TECHNISCHE UNIVERSITÄT BERLIN

An Extensible and Customizable Framework for the Management and Orchestration of Emerging Software-based Networks

vogelegt von
Master of Science
Giuseppe Antonio Carella
geb. in Brindisi, Italien

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

Promotionsausschuss:

Vorsitzender :	Prof. Dr. Rafael Schaefer
Gutachter :	Prof. Dr. Thomas Magedanz
Gutachter :	Prof. Dr. Paolo Bellavista
Gutachter :	Prof. Dr. Odej Kao

Tag der wissenschaftlichen Aussprache: 02.02.2018

Berlin, 2018

Abstract

The 5th Generation Mobile Telecommunications (5G) is supposed to drastically change network operators' infrastructures.

The evolution of telecommunication networks has always been influenced by the parallel evolution within the Information and Communication Technology (ICT) domain. What started with intelligent networks in the 90's, namely the centralization of service programs and data in central computers controlling remote switching layers in order to simplify the service creation, deployment and management, led, at the beginning of the millennium, to Service-Oriented Architecture (SOA) based distributed Session Description Protocol (SDP) on top of converging networks.

Due to the current approach of *"everything is fully connected"*, a more efficient way to manage network operators' resources and infrastructures is required in order to provide always much greater throughput and much lower latency with high availability and higher connectivity density. The transition towards *"everything as a software"* is the enabler for this transformation where software-based virtualized network functions can be customized for the particular needs of a particular vertical domain. Therefore, almost ten years later, cloud principles and technologies have again changed the way services are developed and provisioned. The novel concept of *"network slicing"* allows Telecommunication Service Providers (TSPs) to optimize the usage of their infrastructure resources, providing on demand to end-customers different network segments with different capabilities.

This dissertation work will present the extensive research conducted by the author over the last 5 years in which Network Function Virtualization (NFV) concepts and standards emerged and drastically transformed the telecommunication networks. The design evolution presented started when requirements for virtualizing network functions were first described by the research and industrial communities. Based on the author's past experiences gained in realizing cloud-based Service Delivery Platforms (SDPs), the thesis will elaborate the design and development of an extensible NFV Management and Orchestration (MANO) framework suitable for managing and orchestrating any kind of software-based networks.

This work was developed in parallel to the international Software-Defined Networking (SDN) and European Telecommunications Standards Institute (ETSI) NFV standardization activities, and the implemented open source Open Baton project is the result of an agile design and development process, initiated and managed by the author. Today Open Baton represents one out of four globally recognized ETSI NFV MANO frameworks, enabling early 5G prototyping and standardization.

The author, a senior scientist at the Technical University of Berlin, conducted this work in the context of several European research projects, including the supervision of several bachelor and master theses and integrated into various industry SDN/NFV testbeds in ongoing research projects as well as several industry solutions.

Zusammenfassung

5G wird einen enormen Einfluss auf die Infrastrukturen der Netzbetreiber haben.

Die Entwicklung von Telekommunikationsnetzwerken wurde seit jeher von der Entwicklung im Bereich der Informations- und Kommunikationstechnik (IKT) beeinflusst. Um die Erstellung, Bereitstellung und Verwaltung von Diensten zu vereinfachen, wurden in den 90er Jahren intelligente Netzwerke erforscht, in denen Dienstprogramme und Daten in zentralen Computern zusammengefasst sind, die entfernte Vermittlungsebenen steuern. Das führte zu Beginn des Jahrtausends zur Entwicklung eines Ansatzes für verteilte und konvergierende Netzwerke, basierend auf serviceorientierten Architekturen (SOA) und Dienstplattformen (SDP).

Da mit dem Ziel „alles vollständig zu vernetzen" höhere Datendurchsätze, niedrigere Latenzzeiten, hohe Verfügbarkeiten und höhere Konnektivitätsdichten benötigt werden, wird ein deutlich effizienterer Weg zur Verwaltung der Ressourcen und Infrastrukturen der Netzbetreiber erforderlich. Der Übergang zu „alles als Software" (engl. "Softwarization") ist Treiber dieser Transformation, wodurch softwarebasierte, virtualisierte Netzwerkfunktionen (NFV) für die speziellen Bedürfnisse eines bestimmten, vertikalen Anwendungsfalles zugeschnitten werden können. Cloud-Prinzipien und -Technologien haben daher, fast zehn Jahre später, erneut die Art und Weise, wie Dienste entwickelt und bereitgestellt werden, weiter verändert. Das aufgekommene Konzept "Network Slicing" ermöglicht Telekommunikationsdienstanbietern (TSP) die Nutzung ihrer Infrastrukturressourcen zu optimieren und den Nutzern auf Anfrage verschiedene Netzwerksegmente mit zugeschnittenen Eigenschaften bereitzustellen.

Diese Dissertationsarbeit spiegelt die umfangreiche Forschung des Autors während der vergangenen 5 Jahre wider, in der NFV-Konzepte und -Standards entstanden und Telekommunikationsnetze maßgeblich beeinflusst wurden. Die Designevolution, die im Zuge dieser Arbeit vorgestellt wird, begann als die Anforderungen für die Virtualisierung von Netzwerkfunktionen von der Forschungs- und Industriegemeinschaft beschrieben wurden. Basierend auf den bisherigen Erfahrungen des Autors bei der Realisierung von Cloud-basierten Dienstplattformen (SDP), wird sich diese Arbeit auf den Entwurf und die Entwicklung eines erweiterbaren NFV Management and Orchestration (MANO) Frameworks zur Verwaltung und Orchestrierung jeglicher Art von softwarebasierten Netzwerken fokussieren.

Der Autor, ein leitender Wissenschaftler an der Technischen Universität Berlin, führte diese Arbeit im Rahmen mehrerer Industrie- und europäischer Forschungsprojekte durch und betreute während dieser Zeit eine Vielzahl von Bachelor- und Masterarbeiten. Die entwickelten Konzepte wurden umgesetzt und haben zu verschiedenen Open-Source-Projekten beigetragen. Dazu gehört auch Open Baton, das als Softwareplattform für die prototypische Entwicklung von Netzwerk und Diensten im 5G dient. Darüber hinaus wurden die Konzepte in laufenden Forschungsprojekten im industriellen Bereich angewandt und in diverse Testbeds integriert.

Acknowledgments

Stay hungry, stay foolish.
Steve Jobs

During the years, self-motivation has been the main driver for reaching this objective. However, each one of us, alone, is worth nothing. I would like to thank all the people who believed in me, and who supported me.

First of all, I would like to thank Professor Magedanz for pushing me forward with his big vision and believing in me during all these years, and also Prof. Dr. Paolo Bellavista and Prof. Dr. Odej Kao for their support and valuable suggestions.

A special thank goes to my team, without whom it would not have been possible to reach such objective. We have started from scratch, and we have reached the impossible. I'm really proud of you (my friends!) and please remember, impossible is nothing! In alphabetical order: Nico Bove, Thomas Briedigkeit, Olek Gozman, Ahmed Medhat, Marcello Monachesi, Flavio Murgia, Philipp Kuhnz, Radoslav Vlaskovski, and many others who have left or joined the team later, as well as other colleagues at TU Berlin and Fraunhofer FOKUS who supported me anytime (the list is too long for being expanded here!). Thanks also to Florian Ermisch for his incredible skills in administrating our infrastructure. Thanks to Birgit Francis and Oana Vingarzan for their support. Very very special thank to Lars Grebe, Lorenzo Tomasini, and Michael Pauls for their superior continuous support.

Special gratitude goes also to Niklas Blum and Florian Schreiner for their initial supervision and for being always present during these years, as well as to Dragos Vingarzan, my mentor, always supporting my crazy ideas. Many thanks also to my friends at CND: Alberto Diez, Jakub Kocur, Alexandru Russu, Valentin Vlad, and all the others.

I would like to thank also people I've met during this journey and who helped me enhancing my professional skills: Thomas Micheal Bohnert, Gino Carrozzo, Paolo Crosta, Andy Edmonds, Luca Foschini, Luis Lopez, Roberto Minerva, Fatih Nar, Prakash Ramchandran, Junnosuke Yamada, and many others.

Thanks to my in-law relatives for supporting me during these years and my best friend Ilaria for being always present in my life. Very special thank to my sister and my parents for giving me the possibility to become what I'm.

Finally, the most important thank goes to my wife Laura and my daughter Lena. I could not make it without their infinite patience and love.

Giuseppe Carella

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

The continuous evolution of ICT technologies is paving the way towards a radical transformation on how telecommunication services are currently managed. The introduction of all-Internet Protocol (IP)-based NGN reduces network infrastructures to a composition of multiple network functions collaborating together for providing feature-rich communication services to end users. At the beginning of this work classical NGN infrastructures were comprised of diverse Network Functions (NFs) implemented as monolithic hardware appliances and designed in a certain way for providing a desired functionality for a specific vertical domain.

The concept of software-based networks appeared since the seminal work on programmable networks and mobile code started with the introduction of Java in 1995. *"All a user will have to do is go to the C prompt and type 'Java OS,' which will bring up the HotJava Web browser and the HotJava views,"* mentioned Alan Baratz, president of JavaSoft, who defined this as the first software-based network [1][2][3].

Those NFs, also defined as network nodes, are *functional building blocks within a network infrastructure, having well-defined external interfaces and well-defined functional behavior*[4]. After some interesting experiences about network programmability, like active networks [5] and intelligent networks [6], the transition towards *"everything as a software"*, and in particular the possibility of being able to control network capabilities offered by those NFs via defined APIs, gained the attraction of major network operators and academic institutes that, back in 2011, launched the Open Networking Foundation (ONF) fostering the possibility of defining protocols for allowing dynamic control of network resources [7], namely SDN.

From a technical perspective, this transition towards *"everything as a software"* was primariliy enabled by the introduction of cloud computing technologies. In fact,

the evolution of telecommunication networks has always been influenced by the parallel evolution within the ICT domain as shown in fig:evolution. Virtualization, the key enabler technology in cloud computing, gained momentum in the ICT domain providing cost-efficient means for ICT and Over-The-Top (OTT) communication services infrastructure consolidation [8]. Certainly, virtualization of hardware resources not only reduces infrastructure costs, but also improves time for provisioning new resources and increases flexibility in management operations.

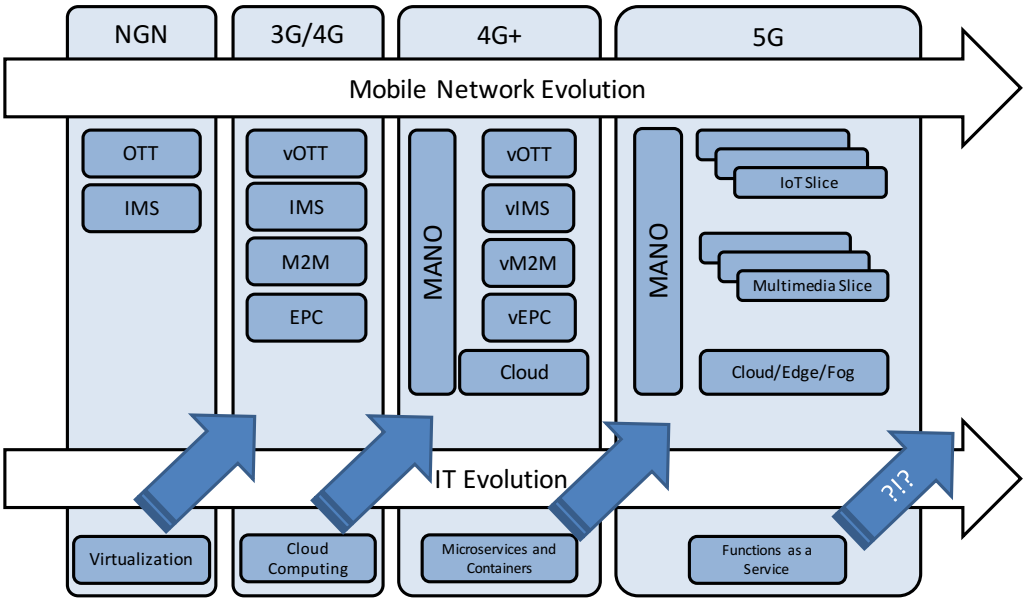

Figure 1.1: Mobile Network Evolution vs IT Evolution

On the one hand, the requirement of always being able to cope with the increasing number of users' demands is transforming operators in just dumb pipe providers while OTT communication service providers are gaining momentum utilizing network operators' resources and infrastructures in a more efficient way [9]. On the other hand there is an increasing number of requirements coming from the 5th Generation Mobile Telecommunications addressing the demands and business contexts of 2020 and beyond. Telecommunication Service Provider have urgent needs in transforming their network infrastructures, especially for accommodating the requirements of the approach of *"everything is fully connected"*, providing always much greater throughput and much lower latency, with high-availability and higher connectivity density[10]. Furthermore, *"commoditization"* represents the way operators are trying to reduce the gap between revenues and capital spending. This transformation could at least slow the negativity, which is, in return, also placing some limits on capital spending towards vendors' equipment. All in all, what TSPs are targeting is to simplify the way NFs are managed and orchestrated, especially

lowering costs of the infrastructure used for executing them.

A large number of NSs, defined as a composition of NFs, are nowadays provided to end users via network operators' infrastructures. Thus, to maintain the desired level of Service Level Agreement (SLA) also in case of high peak situations, operators' infrastructures are usually over-provisioned, sometimes by an order of magnitude, generating low resource utilization rate during typical times. The situation is that an increasing number of TSPs are entering a phase in which infrastructure costs required to cope with the growing traffic demand are not sustainable anymore by the gradient of revenue increases (as shown in Figure 1.2), and, therefore, a radical transformation is required in order to invert this trend and decrease the revenue gap [11]. This is mainly due to the fact that traditional telecommunication infrastructures are very intensive in hardware. Therefore, costs for maintaining such infrastructures are very high.

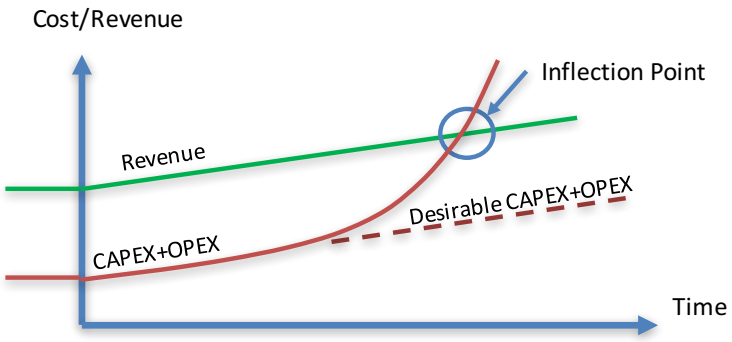

Figure 1.2: Cost/Revenue Over Time

This transformation is very much influenced by the *software-defined era*. TSPs are forced to adapt to novel paradigms shifting the focus from harwdare to software [12]. Hence, if elastic resource provisioning is desirable for best-effort service delivery models, elastic scalability is definitely crucial for QoS/SLA sensitive services, such as most telco services[13][14]. Therefore, these technologies and trends are critically important for reducing Capital Expenditures (CAPEX) and Operating Expenditure (OPEX) in TSPs infrastructures and justifying the Return on Investment (ROI) for virtualizing it [15].

Network management plays a critical role for achieving this transformation. Network management is definitely not a novel paradigm. Since the early availability of distributed systems over large networks, TSPs started defining standard interfaces and protocols for managing network devices, with the general objective of meeting real-time, operational performances and guaranteeing QoS at a reasonable cost. Network management involved a set of operations applied to the life cycle of a network component.

Orchestration, a term broadly used in the cloud domain[16], has lately also been

applied in the context of network management. Basically, the telecommunication service domain started a transition towards a different way of building networks, getting closer to what the ICT domain has been doing over the last decade, i.e., providing services as composition of multiple software components. Considering the large advances in software design and development achieved in the ICT domain, the TSPs started adopting those new technologies and applying those novel architectural principles to their NGN infrastructures while moving towards the latest generation of mobile networks. Although a significant amount of research was conducted in the ICT domain, the elastic provisioning of 3rd Generation Mobile Telecommunications (UMTS, CDMA2000) (3G)/4th Generation Mobile Telecommunications (LTE, WiMAX) (4G) network functions was widely unexplored and still a challenging issue for the telecommunication domain[13]. Most of the solutions presented were either analyzing state of the art in network virtualization[17] or proposing basic concepts for virtualizing specific network functions[18][19][20][21], without considering important characteristics like elasticity and flexibility offered by cloud computing technologies.

Nevertheless, back in 2012, with the great success of cloud computing and virtualization technologies, a rather large group of operators and vendors published a white paper [22], paving the way to a new Industry Specification Group (ISG) group created by ETSI with the scope of defining an architecture for the development of virtual network infrastructures by porting and further adapting NFs to the specific cloud environment. Figure 1.3 shows a simplified version of the architecture presented by the ETSI NFV group.

Migrating standard 3G/4G functions, such as the 3rd Generation Partnership Project (3GPP) Evolved Packet Core (EPC) and the 3GPP IMS, from dedicated hardware-based appliances to software-based artifacts requires the transformation of network operators' infrastructures towards multisite cloud-based datacenters built on top of commodity hardware.

As a first step in order to achieve this objective, those NFs have to be redesigned to fulfill the requirements of a highly flexible and dynamic environment. In particular, they will need to support elasticity considered one of the major characteristics introduced by cloud computing, but maintain, and further improve, the performances reached with previous dedicated hardware solutions. Novel principles like microservices architectures and cloud-native applications should be adopted for designing highly scalable network services. Nevertheless, the transformation towards cloud-based multisite infrastructures requires the introduction of novel management and orchestration paradigms allowing a more efficient allocation of resources on an on-demand schema on such distributed infrastructures.

The requirement of moving processing power, and particularly data, towards the edge of the network for reducing latency between users and services, foreseen by the work conducted in the context of Mobile Edge Computing (MEC) [23] required in future 5G network infrastructures, requires a novel distributed environment, in which some of the sites maybe implemented with lightweight hardware equipment on the edge nodes. The industry-standard servers and cloud computing technologies

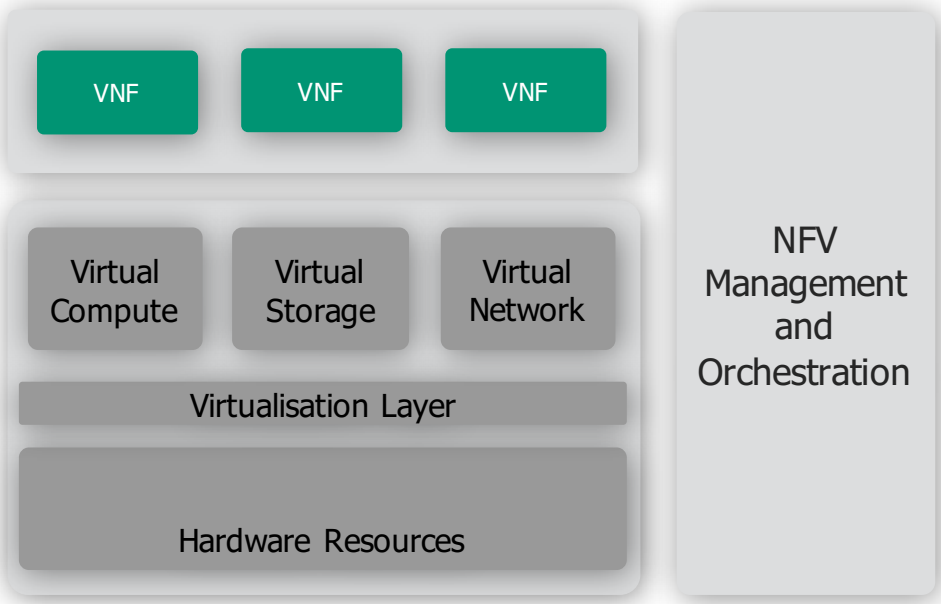

Figure 1.3: ETSI NFV Architecture - Simplified Version

represent two of the most important enablers for this transformation process [24].

Therefore, the path forward is clear: A radical change of the operators' network infrastructures is required in order to support this huge transformation, otherwise the chances to reduce revenues will be very low. NFV and SDN trends represent important enablers for restructuring operators' networks.

In particular, NFV proposes a new set of components part of the MANO domain, differing from legacy network management approaches providing a more flexible solution for incorporating different vendors' solutions reducing time-to-market of new network functions. Those novel MANO components are crucial for bringing the desired flexibility in NGN infrastructures, thus, most of the TSP are moving towards this architecture using a bottom-up approach. First of all, they need to virtualize their infrastructure, secondly they have to identify the required MANO components for simplifying the management of their networks' life cycle.

As a result, different competitive solutions are under development leveraging the power of open source to foster innovation in the telecommunication industry to provide an environment which could answer to those research challenges envisioned by the industrial and academic communities.

1.2 Problem statement, and Major Keyword Definition

The above described evolution outlines several research challenges and imposes a transformation of operators' network infrastructures. Management and orchestration of network services is considered one of the major challenges due to the complexity that is introduced by network functions implemented as software components. The more NFs are decomposed in micro services, the more complex is the task of the MANO framework to manage their complete life cycle, also due to the heterogeneity of the management systems used for controlling them individually.

Focusing on the telecommunication domain, managing networks has always been a nontrivial task. Back in 1996 Saydam and Magedanz provided a comprehensive definition of network and service management in a Journal about Network and System Management [25]:

> *"[...] Network management involves the deployment, integration and coordination of all the hardware, software and human elements to monitor, test, poll, configure, analyze, evaluate, and control the network and element resources to meet the real-time, operational performance and QoS requirements at reasonable cost. Service management involves the creation, access, usage, and management of value-added services using the logical, virtual, and physical network resources and the network management systems. The separation of service, service management, and network resources is crucial in creating open, transparent, and reconfigurable services [...]"*

Such definition highlights the importance of separating services and networking resources. The challenge for NFV is that a new management model must provide efficiency across an entire set of software stacks. Functions that have usually been implemented as monolithic hardware-based solutions, have to be redesigned as VNFs following a cloud-native approach and microservices principles. Furthermore, the split of the software elements from the hardware components introduces an additional level of uncertainty, requiring new mechanisms for ensuring QoS and SLAs expected from such services. What the ETSI NFV ISG appears to aim for is the creation of a network management model that plugs into current network management systems and OSSs/Business Support Systems (BSSs) by offering interfaces from/to NFV processes and elements.

In effect, this means that from the end user perspective, a VNF, or a complex composition of individual VNFs defined as NS, emulating the behavior of a physical appliance (like a firewall), would be a virtual form of that physical device and be managed in the same way. While this approach would address the stated goal of exploiting virtualization, it also suggests that overall service deployment and management practices would change little as NFV is deployed. That makes it difficult to secure major changes in operating efficiency or service agility. Very recently the

ETSI NFV ISG identified some priorities that should be addressed for fulfilling the requirements of future 5G networks, published as a white paper in February 2017. One of the major priorities has been identified in the *end-to-end Service Management*[26]:

> *"One of the key challenges that must be addressed is the definition of a complete management and orchestration framework. Such a framework should exploit and leverage NFV features (e.g., on-demand instantiation and scaling of VNFs) together with additional automated network capabilities for guaranteeing reliability and service assurance. Coordination among a) resource-oriented management tasks performed by MANO, and b) Fault-management, Configuration, Accounting, Performance, and Security (FCAPS) management of network application is needed. [...]"*

Management and orchestration is still considered a critical challenge for ensuring reliability and guaranteeing the desired level of Quality of Experience (QoE) to the end users. Moreover, the ETSI NFV ISG identified the need of coordination between MANO functions handling the NS deployments and OSSs addressing aspects defined by the FCAPS model introduced several years ago by the International Organization for Standardization (ISO)[27]. Figure 1.4 shows a very simplified view of the NS life cycle. Typically TSPs deploy their NSs for serving end customers over a rather certain period of time. For instance, considering the mobile core network use cases, TSPs are deploying necessary NS for serving end customers for at least a couple of years. Moving towards a fully virtualized environment, the expectation is to reduce the deployment process defined as the period from the selection of the required NS up to the actual activation, from months to minutes. This means that the deployment phase may impact less than 1% of the overall life cycle. Therefore, correctly handling the state of the NS over the runtime phase becomes crucial for maintaining the appropriate QoS towards end users.

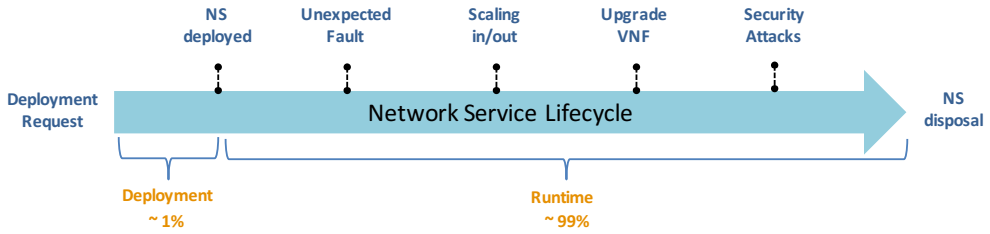

Figure 1.4: Simplified Network Service Life Cycle

Furthermore, NFV proposes an architecture where the separation of roles be-

tween the VNF Function Provider (VNFP), the TSP, and the NFVI Provider (NFVIP) is pretty much clear. This implies a different approach in the way NS are provisioned. The TSP interacts with the NFVIP for building up the required NFVI. The VNFP provides VNFs to the TSP in a portable form, typically a package, on boarding them in their MANO framework[28]. The TSP composes complex network service topologies wiring together multiple VNFs, and executes the automated deployment procedure. The deployment may require the execution of several tasks involving heterogeneous technologies.

Service provisioning and deployment is not a novel topic for the ICT domain. However, most of the existing technologies are not considering the clear separation between the VNFP and the Service Provider (SP), requiring a TSP to define also the management plan of the composed service [10], including details about low-level operations (i.e., installation procedures) that need to be executed for each individual VNF. Those details are typically known only by the VNF developer, making much more complicated the role of the TSP whose objective is to deploy the end-to-end network service. What is actually required by a TSP is a MANO framework providing an inventory of the available VNFs and their capabilities, and providing mechanisms for composing them in the end-to-end NS.

Although there are no doubts about the fact that the virtualization trend, better definable as *cloudification*, is the way to go for simplifying management operations, at the same time, a huge transition towards open source solutions started in different telecommunication-oriented domains. Based on a recent survey conducted by Gigaom Research involving around 600 North American operators (300 enterprises and 300 service providers), 95% view open source SDN and NFV technologies positively as they allow fast prototyping of new functionalities without requiring huge investments in hardware and software components [29]. However, back in 2013, implementing a NFV compliant solution was a not so trivial task due to the lack of maturity of the specifications provided by the ETSI NFV ISG not yet ready with the normative work.

Nowadays the situation is becoming a bit uncontrollable: There are many open source solutions trying to solve the same problem and sometimes they are not compatible with each other due to the fact that they are based on proprietary information models and interfaces due to the lack of maturity of the respective standards when those projects were started. Furthermore, for TSPs interoperability among the different solutions is a crucial aspect: Axel Clauberg, VP of aggregation, transport and fixed access at Deutsche Telekom AG, coined the term *"zoo of orchestration"* at a big telecom event referring to the proliferation of incompatible NFV and SDN implementations [30]. Creating new silos is exactly what TSPs do not want because, although they may be open source, it will transform just into a new set of scenario-specific solutions.

On the one hand, current solutions available in the open source ecosystem are still focusing on aspects related to the deployment and configuration of NSs without considering aspects related to the FCAPS model. On the other hand, most of the scientific work already conducted focuses mainly on specific research challenges

providing solutions validated with simulators, without considering the complexity of a real environment.

What is actually needed is a framework capable of i) managing heterogeneous resources at the infrastructure level, and ii) providing a unified model for orchestrating heterogeneous software-based networks throughout their life cycle, including the runtime phase. Such framework should be designed for being further extended and customized for the needs of a particular use case, and should maintain compatibility with the standardization work for allowing interoperability between other existing open solutions.

1.3 Key Questions Addressed by the Dissertation

Therefore, the main objective of this thesis is to design and implement an extensible and customizable framework for the management and orchestration of emerging software-based networks. It should support heterogeneous types of VNFs and orchestrate their life cycle on top of a distributed NFV-based infrastructure in order to satisfy the requirements of a particular vertical domain (Internet of Things (IoT), automotive, e-health, etc.).

Hence, the main aim of this work is to answer the following main research question: ***How to design an extensible and customizable open source Network Function Virtualization (NFV) Management and Orchestration (MANO) compliant framework supporting heterogeneous vertical domain requirements on a multisite NFV Infrastructure (NFVI)?***

The secondary research questions derived as aspects of the main research question are:

- **Q1**: How to design a framework for end-to-end managing and orchestrating the whole life cycle of network services?
- **Q2**: How to ensure Quality of Service (QoS) and Service Level Agreement (SLA) levels required by Next Generation Network (NGN) throughout their life cycle fulfill the requirements of the FCAPS model?
- **Q3**: How to develop such framework for being further extended and customized by an open community?

1.4 Scope of the Thesis and Major Contributions

With the radical transformation towards software-based network infrastructures initiated by some of the tier one operators, it is clear that there are several assumptions underlying the work of this thesis.

First of all, the adoption of those new technologies will require a transformation of the network operators' infrastructures.

Even though scalability represents one of the major benefits introduced by the virtualization of network infrastructures, it is clear that such framework can only

provide generic functionalities for supporting scalability operations, and it is the task of the VNFP to define scalability mechanisms at the network service level.

To answer the research questions presented above, the scope of this thesis is focused on the MANO domain, providing the design evolution and implementation process of a framework named MANO4X, compliant with the most relevant standards in the domain, and capable of fulfilling the requirements of network service virtualization and life cycle management. A particular care will be given to requirements of network functions constituting future 5G software-based networks. The heterogeneity of the multisite infrastructure will be a key element driving the design of the solution, particularly devoted to maintaining the level of required QoS expected by those virtualized network services.

Therefore, the author proposes the concept of *"event-based orchestration"* as a major contribution while designing the MANO4X framework, combining several novel design paradigms for distributed applications, like microservices and cloud-native applications, with classical approaches like Model View Control (MVC) and SOA, maintaining compatibility with the proposed ETSI NFV architecture and information model. Events can be of any type, either generated by humans or by other active components of the system. Figure 1.5 provides a very high-level overview of the environment in which the MANO4X framework plays a central role across different horizontal layers.

