vnfd-VM
43       vm-flavor:
44           memory-mb: '1024'
45           storage-gb: '0'
46           vcpu-count: 1
47   vendor: OSM
48   version: '1.0'
```

FIGURE 19 USING POSTMAN TO APPLY A GET METHOD ON THE /VNF_PACKAGES RESOURCE.

POST method applied to the /vnf_packages resource

In order to create a new VNF package resource at the NFV MANO, i.e. the OSM, we need to apply a POST method on the URI resource /vnf_packages. Namely, the complete URI structure for the CTTC testbed is "https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages". As in the previous RESTful API, we used postman to emulate the OSS and to send the request to the OSM. In Figure 20 we display the postman interface that shows the POST query. Observe that it returns an "id" parameter, which corresponds to the identification for the VNF package resource that the OSM has created. Figure 20 also shows the web interface of the OSM, where we can observe that effectively a VNF package resource has been created with the id mentioned above. Observe that the VNF package resource is void, as we need to upload the VNF package content. This operation will be presented below. The curl code associated to the REST API in Figure 20 is displayed next:

```
curl -X POST \
 https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages \
 -H 'Accept: */*' \
 -H 'Accept-Encoding: gzip, deflate' \
 -H 'Authorization: Bearer jB9X82hWidTqB4ZfxblN9gM7fwzh82sQ' \
 -H 'Cache-Control: no-cache' \
 -H 'Connection: keep-alive' \
 -H 'Content-Length: 0' \
 -H 'Cookie: session_id=c274a7fde7058c47202c08e6b793471136ca1887' \
 -H 'Host: 10.1.14.248:9999' \
```

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
-H 'Postman-Token: 59a225ee-de4f-4259-a72c-dfe328fc9aa4,1e6880e4-f56c-4ba6-8447-296116818026' \
-H 'User-Agent: PostmanRuntime/7.20.1' \
-H 'cache-control: no-cache'
```


FIGURE 20 POSTMAN TO APPLY A POST METHOD ON THE /VNF_PACKAGES RESOURCE AND RESULT IN THE OSM.

PUT method applied to the /vnf_packages/{vnfPkgId}/package_content resource

This operation permits to upload the content related to a VNF package, i.e. its VNF descriptor. Thereby, this RESTful API requires first to carry out the previous POST request described above, which creates the VNF resource at the OSM. In fact, the *{vnfPkgId}* placeholder in the URI resource must be substituted by the identification obtained by the POST operation. Observe that according to Figure 20 this identification reads "*618b6f75-b679-45b3-941e-7434fcdca0a3".* Therefore, the complete URI resource for the CTTC testbed reads *https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages/618b6f75-b679-45b3-941e-7434fcdca0a3/package_content*.

Also, it is important to note that in the body of the RESTful command we attach the .yaml file that represents the VNF package descriptor, i.e. the content of the VNF package, which will be used by OSM. And another important detail is that in the headers of the RESTful command we must specify that we are sending a .yaml file. We do by setting the "Content-Type" and the "Accept" keys to the "application/x-yaml" value. Thereby, bearing in mind all these considerations, we present in Figure 21 the complete RESTful command that we need to run to upload a VNF package content to OSM. As in the previous cases, we did the implementation using postman. In Figure 22 we show the effect that this PUT REST API has on the OSM. Comparing this figure to Figure 20 we can see that now we have content for the VNF package. Next, we provide the curl code associated to the PUT command that we have just described.

```
curl -X PUT \
  https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages/618b6f75-b679-45b3-941e-7434fcdca0a3/package_content \
  -H 'Accept: application/x-yaml' \
  -H 'Accept-Encoding: gzip, deflate' \
```

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

-H 'Authorization: Bearer jKPEEepXUQ3yIGUS6fSygXBmANFykIyL' \
-H 'Cache-Control: no-cache' \
-H 'Connection: keep-alive' \
-H 'Content-Length: 860' \
-H 'Content-Type: application/x-yaml' \
-H 'Cookie: session_id=c274a7fde7058c47202c08e6b793471136ca1887' \
-H 'Host: 10.1.14.248:9999' \
-H 'Postman-Token: 2154db23-fcd6-4079-91a0-ce50e6773e71,34681427-815c-4fe6-8d2b-988a304f86a7' \
-H 'User-Agent: PostmanRuntime/7.20.1' \
-H 'cache-control: no-cache'

FIGURE 21 PUT METHOD IN POSTMAN TO UPLOAD THE CONTENT OF A VNF PACKAGE.

FIGURE 22 EFFECT OF THE PUT METHOD IN OSM.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

4.3.3.2   OPERATIONS RELATED TO THE NSD MANAGEMENT INTERFACE

This interface allows OSS to control externally the operations related to the NS descriptors in the NFV MANO, i.e. the OSM. More specifically, we can perform operations like:

- Query the information of the available NS descriptors at the NFV MANO.
- Create a NS package resource.
- Upload the content of a NS package, i.e. a NS descriptor.

The simplified URI resource structure is shown in Figure 23. Moreover, in Table 6 we present the HTTP methods that can be applied to the URI resources to manage the NSD interface. Below we give complete details on how we implemented all these HTTP REST APIs in the CTTC testbed.
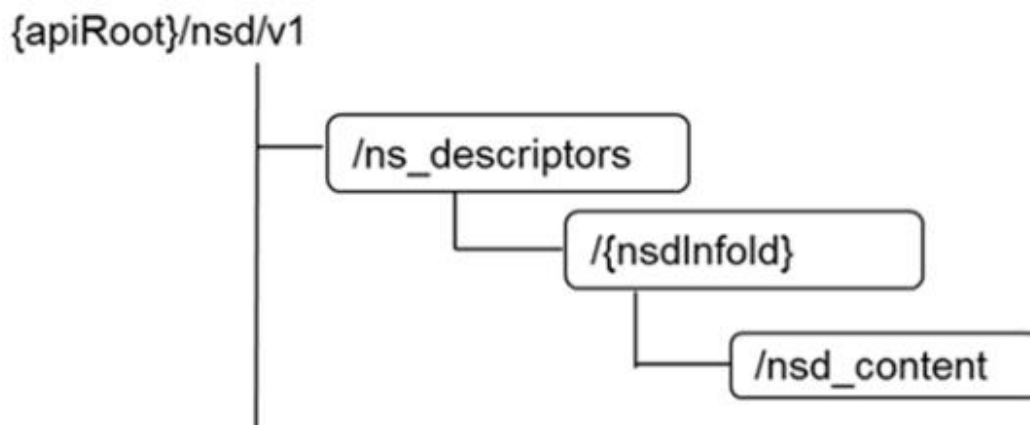
FIGURE 23 SIMPLIFIED VIEW OF THE URI RESOURCE STRUCTURE FOR THE NSD MANAGEMENT INTERFACE [5].

TABLE 6 RESOURCES AND HTTP METHODS FOR THE NSD MANAGEMENT INTERFACE

Resource URI	HTTP method	Meaning
/ns_descriptors	GET	Obtain information on all NS descriptors onboarded in the NFV MANO, i.e. the OSM.
	POST	Create a NS descriptor resource. This creates like a placeholder, and afterwards with a PUT method we upload the NS descriptor content.
/ns_descriptors/{nsdInfoId}/nsd_content	PUT	Upload the content of a NS descriptor.

GET method applied to the /ns_descriptors resource

This method allows to obtain information on all the NS onboarded in the NFV MANO, i.e. in our case in the OSM. We have implemented this REST API using postman, as it is shown in Figure 24. To this end, an important observation is that the complete URI resource in the CTTC testbed reads:

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

- *"https://10.1.14.248:9999/osm/nsd/v1/ns_descriptors"*.

Figure 24 shows the result of running the GET method, i.e. it effectively displays the NS descriptors. For the sake of completeness, we also provide the curl code associated to this request:

*curl -X GET *
  *https://10.1.14.248:9999/osm/nsd/v1/ns_descriptors *
  *-H 'Accept: */*' *
  *-H 'Accept-Encoding: gzip, deflate' *
  *-H 'Authorization: Bearer bLX8XE1VpdwbwleJNyeNz1OPkFn3OoBP' *
  *-H 'Cache-Control: no-cache' *
  *-H 'Connection: keep-alive' *
  *-H 'Cookie: session_id=fee5827f5ac9c0a85cd3bd26976dc3b57ffc3c4b' *
  *-H 'Host: 10.1.14.248:9999' *
  *-H 'Postman-Token: 253e7095-4898-45b9-adab-8fbea67c1461,54dc58fd-6776-4e41-b92d-7d383d5e551f' *
  *-H 'User-Agent: PostmanRuntime/7.20.1' *
  -H 'cache-control: no-cache'

FIGURE 24 GET REQUEST TO OBTAIN NSD INFORMATION, IMPLEMENTED IN POSTMAN.

57

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

POST method applied to the /ns_descriptors resource

This REST operation allows the OSS to create externally a NS package resource in the NFV MANO. To this end, the complete URI resource in the CTTC testbed is *https://10.1.14.248:9999/osm//nsd/v1/ns_descriptors*. Thereby, we applied the POST method to this URI resource using postman as an emulation of the OSS. This is shown in Figure 25. Observe that this request produces an "id", which identifies the new NS package. Also, observe in the OSM web interface, that the NS resource with that "id" is void. With the PUT method of the next section we will upload the NS content. We also provide the curl code associated to this NS creation:

```
curl -X POST \
  https://10.1.14.248:9999/osm//nsd/v1/ns_descriptors \
  -H 'Accept: */*' \
  -H 'Accept-Encoding: gzip, deflate' \
  -H 'Authorization: Bearer VKHja3XKGvH37Oia0OUIInnHiqNVrzVa' \
  -H 'Cache-Control: no-cache' \
  -H 'Connection: keep-alive' \
  -H 'Content-Length: 0' \
  -H 'Cookie: session_id=fee5827f5ac9c0a85cd3bd26976dc3b57ffc3c4b' \
  -H 'Host: 10.1.14.248:9999' \
  -H 'Postman-Token: 8c446a90-26f4-4833-b339-11f39d1a8fce,acd86f84-4861-4ad4-8dc4-cb555fa1d40e' \
  -H 'User-Agent: PostmanRuntime/7.20.1' \
  -H 'cache-control: no-cache'
```


FIGURE 25 POST REQUEST TO CREATE A NS PACKAGE RESOURCE.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

PUT method applied to the /ns_descriptors/{nsdInfoId}/nsd_content resource

This operation is used to upload the content of a NS from the OSS to the NFV MANO (the OSM). This operation requires first to execute the POST operation introduced above, which creates a new NS resource. In fact, we need the "id" produced by that POST operation. More specifically, the partial URI resource to upload the NS content reads ***/ns_descriptors/{nsdInfoId}/nsd_content*** and we need to substitute the *{nsdInfoId}* placeholder by the "id" produced by the POST operation. Therefore, considering the "id" of Figure 25, the complete URI resource to upload the NS content in the NFV MANO has the next structure:

- *https://10.1.14.248:9999/osm/nsd/v1/ns_descriptors/45a070fb-f716-4dc1-b932-42a7a52a540a/nsd_content*

We have execute the PUT method applied to this URI resource using postman, the result is displayed in Figure 26. It is important to put in the body that we want to send a binary file, which is the .yaml file with the NS content, i.e. the NS descriptor. Also, we must specify in the headers that the keys "Content" and "Accept" have associated values "application/x-yaml". Figure 26, also shows the web page of the NFV MANO, where we can see that effectively the NS resource has now an associated content. Finally, the curl code associated to this PUT RESTful request is as follows:

```
curl -X PUT \
  https://10.1.14.248:9999/osm//nsd/v1/ns_descriptors/45a070fb-f716-4dc1-b932-42a7a52a540a/nsd_content \
  -H 'Accept: application/x-yaml' \
  -H 'Accept-Encoding: gzip, deflate' \
  -H 'Authorization: Bearer 1dYtscCemxPDsdEgpxLstPpVy993XZWK' \
  -H 'Cache-Control: no-cache' \
  -H 'Connection: keep-alive' \
  -H 'Content-Length: 710' \
  -H 'Content-Type: application/x-yaml' \
  -H 'Cookie: session_id=fee5827f5ac9c0a85cd3bd26976dc3b57ffc3c4b' \
  -H 'Host: 10.1.14.248:9999' \
  -H 'Postman-Token: a25a044c-3c6e-4992-b1e7-1ad7f7bf969c,c3c62157-b336-471c-8a0f-a934e3c924a3' \
  -H 'User-Agent: PostmanRuntime/7.20.1' \
  -H 'cache-control: no-cache'
```

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
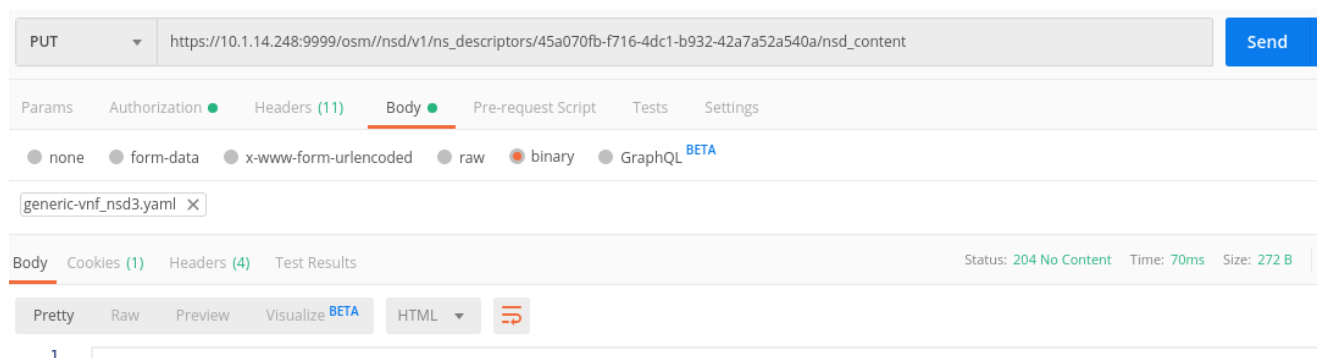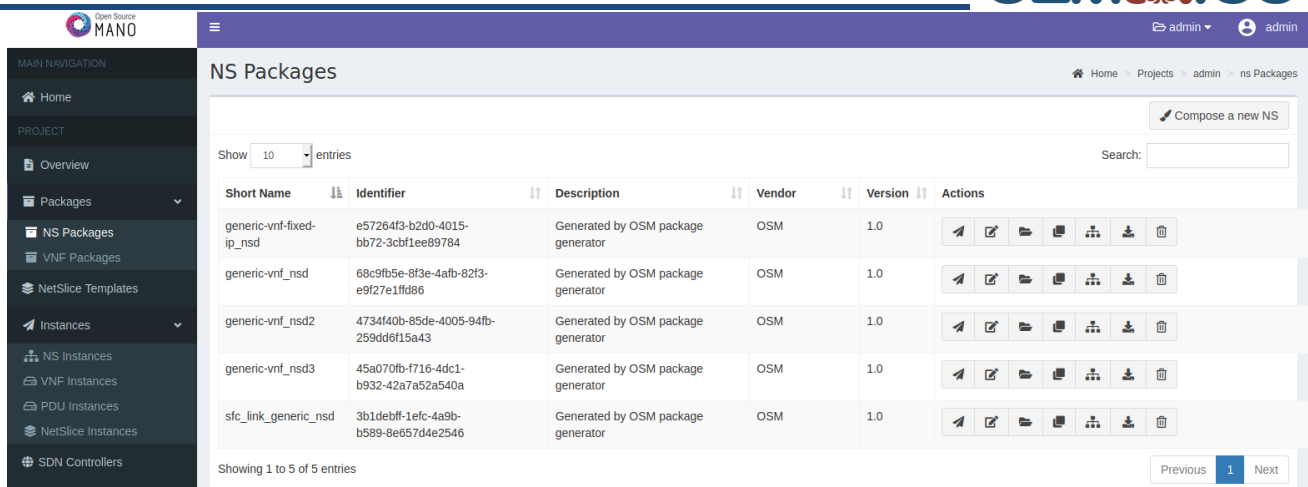
FIGURE 26 PUT REQUEST TO UPLOAD A NS CONTENT.

4.3.3.3   OPERATIONS RELATED TO THE NS LIFECYCLE MANAGEMENT INTERFACE

This interface permits an external OSS to execute operations in the NFV MANO related to the NS lifecycle management. This is a list of some of the possible operations:

- Query information on the NS that are instantiated at the NFV MANO.
- Create a NS instance resource.
- Instantiate a NS.
- Terminate a NS.

In order to perform these operations, we need the specification of the URI resources and the HTTP methods. These are specified in [5] and we show them in Table 7. As in the previous cases, the {apiRoot} for the CTTC testbed reads https://10.1.14.248:9999/osm.

TABLE 7 URI RESOURCES AND HTTP METHODS FOR THE NS LIFECYCLE MANAGEMENT INTERFACE [5]

Resource URI	HTTP method	Meaning
{apiRoot}/nslcm/v1/ns_instances	GET	Obtain information on all NS instances in the NFV MANO, i.e. the OSM.
	POST	Create a NS instance resource. This schedules a resource, and afterwards with a POST method we perform the instantiation associated to this NS instance resource.
{apiRoot}/nslcm/v1/ns_instances/{nsInstanceId}/instantiate	POST	Instantiate a NS.
{apiRoot}/nslcm/v1/ns_instances/{nsInstanceId}/terminate	POST	Terminate a NS instance.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

GET method applied to the {apiRoot}/nslcm/v1/ns_instances resource

The GET method applied to this URI resource allows the external OSS to obtain the information on the NS instances available in the NFV MANO. The complete URI resource has the next structure in the CTTC testbed *https://10.1.14.248:9999/osm/nslcm/v1/ns_instances*. Therefore, we applied the GET HTTP method to this URI resources, using postman as the external OSS. We can see the result in Figure 27. The curl code associated to this GET request is displayed next:

```
curl -X GET \
  https://10.1.14.248:9999/osm/nslcm/v1/ns_instances \
  -H 'Accept: */*' \
  -H 'Accept-Encoding: gzip, deflate' \
  -H 'Authorization: Bearer TNm6zBAcozoFQDy9NJcWw9eHwQdglHUI' \
  -H 'Cache-Control: no-cache' \
  -H 'Connection: keep-alive' \
  -H 'Host: 10.1.14.248:9999' \
  -H 'Postman-Token: 9aad5014-3370-4c7e-b155-27926d267b5f,95cc00a6-ae55-49fd-af58-d61f04b5cdec' \
  -H 'User-Agent: PostmanRuntime/7.20.1' \
  -H 'cache-control: no-cache'
```


FIGURE 27 GET REQUEST TO OBTAIN THE INFORMATION ON THE NS INSTANCES AT THE OSM.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

POST method applied to the {apiRoot}/nslcm/v1/ns_instances resource

This RESTful command allows an external OSS to create a new NS instance resource in the NFV MANO. Figure 28 shows how we have to configure postman to perform this operation. First, observe that the complete URI resource is:

- *https://10.1.14.248:9999/osm/nslcm/v1/ns_instances*

Then, we must apply the POST method to that URI resource. Moreover, it is very important to note that we have to specify the value of a set of key parameters in the body, as it is indicated in [5]:

- *"nsdId".* This key indicates the identification of the NS package that we want to use to create a new NS instance resource. In our case, it is "*4734f40b-85de-4005-94fb-259dd6f15a43*".

- *"nsName".* This is related to the `nsdId`, as it is the name of the NS descriptor that we want to use to create the NS instance resource. In our case it has the value *"generic-vnf_nsd2"*.

- *"nsDescription":* This is just the description of the NS described in the previous bullets. In our case we just set it to *"This is a generic ns instance"*.

- *"vimAccountId".* This is the identifier of the VIM account. The OSM will use this VIM to instantiate the NS on top of a NFVI. In our case its value is *"3c4b3555-7489-457d-9c35-1616483d6915"*.

Also observe that we must indicate that the data in the body is in JSON format. We do that by setting "content-type" and "accept" header keys to the "application/json" value. Finally, running this POST RESTful command yields a new NS instance resource in the OSM, as it is shown in Figure 29. Observe that OSM created the new instance resource, though the status is "scheduled". This means that it is not actually instantiated in the NFVI controlled by the VIM. That is to say, we have just created the NS instance resource. The actual instantiation will be explained in the next RESTful operation. However, note that we need to run this operation because it yields an "id" that is needed when trying to do the instantiation operation. Next, we display the curl code associated to the NS instance resource creation that we have just described:

```
curl -X POST \
 https://10.1.14.248:9999/osm/nslcm/v1/ns_instances \
 -H 'Accept: application/json' \
 -H 'Accept-Encoding: gzip, deflate' \
 -H 'Authorization: Bearer TNm6zBAcozoFQDy9NJcWw9eHwQdglHUI' \
 -H 'Cache-Control: no-cache' \
 -H 'Connection: keep-alive' \
 -H 'Content-Length: 191' \
 -H 'Content-Type: application/json' \
 -H 'Cookie: session_id=c8bc3a82f22cc258ad7da63eb09af9dbc69c5ea3' \
 -H 'Host: 10.1.14.248:9999' \
 -H 'Postman-Token: 5a6f0eb0-1958-4728-a854-d46e265fbfeb,2a6c15ae-e45a-480c-9021-8a7a1f9c2748' \
 -H 'User-Agent: PostmanRuntime/7.20.1' \
 -H 'cache-control: no-cache' \
 -d '{
      "nsdId": "4734f40b-85de-4005-94fb-259dd6f15a43",
      "nsName": "generic-vnf_nsd2",
      "nsDescription": "This is a generic ns instance",
      "vimAccountId": "3c4b3555-7489-457d-9c35-1616483d6915"
}'
```

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

FIGURE 28 POSTMAN CONFIGURATION TO REQUEST THE CREATION OF A NEW NS INSTANCE RESOURCE.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
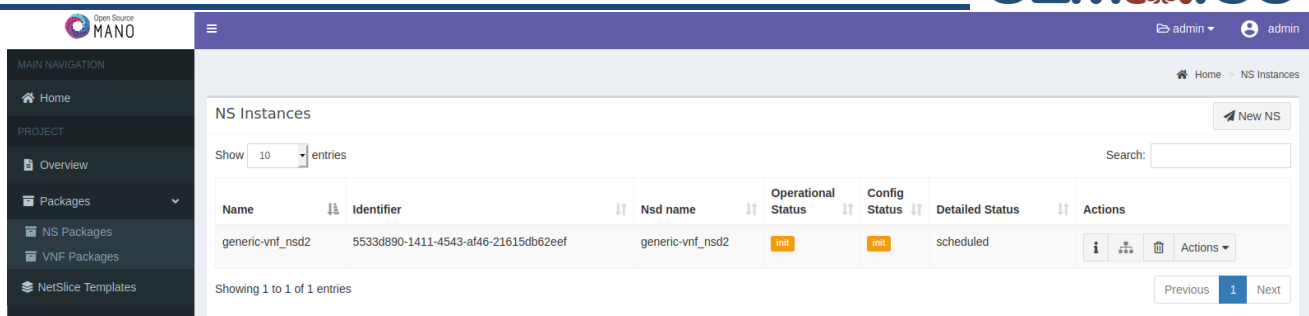
FIGURE 29 CREATION OF A NEW NS INSTANCE RESOURCE IN OSM.
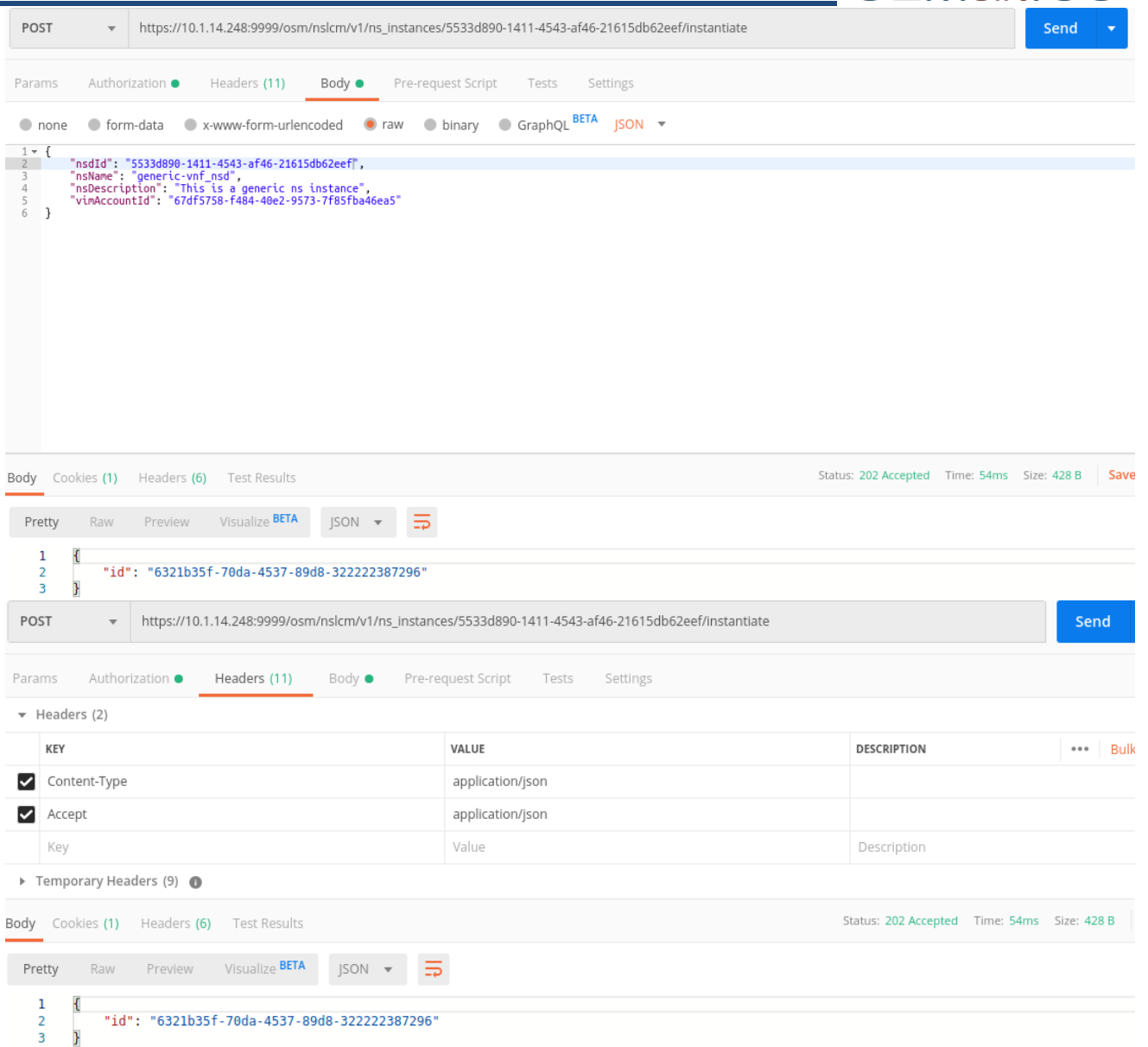
POST method applied to the {apiRoot}/nslcm/v1/ns_instances/{nsInstanceId}/instantiate resource

This operation allows the OSS to order the OSM to trigger the instantiation of a NS. To this end, we need the previous operation, i.e. the one that created the NS instance resource, as it returns the identification of the NS instance resource. This "id" is substituted in the placeholder /{nsInstanceId} of the URI. Therefore, considering the "id" of Figure 28, the complete URI resource to instantiate the NS is:

- *https://10.1.14.248:9999/osm/nslcm/v1/ns_instances/5533d890-1411-4543-af46-21615db62eef/instantiate*

In Figure 30 we show how we used postman to execute the RESTful command that allows to instantiate the NS. Observe that we applied the POST method on the URI resource that we have just described. Also note, that we have to specify a key/value pair in the body. Their meaning is the same than in the previous POST operation. The only difference is that now "nsdid" is set to the NS instance resource obtained in Figure 28. In Figure 31, it is shown that the NS is effectively instantiated. Namely, what is happening in background is that the OSM triggers the instantiation by contacting with the OpenStack VIM, which is the one that manages the NFVI. Then the OpenStack performs the instantiation of the NS on top of the NFVI. Finally, next we display the curl code to perform this NS instance operation:

```
curl -X POST \
  https://10.1.14.248:9999/osm/nslcm/v1/ns_instances/5533d890-1411-4543-af46-21615db62eef/instantiate \
  -H 'Accept: application/json' \
  -H 'Accept-Encoding: gzip, deflate' \
  -H 'Authorization: Bearer TNm6zBAcozoFQDy9NJcWw9eHwQdglHUl' \
  -H 'Cache-Control: no-cache' \
  -H 'Connection: keep-alive' \
  -H 'Content-Length: 190' \
  -H 'Content-Type: application/json' \
  -H 'Cookie: session_id=b54d889744ac37495b344224d040026c0be9aec2' \
  -H 'Host: 10.1.14.248:9999' \
  -H 'Postman-Token: 5d2600fb-cb4a-451f-9270-379713976412,26b38190-2d0e-4ed7-ad53-d17d51b72af7' \
  -H 'User-Agent: PostmanRuntime/7.20.1' \
  -H 'cache-control: no-cache' \
  -d '{
      "nsdId": "5533d890-1411-4543-af46-21615db62eef",
      "nsName": "generic-vnf_nsd",
      "nsDescription": "This is a generic ns instance",
      "vimAccountId": "67df5758-f484-40e2-9573-7f85fba46ea5"
}'
```