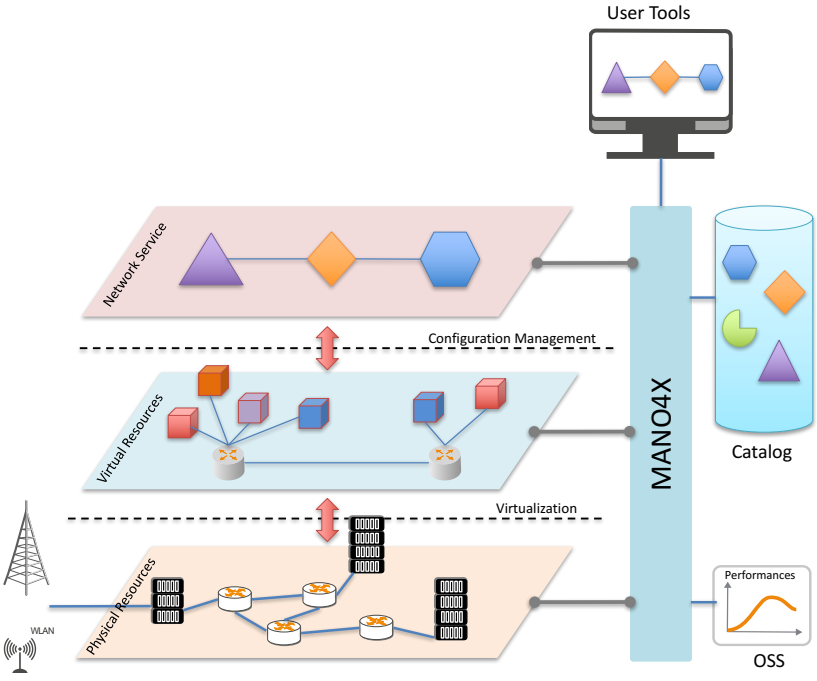

Figure 1.5: A High-Level Overview of the Environment

At the bottom there are physical resources providing computing and networking capabilities. In the middle, cloud computing technologies play an important role

abstracting the physical layer using virtualization technologies. At the top, VNFs (individually exposed through a catalog to the TSP) are composed together in the end-to-end network service required for fulfilling the specific needs of a particular vertical domain. The MANO4X framework orchestrates resources at the three different layers, upon requests executed by the TSP via user tools. During runtime the MANO4X framework coordinates with OSSs for modifying the network service composition in order to maintain the desired QoS level.

The different elements comprising the proposed MANO4X architecture have been categorized in four main surrounding domains which should interact with the central one. The MANO4X framework realizes life cycle management of the network services through the core functional elements of the central domain, orchestrating events generated across the four different domains. Figure 1.6 shows the high-level overview of the different domains identified in the scope of this dissertation.

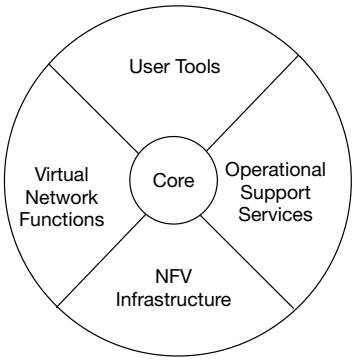

Figure 1.6: Overview of the Main Domains Surrounding the MANO4X Framework

The central domain represents the *core* of the MANO4X framework, comprising components providing brokerage functionalities between the events received from the northbound domain, typically user-driven, and the ones generated by the other three domains (eastbound, westbound, and southbound) typically generated by components contributing to the life cycle management of the end-to-end service.

The proposed framework abstracts the view portion, what users actually see and interact with, exposed via a *northbound* domain comprising *user tools* (CLI, Sofware Development Kit (SDK) or a web-based dashboard) consuming APIs hiding the complexity of the internal orchestration logic, executing life cycle management. In the backend, following microservices principles, each individual component is activated only based on the requirements of a particular use case. This way also the design and development of a particular component providing specific functionalities for a particular scenario could be done independently, and plugged into the framework only if needed.

The *southbound* domain represents the NFVI comprising heterogeneous sites - those being central clouds, MEC nodes, or even FOG devices. The commonalities

between those infrastructures are that they all offer compute, storage, and networking resources as atomic elements. Typically, this domain is comprised of several kinds of Infrastructure as a Service (IaaS) technologies exposing different interfaces for the control of the provided resources. One of the major components belonging to this domain is the Cloud Management System (CMS), also defined as VIM in the ETSI NFV specification, which provides an interface for the on-demand provisioning of those atomic resources. The proposed framework should be able to manage the execution of any kind of compute resources, those being virtual machines, containers, or bare metal, and connect them using any kind of overlay networking technology, also SDN-based.

The *westbound* domain corresponds to the VNF domain. It is the most critical domain for supporting heterogeneous vertical use cases and satisfying interoperability across VNFs provided by different vendors. The ETSI NFV architecture already decouples the network service life cycle management from the VNF life cycle management making use of the VNFM functional entity. Although this logical separation exists as a native separation, MANO4X should support the possibility of incorporating on demand additional VNFMs in a plug-and-play fashion, and accommodate different kinds of VNFMs (either specific or generic).

Last but not least, the *eastside* domain comprises elements usually belonging to the OSS (and consequently BSS) domain contributing to the overall life cycle of the end-to-end network service. Particularly, OSS elements contribute to the runtime phase of the network service execution, ensuring that the aspects defined by the FCAPS model are guaranteed along the overall service life cycle.

The final solution proposed can be considered an extensible[1] and customizable[2] NFV MANO-compliant framework in which different elements could be combined for satisfying the particular requirements of a set of very heterogeneous use cases, being the deployment of the 3GPP EPC on a typical NFVI environment (i.e., OpenStack-based), or the deployment of the 3GPP IMS on top of a MEC node (i.e., container-based).

Open Baton represents the reference implementation of the proposed solution. *"Open"* because of the openness of the solution, considering the major objective of this work to release the source code openly to the community, while *"Baton"* because of the similarities between the music domain and the orchestration domain: As the director needs the baton while managing an orchestra of musicians for playing a particular song, so the administrator needs a tool for managing different VNFs to execute a particular network service.

[1] Extensible without major changes to its architecture and with minimal development efforts.

[2] Customizable for a particular scenario through specific configurations, without requiring modification to the architecture and its implementation.

1.5 Methodology

In order to achieve the results of the work presented in this dissertation, an iterative agile approach based on the Scrum [31] methodology has been performed. The work conducted during this research work has been organized in major releases of a duration of six months, comprising several minor release cycles (sprints in the scrum terminology) of a duration of around two weeks each. Each major release iterated over the architectural solution based on the results of a prototype implementation utilized for validating the design decisions taken in the previous phase. This iterative approach allowed the definition of an always evolving functional architecture, which was validated by several proofs of concept in the context of running large research project collaborations. The initial major release cycle, named the prototype phase, started back in 2012. An overview of the methodology that will be used in this thesis is presented in Figure 1.7.

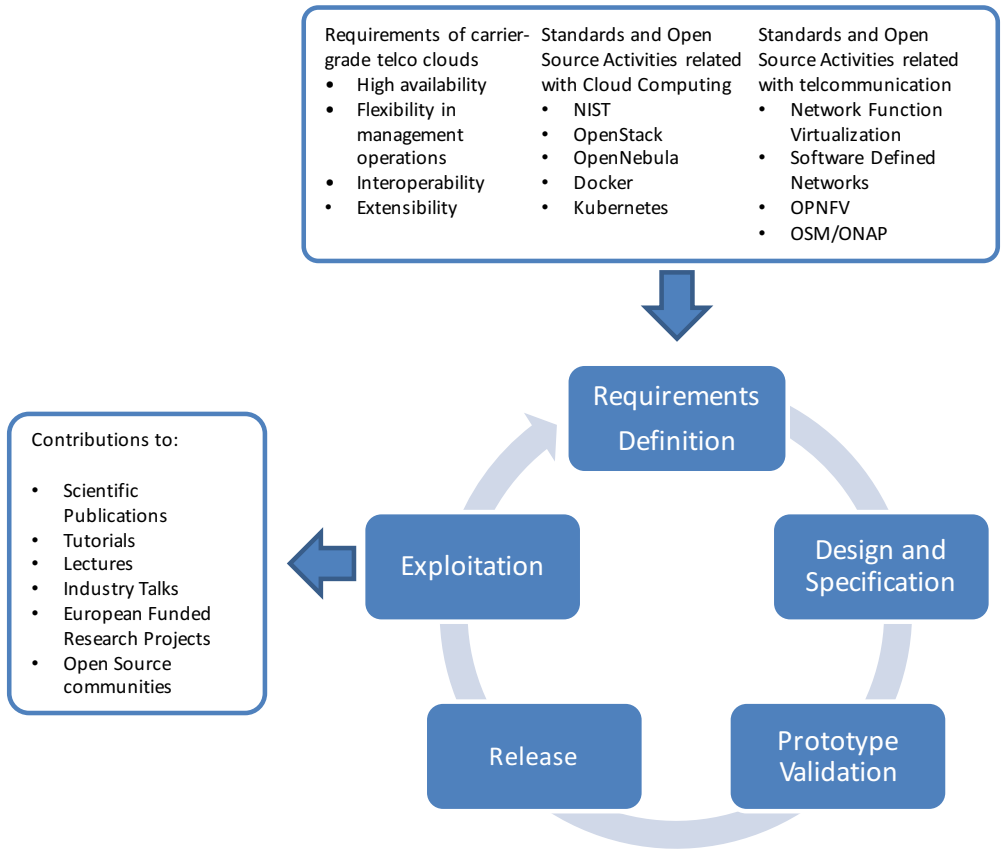

Figure 1.7: Overview of Used Methodology

Considering the iterative agile approach taken for the realization of this research

work, it is important to clarify that the inputs coming from different sources (either standardization activities or advances in open source technologies related to this topic) have always evolved. In practice:

1. A requirement analysis of the major stakeholders (TSPs, VNFPs, and infrastructure providers) as well as a thorough analysis of state-of-the-art and future trends on the topics addressed by this thesis (network management, cloud and virtualization technologies, and NFV), have driven the design and specification of the proposed architecture. Nonfunctional aspects like flexibility, extensibility, and customizability contributed to the final version of the proposed MANO4X framework.

2. One of the most relevant influences is the current virtualization trend initiated by SDN and NFV. Virtualization is paving the way to software-based network services running on top of commodity hardware-distributed infrastructures. This trend provides promising OPEX and CAPEX reductions as well as improved agility in network management operations. ETSI NFV and SDN standardization are playing an important role in defining a common approach.

3. Standardization as well as open source activities in the context of NFV served as influences for the work conducted in this thesis, considering that the alignment to standards, and also standard de facto, is foreseen as one of the major requirements. Other open source solutions, especially the ones claiming compliance with the ETSI NFV architecture, have also been analyzed as influences and finally compared to the proposed designed framework.

1.6 Overview

The rest of the thesis is structured in additional seven chapters as follows.

Chapter 2 – State of the Art provides an overview of the classical network management approaches, and orchestration technologies and specifications that are relevant for the work conducted in this thesis. In particular, a thorough analysis of the cloud computing domain as well as the Network Function Virtualization one will be provided. Furthermore, relevant research work is also considered.

Chapter 3 – The MANO4X Requirements and Features Analysis lists and discusses requirements gathered from different stakeholders in the research scope of this thesis.

Chapter 4 – The Design Evolution of the MANO4X Framework presents the design process of the MANO4X framework as an evolutionary process, presenting the ideas and concepts developed at the early beginning of this research work, and adapted towards the final version of the presented architecture.

Chapter 5 – Specification of the MANO4X Framework describes the design and specification of the MANO4X framework, providing a deep overview of the designed

mechanisms and interfaces. A definition of the functionalities provided by the different components is also provided.

Chapter 6 – Implementation of the Open Baton Framework exposes the implementation details of the implemented Open Baton framework. This chapter provides a detailed view about the selected technologies and software artifacts used for implementing the MANO4X Framework.

Chapter 7 – Validation and Evaluation shows some validation scenarios focusing on the achieved results of the implementation. Details about the evaluation scenarios are given, mainly focusing on those conducted in the context of large research project collaborations.

Chapter 8 – Summary & Outlook concludes the work with a summary, highlighting open research directions.

Chapter 2

State of the Art

The continuous evolution of ICT technologies is paving the way towards a radical transformation of how telecommunication services are currently managed. The evolution towards all-IP-based NGN reduces network infrastructures to a composition of multiple network functions collaborating together for providing feature-rich communication services to end users. Typically, classical NGN infrastructures comprised diverse network functions implemented as monolithic hardware appliances, and connected in a certain way for providing a desired functionality that the network is designed to provide.

On the one hand, network management is definitively not a novel paradigm. Since the early availability of distributed systems over large networks, TSPs started defining standard interfaces and protocols for managing network devices, with the general objective of meeting real-time, operational performances and QoS at a reasonable cost. Network management involved a set of operations applied to the life cycle of a network component.

On the other hand, orchestration is a term broadly used in the IT domain[16], in particular with the arising interest about cloud computing technologies. Orchestration is often discussed in the context of SOA with the main objective of automating system deployments[32].

With the radical transformation currently happening in TSPs' infrastructures, classical NGN NFs are decomposed in hardware and software components with the break-through of cloud computing technologies driven by the virtualization concepts. The great success of cloud computing technologies attracted TSPs that saw the possibility of reducing CAPEX and OPEX while moving towards a horizontal infrastructure where hardware components are based on common architectures.

Decoupling network functions into software and hardware components represents one of the major changes for TSPs that have been used to interoperate with monolithic network functions provided by vendors as black boxes. One of the major aspects of this transition is that network functions that were previously mainly implemented on dedicated hardware, have to be developed as software components running on common hardware infrastructure[33].

Basically, the telecommunication domain started a transition towards a different way of building networks, getting closer to what the IT domain has been doing since its inception, meaning providing services as composition of multiple software components, and automating their life cycle management. Considering the large

advances in software design and development achieved in the IT domain, the TSPs started adopting technologies and architectural principles and applying them to their NGN infrastructures.

This work focuses on designing a framework for managing and orchestrating software-based networks exploiting virtualization technologies provided by cloud-based IaaSs. Thus, this work intends to provide an extensible and customizable framework that could be used for managing and orchestrating network services across heterogeneous cloud-based infrastructures, maintaining compatibility with the ETSI NFV reference architecture.

Therefore, this chapter introduces the concept of network management starting from the classical model, moving towards IT-oriented service orchestration, exposing design principles for managing and orchestrating large scale distributed systems. Furthermore, an extensive overview about the ETSI NFV standardization activities is given, considering the relevance of this standard for the final design of the proposed framework.

2.1 The Evolution of Network Management in Next Generation Network (NGN)

Management is a very broad term applied to several domains of our society, including business, computing, and medicine[1]. The simplest definition of management, mainly applied to the business domain, is given by "businessdictionary" as *the organization and coordination of the activities of a business in order to achieve defined objectives*[2]. The *"What"* is being managed, and the *"How"* management operations are executed, these are the key factors for further reducing this general definition.

Focusing on the telecommunication domain, managing networks has always been a nontrivial task. According to Gartner[3], Network Management comprises *"[...] applications designed to isolate and resolve faults on the network, measure and optimize performance, manage the network topology, track resource use over time, initially provision and reconfigure elements, and account for network elements. Suites that include fault monitoring and diagnosis, provisioning/configuration, accounting, performance management, and Transmission Control Protocol (TCP)/IP application management — but only for networks — are also included here. This network management segment is intended for products that are mainly or entirely network-oriented and used primarily by enterprises [...]"*.

Another important definition was given by Saydam and Magedanz in 1996 in a Journal about Network and System Management [25]: *"[...] Network management involves the deployment, integration and coordination of all the hardware, software and human elements to monitor, test, poll, configure, analyze, evaluate, and control the network and element resources to meet the real-time, operational performance*

[1] https://en.wikipedia.org/wiki/Management_(*disambiguation*)
[2] http://www.businessdictionary.com/definition/management.html
[3] http://www.gartner.com/it-glossary/network-management/

and QoS requirements at reasonable cost. Service management involves the creation, access, usage and management of value-added services using the logical, virtual and physical network resources and the network management systems. The separation of service, service management, and network resources is crucial in creating open, transparent and reconfigurable services [...]".

The term network management is typically associated to the process of administering and managing the computer networks of one of many organisations. Although network management is a subset of system management, network management principles have influenced quite a lot the progresses in the system management domain[34]. Initially, in the ICT domain, networks were small and most of the time local. The job of a network administrator was mainly related to the installation and configuration of Personal Computers (PCs) and devices for serving the needs of a particular organization. Since the early days the process of managing a network infrastructure has been critical for guaranteeing certain levels of QoS for connected users. This definition underlines the fact that network management is broadly used for identifying a set of hardware and/or software tools, namely Network Management System, allowing network administrators to supervise the individual components of a network infrastructure.

The evolution towards NGN infrastructures paved the way to a different way of managing networks, dealing with network functions as services exposing standardized interfaces and communicating over the network. Moreover, the convergence towards an all-IP- based infrastructure as well as the transition towards *"everything as a software"* paved the way to an ICT-like approach for managing distributed systems. Furthermore, automating the process of coordinating and managing those distributed systems, typically referred to as orchestration[4], implies the transition towards a different set of architectures and solutions.

To limit the scope of this thesis, the focus is set on management of software-based network functions. Differentiating between *"what"* is being managed in a network infrastructure represents a crucial aspect for limiting the scope of this research work. It is clear that there should be a distinction between network management, as it was initially conceived, and network function management and orchestration as per the more recent advances within the ICT domain and the telecommunication domain as introduced by the ETSI NFV standardization activities.

In order to fully understand the scope of this work, in the following sections the author i) presents classical network management systems in NGNs, ii) provides an overview about the ICT domain, mainly focusing on the advances of virtualization and cloud computing technologies, iii) and finally gives an overview of the novel concepts introduced by the NFV paradigm.

2.1.1 Classical Network Management Protocols and Interfaces

As already discussed network management can be identified with a set of *"tasks"* executed by a network administrator (also referred to as network manager) for meet-

[4] https://en.wikipedia.org/wiki/Orchestration(*computing*)

ing the particular set of requirements of an organization in alignment with the business requests. In the very early days, a network manager was responsible of installing PCs, printers on a Local Area Network (LAN), configuring Network Interface Controllers (NICs), protocol stacks, user applications, and testing, operating several kinds of devices (i.e., routers, switches, etc.). This kind of job, dealing only with the configuration of network elements, was not enough for guaranteeing the expected performances of end users. Optimize performances, handle failures and network changes, extend network capacity, manage network usages, and ensure security, were some of the issues that were identified and paved the way towards the standardization of network management protocols and tools.

Network management tools and protocols were initially standardized as part of the International Telecommunication Union (ITU) - Telecommunication Standardization Sector (ITU-T) and IETF standardization activities and included protocols and tools for managing the network remotely. To control the network a set of three different protocols and systems were identified (Simple Network Management Protocol[35], Management Information Base[36], and Network Management System), while for monitoring the Remote Network MONitoring protocol was proposed. In particular, Simple Network Management Protocol (SNMP) became the de facto standard for management of TCPIP networks[37].

2.1.1.1 Telecommunications Management Network (TMN)

In 2000 the ITU-T introduced the concept of Telecommunications Management Network as a protocol model for managing open systems in a communication network. The TMN concept has been widely adopted for standardizing a set of protocols and interfaces for managing heterogeneous network elements. The TMN standard definition span across a large number of ITU-T documents as part of the M.3000 recommendation series[38], in particular M.3010[39], M.3400[27] and X.700[40].

TMN is mainly designed for public networks and its main objective is to optimize network functionalities in multivendor environments. This set of recommendations also defined an architecture to support the management requirements of TSP to plan, provision, install, maintain, operate, and administer telecommunication networks and services. TMN defined an architecture for combining various types of Operating Systems (OSs) for the exchange of management information using standardized interfaces including protocols and messages.

The approach introduced by the ITU-T M.3010 recommendations was based on the assumption that a TMN *"can vary in complexity from a very simple connection between an OS and a single piece of telecommunications equipment to a complex network interconnecting many different types of OSs and telecommunications equipment"*[39].

ITU-T M.3200[41] provided the scope of the TMN architecture distinguishing between Telecommunications Managed Areas (TMA), identifying the different resources being managed, and TMN Management Services relating to the set of processes needed to achieve business objectives, while M.3400[27] and X.700[40] defined

the specification and classification of the management functionalities.

Another relevant aspect of the ITU-T M.3400[27] recommendations is represented by the FCAPS that was introduced for categorizing those different categories of network management[42][43]. Recommendations are classified into the following categories:

- **Fault Management**: Prevention, detection, and isolation of unexpected situations (i.e., a malfunction in a network device) in order to guarantee the continuous network operation.
- **Configuration Management**: Manage the overall system life cycle, particularly its configuration, keeping an appropriate level of control of each component (software and hardware) of the network. During runtime modify the network configuration in reaction to external events (i.e., faults, congestion, etc.).
- **Accounting Management**: Control the usage of the network resources in order to properly measure costs and produce appropriate fees.
- **Performance Management**: Monitor and analyze networking metrics (i.e., response time, throughput, packet loss, etc.) identifying risky situations (i.e., network congestion), keeping a history for later analysis.
- **Security Management**: Control access to network and resources, including authentication and authorization mechanisms.

The research work conducted in the context of this dissertation primarily focuses on the aspects related to the configuration management applied to NGN elements. Configuration management has been one of the major aspects of the ETSI NFV ISG, however, before digging into the novel concepts introduced by the NFV it is necessary to i) introduce NGN architectures, and ii) provide an overview of the existing mechanisms for network function management.

2.1.2 Overview of Next Generation Network (NGN) and Key Definitions

Back in 2004, ITU-T introduced the NGN term describing the evolution towards a new generation of network infrastructures based on the all-IP architectural concept[44]. Standard network infrastructures are migrated from circuit-switched networks to packet-switched networks. Although there are several other novel concepts introduced with the NGN evolution, a key aspect related with the scope of this work is the separation of transport and service layers. The ITU-T's general overview of NGN[45] defines a NGN as *"[...]a packet-based network able to provide services including Telecommunication Services and able to make use of multiple broadband, QoS-enabled transport technologies, and in which service-related functions are independent from underlying transport-related technologies. It offers users unrestricted*

access to different service providers. It supports generalized mobility which will allow consistent and ubiquitous provision of services to users[...]" In other words, the main idea is to move towards a common transport layer for supporting different kinds of services so that the service provided is independent from the network.

The NGN service network follows ICT concepts having the IT part mainly devoted to service delivery, and the communication technology part mainly defining the data plane. NGN also aims to tackle important concerns raised from the use of current IP-based services (i.e., QoS and security) [46].

The term convergence describes the combination of all the multimedia services in a single platform. Video, music, television, telephone, and other services available from multiple devices must be accessible from one single device. Although today everything seems already possible, the original idea was to provide full convergence across different kinds of services through a single infrastructure using IP as a single network protocol.

An overview of the NGN architecture as formalized by multiple standardization bodies (ITU-T, 3GPP, and ETSI), is depicted in Figure 2.1.

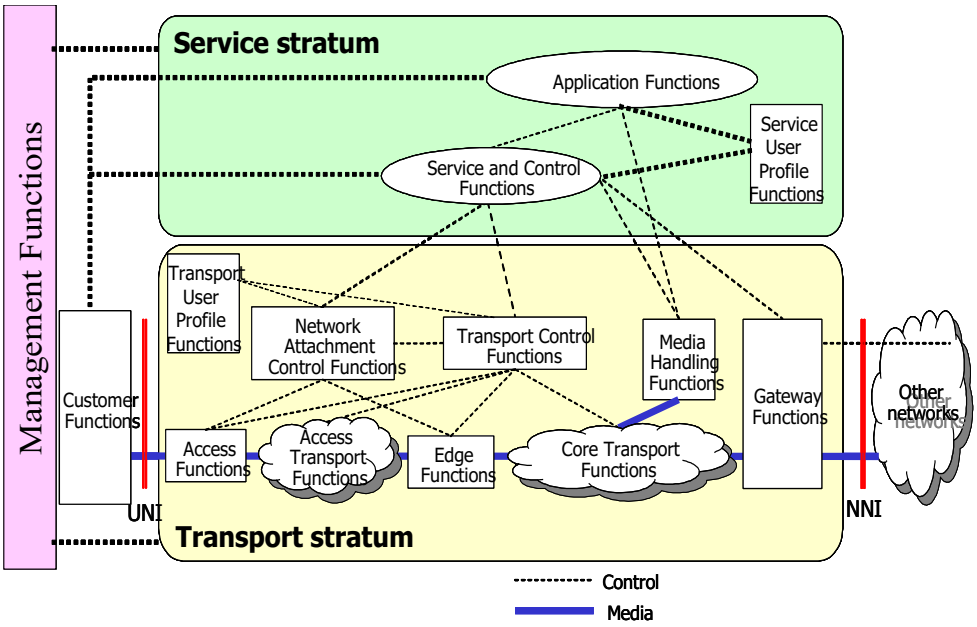

Figure 2.1: NGN Architecture[45]

This picture shows the high-level view of the architecture depicting the major parts as follows [46]:

- Transport Stratum: Comprising various functional elements to provide access, control, and interworking of the common IP transport infrastructure
- Service Stratum: Layer required for supporting several NGN applications
- Gateways to other networks

- Customer profile functions
- Comprehensive management functions to manage the overall NGN environment

As can be seen from the definition given by the ITU-T about NGN, a very complex and flexible environment will be required for managing NGNs.

The first key-added value of NGN architectures is represented by the decomposition of network infrastructures into NFs and interfaces. Each individual NF provides well-defined functionalities based on the input and output information received on its exposed interfaces. A NGN comprises several kinds of NFs that may be connected or chained together for providing a certain business logic required by a particular communication service (i.e., voice, connectivity, etc.).

The second key-added value of NGN architectures is the separation between the transport and service layers, enabling a richer spectrum of services than what TSP typically provided to end customers, in a significantly shorter time. Indeed, the key success of a TSP is the possibility of always creating, deploying, and delivering new services to the market. Therefore, versatile mechanisms for creating new services is the key for the success of future NGN infrastructures.

Several activities have taken place since the time the NGN concept was introduced in order to define an instantiation of the NGN concepts. So far the most accepted model is represented by the IMS[47]. The term service was initially adopted in the context of the IMS architecture. Services (standardized by Open Mobile Alliance (OMA)) like presence, group, and list management, and simple instant messaging, represented and still represent the most basic functional blocks of many NGN services. Principles of service composition showed to be, since the very early stages, very powerful for creating new service landscapes. Lately, the concepts of IMS were further adopted by other standardization bodies and communication services. Nowadays IMS is the most recognized NGN infrastructure providing convergence between legacy telephony services and Voice over IP (VoIP). Figure 2.2 provides an architectural diagram of the IMS functions and reference points.

Looking at the IMS architecture, what can be noticed is the complexity of such systems[48]. One can think of several deployment models of such an architecture. Considering the complexity of the NGN environment, a management system has to be comprehensive, integrated, and extremely flexible. Several different approaches towards classical NGN management were taken in the last decades by different standardization bodies. Those approaches are further presented in the following subsections.

2.1.3 ETSI NGN Management Architecture

TeleManagement Forum uses a business and customer services driven approach to achieving end-to-end automation using integrated Commercial off-the-shelf (COTS) software[49]. The TeleManagement Forum (TMF) focus is on providing pragmatic

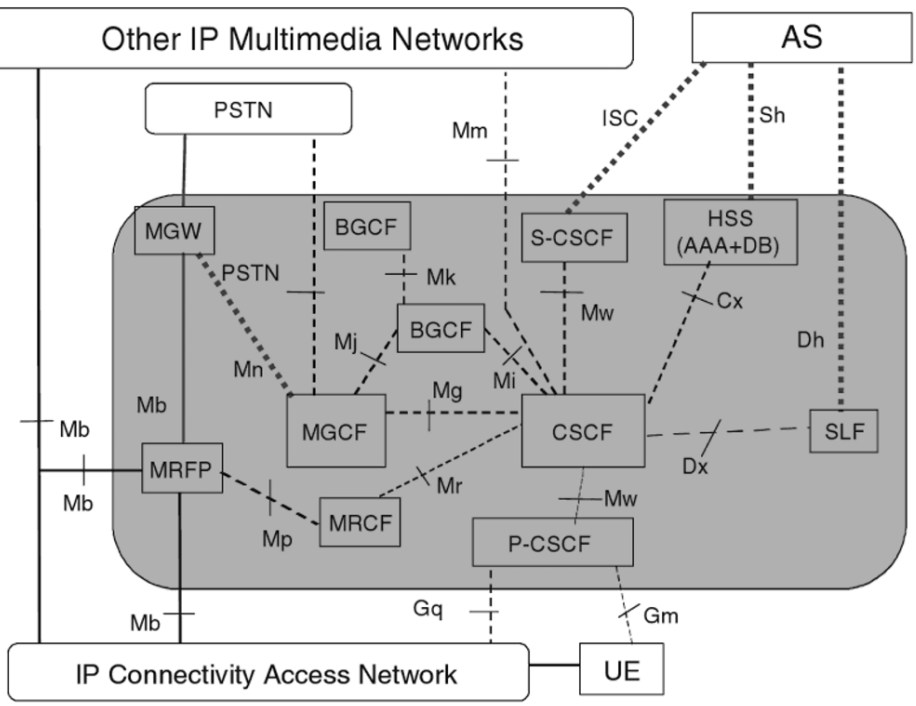

Figure 2.2: IMS Architecture

solutions to business problems and is based on the business layering principles articulated in the ITU-T TMN model.

TMF designed a framework comprising a set of technologies suitable for implementing most of the requirements of an NGN management system. Such system is typically referred to as New Generation OSS (NGOSS). Such OSS should provide the following characteristics to fulfill the NGN business vision and optimize system development speed:

- all systems must be defined in a technology neutral form following a component-based SOA
- a common *information architecture* should be adopted across all existing management applications in order to i) share information across multiple areas of Management, ii) develop capabilities for retrieving end-to-end service measurement data, and iii) provide policy-based management for already existing or yet to be defined services
- a common *business process framework* must be adopted across OSS solutions
- a solution must be provided for managing services independently from network technologies

As it can be noticed, those characteristics of an OSS system may refer to different application areas in NGN infrastructures.

2.2 The Virtualized Cloud Era

Figure 2.3 introduces the different phases during the last decades, providing an overview about the main *"Eras"* defined as a period of time in which a particular development model has been widely used in software design.

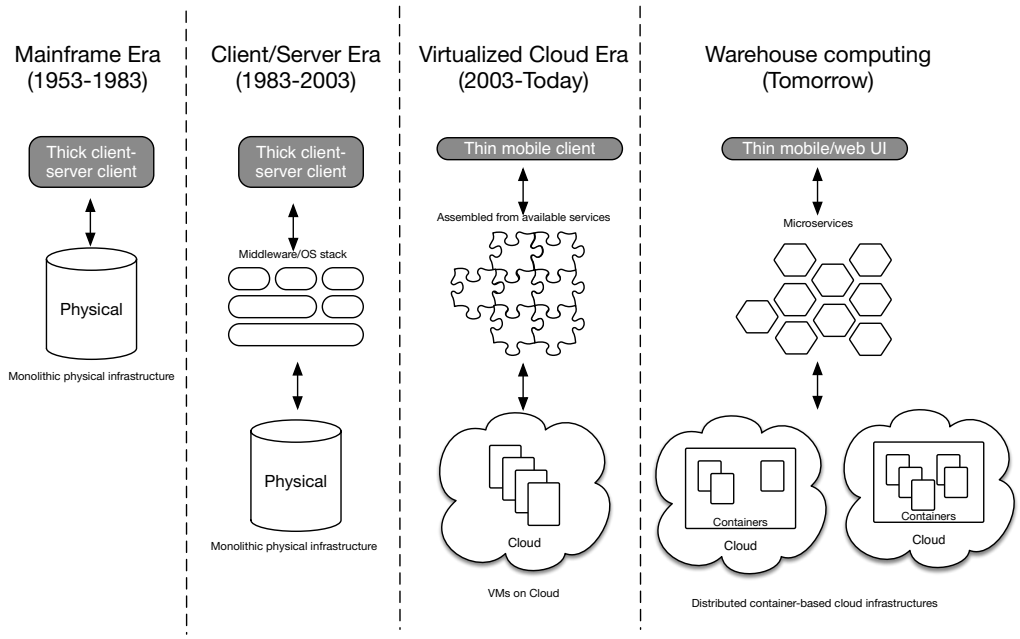

Figure 2.3: Historical Models

Figure 2.3 starts with a very early era when the mainframe was initially introduced as a totally disruptive approach for computing purposes. Although nowadays this phase may not be too relevant considering the always increasing number of improvements in miniaturizing processors, it is important to highlight some pros and cons of such an approach as relevant for its successor phases. This centralized model highlights the monolithic approach of putting together the presentation and business logic with the data storage. A dumb terminal (i.e., a display monitor with output capabilities) connects directly to the mainframe and provides output to the end user. On the one hand, with this approach there was no need to manage client-side applications, and it was rather easy to obtain data consistency. On the other hand, as for any monolithic application, it was rather complex to maintain and develop the code between different releases.