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

FIGURE 30 CONFIGURATION OF POSTMAN FOR THE INSTANTIATION OF A NS INSTANCE.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
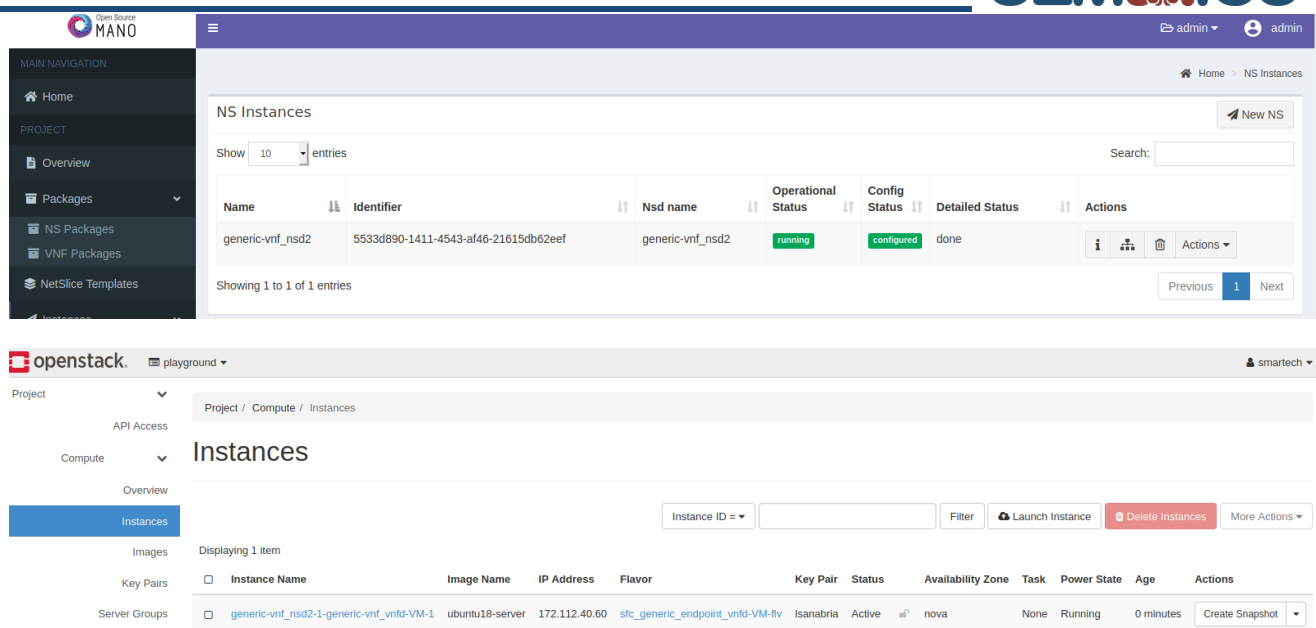
FIGURE 31 CONFIRMATION OF THE NS INSTANTIATION IN OSM AND OPENSTACK.

POST method applied to the {apiRoot}/nslcm/v1/ns_instances/{nsInstanceId}/terminate resource

This operation allows the OSS to terminate a NS instance that is running in OpenStack, through OSM. The "id" that we have to use in the /{nsInstanceId} placeholder is the same than in the previous NS instantiation operation. That is the one that identifies the NS instance resource that had been instantiated. Therefore, the complete URI resource has this expression:

- *https://10.1.14.248:9999/osm/nslcm/v1/ns_instances/5533d890-1411-4543-af46-21615db62eef/terminate*

In Figure 32 we show how we configured postman to execute the NS instance termination operation. Note that we applied the POST method on the above described URI. Also, we specified in the body a set of JSON key/value pairs. Actually, they have almost the same value than in the previous NS instantiation operation above. Only we have a new key "DateTime", which indicates when we want to terminate the NS. By setting it to "0" we are indicating that we want to terminate at this precise moment the NS instance. By executing this operation we can see how the NS has been effectively terminated, see Figure 33. Next, for the sake of completeness we display the curl code to perform the NS termination operation:

```
curl -X POST \
 https://10.1.14.248:9999/osm/nslcm/v1/ns_instances/5533d890-1411-4543-af46-21615db62eef/terminate \
 -H 'Accept: application/json' \
 -H 'Accept-Encoding: gzip, deflate' \
 -H 'Authorization: Bearer K1Z3XBAP0a8Nf0PQT39LYwlGsSJHiVYQ' \
 -H 'Cache-Control: no-cache' \
 -H 'Connection: keep-alive' \
 -H 'Content-Length: 157' \
 -H 'Content-Type: application/json' \
 -H 'Cookie: session_id=0deeebd5e81046d600e55705e2eb5a547b241d33' \
 -H 'Host: 10.1.14.248:9999' \
 -H 'Postman-Token: 312e0eec-40d9-4dd1-b8c2-efe290df437f,de481c0e-e3a5-4c65-9f89-c9bd80fd43f0' \
 -H 'User-Agent: PostmanRuntime/7.20.1' \
 -H 'cache-control: no-cache' \
 -d '{
```

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

> *"nsdId": "5533d890-1411-4543-af46-21615db62eef",*
> *"nsName": "generic-vnf_nsd",*
> *"DateTime": "0",*
> *"vimAccountId": "67df5758-f484-40e2-9573-7f85fba46ea5"*
> *}'*

FIGURE 32 CONFIGURATION OF POSTMAN FOR THE TERMINATION OF A NS INSTANCE.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

FIGURE 33 CONFIRMATION OF THE NS INSTANCE TERMINATION IN OSM.

4.4 Orchestrating a generic Network Service

The NFV Orchestrator leverages a set of software endpoints (or interfaces) in the form of RESTful APIs to realize Network Services (NS). Such interfaces enable the *registration* of (a) VIM(s) and SDN Controllers into the Orchestrator, as well as the specification of the software images which form the basis of the resulting Virtual Network Functions (VNF).

In order to spawn a NS, first, NFVI administrators should specify Network Service descriptors (NSd), which are in turn composed of Virtual Network Function and Virtual Links descriptors (VNFd and VLd, respectively). These descriptors are static YAML files following an ETSI-compliant Information Model (IM) for each of the elements in the NS [3, 4]. For each of the SEMIoTICS use cases, VNFd and VLd should be described and summarized in a NSd. In the following, the process of VIM (OpenStack) registration, and VNF/NS onboarding is described for ETSI's Opensource MANO (OSM). Later, a similar NS is built using lightweight virtualization, that is, Docker containers employing Kubernetes as Orchestrator and VIM. Finally, the benefit and tradeoffs of both approaches with respect to SEMIoTICS are discussed.

4.4.1 A GENERIC VNF-VM EXPOSED THROUGH A ROUTED NETWORK (OSM+OPENSTACK)

The adjective "generic" is conferred to this example because the VNF does effectively nothing. Instead, these sections aim at describing the onboarding and instantiation of a NS.

4.4.1.1 PHYSICAL NETWORK TOPOLOGY AND NFVI

As specified in OSM documentation, the orchestrator should have IP connectivity with both the VIM and the resulting VNFs, while OSM management is realized via a graphical user interface or OSM CLI client as northbound interfaces, as shown in Figure 35.

Assuming a successful installation of OpenStack (VIM) and OSM (NFVO+VNFM) (as shown in Figure 34), the next step is to detail the specifics of each VM composing the NS.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

FIGURE 34 ONBOARDED VIM ACCOUNT: IOTWORLD_OPENSTACK

4.4.1.2 DESCRIPTORS AND ONBOARDING TO OSM

In this generic example, a single default Ubuntu cloud image called `ubuntu` is used. The corresponding VNFd for a `semiotics_generic_vnfd-VM` is shown in Descriptor 1 below.

The VNF should be exposed to the network via a NS. If the VIM was registered to OSM using admin privileges, then the NSd could include arbitrary network names and IPv4 ranges. In the example Descriptor 2 below though, previously mentioned VNF is exposed using an already existing VIM network called `internalNet`. This is particularly relevant when the NFVI owner is not interested in yielding complete control of network resources to tenants, instead it just exposes a set of predefined networks where NS could be spawned.
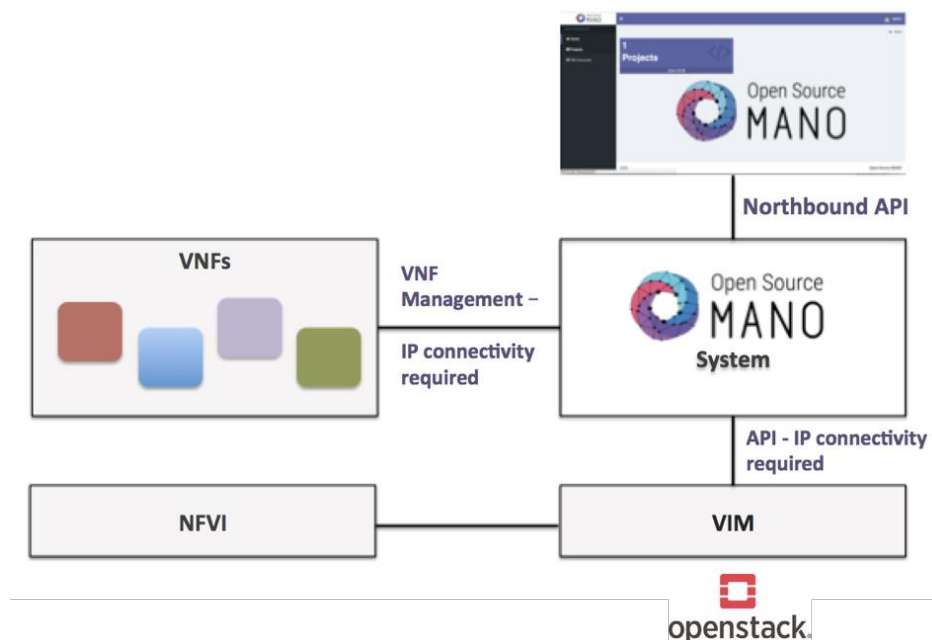

FIGURE 35 OSM TOPOLOGY [31]

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

Once the descriptors are filled with the required information, they should be verified and onboarded to OSM. For this purpose, OSM provides a set of scripts [32] to handle this process and encapsulate the resulting bundle of files for onboarding. The following Figure 36 and Figure 37 show the onboarded VNF and NS descriptors, respectively, through OSM northbound API using a Web browser.
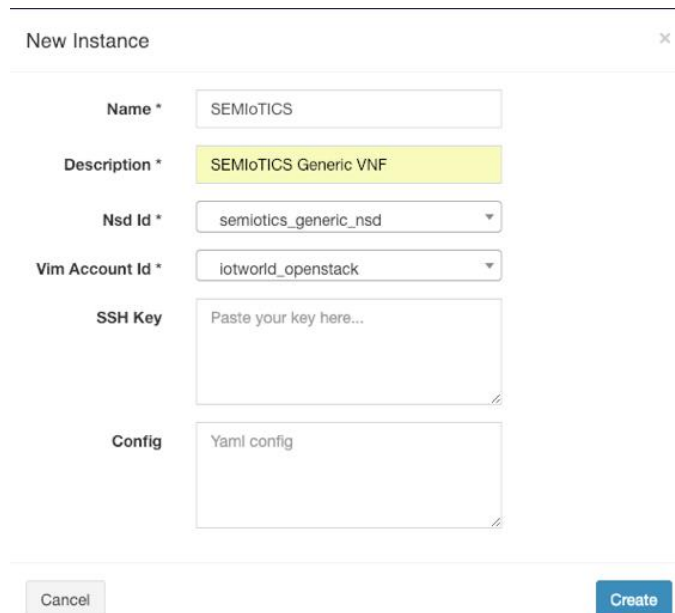
FIGURE 36 VNFD FOR SEMIOTICS_GENERIC_VNFD

FIGURE 37 NSD EXPOSING SEMIOTICS_GENERIC_VNFD-VM

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

4.4.1.3   NS INSTANTIATION

One of the key functionalities of the orchestrator is its ability to re-create/instantiate an onboarded NSd on every registered VIM. In this case, the already-onboarded Descriptor 2 will be instantiated one time on `iotworld_openstack` VIM, as shown in Figure 38.
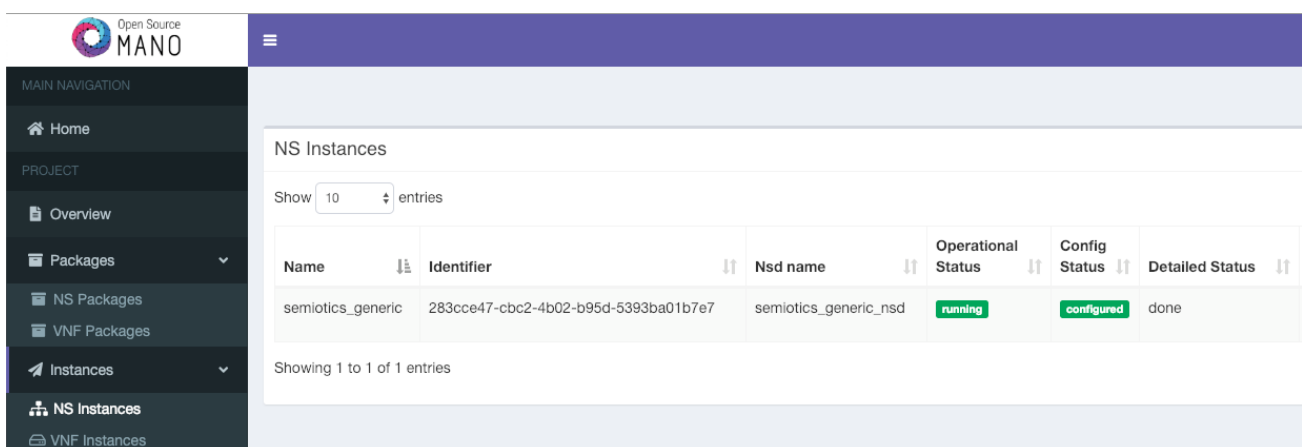

FIGURE 38 NS INSTANTIATION ON VIM

Once the instantiation instruction is executed, OSM will trigger VIM endpoints to relay the information contained in the descriptors (VM specifics, networking, storage). The instantiation process takes approximately 10 seconds, and results could be visualized at OSM GUI, via OSM CLI client, or at the VIM; these are shown by Figure 39, Figure 40, and Figure 41, respectively.