The next temporal phase was the introduction of the client-server architecture that was facilitated by the initial advances achieved in Internet technologies[50]. The so-called middleware [51], defined in an IETF workshop in 2000 as *"those services found above the transport (i.e., over TCP/IP) layer set of services but below*

the application environment" (i.e., below application level APIs), was a major step towards distributed architectures with its multitier architectural model. With this model there was a separation between the different entities composing a service. All the common aspects related to security, concurrency, performances, etc. were handled at the middleware level, however, in most of the cases, a thick (also called fat) client was required for making the user interoperate with the backend service. The approach of thick clients required huge investments in terms of maintainance: Every change would need to be propagated to all existing clients. Common Object Request Broker Architecture (CORBA) was the result of a combination of concepts from client-server architectures with the newly emerging paradigm of object-oriented development [52].

The third phase, the so-called *"Virtualized Cloud Era"*, represents the current "Era". The introduction of virtualization technologies enabled novel mechanisms for conceiving new services and exposing them to end users. Cloud computing represents the main model facilitating the transition towards a different way of defining and delivering services.

According to the definition proposed by NIST in the NIST Cloud Computing Standards Roadmap, dated July 2011, cloud computing is defined as a *"model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models"*[53].

However, the underlying concept of cloud computing dates back to the 1960s when John McCarthy spoke at the Massachusetts Institute of Technology (MIT) Centennial saying that *"computation may someday be organized as a public utility just as the telephone system is a public utility"* [54].

Figure 2.4 shows the different cloud models as presented by the NIST. Those cloud models and characteristics will be further detailed in the next sections.

Based on this definition cloud computing can be seen not as a product, but as a computing model or paradigm. Cloud computing is the evolution of the widespread adoption of virtualization technologies and SOAs. End users do not have the knowledge anymore about the details of the underneath hardware infrastructure, while cloud computing technologies support them [55].

Although cloud computing technologies can be considered mature enough and production-ready, there are still several open research challenges as identified in 2015 by R. Jennings and R. Stadler who provided a comprehensive survey about resource management in cloud computing[56]. In their article they survey the recent literature covering 250+ publications, and they identify major challenges that have to be addressed.

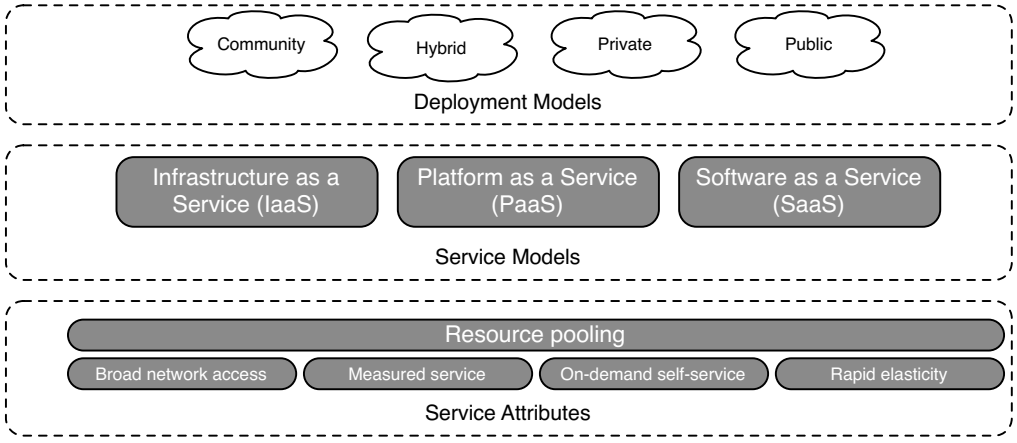

Figure 2.4: Cloud Models as Presented by the NIST

2.2.1 Cloud Computing Benefits

Cloud computing enables new business models and cost-effective resource usage by providing virtualized computing resources as a service in a pay-as-you-go manner. Instead of investing high amounts of resources to maintain their own data centers, companies can concentrate on their main business purchasing resources only when needed. Particularly when combining a privately maintained virtual infrastructure with publicly accessible clouds in a hybrid cloud, the technology can open up new opportunities for business and help consolidate resources [57].

Figure 2.5 provides an overview of the main benefits introduced by cloud computing. In particular, it is important to underline a few of them, like elasticity and pay-per-use model, because they introduced a completely innovative approach in a very static environment like the one of service hosting.

2.2.2 Virtualization and Containerization

Virtualization is the key enabler technology for cloud computing. It represents the capability of executing multiple logical (virtual) instances on top of physical ones [58]. Initially virtualization was considered a method for logically dividing mainframes to allow multiple applications to run simultaneously [59]. In other words, it is the abstraction of physical hardware to appear as multiple logical instances. Virtualization technology enables the sharing (splitting) or combining (aggregation) of computing resources to other virtual ones.

Recently, with the highly increasing interest in cloud computing technologies, virtualization became a very important technology for the industry, being used in many different scenarios and use cases. Although virtualization is applied to different domains, including networking [60][61] and storage, server virtualization represents

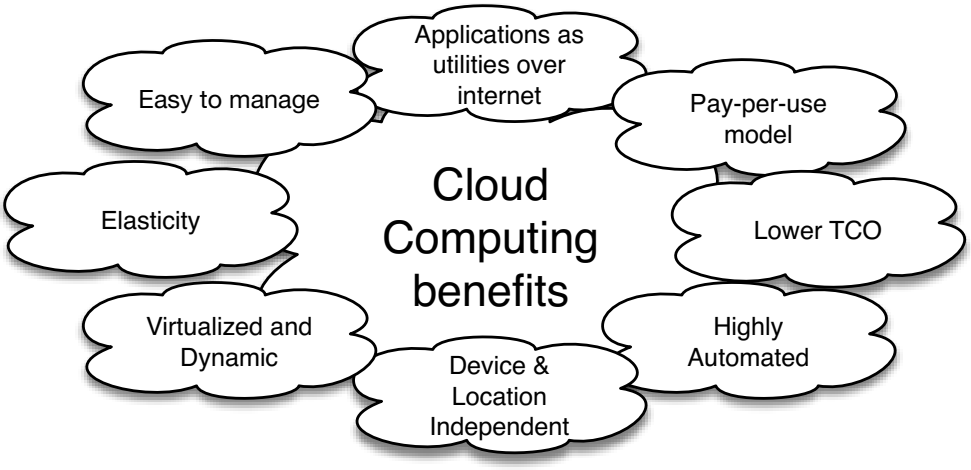

Figure 2.5: Cloud Computing Benefits

the most dominant technique in cloud computing [62].

Server virtualization allows the execution of multiple OS kernels on top of the same hardware [63]. Virtualization technologies allow abstracting physical resources exposing interfaces towards the guest operating systems in a seamless way. Similar to the OS that abstracts hardware components guaranteeing the users access, virtualization allows the usage of physical resources through a common interface, without requiring the knowledge of the available physical resources. There are two main components on which virtualization technologies are based:

- Virtual Machine (VM) being the representation of a server, provided by the hardware which is required for hosting a guest operating system. It could be stored in a form of a disk image, including all the required resources and characteristics. It is important to underline that in the cloud computing space a VM could be moved from a physical server to another.

- Hypervisor, also called Virtual Machine Manager (VMM), being the component managing the guest OSs in execution on a physical server, presenting them a virtualized view of the physical resources available.

There are various ways for supporting virtualization differing for the level of abstraction provided and their performances. There are five different main variants of virtualization[59][63]:

- *Full Virtualization*: The guest operating system does not need to be modified, and all its system calls are intercepted by the hypervisor that is emulating them. This approach is the most generic and flexible one since it completely

separates the guest operating system from the underneath hardware components. However, it introduces some overhead which is negatively influencing performances. Examples of hypervisors of this type are XEN[5][64] and KVM[6][65].

- *Paravirtualization*: This approach requires some modifications to the guest OS since the system calls should be substituted with calls to the hypervisor, which abstracts the underneath hardware providing similar interfaces. It is a solution mainly used in cases where the hardware does not support virtualization, therefore, hypervisor technologies are installed on top of the OS. Some modified versions of XEN support this kind of virtualization.

- *Hosted Virtualization*: The hypervisors are installed on top of the *host* OS. Examples of hypervisors of this type are VMWare Workstation[7] and Virtualbox[8]

- *Emulator*: Emulates the hardware resources enabling software that was compiled for a particular architecture to execute on a different one. QEMU[9][66] is an example of such a hypervisor type.

- *Container-based Virtualization*: A virtualization technology that is not based on the hypervisor entity[67]. This solution envisages a shared host operating system, having a kernel appropriately modified.

In particular, the advent of ***software containerization*** during recent years has drastically transformed the way software is deployed and managed. A software container, also known as jail[10], is an isolated user space instance of an OS. A relatively large number of containers can be executed on the same physical machine, all having a dedicated and isolated file system and running processes. The container behaves like a virtual machine, however, it does not rely on the intermediate layer provided by the hypervisor, therefore, having less performances overhead[67]. Figure 2.6 shows the architectural differences between standard virtualization technologies and containerization ones.

The architectural view depicts the lower amount of layers required by containerization technologies for executing guest OSs. Some of the most relevant container implementations are currently FreeBSD Jails[11], Linux Containers (LXC)[12], OpenVZ[13], rkt[14], and Docker[15].

[5] http://www.xenproject.org
[6] http://www.linux-kvm.org
[7] http://www.vmware.com
[8] https://www.virtualbox.org
[9] http://wiki.qemu.org
[10] https://en.wikipedia.org/wiki/Operating-system-level_virtualization
[11] https://wiki.freebsd.org/Jails
[12] https://linuxcontainers.org/
[13] https://openvz.org/Main_Page
[14] https://coreos.com/rkt
[15] https://www.docker.com/

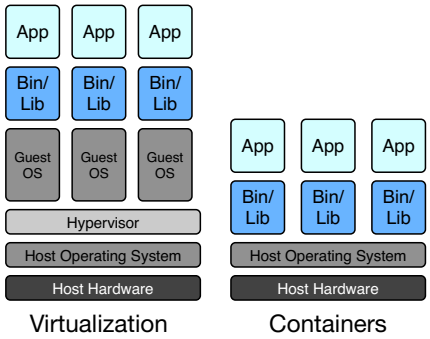

Figure 2.6: Architectural Differences between Virtualization and Containerization

2.2.3 Cloud Computing Service Models

Basically computing resources are delivered over the Internet and elastic scalability can be achieved for any kind of application. Instead of regarding individual machines, cloud computing treats resources as a utility, i.e., computing time and storage are provisioned on demand and paid per usage without the need for any upfront commitment [57]. Figure 2.7 shows the different cloud computing service models.

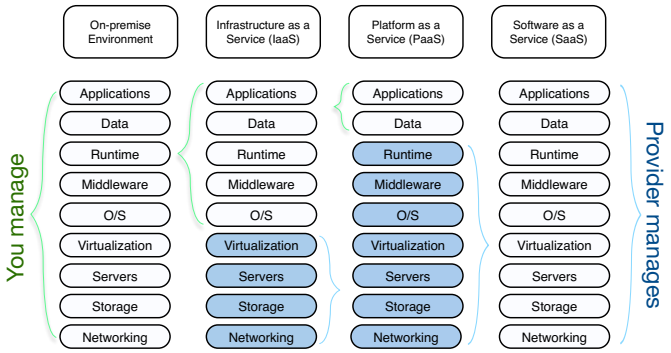

Figure 2.7: Cloud Computing Service Models

According to the NIST document, the *Cloud Computing Standards Roadmap*, three different major cloud service models are recognized: *Cloud Software as a Service*, *Cloud Platform as a Service*, and *Cloud Infrastructure as a Service*[68].

Figure 2.8 depicts an overview of the different cloud service models detailing the different software stacks required by each of the different layers for providing the specific required functionality.

In the following subsections the three different cloud service models are presented and an overview is given of the software stacks depicted in the previous Figure 2.8.

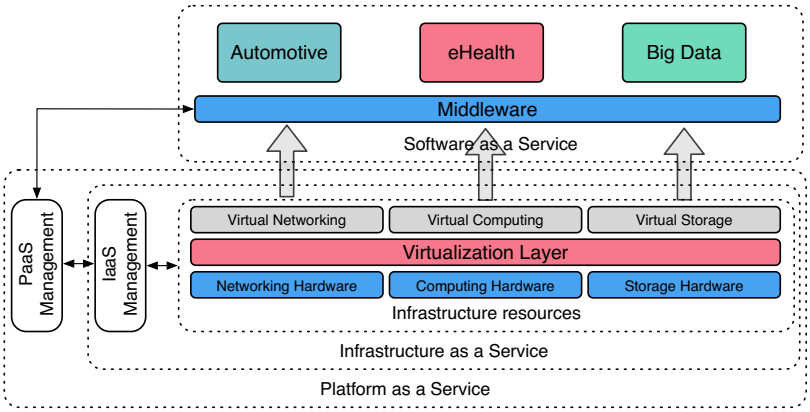

Figure 2.8: Cloud Software Stack

2.2.3.1 Software as a Service (SaaS)

SaaS is an innovative approach for offering the same software to different users in an isolated way via networking. SaaS is defined by NIST as:

> *"The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., Web-based e-mail), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings."*

One of the most important things is that each user thinks he is the only one using that particular software, without necessarily having multiple instances of that software. Indeed, the application should be aware of the different users and should provide the required isolation so that software components could be shared among multiple users. In some cases it is also required that a particular configuration of the software is available for a particular user. SaaS typically comprises a middleware entity, interacting with the PaaS management system for deploying applications and offering them directly to end users.

2.2.3.2 Platform as a Service (PaaS)

The main objective of a PaaS is to host software artifacts implemented by different users on top of a shared execution environment, offering the most common functionalities required. PaaS is defined by NIST as:

> *"The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using*

> *programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment. "*

In order to do so it is necessary to increase the level of isolation that should not be guaranteed anymore only at the virtual machine level, but also at the software component level, which could ideally be hosted and executed on the same server. Therefore, the platform itself should be aware of the multiple users and should avoid that components of a particular user access data and functionalities of components developed by other users. In the same cases it should also support the possibility of sharing components among multiple users, and should provide access to shared storage.

Such requirements are satisfied providing a set of APIs allowing users to deploy their applications on top of the platform. Typically, PaaS offers a set of supporting services (like databases) that could be used on demand by the deployed applications.

The PaaS layer comprises mainly a PaaS management system enabling developers to deploy on demand their applications, and interacting with the IaaS layer for acquiring the virtualized resources needed.

2.2.3.3 Infrastructure as a Service (IaaS)

The IaaS layer comprises infrastructure resources in terms of hardware resources, providing networking, computing, and storage as atomic elements via a virtualization layer.

IaaS is defined by NIST as:

> *"The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software that can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure, but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)."*

The IaaS management component is in charge of exposing an interface for providing virtualized resources as a service either to end users or to another layer of the cloud stack providing a different cloud service model.

2.2.3.4 Open Cloud Computing Interface (OCCI)

OCCI [69] comprises a set of specifications initially released around 2009 through the Open Grid Forum (OGF) organization. The main aim of the OCCI specification is to provide a set of APIs for remotely managing cloud resources provided by different providers. The initial version of the OCCI specification mainly addressed the IaaS

layer proposing a REST-based API. In such models, resources are identified by a Uniform Resource Locator (URL), and users can interact with those resources using standard Hypertext Transfer Protocol (HTTP) methods. Resources could also be linked together based on user needs.

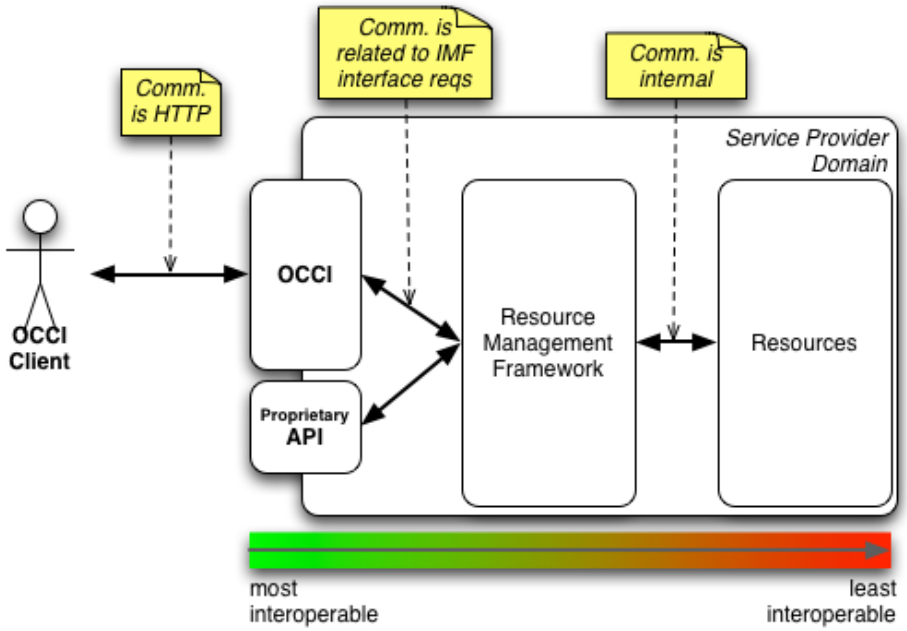

Figure 2.9: The OCCI Architectural Model[69]

The OCCI architecture is rather modular and extensible. The latest version available (1.1) comprises three major modules:

- OCCI *Core* represents the core module of the standard, provides mechanisms and semantics as well as definitions and classes for describing and managing cloud resources. This model is independent from the others and could be used as a stand-alone component in different contexts. It provides an abstraction of the real resources using the class *Resource* providing a set of attributes common across different kinds of resources, interconnected via the *Link* class. The *Entity* abstract class is used together with the *Kind* and *Category* classes for classifying and identifying resources across several cloud providers.

- OCCI *Infrastructure* represents an extension to the *Core* module providing an abstraction specifically for the IaaS layer. The three infrastructural types defined by this module are the *Compute*, *Network*, and *Storage* resources that are connected via *NetworkInterface* or *StorageLink*.

- OCCI *RESTFul HTTP Rendering* describes the serialization format used for the communication between client and server. It is based on the concept of

Resource Oriented Architecture using the HTTP protocol for identifying and controlling resources provided by a cloud provider.

The main motivations behind the OCCI specification are interoperability, portability, integration, innovation, and reusability.

2.2.3.5 OpenStack as De Facto Standard IaaS Solution

OpenStack is an open source project providing an IaaS solution for building private cloud infrastructures. OpenStack represents a cloud management layer. It was released in 2010, initially developed by National Aeronautics and Space Administration (NASA) and Rackspace. Although other similar solutions exist and provide a rather comprehensive set of functionalities required at the IaaS level[70][71], OpenStack represents the standard de facto platform within the ETSI NFV architecture.

The OpenStack architecture follows microservices principles, having loosely coupled modules that have specific functionality. It is also based on a distributed architecture using a message bus for the communication between all components. There are many different projects currently under the OpenStack foundation[16]. In principle, OpenStack is a cloud operating system controlling a large pool of compute, storage and networking resources available in a datacenter. It is composed of 6 key core services and 13 optional services. Figure 2.10 shows the high-level functional architecture of OpenStack including its 6 core services and the dashboard.

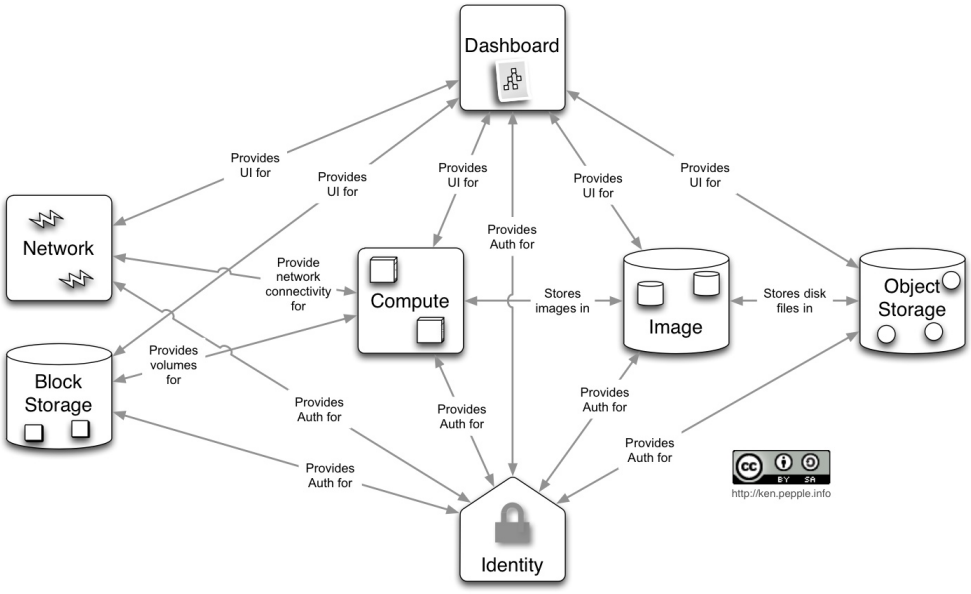

Figure 2.10: The OpenStack Logical Architecture

A quick overview of each service provided by OpenStack:

[16] https://www.openstack.org/software/

- Compute (codename: nova) manages compute hardware resources providing virtualized compute (virtual machines) resources. In practice, nova communicates with the physical servers, also known as compute nodes, on top of which virtual machines need to be deployed. It interacts with the hypervisor transforming the requirements received via nova APIs to the hypervisor information model, and deploying them. In the context of this project, the Compute service is used for on-demand deployments of virtualized compute resources (virtual machines) on top of which VNFs are executing.

- Network (codename: neutron) provides connectivity as a service controlling heterogeneous networking equipments. Connectivity is provided in terms of overlay networks that can be attached to running compute entities. It exposes an API for users to define their networking requirements. In the context of this project, the Network service is used for connecting virtualized compute resources hosting VNFs. This service makes use of those additional technologies in order to realize virtualized networks:

  - OpenVSwitch (OvS)[17][72] as a software switch implementation for instantiating different virtual network layers between VMs.
  - Physical Switch (not necessarily SDN based) for connecting compute nodes.

- Block storage (codename: cinder) exposes storage resources as a service to end users that can be consumed by the Compute service. In practice it creates an abstraction on top of existing block storage management technologies (like Logical Volume Manager) in order to offer them via a simplified API.

- Identity (codename: keystone) provides authorization and authentication mechanisms for all OpenStack services. It supports token-based authentication and it can be further extended to support several types of security mechanisms.

- Image (codename: glance) manages virtual machine disk images that are used by the OpenStack compute service during provisioning.

Considering the complexity of setting up such distributed environments, several "distributions" of the OpenStack platform have been released over the years. Major companies behind open source initiatives like RedHat and Canonical have launched OpenStack installers that can be used for easily setting up a whole environment requiring minimal effort.

2.2.4 Design Principles for Cloud-Native Architectures

The following part of this section presents the different styles of design for distributed architectures, particularly focusing on those architectural models that have driven the design of the proposed MANO4X architecture and that will be presented in

[17] http://openvswitch.org/

the Chapter 4. For the sake of clarity, design principles and methods presented in this section can generally be applied to any kind of software architecture. Concepts presented as follows can be applied to the design process of any kind of software component for increasing extensibility and customizability. Thus, those approaches could also be utilized while designing and developing software-based networks.

2.2.4.1 Service-Oriented Architecture (SOA)

SOA is an architectural model in which software applications are decoupled in modular components, defined as *services*, and made available either to users or to other services via networking [73]. SOA is a collection of auto-contained services communicating with each other, executing a particular business logic using a standard language. A system designed using the SOA approach is constituted by well-defined services, independent from each other. Each service provides a certain functionality and can make use of functionalities provided by other services.

The term SOA is typically used for defining a software architecture to support the usage of Web services for ensuring interoperability between different systems, to allow the usage of single applications as components of the business process and to satisfy the requests from users in a seamless way. SOA is an architectural style with the objective of obtaining a loosely coupled interaction between different software components interoperating with each other. A service is a work unit executed by a service provider for a user with the objective of obtaining a particular result. Both the service provider and the users are roles executed by the software acting on request of the respective owners.

The Organization for the Advancement of Structured Information Standards (OASIS) defined the notion of SOA as *SOA is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.* [74].

The basic principle of a SOA is that developers build different services of a monolithic application, and those services should be installable and reusable for supporting different kinds of applications and processes. The immediate benefits of such an approach are straight forward: Enhanced reusability of the functionalities provided by the software, and flexibility, considering that developers could focus on an individual service without interfering with the consumer of such service. One of the major requirements of a SOA is the decoupling of the interface of the service from its implementation.

A service should be discoverable dynamically, self-contained and modular, defined via interfaces and independent from the implementation, loosely coupled, available on the network, and coarse-grained.

Principle actors in an SOA-based system are:

- *Service Provider*: It is the entity providing services. Such services should be visible on the network, therefore, information about these services should be publicly available via the Service Registry.

- *Service Consumer*: It is the entity consuming functions provided by Service Providers.
- *Service Registry*: It is the entity maintaining all the information about available services (URL, access methodology).

2.2.4.2 Cloud Native and Microservices Architectural Principles

Elasticity, defined as the capability of cloud service infrastructures to dynamically scale (out/in or up/down) compute, storage, and network resources, is one of the most important characteristics of cloud computing[75]. In order to cope with the increasing data traffic, applications should be redesigned specifically for meeting elasticity requirements.

Drawing on SOA principles, cloud native applications are typically based on loosely coupled cloud services leveraging nonblocking communication patterns[76]. Tasks in cloud native applications are broken down by the developers into separate services (meets the flexibility demand and improved time-to-market) and thus can consequently run on several servers in different locations (decentralization of cloud instances). Separate services enable a more selective failure detection within the network and a more efficient use of resources.

By using cloud native applications, the pay-as-you-go pricing model prevents upfront costs and is a step forward in terms of economical reasons as well. If the demand increases, new resources are automatically added based on proactive and/or reactive actions, also known as scaling-out actions. On the other hand, if the demand decreases, resources could be released via scaling-in actions[77]. All those scaling actions should have almost no impact on the QoE perceived by end users, therefore, a cloud native application should, by design, support any kind of runtime modification without lowering performances and QoE[78]. The main aim of these technologies is, therefore, to host and develop applications which is, at best, a cloud native solution. In case of resilience optimization handling with failures of commodity hardware, virtualized resources or services is pointed out[79][80].

Microservices architecture is a novel emergent approach utilized for designing and developing large-scale cloud applications that require high scalability and fast evolution. The idea behind microservices architectures is not completely new as it is an evolution of the Service-Oriented Architecture, defined also as SOA light or SOA fine-grained[81], as it focuses as well on service-oriented distributed architectures. One of the major characteristics of the microservices architecture is the structure of the applications comprised by a certain number of independent services, each one centered on a particular business aspect, therefore, micro (as the name suggests) services communicating with each other for realizing more complex services.

Scalability means that the cloud-native applications have the potential of taking advantage of the cloud features, including rapid elasticity, adjusting capacity by on-demand adding or removing resources. The book *"The Art of Scalability"*[82] introduced a three-dimension scalability model: The scale cube as shown in Figure 2.11.

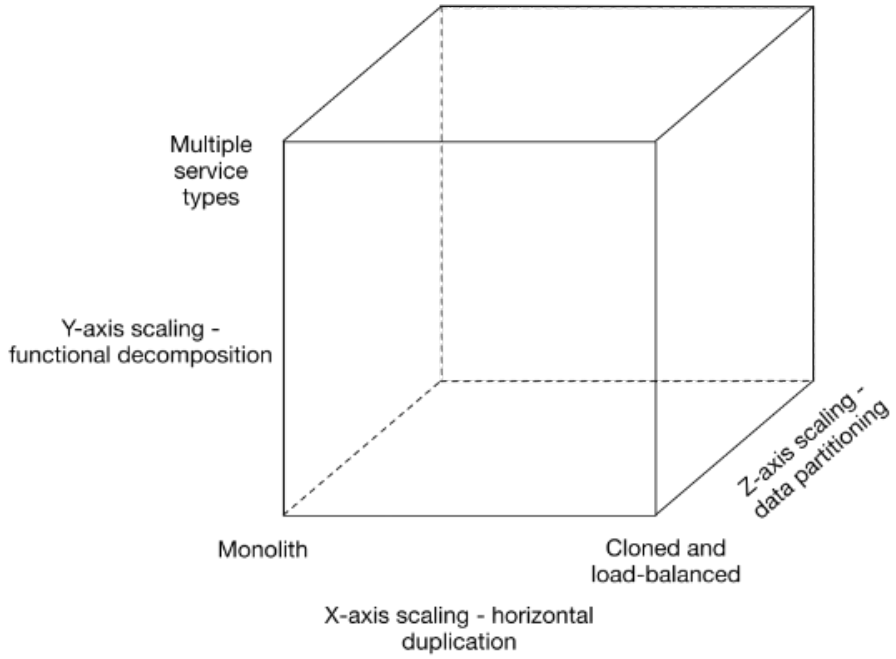

Figure 2.11: The Scale Cube as Presented in [82]

Based on this model the most common approach to scale an application is the one on the *X-axis*, defined as horizontal duplication. Basically, multiple instances of the same application are deployed behind a load balancer.

Similar to the *X-axis* there is the *Z-axis* where each server executes an identical copy of the code with the only difference that each server is responsible only for a subset of the data. In this case there are some components of the system that are responsible for dispatching the information to the most appropriate server, using the attribute inside the request as routing key. Scaling on the *X-axis* as well as on the *Z-axis* improves the capacity of the application and its availability. However, those approaches do not resolve problems related to the complexity of a growing application. This is the reason why it is required to scale also on the *Y-axis*, which deals with the decomposition of an application. The major difference between the *Z-axis* and the *Y-axis* is that scaling on the *Z-axis* precludes the decomposition of a single feature of an application, while scaling Y-axis is about scaling the application based on features. Thus, scaling an application on the Y-axis means to divide a monolithic application into a group of services, each one implementing a set of correlated functionalities.