FIGURE 39 OSM GUI: SUCCESSFUL NS INSTANTIATION

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
iotworld@OSMv5:~$ osm ns-list
+-------------------+-------------------------------------+------------------+---------------+-----------------+
| ns instance name | id                                  | operational status | config status | detailed status |
+-------------------+-------------------------------------+------------------+---------------+-----------------+
| semiotics_generic | 283cce47-cbc2-4b02-b95d-5393ba01b7e7 | running          | configured    | done            |
+-------------------+-------------------------------------+------------------+---------------+-----------------+
```

FIGURE 40 OSM CLI CLIENT: SUCCESSFUL NS INSTANTIATION

FIGURE 41 OPENSTACK VIM: SUCCESSFUL NS INSTANTIATED ON VIM-REGISTERED NETWORK

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
vnfd:vnfd-catalog:
    vnfd:
    -   id: semiotics_generic_vnfd
        name: semiotics_generic_vnfd
        short-name: semiotics_generic_vnfd
        description: Generated by OSM package generator
        vendor: OSM
        version: '1.0'

        # Management interface
        mgmt-interface:
            cp: vnf-cp0

        # Atleast one VDU need to be specified
        vdu:
        # Additional VDUs can be created by copying the
        # VDU descriptor below
        -   id: semiotics_generic_vnfd-VM
            name: semiotics_generic_vnfd-VM
            description: semiotics_generic_vnfd-VM
            count: 1

            # Flavour of the VM to be instantiated
            vm-flavor:
                vcpu-count: 2
                memory-mb: 4096
                storage-gb: 10

            # Image including the full path
              # This image should exist at the VIM
            image: 'ubuntu'

            interface:
            # Specify the external interfaces
            # There can be multiple interfaces defined
            -   name: eth0
                type: EXTERNAL
                virtual-interface:
                    type: VIRTIO
                external-connection-point-ref: vnf-cp0

        connection-point:
        -   name: vnf-cp0
            type: VPORT
```

DESCRIPTOR 1 VNFD OF A GENERIC VNF FROM AN UBUNTU CLOUD IMAGE

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
nsd:nsd-catalog:
    nsd:
    -   id: semiotics_generic_nsd
        name: semiotics_generic_nsd
        short-name: semiotics_generic_nsd
        description: Generated by OSM package generator
        vendor: OSM
        version: '1.0'

        # Specify the VNFDs that are part of this NSD
        constituent-vnfd:
            # The member-vnf-index needs to be unique
            # vnfd-id-ref is the id of the VNFD
            # Multiple constituent VNFDs can be specified
        -   member-vnf-index: 1
            vnfd-id-ref: semiotics_generic_vnfd

        vld:
        # Networks for the VNFs
        -   id: semiotics_generic_nsd_vld0
            name: internalNet
            short-name: internal
            type: ELAN
            mgmt-network: 'true'
            vim-network-name: internalNet
            vnfd-connection-point-ref:
            # Specify the constituent VNFs
            # member-vnf-index-ref - entry from constituent vnf
            # vnfd-id-ref - VNFD id
            # vnfd-connection-point-ref
            -   member-vnf-index-ref: 1
                vnfd-id-ref: semiotics_generic_vnfd
                vnfd-connection-point-ref: vnf-cp0
```

DESCRIPTOR 2 NSD EXPOSING SEMIOTICS_GENERIC_VNFD-VM VIA AN EXISTING VIM NETWORK

4.4.2 A GENERIC VNF-DOCKER EXPOSED THROUGH A ROUTED NETWORK (DOCKER+KUBERNETES)

As opposed to the example shown above, this VNF is not a VM but a Docker container. Containers provide much of the desired isolation of VMs but with faster boot time, mostly due to the use of namespace isolation (based on chroot) which bypasses the requirement of spawning a new Kernel for each container (VNF).

This fundamental difference between VMs and containers (Docker) imply different application/VNF design considerations. For instance, a VM is a complete OS environment, whereas a Docker container only includes what the script/applications within it requires, making it very lightweight and fast to orchestrate. There are several alternatives for Docker container orchestration, namely Docker Swarm [33], OpenShift [34], OpenStack Magnum [35], Kubernetes [36], among others. In this section Docker container orchestration will be performed with Kubernetes.

Based on ETSI's NFVI (see Figure 1), it is safe to assume Kubernetes as the complete set of NFV Management and Orchestration components. That is, it takes care of managing the virtualized infrastructure (compute and storage), networking, and VNF lifecycle management. Furthermore, similar endpoints are exposed so external entities (such as OSS/BSS) could collect information from VNFs and the resources being used.

74

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

4.4.2.1 PHYSICAL NETWORK TOPOLOGY AND NFVI

Orchestration with Kubernetes simplifies the physical topology's minimum requirements. The NFVI will be composed of a single Master and a collection of Minion nodes. The Master takes the role of a VIM, VNFM and NFVO; while Minions work as Hardware Resources, i.e. NFVI, refer to Figure 1. The analogy goes a long way, for instance, gathering container information must be done by triggering the Master's corresponding endpoints (RESTful APIs), and Docker containers are spawned on top of Minions. Networking among containers (or pods) within Kubernetes is also software-defined, often dubbed Cluster Networking [37].

4.4.2.2 DEPLOYMENTS AND SERVICES AS DESCRIPTORS

Contrary to OSM, the instructions on how to build a container from an image and how to expose it to the network, do not need to be previously onboarded to the NFVO. Instead, in Kubernetes the analogous to descriptors are YAML files that follow specific Kubernetes APIs. There are APIs for every aspect concerning an application/VNF deployment, e.g.: deployment (pods, containers), services (networking exposure), volumes (storage), volume claims, labels, and much more [38].

The following Descriptor 3 shows a Docker file. This file is used to build a Docker image[11] called `semiotics/restAPI:v0.1`. Then, Descriptor 4 shows a Kubernetes deployment file, and Descriptor 5 shows a Kubernetes service file.

```
#Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY test.py requirements.txt /app/

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 5200 available to the world outside this container
EXPOSE 5200

# Run test.py when the container launches
CMD ["python", "test.py"]
```

DESCRIPTOR 3 DOCKERFILE TO CREATE AN IMAGE. WHEN RUN, THE CONTAINER WILL BOOT EXECUTING "TEST.PY"

As can be read in Descriptor 4, it specifies labels, anti-affinity rules (even-though empty in this example), as well as a cap in the amount of resources requested during execution. Similar control over the VNF resources can be obtained with OSM+OpenStack. Furthermore, Descriptor 5 details the networking aspects of the VNF, that is, how could it be reached from outside the cluster (this is specified as NodePort type, but administrators could also expose ClusterIPs which are only reachable by pods within the cluster). The aforementioned descriptors are used for orchestrating a Docker container on Minion nodes.

4.4.3 VMs OR DOCKER CONTAINERS FOR SEMIOTICS

SEMIoTICS seeks to provide optimization at various levels of a NFVI (at field, network and cloud layers). That is, better networking routes, VNF scaling, and the concatenation of VNFs at different layers to form Service Function Chains (SFC) as NS. Focusing on the latter, SFC require close cooperation among compute, storage and networking controllers in order to route traffic to the specific VNFs composing the SFC.

---

[11] Docker containers are the runtime version of Docker images.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

Despite the apparent benefits provided by the fast instantiation of Docker containers, the concept of networking VNFs and SFC is not thoroughly supported in Kubernetes. Let networking VNFs refer to containerized routers, switches, or another customized virtual network element. If such a type of VNF would require specific kernel modules, it could only be orchestrated on top of Minion nodes whose kernel is modified in the same manner. This is due to the nature of Docker containers, i.e. containers run a subset of the host's Kernel. This fact imposes a limitation for NS, unnecessarily tying VNFs to specific nodes[12] and hindering the flexibility of the NFVI.

```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: semiotics
spec:
  selector:
    matchLabels:
      run: semiotics
  replicas: 1
  template:
    metadata:
      labels:
        run: semiotics
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: "kubernetes.io/hostname"
                operator: NotIn
                values: [""]
      containers:
        - name: semiotics
          image: semiotics/restAPI:v0.1
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5200
              name: flask-port
          resources:
            limits:
              memory: "100Mi"
              cpu: "2"
            requests:
              memory: "50Mi"
              cpu: "1"
```

DESCRIPTOR 4 KUBERNETES DEPLOYMENT FILE. SPECIFYING NODE ANTI-AFFINITY FIELD AND RESOURCE LIMITATIONS AS EXAMPLES

---

[12] There are cases where VNFs are spawned at specific nodes, e.g.: when using specific hardware, or for reducing delay by placing the VNF physically closer to where it is needed.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
apiVersion: v1
kind: Service
metadata:
  name: semiotics
spec:
  type: NodePort
  ports:
    - port: 5200
      targetPort: 5200
      protocol: TCP
      name: flask-port
  selector:
    run: semiotics
  externalIPs: ["192.168.2.102"]
```
DESCRIPTOR 5 KUBERNETES SERVICE FILE. IT EXPOSES THE DEPLOYMENT "SEMIOTICS" VIA AN SPECIFIC IP AND TCP PORT

On the other hand, VMs as VNF do not suffer from such limitation. Moreover, network controllers for OpenStack (Neutron), external SDN Controllers (such as OpenDaylight and the SSC), and OSM have extensive support for SFC and other 5G technologies such as network slicing. Therefore, it is recommended to continue development of the SEMIoTICS architecture employing the ETSI-compliant combination of OpenStack+OSM.

Having clarified the above, it is also valid to highlight the possible benefits offered by Kubernetes as a backend/cloud application orchestrator. It offers similar virtualization capabilities and management tools and has been widely adopted as an agile platform for constantly improve and constantly develop (CI/CD) web applications on top of a virtual environment (e.g.: Docker containers). Moreover, it provides tools for live updating the characteristics of a deployment (e.g.: scale up/down) with virtually no down-time; a feature that is still to be implemented in OSM[13]. In the end, NFV-MANO as proposed for SEMIoTICS could support Kubernetes at the backend/cloud level as several VNFs belonging to the same tenant network.

4.5 Dynamic management of the NFV resources

4.5.1 SoA ON NFV RESOURCE ALLOCATION

As it has been mentioned above, NS are deployed in NFV as a chain of network functions or VNFs. That chain is so-called SFC. Moreover, VNFs are deployed on top of the NFVI. This means that each VNF is executed within a VM, or other type of virtual environments such as containers, that provide virtual computing, storage and communications resources to execute the VNF properly. Thereby, this section deals with two fundamental questions from an NVF resource management viewpoint [39] [40] [41].

The first one, is where it is more convenient to deploy the VNF from a QoS point of view. Or in other words, in which VM is better to place a given VNF. Thereby, this problem is so-called VNF placement. This VM can be physically located in any part of the network that allows virtualization of its resources. For instance, in SEMIoTICS the VM could be placed at the edge of the network, i.e. at the IoT Gateway, or at the backend cloud.

The second problem treated herein is explained as follows in the form of two statements. First, to determine how many virtual computing resources are assigned to the VM to execute the VNFs. Second, to decide how many virtual communication resources are assigned for the communication between VMs. The VNF placement along with the allocation of virtual computing and storage resources determine the QoS that the NFV provides to a network service. Thereby, in the sequel we deal with optimal allocation of the NFV resource from a network service QoS point of view. In SEMIoTICS, this QoS is determined for instance by a low latency and a reliable communication. Next, the SoA on NFV resource allocation is reviewed.

---
[13] OSM release FIVE.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

First, there are several works that consider geographically distributed clouds to deploy the VMs [42] [43] [44]. The aim of those works is to minimize the operational resource cost to run the service while satisfying QoS constraints related to the service level agreement (SLA), e.g. the maximum delay between the data center and the user. These works only consider that the VMs are deployed in the backend cloud. However, SEMIoTICS considers a two-tier cloud architecture where VMs can be deployed either at the backend cloud or at the network edge, i.e. at the IoT Gateway.

Several works consider this two-tier cloud architecture, e.g. [40] [45]. The most interesting for our purposes is [40]. Namely, [40] treats the problem of placing the VNFs either at the backend cloud or at the cloudlet, i.e. at the edge. Also, they deal with the allocation of computing resources to the VMs that run the VNFs. To this end, they decide the number of CPU cores allocated to the VM running the VNF. In order to decide the VNF placement and the allocation of computing resources they consider an optimization problem that has the next terms in the objective function:

- Minimize the maximum utilization of computing resources of the cloudlet.
- Minimize the amount of computing resources allocated in the backend cloud.
- Minimize the QoS violations. Namely, they consider a QoS model based on the maximum delay acceptable for different traffic types. And consider that a QoS violation occurs when a function of the VNF processing delay exceeds that maximum delay.

In the constraints of their optimization problem, it is worth mentioning a bound on the VNF processing delay related to the SLA agreement. Finally, they show that their optimization problem belongs to the class of Mixed Integer Linear Programming (MILP) problems. Therefore, [40] is interesting but has several drawbacks. First, they do not decide the allocation of virtual communication resources that are needed in the interplay between different VMs of an SFC. Second, they obtain an analytic expression to quantify the delay due to VNF processing, which is based on just an average response time. Namely, it is obtained by modeling the VM as an M/M/1 queue. Also, in this regard, they obtain VNF placement and allocation decisions that are static. That is, the optimization problem is solved without taking into account any kind of temporal or random variations due to the state of the network resources, the state of the computing resources or the services requests. Third, the complexity of a MILP grows quickly as the problem size increases, namely it is an NP-complete problem. Thereby, a MILP problem is not scalable, which is a severe issue for SEMIoTICS, as the optimization problem can have a high dimension due to the massive amount of IoT devices. Last but not least, they do not consider past data to take the resource allocation decisions. This past data can be related to the rate of services request or the state of the network resources. They determine past allocation decisions from which the algorithm could learn the optimal allocation decisions in future time slots.

Another interesting approach for NFV resource allocation is proposed in [41] [46]. This approach solves the drawbacks of [40] as we will see next. The approach proposed in [41] [46] considers time slots to perform the resource allocation task. That is, at each time slot they decide the VNF placement along with the amount of virtual computing and communication resources allocated to the VMs that run the VNFs. Moreover, they consider that there are sources of randomness that affect the resource allocation decision:

- They assume that the virtual computing and communication resources have a time-varying cost, which is parameterized by random parameters that can vary at each time slot. For instance, this is the case when the SFC is deployed in an external cloud and the cloud provider charges a cost for the use of its resources.
- Furthermore, they also consider that another source of randomness is the arrival rate of new services requests at each time slot.