There are several benefits that could be gained from applying a microservice architecture into the telco domain. The major objective is that decoupling components creates a more effective environment for building and maintaining highly scalable applications. Developing and distributing services in an independent way improves agility in reacting to environmental changes. Following is a list of additional benefits that could be gained while adopting microservices concepts:

- *no single point of failure*: By separating components of an application it is less probable that a bug or a problem affects the entire system. Potential faulty services could be isolated, repaired, and redeployed without causing downtime on the functionalities of the entire application.
- *ease orchestration*: Having lightweight services reduces the costs of orchestration, considering the lower configuration required.
- *increase agility*: The source code is much easier to understand considering the lower number of functionalities covered by a microservice. Therefore, developers could concentrate on specific tasks without impacting on the overall application, and without requiring coordination with other developers.
- *flexible programming decisions*: While developing a microservice, developers are free to choose any programming language that fits the best with that particular service.

2.2.4.3 Twelve-Factor Applications

Recently a set of guidelines have been provided online via the Twelve-Factor Application Manifesto[18]. Those guidelines are:

- *Codebase*: Each application exists in a single repository, with a unique versioning system. However, this does not preclude having to execute multiple instances of the same application even with different versions.
- *Dependencies*: The application should explicitly declare dependencies from other entities, preferably managing them with appropriate packaging systems without assuming their existence.
- *Config*: The application configuration should be stored in environment variables so that minimal modifications of the code will be required while moving to a different platform.
- *Backing services*: Supporting services, like databases, log aggregators, message bus, should be treated as independent resources of the application. Therefore, their integration with the application should happen using only simple configuration files.
- *Build, release, run*: Those three phases should strictly be separated. For instance, modifying the code while the application is in execution could be seen as a violation of this design pattern.
- *Processes*: The application should be executed as one or more stateless processes and without shared data. For satisfying this requirement the application should be made up of "Backing services".

[18] https://12factor.net

- *Port binding*: The application should independently expose its own services on the network, binding on its own on ports without making use of intermediate components.
- *Concurrency*: Considering an instance of an application, built using one or more processes (as per point 6), scale-out should be realized using multithread or multiprocessing programming techniques.
- *Disposability*: Fast start-up and graceful shutdown, able to manage unexpected fault situations, increase robustness, since they allow scalability, maintenance, and upgrades in a very fast way.
- *Dev/prod parity*: There should be consistency between the development and the production environment. Developing in an environment very close to the production environment allows continuous release processes, minimizing error situations.
- *Logs*: The application should manage its own log files, but should only write on standard output/error. It is the execution environment with appropriate backing services that should handle output streams as events, aggregating and storing them.
- *Admin processes*: Admin tasks should be executed as one-off processes in the same environment where the application is executed.

2.2.5 Automation and Orchestration

Automated deployment of services is crucial for TSPs for reducing time-to-market of new services. Before the introduction of cloud computing technologies, and particularly Development and Operations (DevOps) principles[83], the process of making available new services to the end user could require a very long, time-consuming process due to the fact that an administrator has to manually execute deployment procedures step by step. Instantiation of compute resources, configuration of networking devices, installation and configuration of software components became a crucial part of the automation of the end-to-end process for rolling out new services and reducing the time-to-market.

Mike Loukides, author of the book "What is DevOps"[84], recalls a blog post of James Urquhart[19] while providing the definition of the DevOps movement: *"[..] modern applications, running in the cloud, still need to be resilient and fault tolerant, still need monitoring, still need to adapt to huge swings in load, etc. But those features, formerly provided by the IT/operations infrastructures, now need to be part of the application, particularly in "platform as a service" environments. Operation doesn't go away, it becomes part of the development. And rather than envision some sort of uber developer, who understands big data, web performance optimization, application middleware, and fault tolerance in a massively distributed environment,*

[19] https://www.cnet.com/news/understanding-the-cloud-and-devops-part-1/

we need operations specialists on the development teams. The infrastructure doesn't go away – it moves into the code; and the people responsible for the infrastructure, the system administrators and corporate IT groups, evolve so that they can write the code that maintains the infrastructure. Rather than being isolated, they need to cooperate and collaborate with the developers who create the applications. This is the movement informally known as "DevOps"[..]"[84].

With DevOps developers and IT managers started collaborating and communicating since the software design phase in order to increase productivity, especially in rolling out newer versions of a particular service. Several technologies have been launched trying to address the main idea of describing "Infrastructure as Code".

Configuration management is the term used for identifying the process of configuring nodes (i.e., servers, etc.) based on the desired final state. Configuration management tools read configurations from files and apply them to nodes in an automated and idempotent way. This way the administrator could replicate the same configuration on multiple nodes using the same approach. Nowadays there are multiple open source tools available supporting automated configuration management: salt[20], puppet[21], chef[22], ansible[23]. Most of these tools are using an *master/agent* approach in which the master node acts as a coordinator of actions executed by agents running in the targeted nodes.

Automation is mainly concerned with single tasks, for instance launching a virtual machine, configuring it for executing a database, stopping a particular service, while orchestration mainly concerns the automation of a workflow, namely a process. The deployment process of an application comprises several tasks which has to involve different systems and needs to be executed in a particular workflow for getting to the desired state. Ultimately, automation focuses on providing programmability to tasks while orchestration allows programming processes. The latter makes use of automation by reusing some of the building blocks. The execution of tasks is the central role of an orchestrator managing the proper execution of a service starting from day-zero operations (onboarding new services) up to day-two operations (managing its life cycle).

Automating the service deployment is definitely not a novel topic. In 2005, Talwar et al. already compared manual, script-, language-, and model-based deployment solutions, particularly addressing their complexity and barriers to first use[85].

Endres et al. discussed the two fundamental approaches for modeling automated deployments of applications, imperative vs declarative models, identifying patterns pointing to frequently occurring problems[86]. Breitenbucher et al. [87] discussed challenges of combining proprietary management services (like the ones offered by public cloud providers) with script-centric configuration management technologies. Although there are differences between the imperative and declarative models, they

[20] https://saltstack.com/
[21] https://puppet.com
[22] https://chef.io
[23] https://ansible.com

propose a combined solution where a standard-based approach is used for generating provisioning plans based on TOSCA, which, later on, are executed fully automatically[88].

2.2.5.1 Topology and Orchestration Specification for Cloud Applications (TOSCA)

TOSCA is a specification proposed by the OASIS enabling interoperable descriptions of applications and cloud infrastructure resources. TOSCA focuses on the semiautomatic creation and management of complex services providing a language for describing relationships between components (using a *service topology*) and their operational behaviors (as *orchestration process*)[89]. The combination between the topology and orchestration, defined as *service template*, allows defining the complete life cycle operations that need to be preserved across deployments in different cloud environments [90].

The main objective is to define a grammar for describing a service template by means of *topology templates* and *plans*. Basically TOSCA provides a *metamodel* for defining services, including both the service structure as well as the mechanisms to manage it. The *topology template* defines the structure of the service while the *plans* define the operations to be executed for instantiating and terminating it as well as for managing it throughout the life cycle. Figure 2.12 depicts the *service template* definition as an architectural diagram.

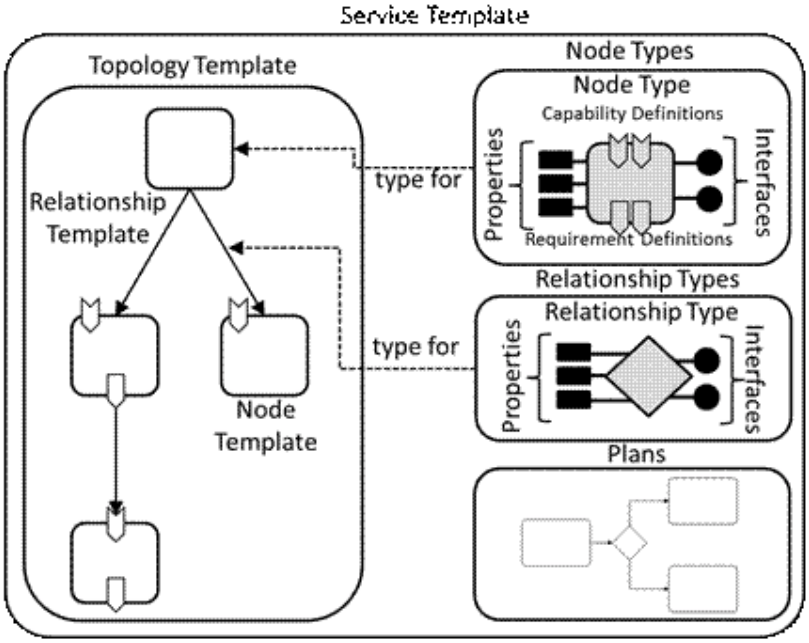

Figure 2.12: The TOSCA Service Template Definition[89]

The *topology template* is composed by a set of *node templates* and *relationship templates*, together defining the model of the service as a directed graph. Typically, a *node template* represents a node in the graph, constituting a component of the service. The *node type* defines the *properties* and *interfaces* exposed by a particular component. *Node types* are referred to by *node templates* as there could be more than one occurrence of the same type of component in a service[89].

The *relationship template* defines the occurrences of a relationship between nodes in a *topology template*. As for the *node template*, for reusability purposes, also the *relationship template* is described by a *relationship type*, defining the semantics and any properties of the relationship. Relationships are defined as unilateral entities providing one source and one target element.

The *service template* instantiation result is the deployed service. In particular, the instantiation process considers the build *plan* defined as part of the *service template* for deploying the service.

TOSCA has been widely used for model-based orchestration in cloud computing. Cloudify[24] and OpenStack Heat[25] (via the TOSCA Heat-translator project[26]) have been two of the early adopters of the TOSCA specification for allowing automatic deployments of complex services across multiple cloud providers.

The initial version (version 1.0) of TOSCA, relying on Extensible Markup Language (XML) Schema 1.0, was published in November 2013[89]. More recently, in December 2016, OASIS published a newer version providing the TOSCA specification in a Yet Another Markup Language (YAML) rendering with the objective of simplifying the authoring of TOSCA *service templates*[91].

TOSCA also defines a packaging solution in order to simplify portability of multiple files between different TOSCA-compliant orchestration engines. The TOSCA Cloud Service Archive (CSAR) may comprise the TOSCA *service template* as well as configuration files, software code, and images required for the instantiation of a particular service.

2.3 Network Function Virtualization (NFV)

In 2012 a large number of TSPs realized that cloud computing was one of the most suitable models for transforming their infrastructures, and in particular get rid of a vendor lock-in that would have allowed a huge reduction in COTS hardware resources. Things changed with NFV paradigms and standards introduced by the ETSI NFV White Paper[92].

ETSI NFV represents a concerted telco operator initiative fostering the development of virtual network infrastructures by porting and further adapting network functions to the specific cloud environment. NFV aims to move such a network architecture from hardware-based appliances to standard servers or a cloud-based

[24] http://cloudify.co/
[25] https://wiki.openstack.org/wiki/Heat
[26] https://github.com/openstack/heat-translator

infrastructure. This opportunity is given by the increase of data center technologies, allowing deployments of network functions as virtualized entities on top of standard high-volume servers. NFV decouples software implementations of NFs from the computation, storage, and networking resources they use. NFs are executed as software components on top of a virtualized environment, meaning Guest OS.

NFV is a new architectural approach, at least from an infrastructure perspective. It is a transformation moving from interconnected monolithic network functions towards a distributed cloud infrastructure, on top of which NFs are executed. Decoupling software from hardware is the first step to achieve this objective. Therefore, a NF is implemented as software, executing on typical linux-based OSs, running on top of virtual machines or containers in the cloud. This transition introduces flexibility in deployment operations, allowing dynamic construction of end-to-end network services, however, requiring different mechanisms for maintaining the same level of reliability, availability, and performances obtained with classic hardware-based NFs. Nevertheless, this flexibility provides the means for novel dynamic operations like placement and scalability. Last but not least, it is important to consider that interoperability with the legacy network should be maintained. Those aspects should be considered while implementing NFV.

The virtualization insulates the NFs from those resources through a virtualization layer. A VNF is a software implementation of a NF that could be deployed in one or more traditional VMs. The VNFs can be instantiated or moved dynamically (orchestrated) in various locations as required. More VNFs can be chained together even with Physical Network Functions (PNFs) to form a NS. The NS is the end-to-end service between two endpoints.

ETSI NFV has defined a large set of virtualization use cases, spanning from the cloudification of the main core network functions such as IMS, EPC, and Radio Access Network (RAN), as well as provided on-demand and complete virtualized infrastructures as IaaS or PaaS to third parties. That enables the elastic deployments of cost-efficient network infrastructures.

One of the main concerns of ETSI NFV is to prove the feasibility of the cloud deployments of the typical NFs through proof of concept trials and prototypes as well as to provide indications for further standardization in the areas of underlying infrastructures, software architectures, networking management, and orchestration to improve performance and grant security of the overall infrastructure. ETSI NFV limits itself to this level of indications, considering that other standardization bodies (i.e., 3GPP) and de-facto open source technologies (e.g., OpenStack) should finalize the specific implementation work.

One of the major benefits is definitely automation: Many operators are currently trying to follow the Continuous Integration / Continuous Development (CI/CD) approach for minimizing the time to get new features into the market where automation represents one of the major methods for achieving that. No manual intervention should be needed for deploying a new feature into the network. Programmability, seen as the major approach for automating the service life cycle, is the key for simplifying the management operation and reducing the costs for maintaining these sys-

tems operational. Furthermore, having a common hardware infrastructure, enabled by virtualization and multitenancy, and being able to deploy several heterogeneous vertical use cases on top, is definitely what operators are looking for.

The outcome of the first phase of the ETSI NFV ISG, spanning between 2013 and 2014, materialized in a set of 11 specifications providing the basis for developing an open, interoperable, and commercial NFV ecosystem, setting the roadmap for future standardization work.

One major outcome of phase 1 was the ETSI NFV reference architecture defined in the ETSI Group Specification (GS) NFV 002 v1.1.1 specification[93]. This architecture is based on three different major approaches: *Decoupling software from hardware*, *Flexible network function deployment*, and *Dynamic operations*.

In 2014, as a result of the different activities carried on by different working groups, the initial version of the NFV architecture was released as shown in Figure 2.13.

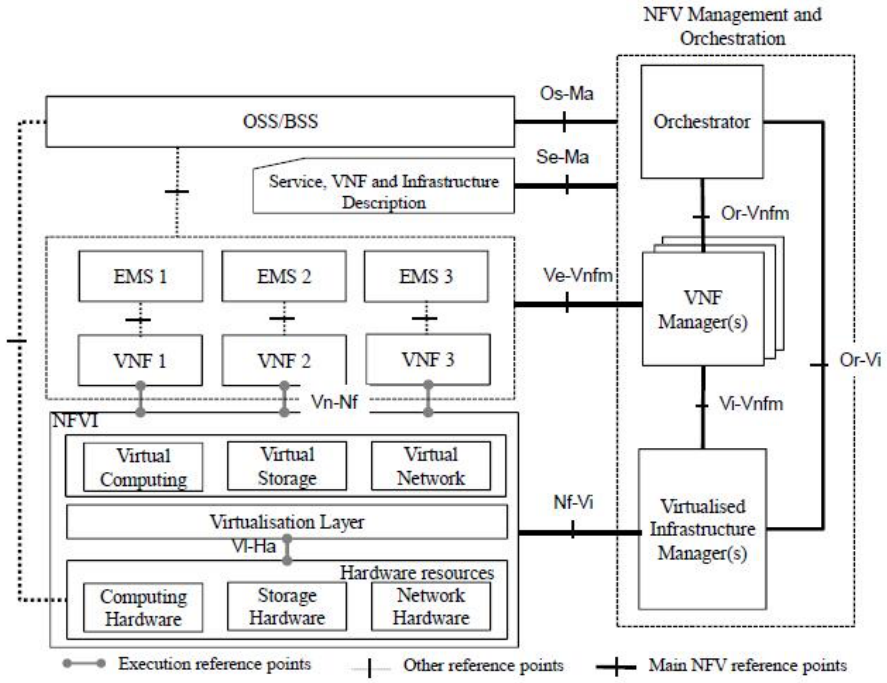

Figure 2.13: ETSI NFV Architecture[93]

First of all, functional elements of the architecture were grouped in three different domains. The domain separation allows different working groups to focus, in parallel, on different aspects within the NFV architecture.

On the bottom left we have the NFVI domain constituted by all the hardware components providing on-demand computing, storage, and networking resources. Hardware resources are typically exposed through a virtualization layer to the other domains where VNFs are executed. This domain has been strongly influenced by

cloud computing methods and technologies where virtual compute, storage, and networking resources are provided on demand to the components part of the MANO domain.

Depicted on the top left is the VNF domain, including the VNFs, the EMS, and the classical OSS/BSS. VNFs could be of any type, and the inital set of use cases focused mainly on Virtualized CPE (vCPE), vIMS, and vEPC. Nowadays, there are many other use cases that have been defined as suitable for NFV-like architectures.

Last but not least, on the right side the MANO domain is depicted including all the components required for managing and orchestrating NSs on top of the NFVI, representing also the main focus point of this dissertation. It includes the NFVO, one or more VNFMs, and a VIM[27]. Starting from the top, the NFVO represents a completely new entity in the network operators infrastructure, providing functionalities for the on-demand deployments of network services on top of the NFVI. The NFVO plays a central role in the architecture and put together resource orchestration with service orchestration. In order to instantiate network services the NFVO makes use of two additional logical entities. Firstly, the VIM that is acting as an intermediate between the NFVO and the NFVI so that virtual resources could be acquired on demand. Secondly, one or more VNFMs are in charge of instantiating and configuring VNFs on top of the NFVI.

In the following subsections a brief overview is given of the different domains, providing more details about the MANO domain as it is a central point for the research work of this dissertation.

2.3.1 The NFVI Domain

The NFVI comprises a set of physical resources hosting VNFs and NSs. These resources can also be virtual and are classified in three groups:

- Compute resources: Physical servers, virtual machines, or, in general, resources with a CPU and memory.
- Storage resources: Physical or virtual storage volume.
- Network resources: Links, networks and subnets, addresses, to allow VNF communications.

2.3.2 The VNF Domain

Considering that the focus of this thesis is to provide an extensible framework for managing NSs, it is important to give an overview of services and their characteristics. As defined in the ETSI GS NFV 002 v1.1.1 specification[93], a network service *"can be viewed architecturally as a forwarding graph of Network Functions interconnected by supporting network infrastructure [..] the underlying network function behavior contributes to the behavior of the higher-level service [..] hence, the*

[27] Multiple VIMs could also exist in case of multisite NFVIs

network service behavior is a combination of the behavior of its constituent functional blocks that can include individual NFs, NF sets, Network Function Forwarding Graph (NFFG), and/or the infrastructure network.

Based on this definition, NFs represent atomic functional blocks of a NS providing individual functionalities which, combined with other NFs, are contributing to the overall network service. The *forwarding graph* represents another important definition that defines the way NFs are communicating with each other inside a NS.

Continuing with the definition provided in the specification, *these network functions can be implemented in a single operator network or interwork between different operator networks.* This means that the NFV architecture should support the placement of NFs across multiple locations in order to interconnect them for providing the end-to-end NS. Figure 2.14 provides a graphical representation of an end-to-end NS.

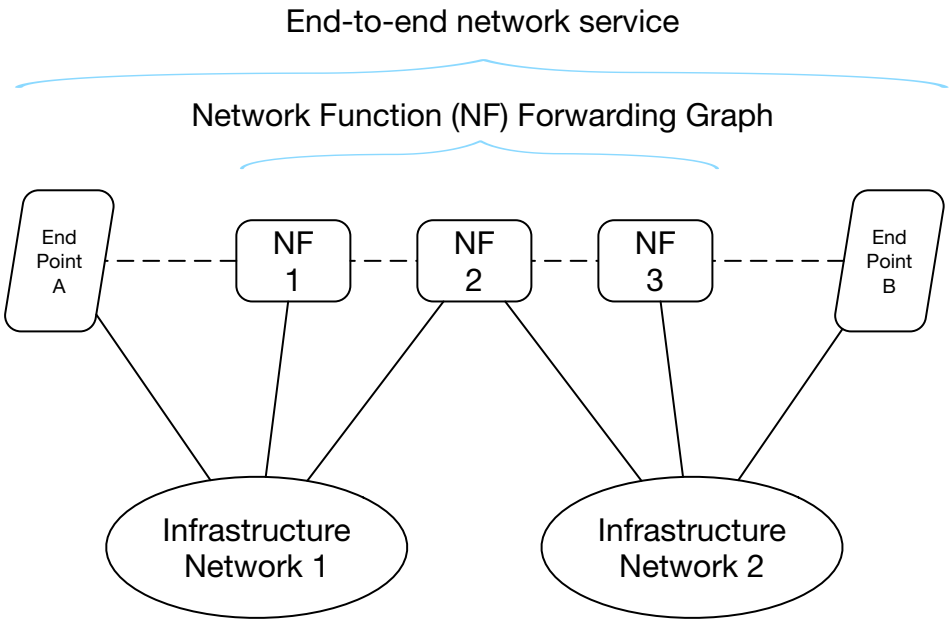

Figure 2.14: Graph Representation of an End-to-End NS

As can be seen from Figure 2.14 the end-to-end NS is composed of end points and a nested NFFG. All the elements are interconnected via network infrastructures (wired or wireless) represented in the figure with dotted lines as logical links. In this example, the NFFG is composed of three NFs, NF1, NF2 and NF3.

The virtualization process of such an end-to-end NS consists of introducing a virtualization layer between the hardware resources in different physical locations, defined PoPs, and the software artifacts implementing the NFs. Figure 2.15 shows the graphical representation of the network service presented in te previous Figure 2.14.

As already mentioned, a VNF is the combination between the NF, implemented

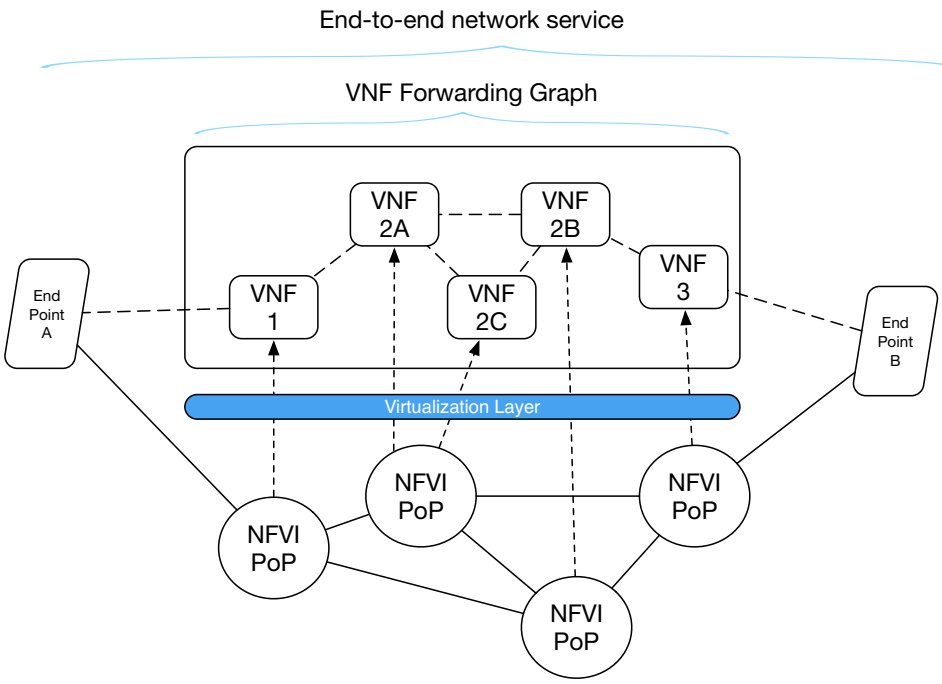

Figure 2.15: Graph Representation of a Virtualized End-to-End NS

as a software component, and the virtual resources it uses underneath. A NS is the composition of one or more VNFs. The NS, and particularly its descriptor called NSD, is the element containing the relationship between VNFs and possibly PNFs. The NSD contains one or more VNFDs, defines their dependencies, provides information about their requirements in terms of virtual network links, describes the life cyle events and their Connection Points (CPs). The VNFDs describe a VNF in terms of its deployment and operational behavior requirements. A VNF can be instantiated either by a single or multiple VNFCs.

A VNFD is generally stored and described as part of a larger NSD, a structure that represents a *network service*. Each network service may potentially be composed of multiple VNFs, interconnected by *Virtual Links*, with possible reciprocal runtime dependencies. A VNF may require the availability of another component of its package to carry out its tasks and to function correctly.

Considering that each VNF can be implemented by different vendors, its management system, the VNFM, can vary between different implementations. The VNFM makes use of the VNFDs while instantiating and managing the life cycle of the VNF. The information provided in the VNFD may also be used by the NFVO to manage and orchestrate network services and virtualized resources on the NFVI. This information model is internally used by the NFVO, by the VNFM and by the VIM.

2.3.2.1 VNF Software Architecture

A VNF can be defined as the software-based version of an NF running on top of virtualization technologies. A VNF instance uses virtualized resources for executing software artifacts. A VNF can be composed by multiple VNFCs, either for high-availability reasons or for decoupling functionalities in different parts. Each VNFC is mapped 1:1 to a virtualization container, typically implemented as a virtual machine. In addition to compute resources, a VNF makes use of networking resources for either interconnecting multiple VNFC instances together or exposing VNFC instances to other VNFs or users via CPs. The external virtual links are part of the network service that combines multiple VNFs together. Figure 2.16 shows a diagram about the VNF composition aspects.

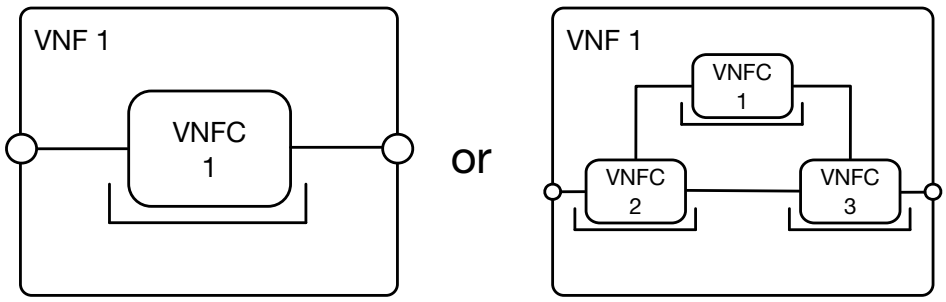

Figure 2.16: VNF Composition

2.3.2.2 VNF States and Transitions

Based on the ETSI GS NFV-SWA 001 V1.1.1 (2014-12) [94] a VNF can assume different states representing the internal status of the VNF. For the sake of clarity, states presented and described in the following relates only to VNF instances, and in particular, it is a prerequisite that before initiating the life cycle, VNF packages are already on board on the MANO framework. Figure 2.17 shows the state transitions of the VNF instance.

The *Null* state represents the initial state when the VNF instance does not exist and it is about to be created. The *instantiate* procedure brings the VNF instance into the *Instantiated Not Configured* state, which means that the VNF instance exists, but it is not yet configured. During the instantiate procedure allocated NFVI resources are required. The next logical transition is triggered by the *configure* procedure that brings the VNF instance into a *Instantiated Configured - Inactive* state. The *start* procedure triggers the state change from *Instantiated Configured - Inactive* to *Instantiated Configured - Active*. Basically, the VNF instance starts performing its provided functionality and is ready to accept any incoming requests from external users or other VNF instances. At the end, the *terminate* procedure deletes the VNF instance, and, therefore, all the NFVI resources associated to it are

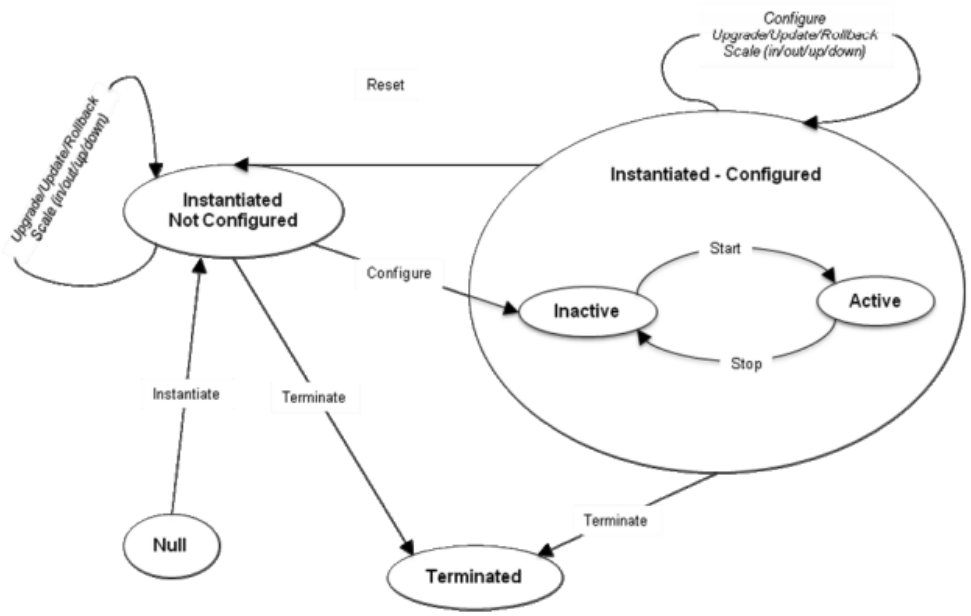

Figure 2.17: VNF Instance State Transitions[94]

deallocated. The state of the VNF instance after termination is *Terminated*.

While the VNF instance is either in the *Instantiated Not Configured* or *Instantiated Configured* state, a set of additional procedures may be executed either for scaling up/down/in/out a number of VNFC instances, or updating/upgrading/rollback a particular version of the VNF.

2.3.2.3 EMS

The EMS is responsible for the FCAPS management of a VNF. The EMS is a component that handles the task of configuring the VNFC instances of a VNF. The EMS may be aware of the virtualization technologies used and collaborates with the VNFM to perform those functions that require exchanges of information regarding the NFVI resources associated with the VNF.

2.3.2.4 OSS/BSS

The OSS/BSS are the combination of the TSP's other operations and business support functions. They usually exchange data with the functional blocks in the NFV-MANO architectural framework, and may provide management and orchestration to legacy systems not covered by NFV-MANO.

2.3.3 The MANO Domain

As mentioned before, the MANO domain comprises these three functional blocks:

- one NFV Orchestrator
- one or more Virtualized Infrastructure Managers
- one or more VNF Managers

The standard also provides a definition about several repositories where to store descriptors and records about NS and VNF instances.