All these sources of randomness are stacked in a state vector. This approach based on carrying out the resource allocation decisions at each time-slot is interesting for the SEMIoTICS purposes. The reason is that we are adapting the allocation decisions to the state of the network and to new network services requests at each time slot, rather than just a static decision as e.g. in [40]. This is particularly, interesting because the IoT data has a streaming and dynamic nature.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

Following with the approach in [46] [41], it is important to explain the system model that they consider. Namely, as it is shown in Figure 42 they assume that each VM runs a given VNF. Also, they model the network services, i.e. the SFC, as a permuted sequence of VNFs, e.g. one service may require $\{f_1, f_2, f_3\}$, whereas another one $\{f_2, f_3, f_1\}$, being f_k the k-th VNF. Furthermore, at the VM there are two types of queues:

- Incoming queues that store the sequence of VNFs to be processed or routed to other VM because they cannot process any of the VNFs in the sequence.
- Outgoing queue that stores the VNF that has been processed along with the other VNFs of the SFC that have to be processed by other VMs. That is, this queue will route the VNF sequence to other VMs to process the remaining VNFs.
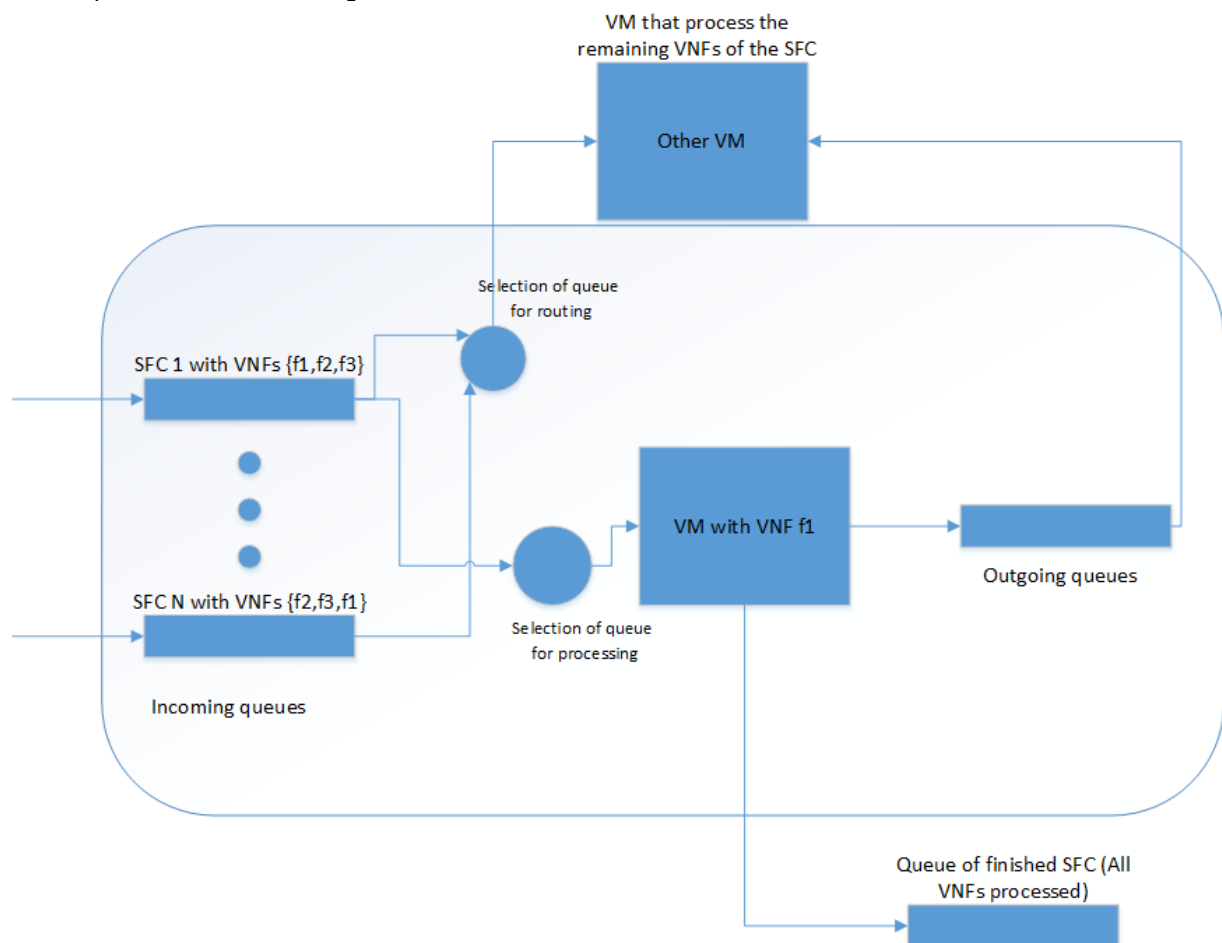
FIGURE 42 MODEL FOR VNF PROCESSING AT THE VMS

These queues follow a recursion model that varies at each time slot. In the case of the incoming queue the terms of the recursion are:

- The state of the queue in the previous time slot.
- The processing rate assigned to process a given VNF at the current VM, which leads to diminish the queue length.
- The communication rate assigned to route the services that cannot be processed at the current VM, which leads to diminish the queue length.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

- The communication rate assigned to the neighboring VMs to route the services that have been partially processed and that may require further processing at the current VM. This leads to increase the queue length.
- The rate of new service arrivals. This leads to increase the queue length.
- The communication rate assigned to the neighboring VMs to route the services that could not be processed. This leads to increase the queue length.

In the case of the outgoing queue, the terms of the recursion are:

- The state of the queue in the previous time slot.
- The processing rate assigned to process a given VNF at the current VM, which leads to increase the queue length.
- The communication rate assigned to route the services that have been partially processed at the current VM and that may require further processing at the neighboring VMs. This leads to decrease the queue length.

Thereby, given the system model described above, [41] [46] pose the NFV resource allocation in terms of a stochastic network optimization problem [47]. This class of optimization problems considers stochastic objective functions and constraints. Moreover, their aim is to minimize a time average objective function for all the time slots subject to time average constraints related to the stability of the network queues [47]. Thereby, in [41] [46] the aim is to find the VNF placements and the allocation of virtual resources that minimize the time-average cost of running the requested SFC in the NFV platform subject to the next constraints:

- Stability of the network queues, which is a time average constraint.
- The queue recursion model explained above.
- Constraints on the processing rates and the communication rates.

This optimization problem has the next generic mathematical expression:

$$\min_{\{x_t, \forall t\}} \lim_{T \to \infty} 1/T \sum_{t=1}^{T} E[\varphi_t(x_t)]$$
$$s.t. \ \ 0 \le x_t \le x_{max}$$
$$q_{t+1} = [q_t + f(x_t) - g(x_t)]^+ \tag{1}$$
$$\lim_{T \to \infty} 1/T \sum_{t=1}^{T} E[q_t] < \infty.$$

Where x_t stacks the VNF placement variable and the virtual computing and storage resources. The function $\varphi_t(x_t)$ is the cost that the platform provider charges for using its resources. The vector q_t stacks the incoming and outgoing queue lengths of the VMs. $f(x_t)$ and $g(x_t)$ are generic function accounting for the increment or decrement of the queue lengths, see above. And the last line of the optimization problem accounts for the stability of the network queues. Last but not least, in order to solve the optimization problem in (1), the authors in [41] [46] follow a data-driven online learning approach. Namely, they are able to solve the problem at each time slot by resorting to the Lagrange dual problem and then, they learn the Lagrange multipliers by using past samples from the state vector. Thereby, rather than in [40] the approach proposed in [41] [46] is data-driven, i.e. it uses past data on the state of the network to decide the allocation of resources. Also another important property of [41] [46] for the SEMIoTICS purposes is that it is a scalable algorithm. That is, to solve the optimization problem they consider that the problem can be high-dimensional, and they employ a type of stochastic gradient average method called SAGA to solve iteratively the problem. The SAGA algorithm has been precisely designed to cope with high dimensional optimization problems [48]. The data-driven online learning approach for NFV resource allocation proposed in [41] [46] is very interesting for the SEMIoTICS purposes. This is because it provides dynamic allocation decisions per time slot, thereby it can adapt to the network state and new network services requirements. This is particularly important due to the heterogeneous and dynamic nature of IoT data. It also learns from the past information on the state of the network. And it takes into account that the problem is high-dimensional, which is another feature of IoT. However, it has an important drawback for SEMIoTICS, because it does not incorporate QoS such as low latency requirements

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

in the optimization problem. In other words, for SEMIoTICS it makes more sense to try to optimize a functional related to the QoS rather than the cost that the infrastructure provider charges for the use of their resources. Or alternatively, to have QoS constraints in the optimization problem.

A more practical approach to deal with the dynamic resource allocation problem in NFV is introduced next and developed in detail in section 4.5.2. This is actually the current SoA to manage dynamically the NFV resources in the implementation of the NFV MANO based on OSM [49]. Thereby, this is the approach that will be considered herein, see section 4.5.2. First, it is worth mentioning that the NFV MANO provides internally a mechanism to manage the virtual resources exposed by the NFVI through the VIM. That is the instantiation, scaling or release of virtual resources assigned to run a NS and the corresponding chain of VNFs, see [4]. The NFV MANO permits to configure or set parameters to control this internal behavior. For instance, it considers threshold parameters that trigger the scaling of virtual computing resources. The NFV MANO allows the users to set up these thresholds through northbound APIs that are so-called descriptors, e.g. network service descriptors.

Last, but not least, it is worth mentioning that the NFV MANO contemplates the possibility that an authorized external entity controls the network service lifecycle management, which includes the proper resource management to run the network service such as the scaling of resources, see [4]. Namely, according to section 7.1.2 in [4], this corresponds to one of the NFV MANO interfaces that is so-called Network Service Lifecycle Management interface. This interface uses the Os-Ma-nfvo reference point described in Figure 1 and permits an external OSS to manage the NFV resources. That is, first the OSS will ask the NFV MANO to obtain metrics about the network service state and the network resources state. Then, the OSS will send control actions to manage the virtual resources that are used to run the network service.

4.5.2   DYNAMIC SCALE OUT OF VNF INSTANCES: A THRESHOLD-BASED APPROACH

Scaling out operations refer to the creation of replicas of a determined VNF. These operations are of particular use when implemented on servers behind a load balancer, or to complement any computation operation with additional workers. In SEMIoTICS, scale out operations are orchestrated by the NFVO, which is instructed before-hand via VNF descriptors the manner of the scaling. That is, which are the available metrics to look at from the VIM telemetry services, and what are the corresponding thresholds that would unleash a scale out, or a scale in[14].

Configuration 7 shows a section of a VNF descriptor that specifies the scaling out criteria, as well as the monitoring parameter that is being watched by the NFVO to comply with such criteria. In summary, the scaling out operation is triggered when the `metric_vim_vnf1_cpu_util` is greater than (GT) `scale-out-threshold` (70%) during `threshold-time` (10) seconds. Conversely, a scale in operation is performed on a replica VNF when the aforementioned metric is detected to be lower than (LT) `scale-in-threshold` (20%) for `cooldown-time` (20) seconds.

Figure 43 shows a sample OSM metrics dashboard containing panels measuring VNF's CPU and Memory usage (as defined in Configuration 7). It can be seen in the figure how the `semiotics_scale_out_1_fast-scale-out-cpu_vnfd-VM-1` CPU utilization metric is maxed out for a period of time, which according to the scaling out rules should trigger the creation of a replica VNF. VM-2 and VM-3 are then created automatically and their respective metrics also appear in the figure[15]. After the scale out operation, the resulting network service topology is shown in Figure 44. Replica VNFs will be scaled in once the observed metric goes below the corresponding threshold during cooldown-time seconds.