2.3.3.1 NFVO

The NFVO is the component responsible for managing virtualized resources provided by the multisite NFVI and orchestrating the life cycle of NSs. These two responsibilities of the NFVO are typically split in two independent functional elements:

- The *Resource Orchestration* satisfied through functions handling the allocation and release of the NFVI resources, like computation, storage and network resources.
- The *Network Service Orchestration* satisfied through the provisioning of functions handling the onboarding, instantiation, scaling, update, and termination of NSs and any operation on their associated Virtual NFFG (VNFFG)

The NFVO uses the *Network Service Orchestration* functions to coordinate groups of VNF instances together to provide NSs that realize more complex functions. It manages their joint instantiation, the required connections between different VNFs, and dynamic configuration as required during the runtime life cycle (e.g., for scaling the capacity of the NS in case of high demand).

Resource Orchestration The NFVO uses its resource orchestration functional element to abstract access to the resources provided by the NFVI to other internal functional elements, avoiding them to depend on any particular VIM interface. Some of the features provided by this aspect are the following[28]:

- Validation and authorization of NFVI resource requests from the VNFMs to control how the allocation of the requested resources interacts within one NFVI-PoP or across multiple NFVIs-PoPs.
- NFVI resource management, including the distribution, reservation, and allocation of NFVI resources to NS and VNF instances; these are either retrieved from a repository of already known NFVI resources or queried from VIMs as needed. The NFVO also resolves the location of VIMs, providing it to the VNFMs if required.
- Management of the relationship between a VNF instance and the NFVI resources allocated to it, using NFVI resource repositories and information received from the VIMs.

- Policy management and enforcement, implementing policies on NFVI resources. This may involve access control, reservation, and/or allocation of resources, optimization of their placement based on affinity, geographical or regulatory rules, limits on resource usage, etc.
- Collection of metrics regarding the usage of NFVI resources by single or multiple VNF instances.

Network Service Orchestration The network service orchestration functional element uses services exposed by the VNFM function and by the resource orchestration function to provide several capabilities, often exposed by means of interfaces consumed by other NFV-MANO functional blocks or external users[28]:

- Management of NSD and VNF Packages, including the onboarding of new NSs and VNF Packages. The NFVO validates the integrity, authenticity, and consistency of deployment templates, and stores the software images provided in VNF Packages in one or more of the available NFVIs-PoPs, using the support of a VIM.
- NS life cycle management through operations like instantiation, updating, querying, scaling, and termination. This also includes collecting performance measurement results and recording events.
- Management of the instantiation of VNFs, in coordination with VNFMs.
- Validation and authorization of any NFVI resource request that may come from a VNF to control its impact on the current Network Services.
- Management of the VNFFG defining the topology of a NS instance.
- Automated management of NS instances, using triggers to automatically execute operational management actions for NS and VNF instances, following the instructions captured in the on-boarded NS and VNF deployment templates.
- Policy management and evaluation for the Network Service and VNF instances, implementing policies related with affinity/anti-affinity, scaling, fault and performance, geography, regulatory rules, NS topology, etc.
- Management of the integrity and visibility of the NS instances through their life cycle; the NFVO also manages the relationship between the NS instances and the VNF instances.

2.3.3.2 VNFM

The VNFM is a NFV-MANO function taking care of the management and orchestration aspects of individual VNFs through the life cycle management of their instances. A single VNF instance is uniquely associated with a given VNFM. This manager may handle several other instances of the same or different types. While a

VNFM must support the requirement of the VNFs associated with it, most of the VNFM functions are generic and do not depend on any particular type of VNF. Like the other functions, the VNFM exposes functionalities to other elements of the NFV-MANO architecture, often as interfaces. These functionalities include[28]:

- VNF instantiation and (if needed) VNF configuration, using a VNF descriptor as a deployment template.
- Checking if the instantiation of VNF instantiation is feasible.
- Update the software contained in a VNF instance.
- Modify a running VNF instance.
- Handle the scaling out/in and up/down of instances.
- Collect NFVI performance measurement results, faults, and events correlated with its VNF instances.
- Provide NFVI or automated healing of VNF instances.
- Handle the termination of VNF instances.
- Handle notifications caused by changes in the VNF life cycle.
- Management and verification of the integrity of a VNF instance through its life cycle.
- Coordinate and handle configuration and event reporting between the VIM and the Element Manager (EM).

Each VNF is defined in a template called VNFD, stored in a VNF catalog that corresponds to a VNF Package. A VNFD defines the operational behavior of a VNF and specifies how it should be deployed providing a full description of its attributes and requirements. NFV-MANO uses VNFDs to create instances of VNFs, to manage their life cycle, and to associate to a VNF instance the NFVI resources it requires; to ensure full portability of VNF instances from different vendors and different NFVI environments, the requirements must be expressed in terms of abstracted hardware resources. The VNFM has access to a repository of available VNF Packages; each package may be present in several versions, all represented using a VNFDs to allow for different implementations of the same function on different execution environments (like different hypervisor technologies and implementations).

2.3.3.3 NFV-MANO Reference Points

Several reference points are defined between NFV-MANO and external functions:

- *Os-Ma-nfvo*: A reference point between OSS and NFVO that involves NSD management (and VNF Packages), management of NS, and forwarding of requests between OSS, plus policy enforcement and event forwarding.

- *Ve-Vnfm-em* and *Ve-Vnfm-vnf*: Two reference points between a VNFM and EM or a VNF, respectively, used by a VNFM for management and control of VNFs.
- *Nf-Vi*: A reference point used by VIMs to control a NFVI, including the management of VMs and forwarding of events, configurations, and usage records.
- *Or-Vnfm*: A reference point between NFVO and VNFM used to authorize the allocation and release of resources to VNFs, to instantiate VNFs and to retrieve and update information regarding VNF instances.
- *Or-Vi*: A reference point between NFVO and VIMs, used by the NFVO to handle NFVI resources, to receive events and reports, and for management of VNF software images.
- *Vi-Vnfm*: A reference point used by VNFMs for NFVI resources information retrieval and allocation from VIMs.

These interfaces allow the NFV-MANO components to receive information about the systems under their management, using standard defined interfaces.

2.3.3.4 Life Cycle Management of a NS

The life cycle of a VNF is composed of different states where the VNF can transit. The transition between states is triggered by VNF management functions such as instantiate, scale, update, upgrade, and terminate VNF. The VNF management operations are executed by the VNFM functional block.

The deployment requirements of a VNF reside in a template. All the information regarding the virtualized resources needed, life cycle management, and dependencies are present in the template and stored in a catalog during the onboarding of the VNF. The template allows to deploy simple and complex VNFs in a standard way, significantly increasing their reuse. The introduction of the network services as a combination of more VNFs requires further orchestration and management functions. These functions, labeled as Network Service Orchestration, regard the life cycle of an NS and include:

- Onboarding of the NSD, storing the relative deployment descriptor in a catalog.
- Instantiate the NS according to the NSD.
- Scale the NS whenever it needs additional resources or needs to reduce the capacity.
- Updating and/or upgrading the NS, it may include changes in the VNF connectivity, VNF dependency, VNF instances.
- Modify the VNFFGs associated with the NS.

- Terminate the network service, it regards different phases like terminate all the VNFs, deallocate the associated NFVI virtual resources in order to restore the initial condition.

2.3.4 Brief Overview of ETSI NFV Phase 2 (2015-2016)

At the end of 2016, ETSI NFV published a set of new specifications defined during Phase 2, including over 30 new work items, including normative specifications. Topics such as interoperability, operations, and collaboration with other industry groups and Open Source initiatives were considered as part of the ETSI NFV work program. ETSI NFV Phase 2 extends the NFV work towards technology adoption, while addressing areas such as testing/validation, performance/assurance, security, stability, interoperability, reliability, availability, and maintainability, including also collaborations with external bodies.

The new Interfaces and Architecture (IFA) working group had the responsibilities of delivering information and data models, as well as information flows for enhancing interoperability between different elements of the NFV architecture. This led to the production of a new set of detailed specifications. As of July 2015, 14 IFA Work Items were active, including the following with direct impact on this research work:

- *IFA001*[95]: Acceleration Technologies; Report on Acceleration Technologies Use Cases.
- *IFA005*[96]: *Or-Vi* reference point - Interface and Information Model Specification.
- *IFA006*[97]: *Vi-Vnfm* reference point - Interface and Information Model Specification.
- *IFA007*[98]: *Or-Vnfm* reference point - Interface and Information Model Specification.
- *IFA008*[99]: *Ve-Vnfm* reference point - Interface and Information Model Specification.
- *IFA011*[100]: VNF Packaging Specification.
- *IFA013*[101]: *Os-Ma-Nfvo* reference point - Interface and Information Model Specification.
- *IFA014*[102]: Network Service Templates Specification.

2.4 Future 5G Network Architectures

The ETSI NFV initiative generated a lot of interest in several other standardization bodies and academic research initiatives. In the following subsections an overview

is provided of the NGMN, IETF, ETSI, ONF, and 5G-PPP standardization efforts and academic initiatives where NFV plays a central role in future 5G network architectures. One of the common aspects that have been identified among all the different activities towards the next generation of mobile networks is the *"network slicing"* concept.

2.4.1 The NGMN 5G Network Architecture

The 5G is supposed to address the requirements and business contexts required by 2020 and beyond. 5G introduces concepts like a fully mobile and connected society, requiring a multilayer diversification of networks in order to enable differentiation of services based on the requirements of each business model. These 5G use cases are expected to demand even higher performances in terms of flexibility, reliability, scalability, security, and latency than what the current NGN infrastructures are providing. 5G network functions are expected to increase traffic by a thousand times, requiring an increase of the average speed connection. Some of the use cases will have rigid requirements in terms of bandwidth usage, others in terms of latency. This means that high flexibility is required in terms of network resources allocation to different network services. Network slicing represents a new concept providing a logical separation of the physical network resources into "slices" where each slice has different characteristics in terms of network capabilities. The NGMN Alliance foresees future 5G network architectures as composed by multiple slices supporting different requirements for each different application domain[103].

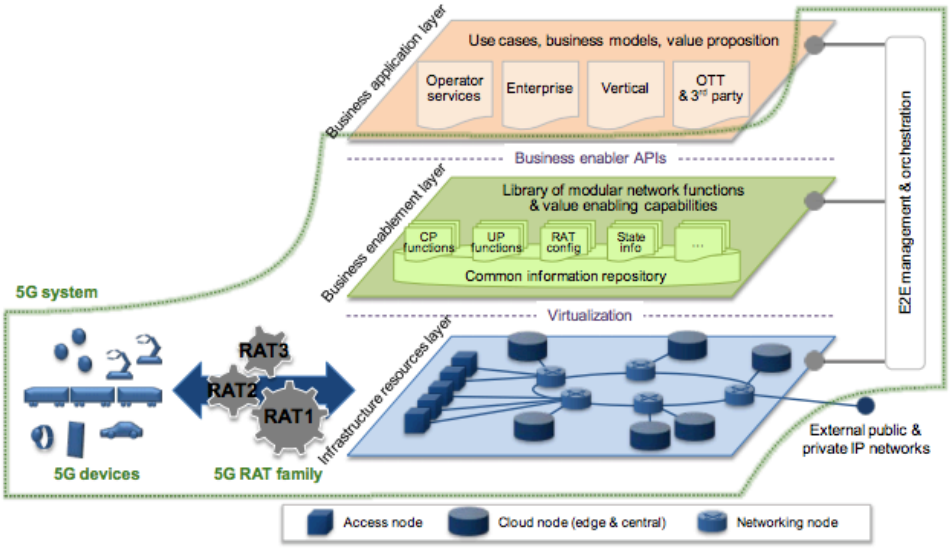

Figure 2.18: NGMN Future 5G Architectures[103]

All technologies that will be used to compose the 5G architecture will rely on logical instead of physical resources, which enables the possibility of delivering the

network on an as-a-service base. This flexibility allows operators to create network topologies on demand and configure the network slices as requested. From the dynamic composition of multiple VNFs it is possible to generate multiple slices. Those slices will be managed on a shared infrastructure that can be controlled by NFV and SDN technologies.

2.4.2 The 5GMF Network Architecture

5GMF released a white paper[104] addressing future 5G networks, focusing also on aspects related to network softwarization and particulary network slicing. 5GMF supports the ideas and concepts already proposed and standardized by the NFV and SDN activities, also proposing an architecture, shown in Figure 2.19, that uses the concept of slices defined as *"a collection of virtual or physical network functions connected by links to create an end-to-end networked system"*. In particular, it classifies the different mobile networks in three major categories: Ultra Reliable and Low Latency Communications (uRLLC), massive MTC (mMTC), and enhancecd Mobile BroadBand (eMBB).

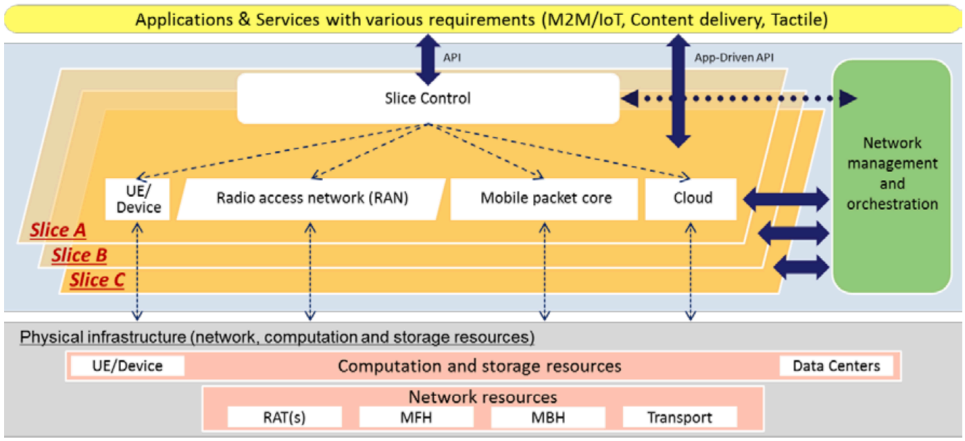

Figure 2.19: 5GMF Network Softwarization Architecture[104]

The proposed architecture is fully compatible with the ETSI NFV architecture, and the role of deploying the different slices is actually executed by the Network Management and Orchestration component. A generic control mechanism of the different slices is provided by the Slice Control element exposing APIs to external users or applications.

2.4.3 3rd Generation Partnership Project

As described in this report[105] published by the 3GPP initiative, the networking requirements would be specific and different for each slice. For example, Machine-to-Machine (M2M) applications will require ultra-low latency without stringent requirements on the network bandwidth, while for Real-time Communication services

guaranteed bitrate is one of the most critical requirements, as already analyzed by the author's publication [13]. To satisfy every use case, several issues have to be solved as described in another report[106] of the 3GPP initiative.

2.4.4 5G-PPP

The 5G-PPP association produced a white paper[107] capturing novel trends and key technology enablers for the realization of the 5G architecture, as shown in Figure 2.20.

5G-PPP envisions that *5G infrastructures will provide tailored network solutions specialized in supporting vertical markets such as automotive, energy, food and agriculture, healthcare, etc. [..] Network slices will contain specialized networking and computing functions that meet the desired Key Performance Indicators (KPIs) of the service providers.*

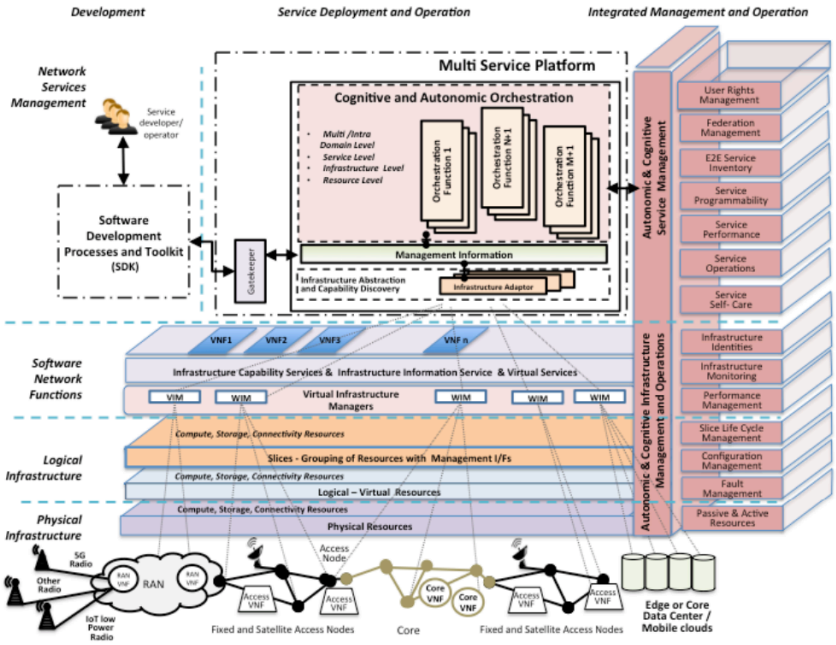

Figure 2.20: 5G-PPP Service & Infrastructure Management and Orchestration Architecture [107]

Based on an indicative list provided by the 5G-PPP white paper, *Slice support (configuration mode selection, adaptation)*, *Flexible placement of VNFs and interfaces*, *Multi-domain orchestration and life cycle management*, are just a few key research issues in the context of service management, orchestration, and control that need to be addressed in future 5G network architectures.

2.4.4.1 IETF

Almost in parallel with the evolution of the ETSI NFV ISG, IETF started a new research group focusing on NFV. The IETF Network Function Virtualization Research Group (NFVRG) primarily focuses on research challenges related to NFV with the main objective of bringing the NFV research community together. In fact, several conferences and workshops were organized during major scientfic events by the IETF NFVRG over the last years.

The areas of interest cover a large set of research challenges that have been identified by this research group, some of them also addressing the MANO aspects in future TSP infrastructures. Among the near-term work items, the IETF NFVRG also identified the *"Policy-Based Resource Management"*, described as follows:

> *"NFV Point of Presences (PoPs) will be likely constrained in compute and storage capacity. Since practically all NFV PoPs are foreseen to be distributed, inter-datacenter network capacity is also a constraint. Additionally, energy is also a constraint, both as a general concern for NFV operators, and in particular for specific-purpose NFV PoPs such as those in mobile base stations. This work item will focus on optimized resource management and workload distribution based on policy."*

During the last years, several drafts were produced [108][109][110] addressing NFV MANO aspects. Results achieved by this research group are planned to be integrated within other standardization activities of IETF Working Groups (WGs) as well as directly provided as inputs to the ETSI NFV ISG.

Very recently, another working group named "NetSlicing" was formed[28], responsible for the definition of network slicing requirements and models. As mentioned in the "Network Slicing - Revised Problem Statement" document draft[111] *"The purpose of the network slicing work in IETF is to develop a set of protocols and/or protocol extensions that enable efficient slice creation, activation/deactivation, composition, elasticity, coordination/orchestration, management, isolation, guaranteed SLA, and safe and secure operations within a connectivity network or network cloud/-data centre environment that assumes an IP and/or Multiprotocol Label Switching (MPLS)-based underlay"*.

Several relevant drafts have been released as part of the the IETF working group: [112][113][114][115].

2.4.5 Software-Defined Networking

Software-Defined Networking represents an emerging paradigm spearating the network's control logic from the underlying routers and switches, which is paving the

[28] https://datatracker.ietf.org/wg/netslicing/about/

way to network programmability paradigms [116]. SDN is focused on the separation of the network control layer from its forwarding layer to make the network dynamic and directly programmable. It is important to clarify that although this thesis covers aspects of management and orchestration of software-based networks, it does not include aspects related to network programmability that are introduced by SDN technologies.

One of the major technological challenges of the SDN paradigm is represented by the centralization of the control logic: This would allow applications in having an abstracted view of the network, as it was managed by a control plane conceptually centralized. It becomes possible to implement control logic abstracting the complexity of the physical layer comprised by heterogeneous networking elements. The control plane has the objective of providing a single and globally centralized view, managing the network topology and distribution of the information required for implementing the application logic. The introduction of different levels of abstraction of the network together with the virtualization of the IT resources increases flexibility in network operators' infrastructure, allowing programmability based on each individual vertical domain.

Based on Scott Shenker [117] and Nick McKeown [118] the main objective for SDN is to redesign the networking architecture introducing appropriate levels of abstraction in order to operate a transformation similar to what already happened in the context of computer architectures where developers could easily implement complex systems without dealing with the low-level details of the physical infrastructure.

Although the vision proposed by the SDN paradigm was very ambitious, most of the time SDN has been wrongly associated with particular aspects having a limited impact: Some are focusing on separating the control plane from networking appliances, others are focusing on providing control interfaces in existing routers. Both innovations are only part of broader and more complex solutions that allow the interaction between the network and the applications running on top.

One of the major outcomes of the SDN activities is the definition of a protocol, named *OpenFlow*, whose main objective is to abstract the networking forwarding elements providing a unified interface for managing flow tables. OpenFlow was initially proposed by several academic institutions (including the Stanford and Princeton Universities) [119], and is currently part of the Open Networking Foundation as principal standardization activity.

2.4.5.1 Service Function Chain

Focusing closer on the mobile core network architecture, being the most relevant part in future 5G infrastructures, the SGi-LAN is the network between the Packet Data Network Gateway (PGW) and Packet Data Network (PDN) (e.g., Internet) where Internet and operators typically Service Functions (SFs) reside. Service Function Chain (SFC) is applied on the SGi-LAN interconnecting a set of SFs, such as Deep Packet Inspection (DPI), Firewall (FW), and Network Address Translation (NAT)

to process the traffic of a specific service. SFC is not a completely novel concept. It has been realized by mobile and fixed network operators for many years.

Using NFV and SDN technologies in applying SFC provides more efficiency in controlling the traffic and elasticity in deploying and scaling up/down virtual SFs. However, there are still challenges that need to be considered while designing and implementing NFV/SDN-based SFC solution[120].

2.4.5.2 Relationship between SDN and NFV

NFV is considered to be complementary to the SDN but not dependent on it and vice versa. An NFV solution can be realized with or without SDN capabilities, providing the possibility to add on demand these functionalities. SDN functionalities consist in an abstraction layer to decouple the control plane to the data plane. The control plane exposes the functionalities to make decisions about where the traffic is sent, the data plane defines the underlying systems that forward the traffic data to a certain destination.

Such technology, combined with NFV, significantly improves network performances, and deployment decisions. Moreover, the SDN software can be deployed in the infrastructure provided by the NFV.

2.4.6 ETSI NFV Priorities for 5G

As presented in Chapter 1, in February 2017 the ETSI NFV ISG published a white paper identifying some priorities that should be addressed for fulfilling the requirements of future 5G networks[26], providing a definition also a definition of network slicing from an NFV perspective: *"Although standardisation bodies and industry forums have produced their own definitions of network slicing, available definitions in different Standards Developing Organizations (SDOs) and industry forums seem to have one thing in common: all of them refer to the creation of multiple logical network instances (i.e., slices) on the same underlying network. The parameters for each "slice" are optimized according to different criteria and possibly used by different tenants/organizations. This is reminiscent of the way NFV is typically used to support use cases such as creating on-demand enterprise customer networks (e.g. vCPE). Indeed, network slices can be viewed as on-demand networks. In NFV parlance, a slice would typically be deployed as one or more NFV Network Service instances. NFV technology thus provides a solid platform to support 5G network slicing. With regard to NFV-MANO, we believe that most features required to enable network slicing are already incorporated in the NFV Architectural Framework [...] however, a few areas do deserve further attention and standardisation work, in particular in the fields of multi-site/multi-tenant orchestration, isolation of resources at different layers, and security enforcement."*

Based on their point of view, most of the features required to enable network slicing are already supported by the ETSI NFV architecture, and the few areas that require more attention are in the context of isolation of resources at different layers.

2.5 Conclusions

This chapter provided an extensive overview of the different research activities and standardization initiatives that have strongly influenced the evolution of management systems in telecommunication networks. As presented, ICT technologies played a central role in the radical transformation on how telecommunication services have been managed. In particular, the evolution in cloud computing technologies and new concepts arising in the area of distributed systems, like microservices architecture and cloud-nativeness, are paving the way to this radical transformation.

The work conducted by the ETSI NFV ISG enabled a novel approach for building future network infrastructures, moving from monolithic hardware appliances towards common cloud-based distributed infrastructures.

Management and orchestration play a central role in this evolution, especially considering the expectations foreseen by different standardization bodies trying to provide a definition for future 5G network architectures in having a common infrastructure providing network slices as a service to different vertical market segments.

Chapter 3

The MANO4X Requirements and Features Analysis

Chapter 2 – State of the Art extensively analyzed the environment in which the research of this thesis belongs. A definition of network management and the evolution towards network service orchestration was presented. The ETSI NFV MANO reference architecture was discussed, and additional insights about research work conducted in the context of the IT and telecommunication service domain were provided.

In order to further elaborate on the research issues addressed by this dissertation, this chapter introduces the MANO4X requirements analysis, finally providing a list of features that shall be supported by the MANO4X framework for satisfying the main research objectives addressed by this dissertation.

The first section focuses on providing an overview of existing requirements gathered from the different standardization activities within the scope of this research work. The second section from presents the list of user stories based on the requirements identified. Finally, a comprehensive list of features, to be supported by the MANO4X framework, is derived from the user stories and requirements presented.

As mentioned in Chapter 1, one of the objectives of this research work is to support different vertical scenarios to be delivered through independent slices on the network operators' infrastructure. The final list of features presented, have been generated considering different system architectures of the vIMS[121], the vEPC[122], and the Virtualized M2M (vM2M)[123] as fundamental elements for building end-to-end network services which have to be supported by the MANO4X framework.

3.1 ETSI NFV Requirements

This section presents the most relevant requirements as discussed in different ETSI NFV specification documents, particularly ETSI GS NFV-IFA 010 V2.1.1[124],

keeping the categorization as presented in the original documents. In particular, nine major categories have been identified and presented in the following.

General

[Gen.4] The NFV framework shall be able to support a network service composed of PNFs and VNFs as a VNFFG implemented across N-PoP multivendor environments that may be instantiated in a single operator or in cooperating inter-operator environments.

Portability

[Port.1] The NFV framework shall be able to provide the capability to load, execute and move VNFs across different but standard N-PoP multivendor environments.

Performance

[Perf.3] For any running VNF instance, the NFV framework shall be able to collect performance related information regarding the usage of compute, storage and networking resources by that VNF instance.

Elasticity

[Elas.1] The VNF vendor shall describe in an information model for each component capable of parallel operation the minimum and maximum range of such instances it can support as well as additional information such as the required compute, packet throughput, storage, memory and cache requirements for each component.

[Elas.2] The NFV framework shall be able to provide the necessary mechanisms to allow virtualised network functions to be scaled with SLA requirements. Different mechanisms shall be supported: e.g. on-demand scaling, automatic scaling.

[Elas.3] The scaling request or automatic decision may be granted or denied depending on e.g. network-wide views, rules, policies, resource constraints or external inputs.

[Elas.4] The VNF user, through standard information model, shall be capable of requesting, for each component capable of scaling, specific minimum and maximum limits within the range specified by the VNF vendor to fulfill individual SLA, regulatory or licensing constraints.

Resiliency

[Res.1] The NFV framework shall be able to provide the necessary mechanisms to allow network functions to be recreated after a failure.

Security

[Sec.3] Management and orchestration functionalities shall be able to use standard security mechanisms wherever applicable for authentication, authorization, encryption and validation.

Service Continuity

[Cont. 2] In the event of an anomaly that causes hardware failure or resource shortage/outage, the NFV framework shall be able to provide mechanisms such that the functionality of impacted VNF instance(s) shall be restored within the service continuity SLA requirements for the impacted VNF instance(s).

Service Assurance

[SeA.3] A (set of) VNF instance(s) and/or a management system shall be able to detect the failure of such VNF instance(s) and/or network reachability to that (set of) VNF instance(s) and take action in a way that meets the fault detection and remediation time objective of that VNF resiliency category.

Operational and Management requirements

[OaM.1] The NFV framework shall incorporate mechanisms for automation of operational and management functions, e.g. creation, scaling and healing of VNF instances based on pre-defined criteria described in the VNF information model, network capacity adaptation to load, software upgrades and new features/nodes introduction, functions configuration and relocation and intervention on detected failures.

[OaM.2] The NFV framework shall be able to provide an management and orchestration functionality that shall be responsible for the VNF and VNF instances lifecycle management: instantiation, allocation and relocation of resources, scaling, and termination.

[OaM.3] The management and orchestration functionality shall be limited to the differences introduced by the Network Function Virtualisation process. The management and orchestration functionality shall be neutral with respect to the logical functions provided by the VNFs.

[OaM.4] As part of VNF life cycle management, monitoring and collection of information related to usage, the management and orchestration functionality shall be able to interact with other operations systems (when they exist) managing the Virtual Network Functions and/or the NFV infrastructure comprised of compute/storage machines, network software/hardware and configurations and/or software on these devices.

[OaM.5] The management and orchestration functionality shall be able to use standard information models that describe how to manage the VNF life cycle.

[OaM.6] The management and orchestration functionality shall be able to manage the lifecycle of VNFs and VNF instances using the information models in combination with run-time information accompanying scheduled or on- demand requests regarding VNF instances and run-time policies/constraints.

[OaM.7] The management and orchestration functionality shall be able to manage the NFV infrastructure in coordination with other applicable management systems (e.g. CMS) and orchestrate the allocation of resources needed by the VNF instances.

[OaM.8] The management and orchestration functionality shall be able to maintain the integrity of each VNF instance with respect to its allocated NFV infrastructure resources.

[OaM.9] The management and orchestration functionality shall be able to monitor and collect NFV infrastructure resource usage and map such usage against the corresponding particular VNF instances.

[OaM.10] The management and orchestration functionality shall be able to monitor resources used on a per-VNF basis, and shall be made aware of receiving events that reflect NFV Infrastructure faults, correlate such events with other VNF related information, and act accordingly on the NFV Infrastructure that supports the VNF.

[OaM.11]: The management and orchestration functionality shall support standard APIs for all applicable functions (e.g. VNF instantiation, VNF instances allocation/release of NFV infrastructure resources, VNF instances scaling, VNF instances termination, and policy management) that it provides to other authorized entities (e.g. OSS, VNF instances, 3rd parties).

[OaM.12] The management and orchestration functionality shall be able to manage policies and constraints (e.g. regarding placement of VMs).

[OaM.13] The management and orchestration functionality shall enforce policies and constraints when allocating and/or resolving conflicts regarding NFVI resources for VNF instances.

[OaM.14] The NFV framework shall be able to manage the assignment of NFVI resources to a VNF in a way that resources (compute hardware, storage, network) can be shared between VNFs.

3.2 List of User Stories

Based on the list of requirements presented in the previous section, the author generated a list of user stories[125] serving as drivers for the definition of the final list of features to be designed and developed as part of the MANO4X framework. User stories have been categorized following the domain classification proposed by ETSI for the definition of the NFV architecture, taking into account the perspective of the three main actors as shown in Figure 3.1: the TSP, the VNFP, and the NFVI Provider.

Most of the user stories listed here have been gathered after a deep analysis of the standardization and research activities which have been presented in the previous chapter.

3.2.1 NFV Infrastructure Domain

NFVIP eligible to provide a NFVI are required to support certain functionalities and capabilities within their infrastructure. Those user stories are listed below:

- *NFVIP1 - Resource Discovery*: As a NFVI user, I need to be able to discover resources available, so that I'm aware of what resources are provided by the NFVI.
- *NFVIP2 - Quota Definition*: As a NFVIP, I need to be able to set quota, so that NFVI users can't make use of all resources available in the infrastructure.
- *NFVIP3 - Resource Provisioning*: As a NFVI user, I need to be able to provision on-demand resources, so that I can make use of them in a programmable way.
- *NFVIP4 - Resource Upload*: As a NFVI user, I need be able to upload my custom resources (i.e. VM images) on the NFVI, so that I can make use of them during the deployment phase.

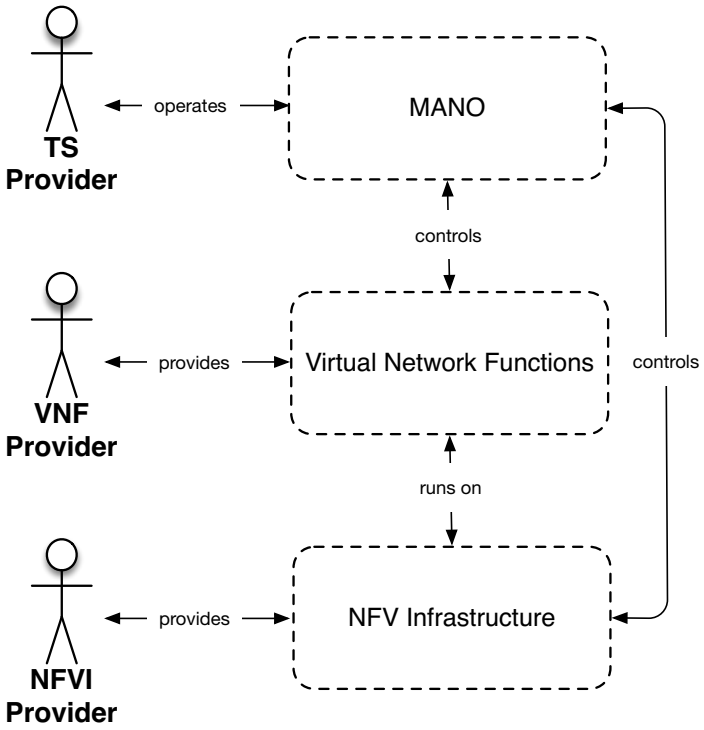

Figure 3.1: Principal Actors and Their Interactions with the Platform

- *NFVIP5 - Resource Configuration*: As a NFVI user, I need to be able to configure provisioned resources, so that in case of any changes required it is not needed a new deployment operation.
- *NFVIP6 - Multiple Networks*: As a NFVI user, I need to attach multiple networks to virtual compute resources, so that for instance data and control plane are separated.
- *NFVIP7 - Virtualization Technologies*: As a NFVI user, I need the NFVI to support different virtualization technologies, so that my virtual compute resources do not have any particular restrictions.
- *NFVIP8 - SLA*: As a NFVI user, I need to be able to establish a SLAs with the NFVIP, so that I can expect certain levels of QoS from the deployed resources.

3.2.2 Virtual Network Function Domain

The main objective of a VNFP is to design and develop VNFs, and provide them to TSP, typically in a standardized packaging format. Depending on the business model, a VNFP could release its VNF package as open source or as closed source. The list of user stories is as following:

- *VNFP1 - Packaging*: As a VNFP, I need to package the VNFs in a standardized format, so that it can be delivered to the TSPs and on boarded on their MANO framework using standard-compliant procedures.

- *VNFP2 - Configuration Management*: As a VNFP, I need to use any kind of configuration management system (i.e. Puppet, Juju, Chef, Salt, etc.), so that I can benefit from any particular feature they provide for the runtime configuration of the VNF.

- *VNFP3 - Specific VNFM*: As a VNFP, I need to be able to integrate my specific VNFM within the MANO framework, so that I could design and develop any particular extensions required by the TSP.

- *VNFP4 - Generic VNFM*: As a VNFP, I need a Generic VNFM, so that I do not need to provide also the VNFM solution to the TSP.

- *VNFP5 - Versioning*: As a VNFP, I need to keep track of different versions of the VNF Package (VNFP) and its compatibility with a particular MANO version, so that any incompatibility issue is avoided while onboarding them on the MANO framework.

- *VNFP6 - Continuous Integration*: As a VNFP, I need a CI/CD system, so that the full integration process is executed automatically from an end-to-end perspective.

- *VNFP7 - Monitoring*: As a VNFP, I need to be able to monitor VNFC instances, so that collected metrics could be further analyzed.

3.2.3 MANO Domain

The most important actor within the MANO domain is the TSP being the one controlling and observing resources which are deployed on the multi-site NFVI. The main objective of the TSP is to compose multiple VNFs in a network service in order to satisfy the needs of its end-users. Therefore, simplified mechanisms for composing new services comprised by heterogeneous VNFs should be supported to avoid vendor lock-in effects. In the following list is given a short description of the main user stories mainly from the TSP perspective:

- *TSP1 - Inventory*: As a TSP, I need to be able to discover available VNFs, so that I know what kind of VNFs are available. Information about the supported VNFs should be part of a global catalog, and their information and data model should be based on open standards, like the ETSI NFV one, in order to allow portability of VNFs from/to different platforms. Nevertheless, the information about the VNF should provide details about its unique type, its dependencies with other VNFs, and what kind of resources are needed for being deployed (e.g., container, VM images, etc.).

- *TSP2 - NS Composition*: As a TSP, I need to be able to select VNFs from the catalog and compose them in a network service, so that new services may be created based on the requirements of a particular vertical domain. This might also include exchange of configuration information between VNFs provided by different vendors.

- *TSP3 - NS Onboarding*: As a TSP, I need to be able to on board a NS composition on the catalog, so that can be stored for future usage.

- *TSP4 - NS Deployment*: As a TSP, I need to be able to deploy a NS selecting it from the catalog and passing runtime configuration elements on the fly, so that specific configuration can be achieved without need of building a new composition.

- *TSP5 - VNF Placement*: As a TSP, I need to be able to specify the PoP where VNFCs should be deployed, so that the NS satisfies latency requirements for a particular set of use cases.

- *TSP6 - NS VNF Monitoring*: As a TSP, I need to be able to monitor the status of NSs, so that I can take any decision during their execution.

- *TSP7 - NS Runtime Management*: As a TSP, I need to be able to execute any runtime operations (exposed through a set of Create Read Update Delete (CRUD) primitives) once the NS has been deployed, so that I can satisfy the requirements of the end users anytime within the lifetime of the NS.

- *TSP8 - NS Termination*: As as TSP, I need to be able to terminate the deployed NS, so that resources allocated to it are released when not needed anymore.

- *TSP9 - Authentication*: As a TSP, I need to be authenticated into the system using well established security mechanisms, so that every request received by the framework is verified before authorizing the execution of any operations.

- *TSP10 - Authorization*: As a TSP, I need to be able to authorize other users based on user roles and policies, so that they can execute only determined operations after being authenticated.

- *TSP11 - Well-defined APIs*: As a TSP, I need to have access to a set of well-defined APIs, so that it is possible to interoperate with the framework in a programmable way.

- *TSP12 - User Tools*: As a TSP, I need a set of user tools (e.g., dashboard, CLI, etc.) for interacting with the platform, so that can execute operations in a simplified way.

3.3 Final List of Features Derived from User Stories

The list of user stories identified in the previous sections served as backlog for the execution of each individual release. For the sake of clarity, it is important to underline that the list of user stories previously presented is the result of an iterative process, being always updated with latest requirements coming from different input sources (standardization bodies, open source communities, etc.). Here is presented a backlog including the minimal set of features required for demonstrating and validating the implementation of the proposed MANO4X architecture:

- *MANO-1 - Inventory*: The MANO4X framework should provide inventory capabilities for storing all the static and dynamic information related to the infrastructure (particularly VNFM endpoints and PoP locations), to the VNFs and NSs (packages and descriptors), runtime information like records of instances deployed, and finally external OSSs.
- *MANO-2 - Lifecycle management*: The MANO4X framework should support lifecycle management of network services.
- *MANO-3 - Multi Tenancy*: Considering that the proposed framework will be employed for several vertical domains, it should support user management, and particular multi-tenancy, so that different deployments maybe logically isolated from each other.
- *MANO-4 - OpenStack support*: OpenStack represents the standard de-facto VIM, thus the MANO4X framework should be capable of interacting with OpenStack for the deployment of virtualized compute, network, and storage resources.
- *MANO-5 - Support for heterogeneous NFVI*: Generic mechanism for integrating heterogeneous cloud technologies. Solving the heterogeneity problem is not a trivial task. Each individual resource management system at the level of the infrastructure exposes a different set of APIs for being managed. An adapter should be designed generically in order to support current and future infrastructure technologies.
- *MANO-6 - Multi-site NFVI*: The MANO4X framework should enable the deployment of a NS on top of a distributed and heterogeneous NFVI. Each individual node in the NFVI should be identified as PoP. Although each PoP may expose a particular version of APIs for being managed, the MANO framework shall support the execution of VNF Packages without requiring any changes on them.
- *MANO-7 - VNF Placement*: A generic module for controlling VNFC instances' placement across multiple PoPs. Considering that the NFVI shall be composed by geographically distributed PoP, the framework shall support the possibility of instantiating a particular VNFC on a selected PoP.

- *MANO-8 - Support for heterogeneous VNFs*: Generic mechanism for integrating various network functions. Heterogeneity at the level of network functions managed needs also to be considered. The framework should be designed in a generic way to support any kind of current and future VNFs.
- *MANO-9 - Generic VNFM*: The MANO4X framework should provide a Generic VNFM supporting the deployment of VNFs using scripting languages. The Generic VNFM should be designed following the guidelines provided by the ETSI NFV specification.
- *MANO-10 - Support for specific VNFM*: The MANO4X framework should provide a simple mechanism allowing VNFP to integrate their own VNFM, so that it is possible to make use of it during the deployment operations of the VNFs under their control.
- *MANO-11 - Monitoring support*: Considering the large number of existing technologies in the monitoring system domain, the MANO4X framework should be designed in such a way that any kind of existing monitoring system can be plugged in and used for monitoring network service deployments across the NFVI.
- *MANO-12 - Manual Scaling support*: The MANO4X framework should provide manual scaling functionality so that individual VNF instances can be scaled in or out via programmable APIs.
- *MANO-13 - Autoscaling support*: Based upon the manual scaling and monitoring support, the MANO4X framework should provide an additional component capable of automatically scale VNFC instances, based on policies defined by the TSP.
- *MANO-14 - Fault Management support*: The MANO4X framework should provide mechanisms for detecting potential faults and executing recovery actions (e.g., switch to standby instances, execute a healing operation, etc.). Those mechanisms should be exposed to the TSP as policies.
- *MANO-15 - Network Slicing support*: The MANO4X framework should support the creation of isolated slices on top of the physical infrastructure. This feature should allow TSP defining the required networking capabilities which should be guaranteed at the physical layer.
- *MANO-16 - SFC management support*: The MANO4X framework should provide mechanisms for establishing a certain data path between VNFs. This mechanism should be compatible with the one proposed by the SFC architecture.
- *MANO-17 - Integration with existing OSS/BSS components*: A generic mechanism for integrating already existing OSS/BSS components should be provided.

- *MANO-18 - User Tools*: The MANO4X framework should provide a set of user tools allowing users to interact with the framework in a simplified and programmable way.

In addition, the author gathered a set of non-functional features as presented in the following:

- Extensibility and customizability: The framework should be extensible without major changes to its architecture and with minimal development efforts. Customization should also be supported, allowing meeting the requirements of a particular scenario through specific configurations, without requiring modification to the architecture and its implementation.
- Scalability and reliability: The framework should be designed following cloud-native principles, so that any component can be horizontally scaled for providing high-availability increasing its reliability.
- Security: The framework should provide mechanisms for authorizing and authenticating users based on well established security mechanisms.
- Standard compliance and openness: The framework should be aligned with standardized architectures and interfaces, and it should provide an open solution available to the public community.
- Lightweight and easy to install: The final solution should be lightweight enough to be executed with minimal hardware resources, and its installation should be easy to realize.

3.4 Conclusion

This chapter introduced a number of requirements seen from the perspective of the different actors involved in the lifecycle management of software-based Networks. Based on this list, and especially after having extensively analyzed the state of the art in Chapter 2 – State of the Art, some potential issues can be identified.

The first issue is represented by the fact that several VNF implementation may differ from the way they are instantiated and managed. Therefore, the MANO4X framework should support the smooth integration of already existing VNF management systems (like Juju) with very low impacts on the integration costs.

The second issue is represented by the heterogeneity of the infrastructure resources. NFVIP could offer the same kind of resource (i.e. a VM) via a different API model. This means that the MANO4X framework should provide a solution for incorporating those heterogeneous system in a simple way without affecting the overall lifecycle of the entire network service. This means that a SP should be able to place a VNF on top of any kind of NFVI, as long as the same kind of virtualization resources are offered.

Chapter 4

The Design Evolution of the MANO4X Framework

Chapter 3 exposed and motivated the requirements and features driving the design evolution and consequent development of the MANO4X architectural framework.

The concept of *"event-driven orchestration"* proposed by this research work is the result of an iterative design process divided into different phases, each of them providing concepts and methods that were reutilized up to the final version of the

proposed architecture. The major objective has been the design of an architecture supporting management and orchestration of network services integrating heterogeneous components. The parallel evolution of the ETSI NFV standardization work also has some influence on some design decisions taken during these phases.

It is important to underline that this thesis does not focus primarily on aspects related to the information and data model, while it intends to reuse the ones already available as part of different standardization activities, thus avoiding introducing yet another proprietary model that will make impossible interoperability and portability of VNFs and NSs across different existing implementations of similar frameworks. Structuring data is consistently crucial for allowing coherent interpretation of the information passed to such complex environments. Information models have typically been used for specifying data semantics for a particular logical element of a specific knowledge domain. A definition about the difference between information and data model utilized in this dissertation is provided by RFC3444 [126].

The solution finally proposed aims mainly at providing an extensible framework adopting and adapting existing information and data models primarily focusing on aspects like extensibility and customizability as the basis for enabling further research in this direction.

4.1 Design Methodology

The design process has been performed following an iterative agile approach based on the scrum [31] methodology. Scrum has become one of the most used design methodologies in software engineering. The main concept of the scrum methodology is to divide the design and development process in smaller cycles that could be executed iteratively. Each cycle, defined as sprint, has as the main objective the realization of particular features selected from an always evolving backlog.

The work conducted during this research work was organized in major releases of a duration of six months, accompanied by minor intermediate release cycles (sprints in the scrum terminology). Each major release comprised an architectural redesign process based on the results achieved in the previous release and validated by a prototype implementation. This iterative approach allowed the definition of an always evolving functional architecture that was validated by several proof-of-concept use cases in the context of running large projects (as will extensively be presented in Chapter 7). Figure 4.1 depicts an overview of the iterative design process executed during this thesis.

In total, ten major releases were executed during the time of this dissertation, divided into three main phases:

- Prototype phase (2012), presented in Section 4.2, having as the major objective the design and implementation of a prototype system capable of orchestrating and autoscaling the 3GPP IMS Application Servers. Such prototype, initially part of the Fraunhofer FOKUS Cloud Broker, further evolved and adapted to become the Elasticity Engine (EE) component utilized in the con-

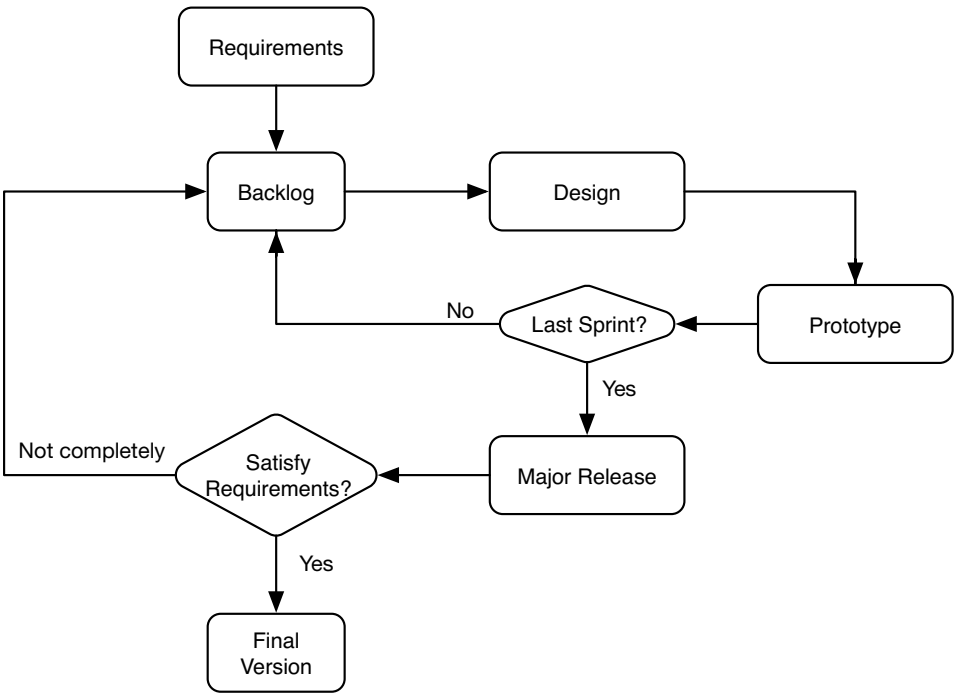

Figure 4.1: Design Process Following an Agile Methodology

text of the BonFIRE project (see Section 7.1.0.1 for more information). During this phase, the author did significant research addressing one of the major issues in cloud orchestration, namely elasticity. Several scientific publications [127][14][13][128][129] were published during this period addressing scalability issues faced in telecommunication networks. This prototype phase helped in further detailing technical requirements for evolving the framework, keeping in mind the extensibility objective.

- Intermediate phase (2013-2014), presented in Section 4.3, focused on designing and developing the very initial version of the MANO4X framework. The design and development work conducted during this period was done as part of the OpenSDNCore[1] project. During this phase the author focused primarily on addressing the need of managing and orchestrating complex software-based NGN infrastructures (i.e., 3GPP EPC and 3GPP IMS), always considering the extensibility requirements for supporting different vertical domains. This prototype was utilized primarily in the context of the Mobile Cloud Networking (MCN)[2] project (see Section 7.1.0.2), a large research project involving several research institutes, operators and vendors in Europe. Several scientific publications [130][131][132][133][134][135][123][136][137][138][122][121] were published

[1] www.opensdncore.org
[2] www.mobile-cloud-networking.eu

during this period mainly addressing the issue of orchestrating complex network services like NGN infrastructures.

- Final phase (2015-2017), presented in Section 4.4, aiming at designing and implementing the final version of the proposed MANO4X framework, being implemented and openly available to the community as part of the the open source Open Baton[3] project. During this phase the author developed the intermediate version of the architecture firstly aligning its definition to the ETSI NFV MANO specification, secondly working on aspects like extensibility and customizibility for supporting a large variety of use cases. The design evolution was done in combination with the research work conducted in the context of NUBOMEDIA[4] and SoftFIRE[5] ICT research projects, and as part of the Fraunhofer Fraunhofer Institute for Open Communication Systems (FOKUS) 5G Playground. Several scientific publications [139][140][141][142][143][120][144][145][146][147][148][149] were published, and several presentations in industry Research and Development (R&D) conferences and technology introduction tutorials as listed in Section A.2 were given during this phase of research.

Each individual phase is presented in the following sections. For the sake of clarity, the author does not provide all low-level details of the prototype and intermediate phases, considering that the main objective of presenting such phases is only to understand the design evolution underlining the limitations and gaps encountered in each phase. Nevertheless, additional details about the research work conducted in each of the first two phases is available as part of scientific publications and technical reports already presented earlier, and with software toolkits as part of the Fraunhofer FOKUS 5G Playground.

4.1.1 Design Evolution Compared to the ETSI NFV One

Considering the relevance of the ETSI NFV specification influences on the research work addressed by this dissertation, it is important to present the design evolution of the MANO4X framework compared to the ETSI NFV timeline. On the top of Figure 4.2 the major phases of this research work are shown and on the bottom the ETSI NFV evolution.

In January 2012 the author was inspired by the increasing achievements in cloud computing technologies and principles, especially by existing CMSs (like OpenNebula and OpenStack) and decided to start investigating mechanisms for *cloudifying* IMS Application Servers (ASs). This triggered the beginning of the prototype phase. At that time, NFV concepts were not yet announced to the public audience. The intermediate phase started immediately after the release of the first version of the

[3] http://openbaton.github.io/
[4] https://www.nubomedia.eu
[5] https://softfire.eu/

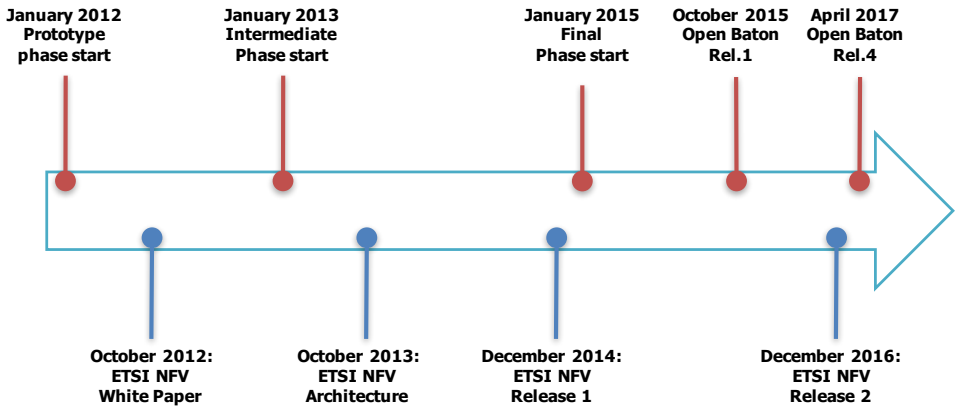

Figure 4.2: Design Evolution Compared to the ETSI NFV one

ETSI NFV white paper published in October 2012, the time when the author decided to further extend the scope of his work towards a more comprehensive solution addressing the research challenges foreseen by the ETSI NFV ISG.

During the intermediate phase, the initial version of the ETSI NFV architecture was published as part of the ETSI GS NFV 002 v1.1.1 specification[93]. Overlapping functionalities between the architecture proposed by the author during this phase and the ETSI NFV one were identified.

The end of 2014 the public release of the first set of specifications including the information and data model as well as reference point definitions were used as inputs for the beginning of the final phase, in which the author decided to redesign the intermediate solution for providing a reference architectural framework fully compliant with the specification.

4.1.2 Design Assumptions and Architectural Design Principles throughout all Design Phases

Before presenting the different design phases, it is important to clarify some design assumptions that were considered throughout the design phases.

The first assumption regards the way NGN NFs are designed and implemented as software components. The author assumes that NF providers follow cloud-native principles while designing their solutions, particularly addressing aspects related to elasticity and scalability. Moreover, the author considers NF as stateless components, therefore, does not focus on managing the state of particular NFs while executing scaling operations. Nevertheless, the final solution proposed could be further extended for allowing NF-specific functionalities, like state recovery or state sharing between multiple instances of an NF. Furthermore, the author assumes that NFs could execute on any common hardware infrastructure on top of virtualized compute resources.

A second, rather important assumption regards the multisite networking envi-

ronment. It is assumed that inter-site networking (defined as connectivity across geographically dislocated sites) is centrally managed by using a Wide Area Network (WAN) or a layer-3 Virtual Private Network (VPN). The proposed solution does not focus on the dynamic creation of virtual networks across multiple sites, however, it could easily be extended adding an additional functional element that could deal with such functionality.

4.2 Prototype Phase

The demand of mechanisms for flexibly and cost-efficiently outsource NGN infrastructures to multiple external cloud providers has driven the results achieved during this phase. In particular, compared to best-effort Internet services, Real-time Communication (RTC) services have different levels of QoS requirements, calling for management functionality able to cope with all the aspects defined by the FCAPS model[129][150].

Elastic deployments and autoscaling of IMS-based application servers was widely unexplored at that time[128][13]. Some solutions were already addressing Web services' elastic scalability [151][152], but without focusing primarily on the telecommunication domain, where NGN infrastructures put much more emphasis on end-to-end QoS. Those research activities dealing with cloud-based deployments of IMS infrastructures [20][153] did not take into consideration important characteristics like elasticity and flexibility offered by cloud technologies for meeting end-to-end QoS requirements[129].

Therefore, the prototype phase had as the major objective the realization of a solution capable of satisfying the requirement of deploying one or more instances of an NF on a particular cloud infrastructure and runtime autoscale it.

4.2.1 Information and Data Model

The information and data model were based on the concept of the `Service Group`[14]. A `Service Group` is used to indicate the entire set of service components, deployed on either private or public clouds (or both), providing a certain service functionality by assuming, with no loss of generality in practical deployment scenarios, that each VM corresponds either to a `Service Instance` or a `Load Balancer`. A simple representation of the information model is provided in Figure 4.3.

A `Service Group` contains:

- one or more `Service Instances` of a particular service type.
- one `Load Balancer`[6].

[6]In the case of vIMS the Subscriber Locator Function (SLF) acts as a Load Balancer for the HSS, while a standard Session Initiation Protocol (SIP) dispatcher maybe satisfactory for distributing load among different SIP ASs

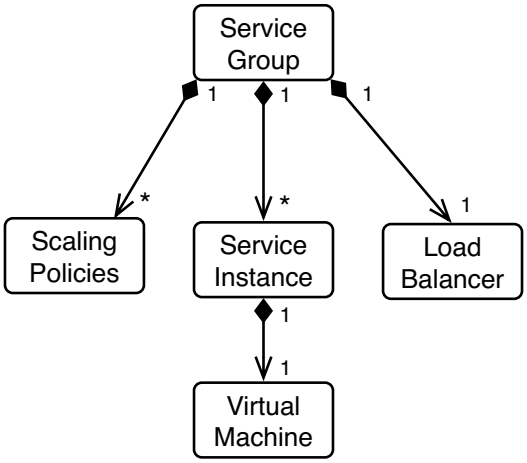

Figure 4.3: Proposed Information Model for Describing a Service Group

- one or more `Scaling Policies` used for describing alarms conditions and actions to be executed.

Considering the vIMS use case, one can think of the AS as the `Service Instance` and the Serving CSCF (S-CSCF) as the `Load Balancer`. In this case the dependency between the two components is mainly unidirectional whereas the S-CSCF requires networking information and the endpoint configuration whenever a new instance of an AS is deployed or an existing one is disposed.

4.2.2 Initial Architecture

The architecture shown in Figure 4.4 has been proposed in order to achieve the goals of having an initial prototype solution supporting the aforementioned requirements,

The proposed solution comprised two major components:

- the cloud infrastructure providing compute, storage, and networking resources needed by a `Service Group`. The cloud infrastructure comprises several Compute Nodes (CNs) managed by the Cloud Management System
- the EE providing functionalities for deploying and autoscaling a `Service Group`

A running instance of the EE was able to handle only one `Service Group` at the time, without any support for multitenancy. It comprised three major components dynamically composing a pipeline:

- Monitoring Aggregator (MA) aggregating metrics and alarms based on conditions defined in the `Scaling Policies`
- Rules Engine (RE) evaluating the alarms and making decisions about which action to execute

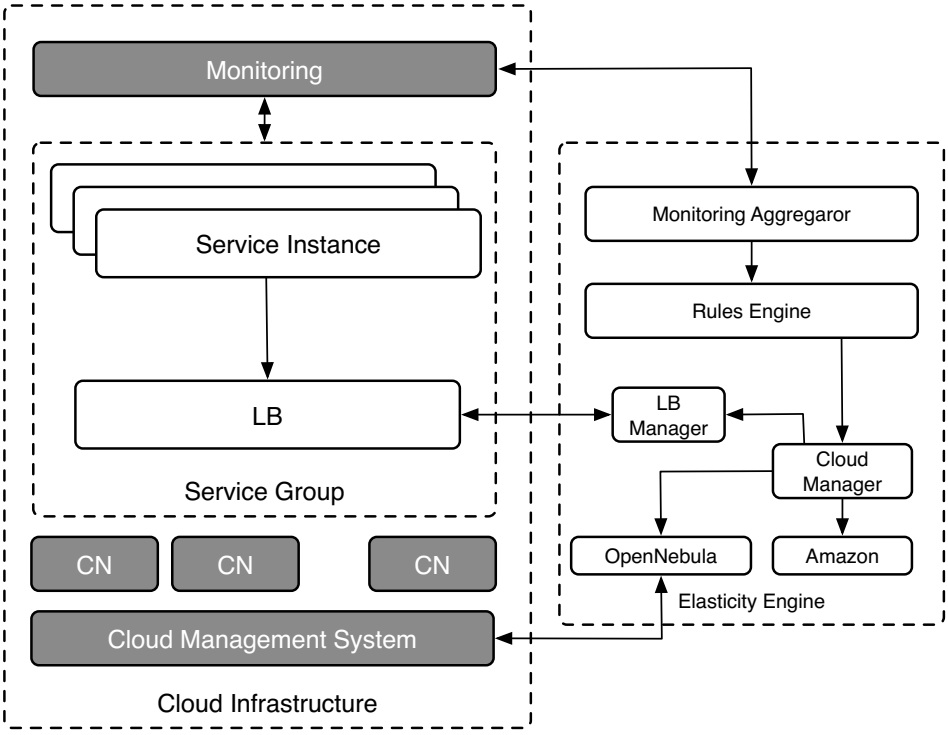

Figure 4.4: Proposed Architecture during the Prototype Phase

- Cloud Manager (CM) managing the deployment of `Service Instances` using adapters for interacting with heterogeneous cloud infrastructures and reconfiguring the `Load Balancer` (via the Load Balancer Manager) whenever a scale out/in action was executed

Each stage of the pipeline receives output data from the previous stage to trigger control actions towards the following one. First the MA receives either raw monitoring data or alarms from the monitoring system and passes them to a RE that analyzes this data and makes a decision based on the current situation of the pool of utilized resources about action requests to send to the CM. Finally, the CM translates those commands into specific requests to the CMS (OpenNebula and Amazon EC2 as supported cloud infrastructures)[127].

4.2.3 Service Group Life Cycle

This initial solution did not support any inventory functionality for storing templates of the desired service to be deployed. The design phase required the creation of a XML Topology file providing the definition of the `Service Group` to be deployed. The on-boarding phase consisted in placing the XML file in a specific path that could be accessed during runtime by the EE. While booting, the EE firstly parses the provided file, and later on starts creating the necessary runtime processes for

handling the life cycle of the `Service Group`.