---

[14] As scale out creates replicas of a VNF, scale in removes such duplicates according to thresholds and a so-called *'cool down'* period.
[15] Even though VM-2 and VM-3 metrics appear in the dashboard at the same time, this does not mean the VMs were in fact created simultaneously.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
scaling-group-descriptor:
-   name: "scale_vdu_autoscale"
    min-instance-count: 0
    max-instance-count: 2
    scaling-policy:
    -   name: "scale_cpu_util_above_threshold"
        scaling-type: "automatic"
        threshold-time: 10
        cooldown-time: 20
        scaling-criteria:
        -   name: "scale_cpu_util_above_threshold"
            scale-in-threshold: 20
            scale-in-relational-operation: "LT"
            scale-out-threshold: 70
            scale-out-relational-operation: "GT"
            vnf-monitoring-param-ref: "metric_vim_vnf1_cpu_util"
    vdu:
    -   vdu-id-ref: fast-scale-out-cpu_vnfd-VM
        count: 1
monitoring-param:
-   id: "metric_vim_vnf1_memory"
    name: "metric_vim_vnf1_memory"
    aggregation-type: AVERAGE
    vdu-monitoring-param:
        vdu-ref: "fast-scale-out-cpu_vnfd-VM"
        vdu-monitoring-param-ref: "metric_vdu1_memory"
-   id: "metric_vim_vnf1_cpu_util"
    name: "metric_vim_vnf1_cpu_util"
    aggregation-type: AVERAGE
    vdu-monitoring-param:
        vdu-ref: "fast-scale-out-cpu_vnfd-VM"
        vdu-monitoring-param-ref: "metric_vdu1_cpu_util"
```

CONFIGURATION 7 MONITORING AND SCALING PARAMETERS INSIDE VNFD
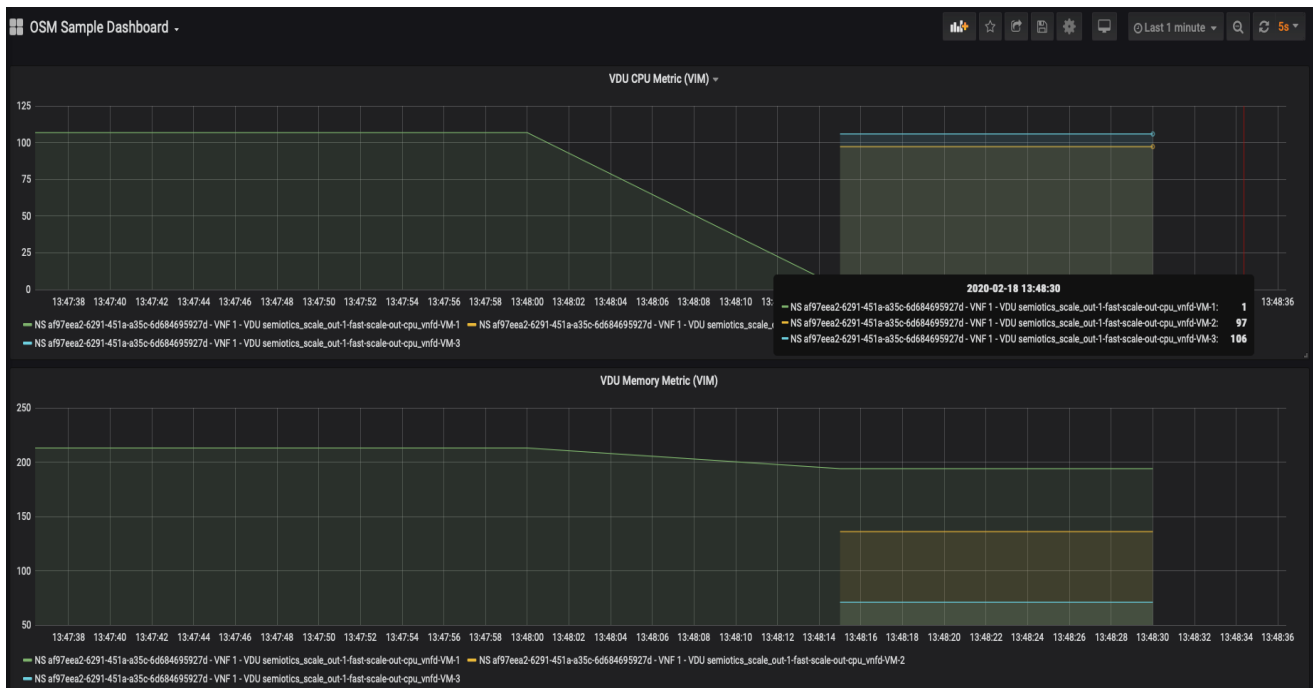
FIGURE 43 SCALED OUT VNF METRICS AS SEEN BY NFVO

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
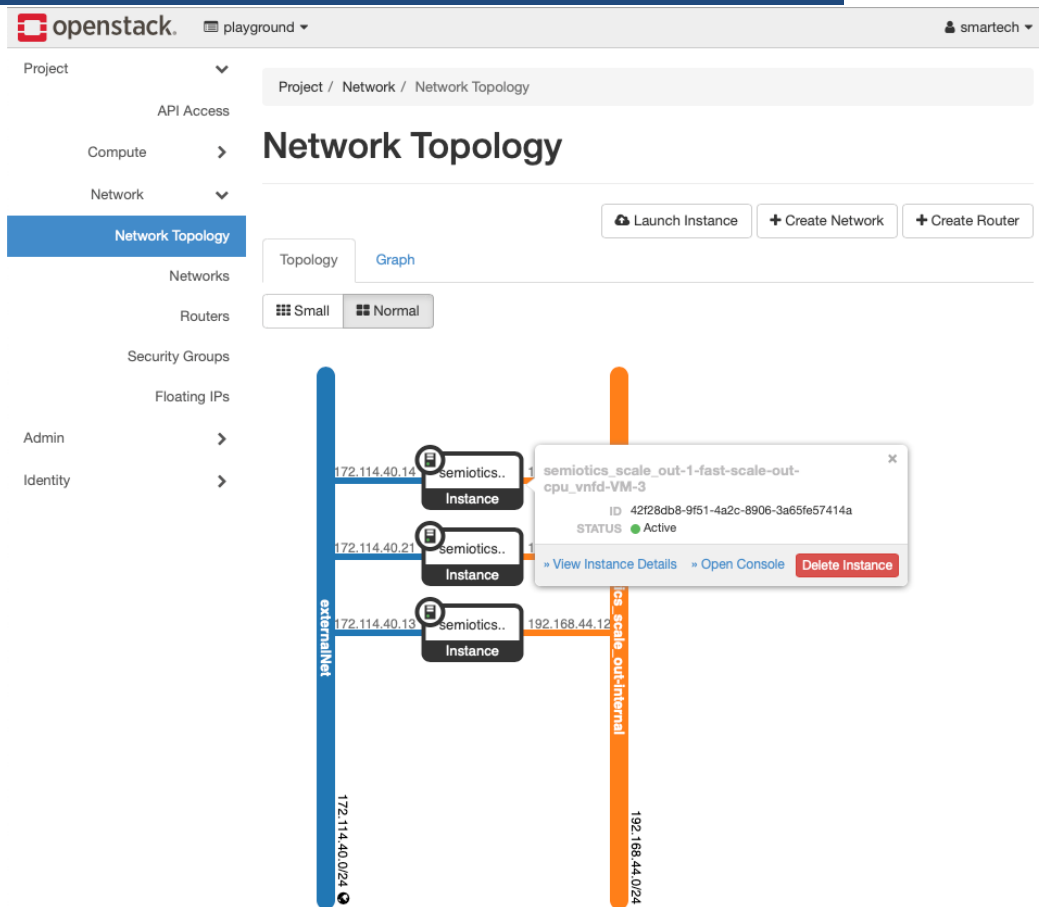
FIGURE 44 TOPOLOGY AFTER SCALE OUT

4.5.3 LOAD BALANCING

The previous sections highlight that the dynamic management of NFV resources, e.g. the computing processing rate, implies the scaling out of VNF instances. This means that the underlying application embedded in a VNF may require dynamically more computing resources. To face this challenge multiple VNF instances, each with the same underlying application, are deployed. However, this scenario poses the next question. How do we distribute the incoming traffic among the VNF instances? This question is solved by the so-called load balancer functional block, whose role is precisely to take that decision.

In Figure 45, we show the role of the load balancer within the NVF ecosystem. The load balancer can be deployed as the functionality of a VNF. It just accepts incoming traffic and decides how to split it among a set of N VNF instances. All these VNFs run on top of an NFVI, which is controlled by a VIM e.g. OpenStack. Also, the lifecycle management of the VNFs and the scaling operations are controlled through the NFV MANO, e.g. the OSM.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public
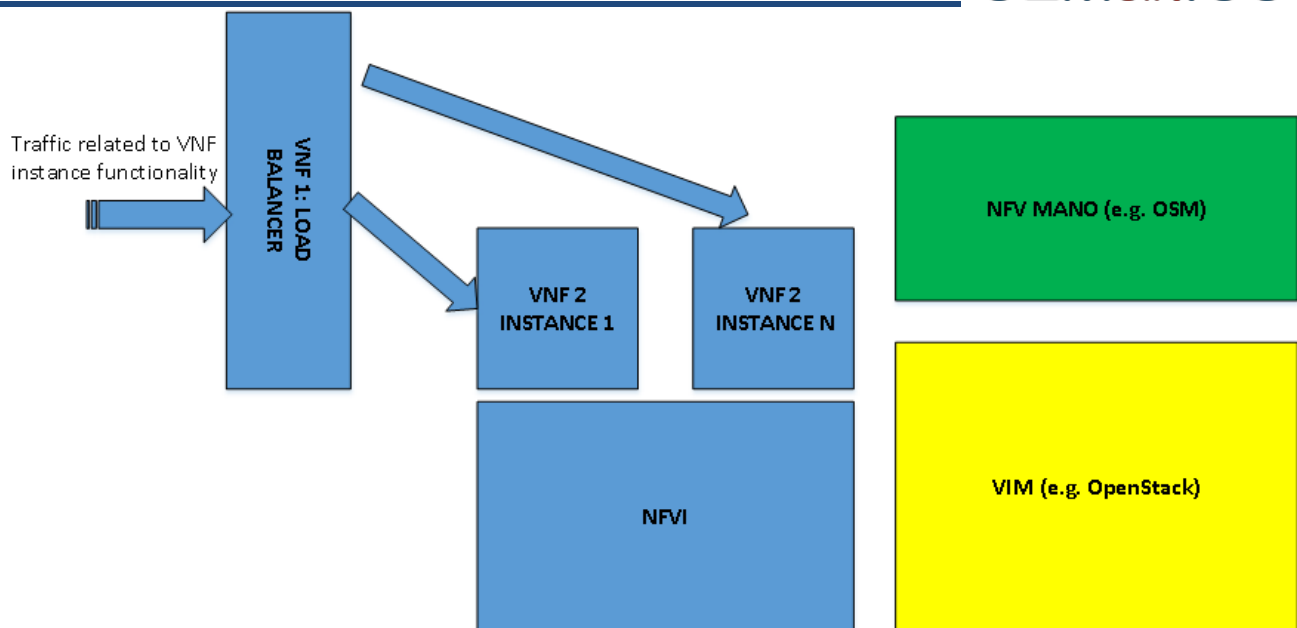
FIGURE 45 LOAD BALANCER TO DISTRIBUTE THE TRAFFIC AMONG THE VNF INSTANCES.

There are several open source implementations of load balancers, e.g. Nginx or HAProxy are among the most popular ones [50]. It is important to point out that load balancers are within the context of the OSI layers 4 and 7. Next, for illustration purposes, we will show a load balancing experiment that uses Nginx [51] as the load balancer.

In this experiment, Nginx implements several functionalities. It is a web server, a load balancer and a reverse proxy server. Namely, it accepts http requests on the port 8080 from web browser clients. Then, to process these requests it distributes the incoming traffic to several backend web app servers. Each backend server has the same functionality, i.e. they are just instances. The backend servers process the traffic and send the response to the Nginx, i.e. the web server. And finally, the Nginx sends the response to the client, i.e. the web browser client. For illustration purposes, we consider that the backend instances receive the traffic on the port 3000. And these backend servers run a node.js code that just returns a "hello world" type message to the client.

For illustration purposes, to carry out this experiment we used docker and docker-compose. More specifically, in a docker container we deployed a Nginx image and we exposed the port 8080. The Docker file reads:

```
# Use the standard Nginx image from Docker Hub
FROM nginx

# Copy custom configuration file from the current directory
COPY nginx.conf /etc/nginx/nginx.conf

EXPOSE 8080

# Start Nginx when the container has provisioned.
CMD ["nginx", "-g", "daemon off;"]
```

Moreover, to implement the load balancing scenario described above, we have to write a configuration file to control how the Nginx works. This configuration file is so-called Nginx.conf:

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
events { worker_connections 1024; }

http {

 resolver 127.0.0.11 valid=10s;

 server {
    listen 8080;
    server_name localhost;

    location / {
       set $web_var webapp;
       proxy_pass http://$web_var:3000;
       proxy_set_header Host $host;
    }
  }
}
```

The *proxy_pass* directive indicates the host direction of the upstream or backend servers, i.e. the multiple instances that implement the web app and that permit to balance the load. It is important to define it as a variable, because in this way Nginx updates dynamically the list of upstream servers. Otherwise, it won't discover a server that joined dynamically the net [52]. Also note that we have to use the same network alias for all the upstream servers. In this case "webapp", otherwise it is not possible to implement the dynamic load balancer, as we cannot discover new servers.

Also, it is mandatory to specify a DNS server through the *resolver* directive. Nginx will ping this DNS server to resolve the hosts' names of the upstream servers [52]. Note that 127.0.0.11 is the Docker embedded DNS server. In this regard, it is worth mentioning that in a load balancing experiment using OpenStack we should substitute this DNS server by the one provided by OpenStack, e.g. OpenStack Designate that is a DNS as a service component for OpenStack [53]. Finally, in the Nginx.conf file above, the flag valid=10s specifies the refresh rate to rediscover new hosts [52].

Following with the experiment setup, each backend server instance that implements the same web app, is implemented as a docker container. To this end, we used the next docker file to implement an instance of the web app server within the container:

```
# Use a standard Node.js image from Docker Hub
FROM node:boron

# Create a directory in the container where the code will be placed
RUN mkdir -p /backend-dir-inside-container

# Set this as the default, working directory.
# We'll land here when we SSH into the container.
WORKDIR /backend-dir-inside-container

# Copy all the node.js code inside src to our directory in the container
ADD ./src /backend-dir-inside-container

# Our Nginx container will forward HTTP traffic to containers of
# this image via port 3000.
EXPOSE 3000
```

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

 # This executes the node.js code that implements the web app
CMD ["node", "index.js"]

Note that the docker container for the load balancer (Nginx) and the instances of the backend servers can be implemented as VNFs as well, the concept is the same. At this point, in order to deploy and configure the docker containers that implement our load balancing experiment, we used docker-compose. More specifically, the docker-compose yaml file reads as follows:

```
version: '3.5'
services:
  backend:
      build:
        context: ./backend
        dockerfile: Dockerfile
      tty: true
      networks:
        load_balancer:
          aliases:
            - webapp
      volumes:
        - './backend/src:/backend-dir-inside-container'

  loadbalancer:
      build:
        context: ./load-balancer
        dockerfile: Dockerfile
      tty: true
      links:
          - backend
      ports:
          - '8080:8080'
      networks:
        load_balancer:
      volumes:
          - './load-balancer:/load-balancer-inside-container'

  volumes:
    backend:

  networks:
    load_balancer:
      name: LB_net
      driver: bridge
```

We can see that this docker-compose file is basically setting up two type of services, i.e. docker containers. On the one hand the Nginx that implements the web server and the load balancer. On the other hand, the backend server that implement the web app. Following with the description of this docker-compose file, in the backend services we must specify the same alias for all the backend services. In this case it is called webapp. Note, that we will run several backend services with the same alias webapp when we run the docker-compose, through the –scale option. E.g., "docker-compose up --scale backend=5 –build" builds and runs 5 backend services.

Also, it is important to mention that both the backend services and the nginx belong to the same docker network. In this case, they will belong to the LB_net network. Note that we can specify the name of the network, but we have to be careful to put in the beginning of the document version: '3.5', since in previous versions this option was not available [54].

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

FIGURE 46 DEPLOYING THE NGINX WEBSERVER/LOAD BALANCER ALONG WITH THE BACKEND SERVER INSTANCES.

At this point, we are ready to deploy the docker containing the Ningx, i.e. the webserver along with the load balancer, and 5 docker containers that are 5 instances of the backend web app servers. We use the docker-compose file described above, and in a new bash terminal we run this docker-compose command:

- docker-compose up --scale backend=5 –build

The result can be observed in Figure 46. We can see that effectively the nginx was deployed successfully, this is the label "loadbalancer_1". Also, we can see that we have deployed 5 backend server instances, which have the label "backend_x", where x belong to the set {1,…,5}. Also, we can see that the backend server instances are effectively waiting for traffic on port 3000.

Then, we open a web browser client, we point to this direction http://localhost:8080/, and we refresh the web browser. This is the incoming traffic and the Nginx serves this traffic by hearing on port 8080 and then by balancing the load among the backend servers. In we can see that effectively the requests are balanced among the backend server instances, i.e. we are balancing the traffic load as we wanted.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

FIGURE 47 NGINX PERFORMS THE LOAD BALANCING AMONG THE BACKEND SERVER INSTANCES.

Next, we want to demonstrate that we can balance the load dynamically. That is, if more traffic arrives or a new backend server instance is created, then the Nginx is capable of taking into account the new instance and send traffic to it. To this end, we created a new docker-compose file that deploys a docker container with the new backend server instance. It also allows this docker to join the existing network "LB_net", where the Nginx and the other backend servers lie. The docker-compose file has the following content:

```
version: '3.5'
services:
  backend_new:
      build:
        context: ./backend
        dockerfile: Dockerfile
      tty: true
      networks:
        load_balancer:
          aliases:
            - webapp
      volumes:
        - './backend/src:/backend-dir-inside-container'

  volumes:
    backend_new:
```
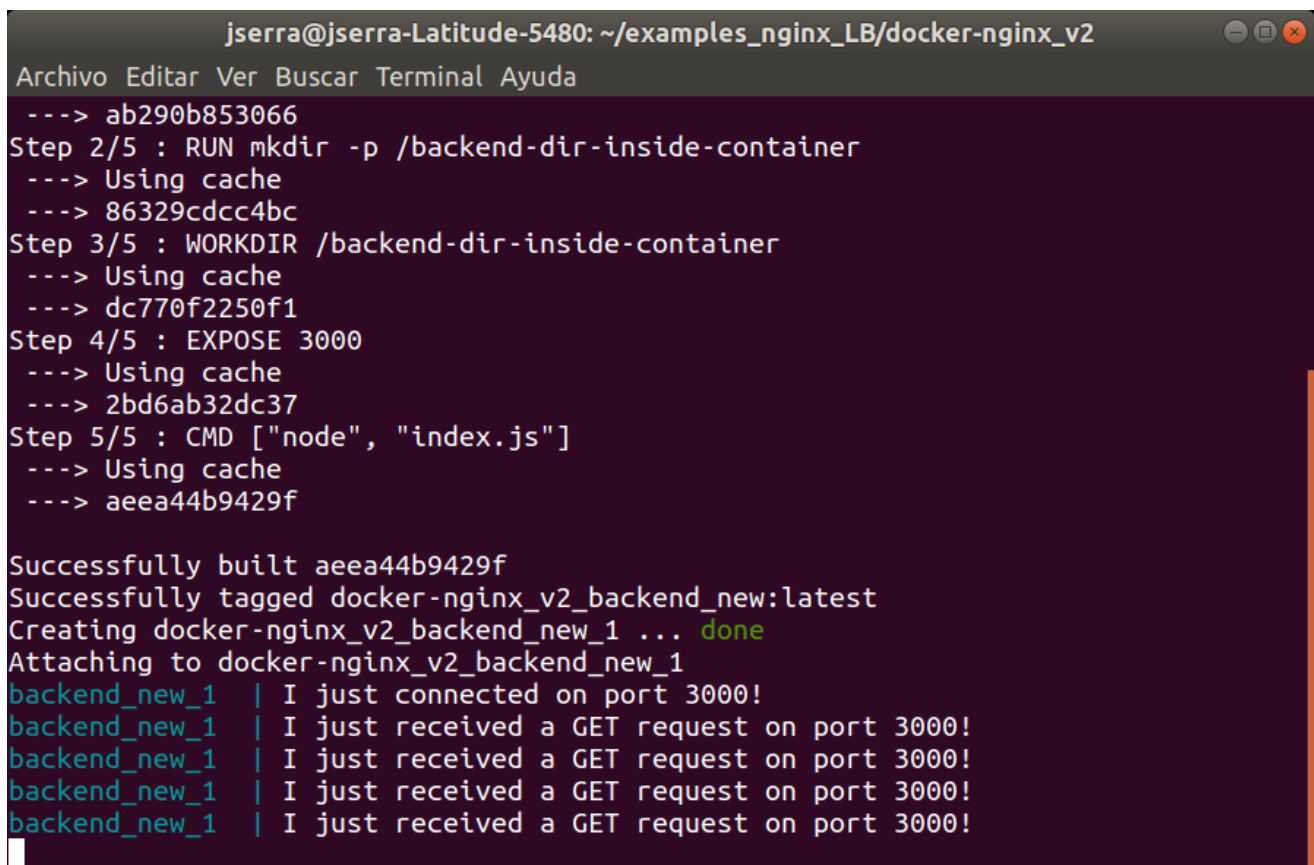
780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

```
networks:
  load_balancer:
    external:
      name: LB_net
```

Therefore, to deploy the new backend instance, we run this docker-compose command:

- *docker-compose -f docker-compose_addcont.yml up –build*

Then, we keep sending new traffic requests to the Nginx, by refreshing the web browser client. In Figure 48 we can see that the Nginx discovers the new backend server instance and that sends some traffic requests to it. Obviously, the rest of the traffic requests are balanced among the former backend server instances. Therefore, we attained our objective consisting of balancing the traffic load dynamically.