4.2.3.1 Dynamic Service Placement

As mentioned, one of the major targets during this phase was the instantiation of services across a multisite heterogeneous infrastructure. During this phase, the author contributed to the service placement problem, providing an algorithm that uses static profiles based on TSPs' KPIs preferences, and monitoring data gathered from the MA (i.e., available bandwidth between endpoints) for selecting the most suitable location where to dynamically instantiate service instances[128]. The first step of the proposed algorithm involves the TSP determining a weight value for each KPI. Let us define CP_i as the i-th cloud provider, with $i \in [1, n]$, and K_j as the j-th KPI value, with $j \in [1, m]$. We call λ_{ij} the weight assigned by the TSP to the j-th KPI for the i-th cloud provider, where:

$$\sum_{j=1}^{m} \lambda_{ij} = 1, \forall i \in [1, n] \tag{4.1}$$

In addition, different KPIs are normalized in order to create a sort of provider ranking table. Defining with V_{ij} the monitored value for a KPI K_j for the cloud provider CP_i (i.e., the current latency measured between the TSP infrastructure and a given cloud provider) and MV_j the maximum value for this column, for each KPI and cloud provider the normalized value NV_{ij} is:

$$NV_{ij} = \frac{V_{ij}}{MV_j} \lambda_{ij} \tag{4.2}$$

Choosing the best provider simply means selecting the one with best normalized KPI sum. More precisely, by identifying with KPI+ the set of KPIs to maximize and with KPI- the others, we define:

$$T_i = \sum_{j\, in\, KPI+} NV_{ij} - \sum_{j\, in\, KPI-} NV_{ij} \tag{4.3}$$

where T_i is the score of the i-th provider.

4.2.3.2 Service Elasticity

Another important contribution regards the service elasticity concept, particularly dynamic scalability of the deployed service instances[13][129]. The critical point is that overloaded services cannot maintain the expected satisfactory QoS causing an immediate QoE deterioration. Scaling operations are typically used by cloud service providers for elastically increasing or decreasing the capacity of their offered services based on specific KPIs. Scaling out or in implies adding or removing on-demand capacity based on the needs.

In the following subsections algorithms are introduced that have been used as part of the RE logic. It is important to highlight that the algorithms proposed

assume that multiple instances of the same service behave similarly under load conditions. This means that the averaged KPI value for all service instances is almost equal to the actual individual value. Such condition can be commonly achieved making use of dynamic load balancing algorithms where the load balancer entity is dynamically checking KPIs of the destination service instance for deciding where to dispatch upcoming requests[154].

4.2.3.3 Threshold-based Scale-out Procedure

The first step of a service in/out scaling process is to analyze the information collected by the MA. Scaling policies, defined in the form of a set of triggers<–>actions, are managed during runtime by the RE, taking the final decision whether to execute a scaling action or not. When a simple load function goes above a certain threshold for a sufficient time interval (to avoid bouncing effects and too reactive behaviors), MA sends a trigger to RE that is in charge of deciding whether to scale out based on the overall load of the `Service Group`. MA can also send a notification because a VNF is in an incorrect state, namely there is at least one process in the VM that is not working properly and, for instance, the CPU is consequently getting overloaded.

In order to avoid such situations, the RE also considers the total number of service instances that are over the threshold, in addition to the average of the CPU usage. Basically the action is triggered only when more than 50% of the units' part of the `Service Group` have a CPU usage over the threshold. The RE decisions depend on the overall resource utilization (current and in a limited time window) of the `Service Group` under control. In case it decides to scale out it requests the CM the instantiation of a new service instance by choosing the most suitable provider through the decision algorithm described in the previous subsection.

4.2.3.4 Optimized Scale-in Procedure

A scale-in operation is required in order to minimize the costs of resources utilization. The most common approach is to implement scaling-in also based on thresholds. It is important to consider that in a typical elastic `Service Group` the removal of a VM hosting a service instance causes a redistribution of the load to the other ones. Therefore, scaling-in operation has to be efficient, but at the same time should avoid fluctuations, where the load redistribution causes the triggering of scale-out actions.

Therefore, the proposed scaling-in algorithm is based on the assumption that a VM can be removed from the `Service Group` *only if* its removal does not bring the other VMs in a state where the scaling-out mechanism would be triggered. To fulfill such requirement, RE analyzes every single VM to check if it is a good candidate for removal according to the proposed algorithm.

To better understand the problem, one can think of the following example shown in Figure 4.5 with two VMs having the thresholds set to 30% for scaling-in and 70% for scaling-out. The two graphs on the left show the individual CPU usage of two VMs. As it can be noticed by the graph on the right, when one of the two VMs is removed, the load on the remaining one will consequently increase.

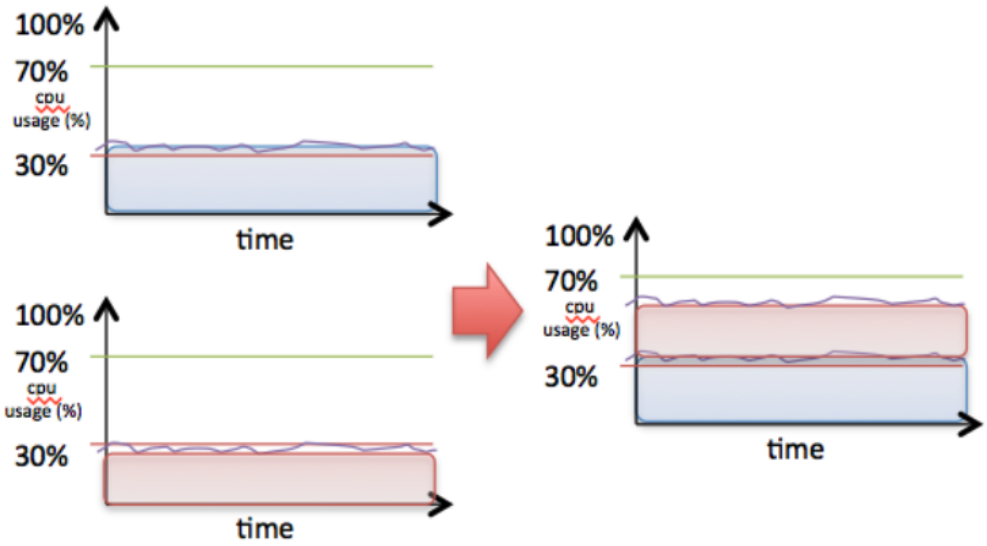

Figure 4.5: Example of the Implication of the Scale-in Operation

This approach works properly for a rather small `Service Group`. It is important to understand that when the number of service instances start increasing exponentially, the usage of the resources is not optimized anymore. For example, considering the case when 1000 service instances are deployed, scaling-in will cause the load redistribution of the removed one on the 999 remaining ones. But this situation might freeze for another unpredictable time period until the average value goes under the scaling-in threshold (30%) again. This is not the most optimized solution considering that those service instances only use around 30% of their capacity.

Furthermore, considering that each VM can be deployed on different physical machines, the associated performance may be significantly different. For this reason, in cases where the decision made is based on average values, it is important that for each type of metric a specific KPI is considered. For instance, for the CPU usage case, while calculating the Total Power (TP) consumption, as the simple sum of all the CPU usage of each service instance in the `Service Group`, it is taken into account the BogoMips[7] value, i.e., ρ_i, a common index calculated in every Linux system to calibrate an internal busy-loop.

Based on these considerations, in this formula, N is the number of VMs in the `Service Group` and U_i is the current CPU utilization of the i-th VM.

$$TP = \sum_{j=1}^{N} \rho_i U_i \tag{4.4}$$

Then RE determines the list of candidate VMs as the ones in a given `Service`

[7] https://en.wikipedia.org/wiki/BogoMips

`Group` for which the following property holds:

$$C_i = \frac{TP}{\sum_{j=1}^{M} \rho_i} \tag{4.5}$$

where UT is the Upper Threshold defined in the scale-out policy and M refers to the subset of VMs excluding the i-th VM. After creating this list a simple algorithm can easily find the candidate that best improves the performance of the whole `Service Group` (this holds because of the round-robin weight-based algorithm):

$$candidate = i - th\, machine\, with\, max\, C_i \tag{4.6}$$

This formula can easily be extended in order to support more sophisticated and service-specific metrics.

4.2.4 Limitations Encountered during the Prototype Phase

This initial phase helped the author in having a better overview of the research issues investigated in the context of his research work, and some of the algorithms proposed for optimized scaling and placement were reused in the follow-on phases. Although concepts investigated during this period satisfied some of the requirements identified in the initial phase of design, like dynamic deployments of software-based networks and elasticity, the solution was not satisfactory for handling complex network services. Dependencies between network functions were only managed for a specific kind of service, either SIP- or HTTP-based. A generic approach for solving runtime dependencies between service instances was not supported by this solution.

The monolithic architectural pattern provided a fast way for designing a prototype solution satisfying the aforementioned requirements, with low development efforts. Deployment resulted to be also simple to handle, however, there have been several drawbacks that became significant to be considered while moving to the second intermediate phase. Nevertheless, during the release cycles of this initial prototype, several additional features and user stories were not considered as part of the backlog.

First of all, the solution was pretty tied to the particular cloud infrastructure chosen. Extensibility, one of the major requirements identified as a research challenge by this dissertation, was pretty hard to achieve with the proposed solution. Moving towards a new cloud infrastructure required several changes in the internal codebase of the proposed framework, hence, would have slowed down developments of additional features.

Furthermore, the system supported only the deployment of a single type of service with strong impacts also on how those service instances need to be packaged for being managed during runtime. No contextualization of running VMs was executed. The approach taken was based on the fact that the service to be deployed and scaled was already preinstalled and preconfigured in the VM image disk. Although this approach goes in contrast with the cloud-native principles presented in

Chapter 2, there are still many solutions adopting this mechanism as a primary way for distributing them.

Although scaling conditions could be defined via scaling policies, the algorithm implementing the logic of solving a particular issue was directly embedded in the RE codebase. Design of a new algorithm would have required a redesign of the RE component itself.

Continuous development was also complex to achieve, considering that any changes on a single part of the system were affecting the entire application, requiring its full redeployment. Last but not least, scaling the framework itself was not an option. Although some clustering solutions would have been enough for statically increasing the number of parallel instances, there was no way of scaling each particular component independently.

4.3 Intermediate Phase

The experiences gained with the design and implementation of the EE during the prototype phase paved the way to a more complex solution having as main objective the deployment of a complex software-based network service. Compared to the previous phase, the major difference was the integration of multiple elements coming from the NGN domain. Some of the requirements that have driven this second phase, have been gathered analyzing different system architectures of the vIMS[121], the vEPC[122], and the vM2M[123].

Furthermore, the technological advances of the OpenStack open source IaaS solution as well as the publication of the first ETSI NFV white paper enabled a redesign of the cloud management modules in order to be easily extended for supporting novel heterogeneous cloud infrastructures.

A redesign process was executed taking into account the limitations encountered during the prototype phase, and for allowing the seamless integration of different kinds of NFs. The proposed functional architecture included the adapter concept as intermediate entity dealing with the management of NF instances.

4.3.1 Intermediate Architecture

The experiences and lessons learned during the prototype and intermediate phases as well as the knowledge acquired analyzing the different network management solutions presented in the Chapter 2 have driven the definition of the first version of the extensible and customazible MANO4X framework, designed in order to simplify end-to-end network service orchestration.

The major architectural changes applied in this intermediate design phase mainly took in consideration the heterogeneity aspects of the environment that has to be managed by the MANO4X framework, both from the perspective of the infrastructure and NFs.

The first assumption is that the infrastructure could be comprised of multiple *sites*, managed by different CMS technologies exposing (in most of the cases) hetero-

geneous APIs. Solving the heterogeneity at the infrastructure level is not a trivial task, considering that different existing technologies do not expose a common interface for offering the on-demand provisioning of compute, storage, and networking resources.

The second assumption is that for supporting the needs of different vertical domains the MANO4X framework should be capable of composing and orchestrating services comprising different kinds of NFs. Orchestrating heterogeneous NFs requires interoperability between different configuration management systems. Therefore, another key aspect that was considered during this phase was the capability of the framework to integrate existing solutions for managing any kind of software component, obtained decoupling the orchestration logic from the actual configuration management of NFs. The proposed solution is presented in Figure 4.6.

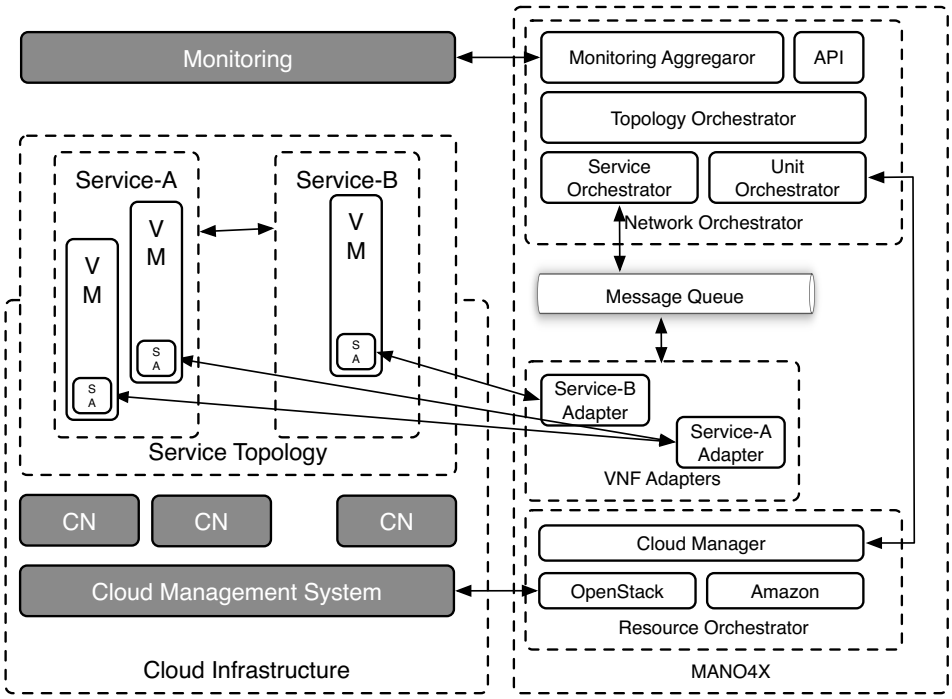

Figure 4.6: Proposed Architecture during the Intermediate Phase

This architectural diagram highlights the first distinction between this version and the previous one, especially with the introduction of the concept of *service topologies*. *Services* exist as *types* and *instances*. A *service type* is used for describing what kind of *services* are provided by this entity. In the context of this research work it is assumed that a service can be implemented by one or more software components, exposing certain functionalities via either standardized or proprietary input/output interfaces. *Services* could be of any *types*. In the telecommunication domain a service corresponds to an NF. Thus, the difference between service *types* and *instances* can be exemplified: A *service type* could generally be used to describe

a standard NF, basically providing information about its capabilities and exposed interfaces, however, a concrete instance running on a particular compute resource represents a *service instance*. A service can be instantiated multiple times in multiple instances.

The *service topology*, represented in Figure 4.6 as a dotted-lined rectangle, is a collection of *services*. A *service topology* exists in the form of *template* and *instance*. A *service topology template* is the result of the design process executed by the TSP selecting services from a catalog. There could be several flavors of the same *service topology*, each one describing a particular end-to-end service supporting the requirements for a particular vertical domain. A *service topology instance* is the result of the overall orchestration process executed by the MANO4X framework.

Following on with the approaches made during the initial phase of design, the cloud infrastructure is composed by several Compute Node managed by a Cloud Management System. In this second phase, the multisite requirement was supported with the introduction of a *resource orchestration* layer abstracting the complexity of the infrastructure topology providing a unified interface exposing CRUD functionalities towards the Service Orchestrator (SO). As already mentioned, typically the semantics and primitives offered by the CMS are implementation-specific (for instance OpenStack, OpenNebula, etc. expose their own proprietary APIs), therefore, creating an abstraction layer providing a common interface across multiple cloud resources is crucial for deploying compute resources on heterogeneous cloud infrastructures. This layer exposes an interface compliant with the OCCI specification presented in Chapter 2 limiting its functionality only to the provisioning of virtual compute resources also defined as *units*[135][130][131].

The SO layer comprises all the functional elements driving the life cycle management of a *service topology*. It exposes a set of APIs that could be consumed by the TSP for retrieving services, creating *service topology templates*, and triggering their deployments. The Unit Orchestrator (UO) functional element of the SO manages virtual compute resources consuming the interface exposed by the Resource Orchestrator (RO). This allows the SO to orchestrate services across the multisite infrastructure. The SO entity usually represents the entry point for the TSP dealing with the life cycle management of service topologies.

The Service Adapter (SA) layer comprises all the individual SA entities in charge of managing the life cycle of a particular *service type*. The SA is an intermediate entity between the SO and the actual *service instance*, providing an abstract interface to the SO for triggering the execution of the installation and the configuration of a *service instance*.

The communication between the SO and the SAs was realized over a message queue. The specific communication mechanism selected is the pub/sub model, allowing fully decoupling the SA layer from the SO. Basically, a topic-based pub/sub mechanism was used for distributing messages to multiple subscribers. As shown in Figure 4.7 a `messageSelector` parameter was used for targeting the receiver of the messages published.

Whenever a new SA is activated, it registers to the SO, particularly to the

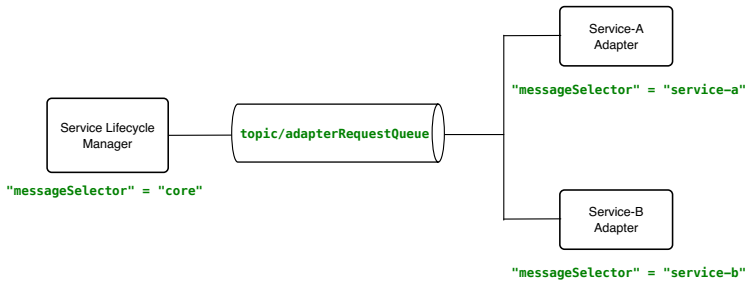

Figure 4.7: Message Queue Configuration

Service Lifecycle Management (SLM) module, announcing its ‘‘`messageSelector`’’ parameter that can be used later on by the SO to send life cycle management operations.

4.3.2 Information Model

As mentioned before, a common information and data model is crucial for allowing interoperability between different orchestration solutions, however, back in 2013, the ETSI NFV ISG had not yet released the information and data model for describing network services, which could have been a perfect match for defining a service topology. Therefore, the approach taken during this phase was to adopt and further adapt the model introduced during the prototype phase. The approach outlined in this phase is mainly based on the practical need perceived by the author while further investigating aspects related with management and orchestration of complex software-based networks, like the vEPC and vIMS use cases. In order to simplify the portability of such templates across multiple MANO frameworks, a common format based on the TOSCA specification was adopted. A representation of the *service topology template* in a TOSCA format is provided as part of the Appendix B. Figure 4.8 provides the Unified Modeling Language (UML) representation of the *service topology template*.

A *service topology* consists of multiple *service containers*. A *service container* defines all the properties required for instantiating the virtualized compute resources on top of which the *service instances* are deployed. For example, properties like *disk image name*, *flavours type*, *min* and *max number of instances*, are all defined as part of the *service container*. In addition, the *service container* references the *subnets* object defining what kind of virtual networks should be used for this specific container, and the *service* object. The *service* object defines generic *configuration parameters* (like the port numbers that should be used for a particular interface, etc.), the *service type* that is used by the SO in order to identify which SA is responsible for this particular service instantiation, as well as the *autoscaling policy*, following the model adopted in the prototype phase. Last but not least, the *service* object contains a *requires* parameter that is used for defining which other *services*

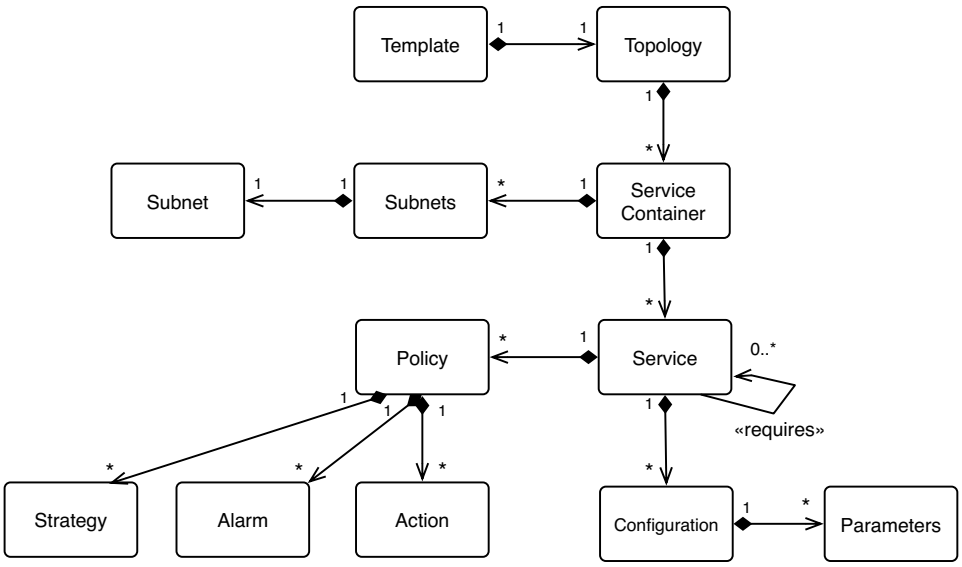

Figure 4.8: Service Topology

are required for being fully functional.

A logical representation of the *service topology instance* designed during this phase is provided in Figure 4.9.

The major difference between the *service topology template* and the *service topology instance* object is represented by the *Relation Element*. As mentioned, solving dependencies (also defined as relationships) between multiple services represents one of the most challenging operations executed by any management and orchestration system as it requires maintaining compatibility also between services implemented by different vendors. Thus, the concept of the *Relation Element* exemplifies the *service topology instance* graph, identifying every single atomic element of the graph. A *Relation Element* is basically constituted by a *unit* and a *service instance*.

4.3.3 Service Topology Life Cycle

A *service topology instance* is the result of the orchestration process executed by the SO entity receiving as input the *service topology template*. A *service topology template* provides information about the service components and their *relationship*. The concept of *relationship* represents a crucial aspect in the service orchestration process, therefore, it is important to provide additional details about the way *relationships* are defined and realized.

4.3.3.1 Design and On-Boarding Phases

The first step of the life cycle is represented by the *design* phase. This operation consists of two different steps executed by two different actors: The *SP* and the

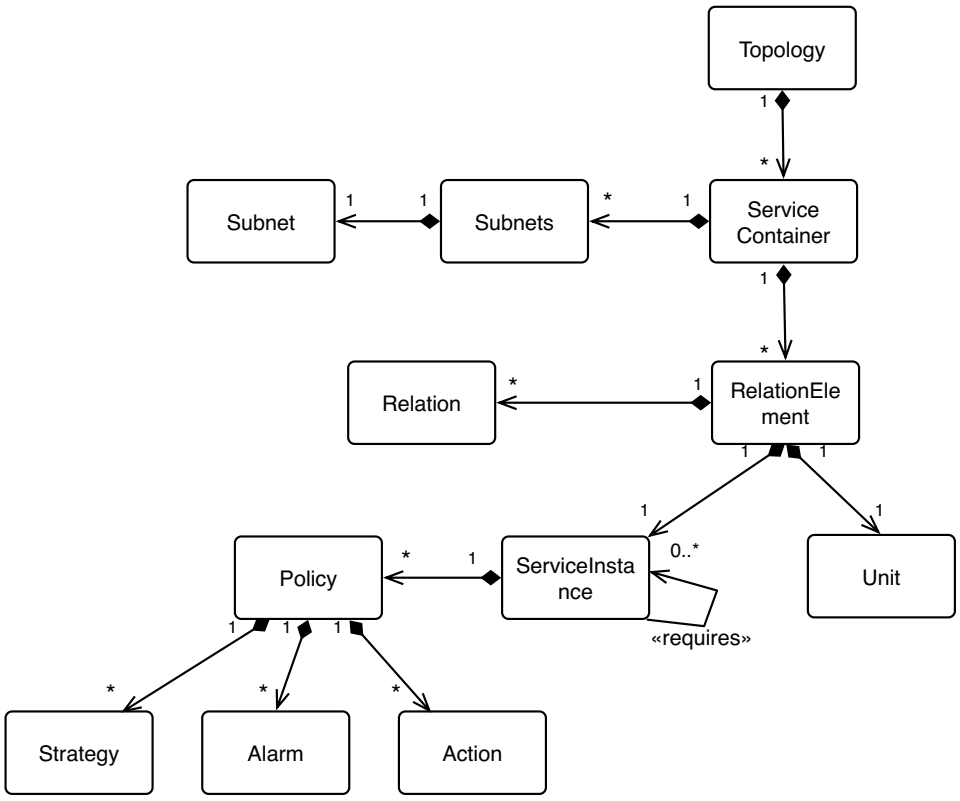

Figure 4.9: Service Topology Instance

TSP[8]. This phase consists of designing and implementing a service, and creating a portable package that could be distributed to any TSPs. Considering that a service could be of any type, the SP should package the solution in a specific format which is portable across multiple orchestration solutions, and a description of the service should be based on the information model previously introduced. The approach taken by the author during this phase was to define a service package as the combination between:

- Disk Image: Containing the software components of the service that could be instantiated in form of virtual compute resources.
- SA: An executable binary in charge of the life cycle of this particular *service type*.

During the on-boarding phase the SP delivers the package to the TSP that installs its content inside its MANO4X framework. The launch of the SA process represents the service registration phase. In fact, the SA publishes over a message bus a registration message including a description of the service under its control.

[8]In this context, the SP corresponds to the VNFP actor identified in Chapter 3

The SO entity receives the registration message and stores the service definition in its catalog.

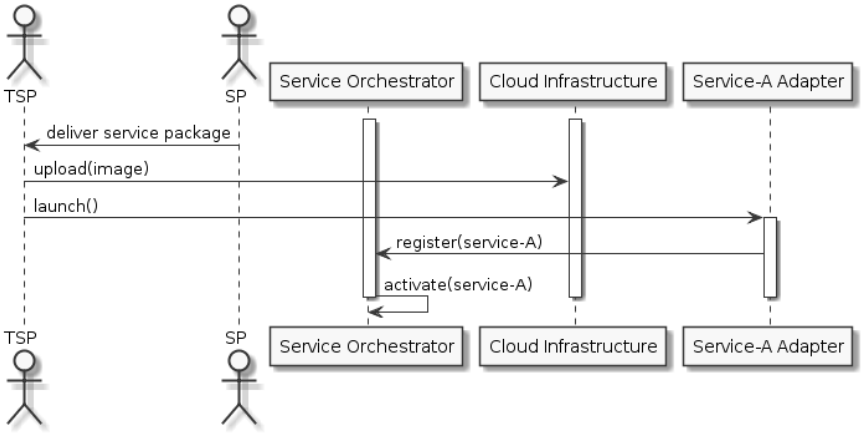

Figure 4.10: On-Boarding Service Packages on MANO4X Framework

The second design phase is represented by the *service topology* composition. The TSP could either create the *service topology* manually combining different services in a *service topology template*, or using a Graphical User Interface that could facilitate its creation via drag-and-drop functionality. While generating the service topology template, the TSP could specify requirements in terms of:

- **site locations**: the TSP could (optionally specify the sites where such service should be deployed. In cases where the TSP does not specify any particular site, the orchestration process should decide based on placement algorithm decisions which site to select for the specific deployment.
- **infrastructure resource capabilities**: the TSP could specify how many compute resources should be used for a particular service instance.
- **service specific configurations**: each service may expose configuration parameters (defined as key-value sets) which could dynamically be configured by the TSP at each deployment. For instance, in case of a Domain Name System (DNS) service, the TSP could decide either to modify the default realm with its own specifics or modify the number of a port exposed by such service once instantiated.
- **autoscaling policies**: policies could be of any types, and the TSP could provide them as part of the service topology to instruct the operational support service during the runtime phase.

The result of this operation is the *service topology template*, which is uploaded to the SO catalog by the TSP. The separation of the design phase and the on-boarding phase is critical for understanding the orchestration process further outlined in the

following. While creating a *service topology template*, the TSP should not be aware of the details of the service life cycle management process that is completely delegated to the SO entity. The TSP focuses on the overall end-to-end service required for their end customers. While composing the *service topology template*, the TSP selects services based on their capabilities in terms of functionalities provided. The low-level details of how these services will be instantiated, and which virtualization technologies are required for their execution, are defined by the SP and consumed by the SO and SA during the deployment phase.

4.3.3.2 Deployment Phase

The actual orchestration process starts with the execution of the deployment operation triggered by the TSP selecting a *service topology* from the catalog. The orchestration process consists of several intermediate steps that are executed with the objective of generating a *service topology instance* based on the *service topology template* received as input. As already mentioned, the service topology template could be instantiated several times (also in parallel), however, *service topology instances* differ between each other because of different runtime information.

To better understand the deployment process, in Figure 4.11 a simple *service topology* is shown.

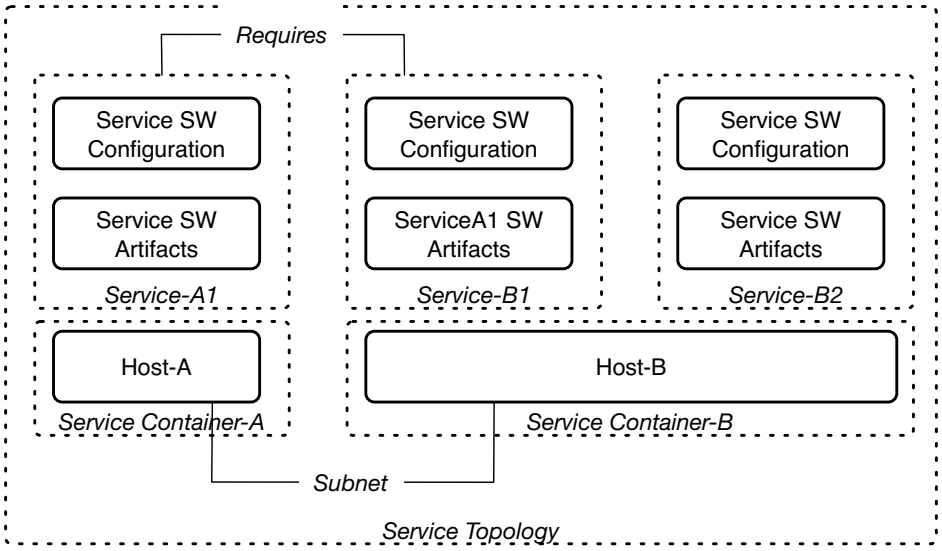

Figure 4.11: High-Level View of a Service Topology

The first step in the orchestration process is the instantiation of the infrastructure resources required by the different service instances. This step consists of several operations involving the SO and the RO. As the initial step, the SO iterates over the services defined in the *service topology* and starts instantiating via the RO entity, the subnets required by the *service topology*, as well as the number of units required

by each individual service, based on the properties defined in the service container. The SO (using the UO functional element) starts interacting with the RO of the selected sites in order to instantiate the compute units required. The result of this process is the infrastructure resource graph instance, comprising running compute and subnets resources across the different sites. This means that the infrastructure resources needed for hosting the service instances are ready.

The second step executed by the SO is the instantiation of the service itself. This step handled by the SLM involves the SA entities, being the components in charge of instantiating a service. The interaction between the SLM and the SA has been realized over a message bus. Table 4.1 defines the three major phases executed for each individual service instance of the service topology during the instantiation procedure.

Lifecycle Event	**Description**	**Input**
INSTALL	Instantiation phase	Service Template
ADDRELATION	Relation management phase	Relation Element
START	Start phase	-

Table 4.1: Lifecycle phases during the deployment phase

During the INSTALL phase, the SO requests the SA to instantiate services under its control, passing as input the *service template* including the information about infrastructure resources allocated to that particular *service instance*. In order to install the required software artifacts, the SA can make use of any mechanism/technology for instrumenting the virtualized compute resources allocated to the service component under its control. An example of such process is the execution of installation scripts comprising the download and installation of the required software components.