FIGURE 48 NGINX PERFORMS DYNAMIC LOAD BALANCING DISCOVERING A NEW BACKEND SERVER INSTANCE.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

5   NFV INTERFACES WITHIN THE SEMIoTICS FRAMEWORK

This section deals with the description of interfaces among the blocks of an NFV platform. To this end, and in the line of the approach proposed in the previous sections, the ETSI NFV specification is taken into account [4] [2]. Thereby, these interfaces are the ones that were mentioned in section 1.2 and that are described in detail now in this section. Also, the interface between the NFV MANO and the SDN controller is specified.

5.1   NFV MANO-NFVI

This interface in the ETSI nomenclature is denoted as the **Nf-Vi** reference point. It is the responsible to establish the communication between the NFVI and the VIM. That is, it connects all the virtualized network with the block that manages the resources of this infrastructure. The Nf-Vi reference point must support the next capabilities [4] [2]:

- Assignment of virtualized resources after an allocation request.
- Forwarding of virtualized resources state information.
- Hardware resources configuration, information exchange and events capture.
- Information exchange with external SDN Controllers.

5.2   NFV MANO-VNFs

This interface corresponds to the communication between the VNF manager sub-block of the NFV-MANO (see section 4), the VNFs that are deployed on top of the NFVI and the Element Management System (EM). Recall that the EM provides the VNF with several management functionalities, such as configuration or fault management for the network function provided by the VNF. The EM may be aware of virtualization and collaborate with the VNF Manager to perform those functions that require exchanges of information regarding the NFVI Resources associated with the VNF.

Thereby, the interface of this section allows the VNF manager to control, deploy and configure the VNFs. In the ETSI NFV nomenclature, the interface between the VNF manager (or the NFV-MANO) and the VNFs is called **Ve-Vnfm** reference point. And it is divided in two reference points. The Ve-Vnfm-em reference point connects the VNF manager with the EM, whereas the Ve-Vnfm-vnf connects the VNF manager with the VNF. The **Ve-Vnfm-em** is the interface to support the next functionalities [4] [2]:

- VNF instantiation.
- VNF instance query, to retrieve any run-time information.
- VNF update, to update the configuration.
- VNF instance scaling, to scale up or down the virtual resources allocated to the VNF.
- VNF instance termination.
- Forwarding of configuration and events from the EM to the VNF manager and from the VNF manager to the EM.

It is important to mention that the Ve-Vnfm-em is only used when the EM is aware of the virtualization. On the other hand, the **Ve-Vnfm-vnf** interface supports the same first five functionalities than the Ve-Vnfm-em plus these other ones:

- Forwarding of configuration and events from the VNF to the VNF manager and vice versa.
- Verification that the VNF is still alive or functional.

5.3   Between NFV MANO sub-blocks (Orchestrator, VNF manager, VIM).

Recall that the NFV MANO has three sub-blocks: the orchestrator, the VNF manager and the VIM, see section 4. Thereby, this section describes the interfaces between these blocks. First, the interface between the

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

orchestrator and the VIM is called **Or-Vi** in the ETSI-NFV nomenclature. The **Or-Vi** reference point supports the next functionalities [4] [2]:

- Orchestrator requests for NFVI resource reservation.
- Orchestrator requests for NFVI resource allocation, release or update.
- Forwarding from the VIM to the orchestrator of the next information. NFVI resources configuration, events and state information.

The interface between the orchestrator and the VNF manager is called **Or-Vnfm** in the ESTI NFV nomenclature and supports the next functionalities [4] [2]:

- Allocation, authorization, validation, reservation or release of NFVI resources for a given VNF.
- VNF instantiation.
- VNF instance query, update, scaling or termination.
- Forwarding of VNF events or state information that may impact the network service.

Finally, it remains the interface between the VNF manager and the VIM. This is called **Vi-Vnfm** in the ETSI NFV nomenclature and it supports the next functionalities [4] [2]:

- Information retrieval regarding the NFVI resources reservation.
- NFVI resources allocation or release.
- Exchanges of information regarding the configuration, events or state of NFVI resources used by a VNF.

5.4 Interface between NFV MANO and service providers, users or external management units

It is important to have an interface between the orchestration block of the NFV platform, i.e. the NFV-MANO, and the users, service providers or even external units that manage the needs of the network service. For instance, in SEMIoTICS this interface can connect the NFV-MANO with the SEMIoTICS Pattern Engines to gather information of the NFVI and trigger the creation/modification of a NS. Additionally, it can also connect the NFV-MANO with an external block that computes automatically and dynamically the virtual resource of the NFVI that network service instance needs to run with an optimal QoS, i.e. it could implement the algorithms of section 4.5. These external blocks are known in the ETSI-NFV specification as OSS/BSS, whereas the interface that connects the ETSI-NFV with the OSS/BSS is known as **Os-Ma-Nfvo** reference point [4] [2]. Thereby, the **Os-Ma-Nfvo** reference point supports the next functionalities [4] [2]:

- Request for network service (NS) lifecycle management: NS instantiation; update; query (retrieving information on NFVI resources related to the NS); NS instance scaling (e.g. increase, decrease allocation of resources); NS instance termination.
- Requests for VNF lifecycle management.
- Forwarding of NFV related state information. For instance, NS instance performance measurements, usages of NFVI resources, number of VMs assigned to a NS instance.
- Policy management exchanges. That is, authorization, access control or resource allocation information related to the NS instances and the NFVI.

5.5 NFV MANO-SDN Controller

Service Function Chains deployment can require traffic traversal and thus a service deployment across virtually or physically dislocated VNFs. In deliverable D3.1, we discuss the SFC Manager component that is able to handle service function chaining of network functions by collecting information about the placement and IP addresses of the VNFs assigned to the SFC, as well as its traversal order.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

We foresee the interaction between the SEMIoTICS SDN Controller (SSC) and the NFV MANO, required for population of the expected VNF information to feed that input. Following a spin-up of a number of VMs assigned to the chain, the MANO will provide the controller with the necessary addressing data and the order information and confirm the successful establishment of network flows by the SSC.

The SSC's SFC Manager exposes a number of interfaces that various components, including MANO, can use to provide and receive information about service chains that need to be built: e.g., which tenants want to use them, which destinations are being accessed, what applications the traffic pertains to and, as mentioned above, about the service instances of the network functions. The functions of the chain can be physical appliances or virtual machines running in NFV Infrastructure.

Having the SFC Manager as a logical component in the SSC (separate from MANO) offers the advantages of having one interface to business applications, and the application does not need to be aware of the underlying SFC.

Internally, the SFC manager invokes the VTN Manager (also, ref. D3.1) in order to register external ports of the SDN transport network (which is being used for SFC) and to declare and associate service instances to those external ports. The service instances in chains required by our use cases are expected to include Firewalls, IDS, DPI, and HoneyPot VNFs.

5.6   NFV MANO-Pattern Engine and Pattern Orchestrator

This section describes the interfaces between the NFV MANO and the blocks that are responsible to extract network patterns that drive the proper configuration of the VNF and NS requirements. These are the Pattern Engine and the Pattern Orchestrator.

As it has been mentioned in the previous sections, the Pattern Engine has a direct link with the NFV MANO. Its role is to ask for updated network state metrics and to configure the VNF and NS descriptors taking into account the extracted patterns, i.e. with the information that provides the Pattern Orchestrator. Thereby, the interface between the Pattern Engine and the NFV MANO that we consider is the one that the NFV platform provides for external controllers and services, i.e. for OSS. This corresponds to an **Os-Ma-Nfvo** reference point according to the ETSI NFV argot and supports all the functionalities that we need for the Pattern Engine, as it is described above in section 5.4.

Moreover, we consider that the Pattern Orchestrator has not a direct link with the NFV MANO. That is, it communicates with the Pattern Engine, which then communicates with the NFV MANO. Moreover, the interface between the Pattern Orchestrator and the Pattern Engine is based on RESTful HTTP APIs.

5.7   NFV-level intelligence through dynamic reconfiguration enablers

As previously covered in Section 1.2, Network Services (NS) are composed of virtual and physical network functions (VNFs, and PNFs, respectively) connected together via virtual or physical links. The specification of the properties of each element within a NS, i.e. VNFs and virtual links, are collected in ETSI-standardized Network Service Descriptors (NSd), which in turn are composed of VNF descriptors (VNFd) and Virtual Link descriptors (VLd). Such descriptors are then onboarded to the NFV Orchestrator (NFVO), which then uses it as blueprint for realising the NS via API calls to the Virtualised Infrastructure Manager (VIM).

SEMIoTICS envisions two types of NS reconfiguration: 1) descriptor-based, and 2) live NS reconfiguration. The following provides insight into these two types, as well as their caveats and enablers.

- **Descriptor-based NS reconfiguration**: it implies the update of an onboarded or yet-to-onboard descriptor. Any authorised party interested on a modification of a service (e.g. based on a pattern) should perform the modification in the descriptor itself (written in YAML), and then onboard/update it

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

at the NFVO previous orchestration. This type of reconfiguration is far reaching, meaning that it is virtually possible to modify all the elements of the NS.

- o A specific example of this type of reconfiguration relates to the adjustment of scale-out operations' thresholds. An external entity can decide to change the scale-out trigger from 80% of vCPU usage to 70%, or determine that the maximum number of scaled-out instances should be 4 instead of 3.
- **Live NS reconfiguration**: this assumes a NS is already running on top of the NFV infrastructure. Updating a running NS via NFVO is limited to the change of collected metrics (this implies updating the corresponding VNFd). Nevertheless, leveraging VIM's APIs it is possible to change network-level QoS policies in real time[16]. Live modification of NS is limited to the available APIs at NFVO and VIM.

The SEMIoTICS NFV component deals with VNFs and their properties (e.g. vCPU, images, storage, placement, etc.) rather than with network properties (which are delegated to SEMIoTICS SDN Controller, see D3.1). In SEMIoTICS, it is expected that any reconfiguration of NS (be it descriptor-based or live) would be performed by the Global Pattern Orchestrator (or any other authorised Pattern enforcement engine) via the NFV Management and Orchestration (MANO) Operations/Business Support System endpoint (refer to Figure 1)[17].

All in all, intelligence at the NFV level tightly correlates with allowing authorized external elements to interact with NSd and in some specific instances with VIM's APIs. Therefore, requirements encompass connectivity among NFV MANO elements, as well as the exposure of endpoints to other authorised SEMIoTICS components. From D2.3 (and Section 2.1 in this deliverable), the specific requirements for this functionality are: R.NL.8, R.NL.9, R.NL.10 (OSS/BSS operations through Os-Ma-Nfvo endpoint in Figure 1), and R.NL.11.

---

[16] There are operations that would inevitably incur in down time (e.g. VM scale up/down), although some of these issues can be leveraged at the application level (e.g. using load balancers, replicas, etc.).

[17] It is also possible for authorized components to reach the VIM APIs for a greater set of operations, nevertheless, these should be carefully specified and managed in order to avoid security risks (e.g. misconfiguration of VIM, mismanagement of resources, etc.).

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

6 CONCLUSIONS

This deliverable has presented the NFV technology as a cornerstone to face the networking challenges posed by the SEMIoTICS project. These are the network scalability, dynamicity and flexibility demanded by IoT devices and applications along with the support for network services that require different QoS needs in terms of latency, reliability, security or privacy.

To this end, the NFV technology has been introduced in section 1 to motivate its use in SEMIoTICS. Also, the main NFV building blocks have been described. In section 2 the link with the requirements of SEMIoTICS, presented in deliverable D2.3, have been established. In that section we also present the link with the SEMIoTICS KPIs and its architecture. In NFV, communication, computing and storage resources stemming from the network are virtualized. Network services are deployed on top of them in the form of a chain of virtualized network functions, thereby they are so called SFC and VNF, respectively. Therefore, section 3 has described VNFs and SFCs that are relevant for SEMIoTICS in terms of security, privacy and dependability, which includes both latency and reliability.

The virtualized network services, i.e. SFC, need a manager entity that guarantees their services requests, their deployment on top of the NFV virtual resources, the monitoring of their performance and the management of their lifecycle. The above-mentioned management entity is so called NFV MANO and it has been presented in section 4. Namely, the main functional blocks of an ETSI compliant NFV MANO have been explained. Then, its practical implementation based on OSM and OpenStack has been thoroughly explained. Also, in this regard, we have described how to enable SFC and the monitoring of NFV resources, in the ecosystem based on OSM and OpenStack. Afterwards, section 4 has presented the interaction between the NFV MANO and other components of SEMIoTICS, such as the Pattern Orchestrator. In this regard, we presented the implementation of northbound interfaces based on REST APIs. Also, two alternatives to implement the NFV MANO have been discussed, one based on OSM plus OpenStack and the other based on Kubernetes. In this regard, we have concluded that the OSM plus OpenStack is more suitable to support the networking functionalities demanded by the virtualized network services. Section 4 has also treated the problem of managing NFV resources dynamically. To this end, we have presented the SoA and a practical implementation based on monitoring the NFVI metrics and evaluating thresholds on these metrics that trigger the scaling out process. Last, but not least, section 4 presents how to balance the traffic load among VNF instances, which arise in scaling out processes. Finally, section 5 has presented the ETSI compliant interfaces between all the building blocks of an NFV platform and the SDN controller.

6.1 NFV Component implementation status

The deployment of the NFV Component entails several procedures. First, network-level requirements need to be satisfied (R.NL.1-4, refer to D2.3 for more details). Then, virtualization-ready nodes, or compute nodes, should be placed throughout the SEMIoTICS architecture, particularly where VNFs are to be orchestrated. Lastly, VIM controller and NFVO should have network connectivity to the compute nodes (R.NL.11). All of these elements conform SEMIoTICS NFV Component.

Following SEMIoTICS implementation cycles, the following actions were taken and successfully completed at the current stage of the project:

1. **Deploy VIM instance and NFVI**
   Compute nodes were setup to emulate the Field and Network Layers of the SEMIoTICS architecture. Therefore, it is possible to instantiate VNFs at the Field layer (emulating virtual gateways in UC3, for example), and at the network layer (using VNFs as software SDN switches). An extensive step-by-step guide was developed and uploaded to the project's Gitlab repository[18] so other partners could replicate this work if needed. It uses a simple, single layer topology as example for deploying the VIM component. The above-mentioned compute nodes make up the NFVI, as they admit the virtualization

---

[18] The VIM deployment guide can be found at SEMIoTICS' Gitlab repo under the following path: SEMIoTICS/NFV Orchestration/VIM.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

of their resources. Moreover, the orchestrator of the NFVI, i.e. the VIM was deployed. This VIM is the OpenStack, see section 4 for further details.

2. **Attach NFVO (OSM) to VIM**
   From a centralized position, the NFVO is now able to orchestrate complete NS or network slices traversing the different layers of the SEMIoTICS architecture.

3. **Share access to NFV component with integrator**
   A VPN server was setup in order to grant access to the NFV Component to other members of the consortium. This is specially tailored to the integrator[19].

Last, but not least, it is important to stress that the NFV MANO implementation based OSM and OpenStack has been thoroughly described in section 4.2.

6.2 Future work

This deliverable D3.8 is the last of task 3.2. However, it is interesting to highlight the future work in the rest of SEMIoTICS project related to NFV.
One of the next steps involving the NFV component relates to the integration with other SEMIoTICS components and network services for use cases, this is task 3.5 and work package 5. In this document, API definitions of the NVF component and the information/functionality they provide were described. Next, these API need to be consumed by other components of SEMIoTICS, such as the SEMIoTICS SDN Controller and Pattern Orchestrator, to relay network-related management and enable SPDI Patterns implementation, respectively.

To achieve the aforementioned, the following set of actions need to be performed:
- Define what are the requirements of other SEMIoTICS components related to NFV. Such as:
  - Telemetry.
  - Onboarding.
  - Orchestration.
  - NFV descriptor updates.
- Specify what are the endpoints involved in satisfying such requirements.
  - Based on Figure 1.
- Perform integration tests.

6.3 Technical choices for SEMIoTICS, SoA and beyond SoA

The aim of this subsection is to highlight which of the technical content, presented above, is related work, SoA or beyond SoA. Also, we stress which technical content is considered for the implementation of the SEMIoTICS project.

Section 3.1 proposes security, privacy and dependability functionalities in terms of VNFs for SEMIoTICS. This yields flexibility, programmability and dynamicity, which are distinguishing features of SEMIoTICS rather than using traditional approaches in networking based on static, monolithic and rigid approaches. Section 3.2 is related to 3.1 as it leverages chains of VNFs to build SFC that provide security, privacy and dependability services for SEMIoTICS. We use some existing VNFs and SFC implementations applying the appropriate adaptations to satisfy the needs of the specific use case (use case 2). This is actually part of the incoming work package 5.

The content presented in section 4.1.1 and also section 4.2.1 describes the VIM functionality and the VIM APIs for computing, storage and networking purposes, bearing in mind the SEMIoTICS framework. To this

---

[19] Information about the topology of such tunnel can be found at the project's Gitlab repository. Specifically, under the following path: SEMIoTICS/NFV Orchestration/CTTC-tunnel.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

end, the OpenStack VIM is used. OpenStack is a SoA orchestrator to manage virtualized infrastructures and it is the one that will be used in SEMIoTICS to manage the NFVI.

The content presented in sections 4.1.2 and 4.1.3 deals with the NFV orchestrator and VNF manager sub blocks of the NFV MANO. They are described according to the ETSI standard specifications and they are implemented in the SEMIoTICS project using the OSM open source software, as it is described in section 4.2.2. OSM is SoA and it is ETSI standard compliant, as it implements the technical content described in sections 4.1.2 and 4.1.3.

Regarding the NFV MANO implementation aforementioned, we have implemented the ability to gather real-time metrics of the NFVI, i.e. telemetry services, and we have enabled the possibility of forming SFC. This requires enabling optional functionalities of both OpenStack and OSM. This is explained in section 4.2 and can be considered in the intersection between SoA and beyond SoA, as these are not options that can be found by default in the OSM and OpenStack.

Section 4.3 describes the interaction between the NFV MANO and the Pattern Engine, which in turn interacts with the Pattern Orchestrator. The two latter are two important SEMIoTICS components mainly developed in other tasks and implement the intelligence and pattern-driven automation of SEMIoTICS. They are among the key contributions of SEMIoTICS project. Therefore, this material can be considered as beyond SoA, as the interfaces and the integration are novel.

The material presented in section 4.4 presents a real experiment comparing two alternatives to implement the NFV MANO for SEMIoTICS. One of them relies on OSM and OpenStack, whereas the other uses Kubernetes and Docker. This comparative can be considered as beyond SoA, as compares SoA options to implement the NFV MANO and selects the most adequate bearing in mind the SEMIoTICS characteristics. Section 4.5 describes the dynamic resource management in NFV. The threshold-based approach can be considered as SoA.

Finally, section 5 presents the interfaces between the NFV sub blocks and between NFV and other SEMIoTICS components. Namely, sections 5.1 to 5.4 present the ETSI compliant interfaces between NFV sub blocks. Thereby, this is SoA and it is used within the SEMIoTICS framework. Section 5.5 is the interface between the NFV MANO and the SDN controller and it is SoA that is being used in SEMIoTICS. Section 5.6 is the interface between the NFV MANO and the Pattern Engine, which is an innovative SEMIoTICS component. The implementation of this interface is described in section 4.3 and can be considered beyond SoA.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

7 REFERENCES

[1] F. Yousaf, M. Bredel, S. Schaller and F. Schneider, "NFV and SDN, key technologie enablers for 5G networks," *IEEE Journal on Selected Areas in Communications,* vol. 35, no. 11, 2017.

[2] ETSI, "ETSI.org: Network Functions Virtualisation (NFV); Architectural Framework," 10 2013. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf. [Accessed 23 October 2018].

[3] ETSI OSM, "OSM Information Models," 2019. [Online]. Available: https://osm.etsi.org/gitweb/?p=osm/IM.git;a=tree;f=models/yang;h=ac67adaec00123ef4a68911ff0082 fb35556b03a;hb=HEAD. [Accessed January 2019].

[4] ETSI, "ETSI.org: Network Functions Virtualisation (NFV); Management and Orchestration (ETSI GS NFV-MAN 001)," December 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf. [Accessed November 2018].

[5] ETSI, "ETSI GS NFV-SOL 005. Network Functions Virtualisation (NFV) Release 2.," ETSI, 2018.

[6] M. Falchetto and others, "Requirements specification of SEMIoTICS framework," SEMIoTICS deliverable D2.3, 2018.

[7] Netflix, "Github Repository: Netflix/FIDO," [Online]. Available: https://github.com/Netflix/Fido.

[8] T. Koulouris, M. C. Mont and S. Arnell, "SDN4S: Software Defined Networking for Security," 2017. [Online]. Available: https://www.labs.hpe.com/techreports/2017/HPE-2017-07.pdf.

[9] P. Quinn and T. Nadeau, "Problem Statement for Service Function Chaining," 2015. [Online]. Available: https://tools.ietf.org/html/rfc7498.

[10] S. Kumar, M. Tufail, S. Majee, C. Captari and S. Homma, "Service Function Chaining use cases in data centers," *IETF SFC WG,* 2015.

[11] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez and J. Uttaro, "Service Function Chaining use cases in mobile networks," *Internet Engineering Task Force,* 2015.

[12] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert and C. Meirosu, "Research directions in network service chaining," in *IEEE SDN for Future Networks and Services (SDN4FNS)*, Trento, Italy, 2013.

[13] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications,* vol. 36, no. 1, pp. 16-24, 2013.

[14] L. Vokorokos, M. Ennert, J. Radušovský and others, "A survey of parallel intrusion detection on graphical processors," *Open Computer Science,* vol. 4, no. 4, pp. 222-230, 2014.

[15] A. Bremler-Barr, Y. Harchol, D. Hay and Y. Koral, "Deep packet inspection as a service," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, Sydney, Australia, 2014.

[16] OpenStack, "OpenStack Ironic Project: Bare metal provisioning," [Online]. Available: https://wiki.openstack.org/wiki/Ironic.

[17] OpenStack, "Compute API," [Online]. Available: https://developer.openstack.org/api-guide/compute/.

[18] Canonical Ltd., "Linux Containers," [Online]. Available: https://linuxcontainers.org/.

[19] OpenStack, "OpenStack Docs: Server concepts," [Online]. Available: https://developer.openstack.org/api-guide/compute/server_concepts.html.

[20] J. Denton, Learning OpenStack Networking (Neutron) Second Edition, Birmingham, UK: Packt Publishing Ltd., 2015.

[21] OpenStack, "OpenStack Docs: Networking API v2," [Online]. Available: https://developer.openstack.org/api-ref/network/v2/.

[22] OpenDaylight, "OpenStack and OpenDaylight," [Online]. Available: https://wiki.opendaylight.org/view/OpenStack_and_OpenDaylight.

[23] OpenStack, "OpenStack Docs: Block Storage," [Online]. Available: https://developer.openstack.org/api-ref/block-storage/v3/.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

[24] DevStack. [Online]. Available: https://docs.openstack.org/devstack/latest/. [Accessed February 2020].

[25] "OpenStack Ansible (OSA).," [Online]. Available: https://docs.openstack.org/openstack-ansible/latest/. [Accessed February 2020].

[26] "OSA Deployment Guide.," [Online]. Available: https://docs.openstack.org/project-deploy-guide/openstack-ansible/stein/. [Accessed February 2020].

[27] "OSA User Guide.," [Online]. Available: https://docs.openstack.org/openstack-ansible/stein/user/index.html. [Accessed February 2020].

[28] "YAML," [Online]. Available: https://yaml.org. [Accessed February 2020].

[29] "Service Function Chaining Extension for OpenStack Networking.," [Online]. Available: https://docs.openstack.org/networking-sfc/latest/. [Accessed February 2020].

[30] "Tcpdump," [Online]. Available: https://www.tcpdump.org. [Accessed February 2020].

[31] ETSI OSM, "Assumptions about interaction with VIMs and VNFs," [Online]. Available: https://osm.etsi.org/wikipub/index.php/OSM_Release_FIVE. [Accessed January 2019].

[32] ETSI OSM, "Creating your own VNF package," [Online]. Available: https://osm.etsi.org/wikipub/index.php/Creating_your_own_VNF_package. [Accessed January 2019].

[33] Docker, "Swarm mode overview," [Online]. Available: https://docs.docker.com/engine/swarm/. [Accessed January 2019].

[34] Red Hat OpenShift, "OpenShift," [Online]. Available: https://www.openshift.com/. [Accessed January 2019].

[35] OpenStack, "Magnum," [Online]. Available: https://wiki.openstack.org/wiki/Magnum. [Accessed January 2019].

[36] Kubernetes, "Kubernetes," [Online]. Available: https://kubernetes.io/. [Accessed January 2019].

[37] Kubernetes, "Cluster Networking," [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/. [Accessed January 2019].

[38] Kubernetes, "Kubernetes Reference," [Online]. Available: https://kubernetes.io/docs/reference/#api-reference. [Accessed January 2019].

[39] J. G. Herrera and J. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE Transactions on Network and Service Management,* vol. 13, no. 3, 2016.

[40] F. B. Jemaa, G. Pujolle and M. Pariente, "QoS-aware VNF placement optimization in edge-central cloud architecture," in *IEEE Global Communications Conference (GLOBECOM)*, Washington DC, USA, December 4-8, 2016.

[41] X. Chen, W. Ni, T. Chen, I. B. Collings, X. Wang, R. P. Liu and G. B. Giannakis, "Multi-timescale online optimization of network function virtualization for service chaining," arXiv:1804.07051.

[42] S. Son, G. Jung and S. C. Jun, "An SLA-based cloud computing that facilitates resource allocation in the distributed data centers of a cloud provider," *Journal of Supercomputing,* vol. 64, no. 2, pp. 606-637, 2013.

[43] Q. Zhang, Q. Zhu, M. F. Zhani, R. Boutaba and J. L. Hellerstein, "Dynamic service placement in geographically distributed clouds," *IEEE JSAC,* vol. 31, no. 12, pp. 762-772, 2013.

[44] M. Alicherry and T. Lakshman, "Network aware resource allocation in distributed clouds," in *INFOCOM*, Orlando, FL, USA, 2012.

[45] J. Altmann and M. M. Kashef, "Cost model based service placement in federated hybrid clouds," *Future Generation Computer Systems,* vol. 41, pp. 79-90, 2014.

[46] X. Chen, W. Ni, T. Chen, I. B. Collings, X. Wang, R. P. Liu and G. B. Giannakis, "Distributed stochastic optimization of network function virtualization," in *IEEE Global Communications Conference (GLOBECOM)*, Singapore, 2017.

[47] M. J. Neely, Stochastic network optimization with application to communication and queueing systems, Morgan and Claypool publishers, 2010.

[48] A. Defazio, F. Bach and S. Lacoste-Julien, "SAGA: a fast incremental gradient method with support for non-strongly convex composite objectives," in *NIPS*, Montreal, Canada, 2014.

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017
Deliverable D3.8 Network Functions Virtualization for IoT (final)
Dissemination level: public

[49] "Scaling out OSM," [Online]. Available: https://osm.etsi.org/wikipub/index.php/OSM_Autoscaling. [Accessed February 2020].

[50] [Online]. Available: https://geekflare.com/open-source-load-balancer. [Accessed February 2020].

[51] [Online]. Available: www.nginx.com. [Accessed February 2020].

[52] "nginx," [Online]. Available: https://www.nginx.com/blog/dns-service-discovery-nginx-plus/. [Accessed February 2020].

[53] "OpenStack Designate," [Online]. Available: https://docs.openstack.org/designate/latest/. [Accessed February 2020].

[54] "Docker networking," [Online]. Available: https://docs.docker.com/compose/networking/. [Accessed February 2020].