After this phase, each individual component is available as a stand-alone component, however, not contextualized for interacting with other components. Inter-service dependencies represent one of the major issues that need to be solved during the deployment phase by the SO. Following the example presented before, it is clear that for the proper functioning of the service composition, it is required that each relation (defined by the *requires* line) between service instances is satisfied before starting the service instance components. For example, the typical problem is represented by dynamic address resolution of the service instance peers that are composing the service.

The approach proposed in the following is a generalized approach that could be applied to any kind of services. Basically, each service declares:

- what information is `required` by this service instance from other peers
- what information is `provided` by this service instance to other peers

Clear separation between the service topology specification and the service management operations enables to further elaborate a generic orchestration logic. The

logical correlation between the service topology specification and the executable orchestration logic is performed by the SO function.

4.3.3.3 Runtime Phase

The experiences gained during the prototype phase about elastically autoscaling service instances were adopted and further extended as part of the runtime phase of the service life cycle. Basically, the architectural components designed in the previous phase were integrated in this version of the architecture as additional functional elements providing policy management functionalities. After the deployment, the RE execution is activated based on the *policy* contained in the service object. The mechanism for detecting the need to scale, and deciding what action to trigger, remained pretty much the same.

The major difference between the previous phase and the one outlined in this section is represented by the integration between the RE and the SLM as intermediate entity towards the CMS. In the previous version, the RE was directly interacting with the CM entity for requesting the instantiation or disposal of a VM hosting a certain service instance. In this current approach, the RE interacts with the SLM for requesting the addition or removal of a service instance from an already deployed service topology instance.

4.3.4 Limitations Encountered during the Intermediate Phase

This intermediate architectural solution, followed by its implementation as part of the Fraunhofer FOKUS Open Software Defined Network Core (OpenSDNCore) toolkit[9], provided the basis for managing and orchestrating complex end-to-end network services on top of a multisite cloud infrastructure. This solution satisfied additional requirements, like composition of multiple service types in a service topology, however, additional limitations were identified during this intermediate process.

The first limitation encountered was that, although the SA entity provided a mechanism for separating the overall end-to-end service life cycle from the management of a particular service, the solution adopted required a dedicated SA component for each type of service to be managed. One of the major drawbacks of such solution is that the SA entity is also responsible for publishing the service description in the SO catalog. This was a major limit in terms of extensibility as the integration of a new service into this architecture would require the deployment of a new SA entity.

Furthermore, management of virtual networks was not realized. The assumption made during this phase was that the TSP would prepare, in advance, the virtual networks on its cloud infrastructure required for hosting service topology instances.

Last but not least, runtime management only covered elasticity aspects of a service. Although the approach taken was policy-driven, autoscaling functions were

[9] http://www.opensdncore.org/

embedded within the service orchestration entities.

4.4 Final Phase

The final phase was mainly influenced by the parallel evolution of the ETSI NFV ISG standardization work. In 2015 the ISG completed the first phase of its work providing 11 comprehensive specifications covering several aspects related to the different NFV domains, including MANO [155]. Most importantly, a specification was provided concerning the information and data model for describing NSs and VNFs, as already presented in Section 2.3.

After a conceptual study conducted by the author about the major differences between the models and concepts defined in the previous phases, and the ones introduced by the ETSI NFV ISG as part of the first specification set, the decision made was to adopt and further extend the ETSI NFV MANO information and data model as the core model of the MANO4X framework, while keeping the orchestration and runtime management processes proposed in the previous phases. This was mainly achieved due to the fact that the information and data model proposed by ETSI had a large set of similarities with the one proposed by the author during the intermediate phase.

Nevertheless, most of the work carried out during this period was mainly devoted to the design of the framework itself, targeting extensibility and customizability as major driving requirements. The final version of the architecture, provided at the end of this phase, can be considered the most comprehensive ETSI NFV MANO compliant solution combining several innovative design principles and concepts investigated during the research work.

4.4.0.1 Extensions to the ETSI NFV Information and Data Model

The final approach used in this thesis intends to reuse the information and data model proposed by the ETSI NFV MANO specification and extend it with any additional parameter needed for supporting the end-to-end life cycle. For the sake of clarity, the reference information and data model considered during the period of this research work is the first version published at the end of 2014[28]. The second version of this model, which was published at the end of 2016, is considered an evolution of the first one, therefore, concepts and extensions applied in the context of this research work can also be applied to the newest version available.

As already presented in Chapter 2, the NFV information model is generally divided in two sets of entities: descriptors and records. Information inside descriptor elements are rather static, mainly used for initiating the on-boarding process of VNFs and NSs. Information inside record elements are instead dynamic, generated by the deployment process and modified during runtime during the life cycle. The cause of this variation can be a life cycle operation completion or even a result of an operation executed because of an expected (i.e., scaling) or unexpected (i.e., fault)

event. Table 4.2 provides a mapping between the information model proposed by ETSI NFV and the one defined during the intermediate phase.

ETSI NFV Information Model	**Intermediate Information Model**
NS	*Service Topology*
NSD	*Service Topology Template*
Network Service Record (NSR)	*Service Topology Instance*
VNF	*Service*
VNF Record (VNFR)	*Service Instance*
VNFC	*Service Instance Component*

Table 4.2: Mapping between the ETSI NFV Information Model and the Intermediate one Proposed in this Thesis

As it can be noticed from Table 4.2, the information model proposed by the author during the intermediate phase can be mapped 1:1 to the one proposed by ETSI NFV. Basically, the *service topology template* corresponds to the NSD as it provides static information about the composition that has to be deployed, while the *service topology instance* corresponds to the NSR as it contains all runtime information generated during the life cycle management operations. The NSD references VNFDs, Virtual Link Descriptors (VLDs), VNFFG Descriptors (VNFFGDs) describing the different elements composing the network service.

Few modifications have been applied to the data model of the different descriptors for supporting the different features required during life cycle management. In most of the cases, extensions were needed also because the specification did not provide a detailed definition of these parameters. Starting with the NSD, a new field, the *vnf_dependency* attribute, has been added in order to describe dependencies between VNFs. This object comprises a set of parameters exchanged by a pair of VNFs. Following the dependency resolution approach designed during the intermediate phase with the concept of *Relation Element*, a dependency between two VNFs is exemplified in terms of *source* and *target* VNF.

Focusing on the VNFD, extensions were mainly done for allowing VNF placement, as well as policies required by the external OSSs components. In particular, a new parameter *VimInstance* is added to the VNFD, which can be used for defining the list of potential PoPs where VNFCs should be deployed. During the deployment phase, the NFVO can utilize any kind of VNF placement algorithm for selecting the most suitable PoP where to deploy the VNF.

4.4.1 Mapping Between the Intermediate MANO4X Architecture and the ETSI NFV Architecture

Following on with the mapping between the approaches taken in the ETSI NFV ISG and the ones presented during the intermediate phase of this research work, it is important to highlight also similarities between the two architectural models. Figure 4.12 shows the mapping between the architecture proposed in the intermediate phase and the ETSI NFV one.

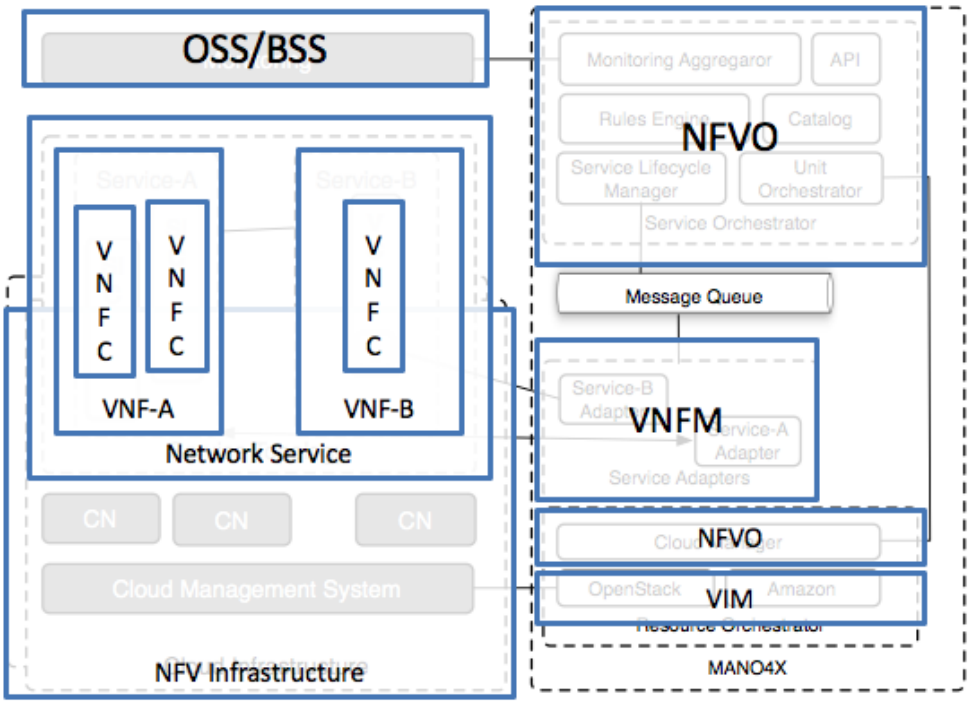

Figure 4.12: Mapping between the ETSI NFV Architecture and the MANO4X Intermediate Version

As it can be noticed, there have been similar approaches between the two different architectural models. The following Table 4.3 provides an overview of the mapping between functional components proposed by ETSI NFV and the MANO4X intermediate architecture.

Most of the functions of the SO and the RO can be mapped on the NFVO functional element. In fact, the NFVO comprises functionalities for managing the life cycle of virtualized resources as well as network services. The SA corresponds to the VNFM as it is basically an intermediate element between the orchestration layer and the actual VNF. The NFVI is by definition a cloud infrastructure, initial Proof of Concepts (PoCs) showcased the NFVI implemented using OpenStack[156]. The OSS/BSS layer, especially the OSS one, correspond to the Monitoring System, MA and RE as they enhance the orchestration capabilities providing runtime

ETSI NFV ISG	MANO4X Intermediate Phase
NFVO	Most of the modules of the SO & RO
VNFM	SA
VIM	CMS
NFVI	Cloud Infrastructure
OSS/BSS	Monitoring System, MA and RE

Table 4.3: Mapping between ETSI NFV and MANO4X Intermediate Architecture

management functionalities, fulfilling the FCAPS model requirements.

4.4.2 Architectural Design Principles

The aforementioned limitations identified in the first two design phases put strong emphasis on the need of an extensible architecture where different components could be introduced using plug-and-play mechanisms. The intermediate version of the architecture provided the means for understanding the complexity of building up an extensible and customizable framework. The concept of *service adapters* allows decoupling the management and configuration of service instances from the general orchestration function. However, the approach made was not scalable enough, especially for large and complex network services.

The *Hollywood Principle "Don't call us; we'll call you (if we want you)"* together with SOA and microservices architectural patterns, have driven the design of the final version of the architecture. In Chapter 2 microservices architectural patterns were introduced as a novel concept for building distributed applications with loosely coupled services. As it was extensively discussed in Chapter 2, microservices architecture suppose that a particular application must be decomposed in several microservices.

Those patterns share some common objectives[157] that have driven the design of the proposed final version of the MANO4X framework architecture:

- *Domain Driven Design*: The architecture should be decomposed in different domains so that developers could focus on the core logic of each individual domain.
- *Information Hiding*: Each component that is part of the final architecture should have an independent life cycle, being still part of the big structure. In order to increase the ability to scale independently of the others, components should use ligthweight communication mechanism (i.e., REST over HTTP, or Pub/Sub) in order to hide implementation details.
- *Decentralization*: Each individual component makes use of its own persistency layer, without having a single logical database across a range of applications.

- *Failure Isolation*: Decoupling the architecture into multiple microservices reduces the risk of failure of the overall system.

From this perspective, the designed solution enables the integration of any kind of infrastructure resources and configuration management through the employment of the *adapter* design pattern. In order to realize such distributed architecture the Event-Driven Architecture (EDA) design pattern was employed. The main idea is to have a central message bus acting as a broker between the different domains, distributing events generated by the central orchestrator entity to external entities contributing to the overall service life cycle.

The final approach, the key contributions of this dissertation, is hereby defined as *"event-driven orchestration"*, providing an extensible and customizable framework with a predefined state machine in which each state transition is based on a particular event, being a user request or a message from another internal component.

4.4.3 The Final Architecture of the MANO4X Framework

Following the *Domain Driven Design*, the final architecture proposed comprises different elements that have been categorized in four main surrounding domains interacting with the main components as part of the central domain. Figure 4.13 shows the high-level overview of the different domains identified in the scope of this dissertation.

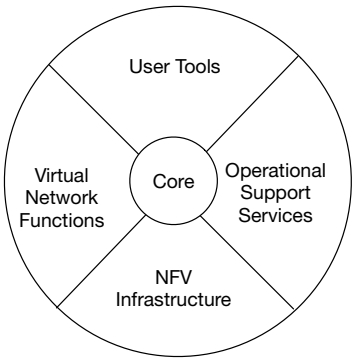

Figure 4.13: Overview of the Main Domains Composing the MANO4X Framework

The central domain represents the core of the MANO4X framework, executing the network service life cycle management using an event-driven engine brokering event received from the northbound domain, typically user-driven, and the ones generated by the other three domains (eastbound, westbound, and southbound). Using microservices principles, each individual function acts as a stand-alone component being activated/deactivated based on events generated during the life cycle. In this way, also the design and development of a particular function providing specific features for a particular use case can be done independently, and can be plugged into

the system whenever it is needed. This approach satisfies the *information hiding*, *decentralization*, and *failure isolation* design patterns aforementioned.

The proposed framework abstracts the view portion, what users actually see and interact with, exposed via a northbound domain comprising user tools (CLI, SDK, or a web-based dashboard), from the internal orchestration logic. The main objective is to expose towards this domain the view layer providing functionalities for controlling the life cycle of network services and VNFs, consumed either via an APIs, SDKs and/or Graphical User Interface (GUI).

The southbound domain comprises heterogeneous infrastructures: central clouds, MEC nodes, or even FOG devices. The commonality between those infrastructures is that they all offer compute, storage, and networking resources as atomic elements. One of the major components belonging to this domain is the VIM that provides an interface for the on-demand provisioning of those atomic resources. Typically, this domain is comprised by several kinds of IaaS exposing different interfaces for the control of the individual resources. The proposed framework should be able to manage the execution of any kind of compute resources such as virtual machines, containers or bare metal, and to on-demand connect them to each other. A driver-based mechanism, following the adapter design pattern, has been designed in order to support interoperability with multiple infrastructure providers. Even though OpenStack[158] represents the standard de facto implementation of the VIM[159] the MANO4X framework should be decoupled from it, and should be extensible enough to easily accommodate other VIM implementations.

The westbound domain corresponds to the VNF domain. It is the most critical domain for supporting heterogeneous vertical use cases and satisfying interoperability across VNFs provided by different vendors. The ETSI NFV architecture already decouples the network service life cycle management from the VNF life cycle management making use of the VNFM functional entity. Although this logical separation exists as a native separation, MANO4X should support the possibility of incorporating on demand additional VNFMs in a plug-and-play fashion, and accommodate different kinds of VNFMs (either specific or generic).

Last but not least, the eastside domain comprises elements usually belonging to the OSS (and consequently BSS) domain contributing to the overall life cycle of the end-to-end network service. Particularly, OSS elements contribute to the runtime phase of the network service execution, ensuring that the aspects defined by the FCAPS are guaranteed along the overall service life cycle.

4.5 Conclusion

The present chapter introduced the design evolution of the MANO4X framework. Following an agile methodology, three major phases were executed during the research work conducted by the author. Each individual phase contributed to achieving the overall research objectives.

In particular, the results achieved during the final phase of design have integrated

concepts and methods designed in the previous phases, as well as influenced by the evolution of the ETSI NFV ISG activities.

The final solution proposed can be considered an extensible and customizable NFV MANO-compliant framework in which different elements could be combined together for satisfying the particular requirements of a set of very heterogeneous use cases, being the deployment of the 3GPP EPC on a typical NFVI environment (i.e., OpenStack-based), or the deployment of the virtual caching system on top of an MEC node (i.e., container-based).

Chapter 5

Specification of the MANO4X Framework

The previous Chapter 4 – The Design Evolution of the MANO4X Framework introduced the design evolution of the MANO4X architectural framework and provided the MANO4X reference architecture as a result of different design phases.

The final architecture proposed supports the integration of various types of network services and virtualization technologies, and provides a horizontal solution to support multiple vertical stakeholders. The aim of this architecture is to satisfy the list of identified user stories and to support the specified, relevant functional and nonfunctional requirements while taking into consideration heterogeneous and distributed NFVIs.

This chapter presents the specification of the main functional elements of the MANO4X framework. The first part elaborates the different domains, particularly focusing on the functional elements comprising the MANO4X functional architecture. The second part outlines the high-level procedures executed for managing the entire life cycle of a network service.

5.1 General Overview

The MANO4X framework is designed for managing and orchestrating software-based network services on top of a multisite NFVI. As already introduced in Section 4.4.3, the framework is composed of a central domain surrounded by four other individual domains contributing to the overall life cycle. The functional architecture is depicted in Figure 5.1. It includes the interfaces with the NFVI as well as external entities, like VNFMs and OSSs.

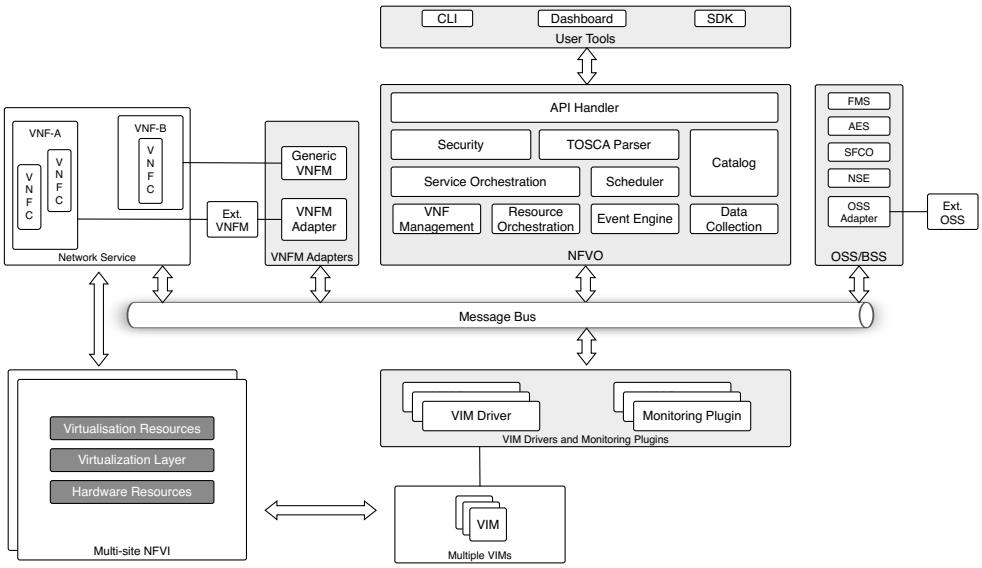

Figure 5.1: MANO4X Framework Functional Architecture

Following the domain separation a short overview is given about the functional elements of each domain serving as intermediate elements between the MANO4X framework and the external entities.

The central domain comprises the NFVO and message bus. The NFVO orchestrates NSs through their life cycle while the message bus provides a loosely coupled communication mechanism for integrating elements from the other domains.

The user tools north domain is composed of the dashboard, the CLI, and a set of SDKs, providing support for interacting with the MANO4X. All these tools interact with the NFVO through REST APIs, and no state is maintained on the client side except for authorization and authentication information (i.e., security tokens).

The south domain comprises the VIM drivers and monitoring plugins allowing the integration with the multisite NFVI. VIM drivers can be of different types, depending on the actual cloud technologies used as part of the NFVI (e.g., OpenStack, OpenNebula, etc.). In particular, a VIM driver type corresponds to the type of APIs exposed by a VIM.

The west domain includes one or more VNFMs handling the life cycle of a particular VNF and their respective EMSs.

Last but not least, the east domain comprises external OSSs handling a particular aspect of the network service life cycle (i.e., scalability, fault management, security, etc.). This domain comprises OSSs designed as part of the MANO4X framework (i.e., the FMS, the AES, the SFCO, and the NSE) as well as OSS adapters allowing integrating existing OSSs external to the MANO4X framework.

In the following subsections a description is provided about each domain and their respective components.

5.2 Central Domain: NFVO and Message Bus

The central domain comprises the two main functional elements of the MANO4X framework: The NFVO and the message bus. The NFVO represents the main entity within the MANO4X architecture orchestrating the life cycle of software-based network services, while the message bus is the messaging system allowing decoupled communication between the NFVO and functional elements composing other domains.

The NFVO has a central role within the framework driving the life cycle management operations. It exposes a REST-based northbound API that can be consumed through the user tools provided by the north domain.

All in all, the major responsibility of the NFVO is to correctly handle the order in which certain operations are carried out, executing life cycle management of the VNFs belonging to a certain NS. This encompasses tasks such as:

1. Allocating infrastructure resources on the multisite NFVI.
2. Satisfying the VNFs dependencies defined in the NS, matching the target of each dependency with a source able to satisfy its requirements (in terms of statically and dynamically defined information).

3. Modifying the configuration contained in the VNF to specify the address at which the dependency source resides in the scope of the Virtual Link (VL) between them.

4. Starting VNFs respecting the order defined by their dependencies. Ordered start could be an optional choice that could be enabled or disabled based on the needs of the TSP.

5. Executing runtime operations (e.g., scaling, healing, etc.) throughout the NS life cycle based on decisions made by external OSSs.

The task defined under 1) requires interactions with the VIM, thus the south domain, in order to allocate infrastructure resources on top of the multisite NFVI. Alongside the NS life cycle management, the NFVO has a global overview of the infrastructure resources available at the multisite NFVI level, interfacing with the VIM of each individual site, over the *Or-Vi* interface. Thus, the NFVO is responsible for granting the allocation of certain resources on a particular NFVI-PoP. Selection of the NFVI-PoP maybe directly provided by the TSP while deploying the NS, or left to the NFVO based on advanced placement algorithms. Information about the allocated virtualized resources are later on provided to the VNFM that handles the life cycle of VNFs, taking care of installing, configuring, and starting the software-based network functions.

The tasks defined under 2) and 4) do not usually require intervention by components external to the NFVO itself which is capable to resolve dependencies through its internal dependency management services, and to handle the ordered issuing of events if configured to do so. Nevertheless, the NFVO functionality concerns more the overall end-to-end network service orchestration as the particular VNF management task is delegated to the VNFM. Although this may be seen as a limitation of functionality, delegating the VNF life cycle management to the VNFM allows defining a generic orchestration logic composing heterogeneous VNFs in a network service, without knowing the details about how VNFs are actually instantiated and configured.

The task defined under 3) instead heavily relies on the VNFM, which needs to be able to modify the configuration of the target VNF to introduce in it the parameters resolved by the NFVO, and on the VIM, which needs to expose to the NFVO exact knowledge about the location of the source VNF. The NFVO uses the *Or-Vnfm* interface to instruct it to carry out the VNF life cycle operations it needs to do on a given function.

The task defined under 5) requires support of the OSSs. Basically, the OSS, based on external conditions and events, may require modifications of the deployed network services requesting the NFVO to execute particular actions towards the network service.

As it can be seen in Figure 5.1, the NFVO comprises the following functional elements:

- **API Handler**: Exposing a set of REST-based APIs to components from the other domains, and transforming incoming requests to specific API calls to other internal functional elements.

- **Security**: Providing a set of functionalities for the authentication and authorization of users. Furthermore, this functional element ensures multitenancy providing logically separated spaces, called projects, where multiple users can work together utilizing a subset of the resources provided by the framework.

- **Catalog**: Exposing a set of APIs for storing information and data related to the different domains. Those repositories can be used for storing network service descriptors and respective records, as well as maintaining an overview of the different components running in the system (i.e., available VIM drivers and VNFMs).

- **TOSCA Parser**: Providing a parser functionality capable of translating the TOSCA *topology template* into an NSD. Once transformed, the template is stored into the catalog in the form of a descriptor. Having a separated functional element for parsing TOSCA *topology templates* allows decoupling the external description language from the internal information model and orchestration logic.

- **VNF Management**: Providing an abstracted API to other functional elements for allowing them to communicate with VNFMs.

- **Resource Orchestration**: Orchestrating virtualized resources provided by the multisite NFVI. It acts as a broker among other internal functional elements and VIM drivers, abstracting the complexity of the NFVI, allowing instantiating virtualized resources without necessarily knowing the underneath technologies used.

- **Service Orchestration**: Representing the core functional element of the NFVO having the responsibility of managing the overall network service life cycle. It receives external requests from the API Handler, interacts with the *Resource Orchestration* function for acquiring virtualized resources, drives the deployment of network services through the VNF management, and collaborates with external OSS for managing the runtime phase of network services.

- **Scheduler**: Providing a generic scheduler for executing periodic tasks of a different nature.

- **Data Collection**: Retrieving external monitoring information.

- **Event Engine**: Managing event endpoints and generating events to external consumers subscribed to particular life cycle events.

In the following subsections details will be provided about the most important functions of the NFVO. A definition of the interfaces exposed towards functional elements of other domains is provided in Section B.2. The interface naming convention follows the one proposed by the ETSI NFV MANO specification[28].

5.2.1 Catalogs of Descriptors and Records

The MANO4X framework uses a set of catalogs and repositories for storing information and data regarding the different domains. This information is usually on-boarded on the MANO4X framework either using packages or descriptors. To this end, both statically defined information provided by the TSP and dynamically generated information about the running NSs and VNFs, are typically stored in private catalogs hosted within an instance of the MANO4X framework.

Following the ETSI NFV MANO specification, the MANO4X framework provides the following set of catalogs:

- *NS Catalog*: This catalog contains the on-boarded NSDs, including information of the network service composition as set of multiple dependent VNFs, VL and VNFFG. Such catalog is typically populated by TSPs while creating new network services as composition of multiple VNFs available in the VNF catalog.
- *VNF Catalog*: This catalog is used to store information about the on-boarded individual VNFDs and VNF Packages. It supports the creation and management of new VNFs providing details on the software images used by the VNF and their descriptors. The catalog is typically populated by the VNF providers while being queried by the TSP while creating new network services.
- *NSR Catalog*: This catalog contains records of instantiated NSs.
- *VNFR Catalog*: This catalog contains records of instantiated VNFs.

Statically defined information can also be available as part of a public catalog, defined as *marketplace*, acting as a global catalog shared among multiple instances of an existing MANO4X framework. The *marketplace* is a shared catalog containing packages and descriptors that have been validated and tested for a particular version of the MANO4X framework. A TSP can decide to download existing packages and descriptors from the *marketplace* to its local catalog.

In addition to the above list, MANO4X defines an additional set of catalogs for storing information related to components needed for interacting with the NFVI, the VNF, and OSS domains, as well as security-related ones. In particular:

- NFVI Catalog: Contains information about registered VIM Drivers (particularly their endpoint and state) and NFVI-PoPs.
- VNF Catalog: Contains information about registered VNFMs (particularly their endpoint and state).

- OSS Catalog: Maintains an active list of external OSSs subscribed for receiving life cycle changes events of any types.

A definition of the different REST APIs exposed by each different catalog is provided in Section B.2.1.

5.2.2 Security and Multitenancy

Based on the requirements identified in Chapter 3, *authentication* and *authorization* are needed for allowing multiple *users* to interact with the framework and manage their individual resources without interfering with each other.

On the one hand, *authentication* is the most common feature required for knowing *who* the user is. During the installation process, the platform administrator is granted some user credentials, having the rights to perform any kind of operations within the system. Authentication of additional users is supported in the MANO4X framework, following a standard registration procedure accomplished by the platform administrator providing details about new users who need to be registered. For administrative purposes, the MANO4X framework also supports the removal of a particular user from the platform.

On the other hand, *authorization* is the most common feature required for defining *what* each user can do. Typically, authorization requires the definition of different *roles* assigning users different permission levels. Before introducing the different *roles* that can be assigned to *users*, it is important to highlight also the concept of *multitenancy*.

Multitenancy is a mechanism that becomes very prominent in the cloud computing domain, and it refers to the concept of running multiple logically isolated spaces dedicated to one or more users. In the context of this work, a tenant is a *project*, a logically separated space in which multiple *users* could work together utilizing a subset of the overall services managed by the MANO4X framework. In general, to each *project* there could be assigned a limited set of resources defined as *quota*.

In the MANO4X framework, three major *users' roles* have been identified:

- *GUEST*: A *user* able to perform read operations, but without any rights to modify the status of any resources.
- *USER*: A *user* able to perform any actions in a particular *project*.
- *ADMIN*: A *user* able to perform any kind of action on the whole system without any restrictions.

A definition of the authentication and authorization REST APIs exposed by the security functional element is provided in Section B.2.3.

5.2.3 The Message Bus as Primary Internal Communication Mechanism

The communication paradigm between loosely coupled components of the MANO4X framework leverages primarily the *message bus* approach providing a higher degree

of flexibility and extensibility in adding and removing components from the architecture. The MANO4X framework's internal communication is mainly based on topic-based pub/sub asynchronous communication and Remote Procedure Call (RPC) patterns. For the sake of clarity, RPC over the message bus follows the pub/sub communication approach, requiring a queue for handling requests sent to external components and another queue for handling their responses.

Thus, the communication between components is completely asynchronous, reducing the overall risks of employing synchronous communications, especially for requests that may require some time to generate a response (i.e., the allocation of virtualized resources). Another advantage in employing the message bus relates to the scalability problem: There could be multiple instances of the same component handling a particular request, thus allowing horizontal scalability of the framework in case of increasing load. The particular communication pattern employed depends on the interface type exposed by the NFVO.

The NFVO requires six queues associated with the `openbaton-exchange` topic:

- `nfvo.vnfm.register`: Messages on this queue are related to the subscription of a VNFM component that became available in the system.
- `nfvo.vnfm.unregister`: Messages on this queue are related to the unsubscription of a VNFM component that became unavailable in the system.
- `vnfm.nfvo.actions`: Messages on this queue are actions sent by the VNFM .
- `vnfm.nfvo.actions.reply`: Messages on this queue are actions sent by the VNFM expecting a reply message.
- `nfvo.event.register`: Messages on this queue are related to the subscription of an external component interested in receiving events after any life cycle state change.
- `nfvo.event.unregister`: Messages on this queue are related to the unsubscription of an external component not interested in receiving events anymore.

All components have to register themselves to the message bus controlling which messages are allowed to be published and consumed by the component, based on their read/write permissions. The subscriptions to a particular topic can be either specific, allowing a component to subscribe exactly to a particular topic of interest, or generic, allowing a component to subscribe to a set of topics that may be of interest.

5.3 North Domain: User Tools

The different functionalities provided by the NFVO (as described in the previous sections) are exposed to the north domain via REST-based API, on the *Or-Oss* interface. Although this API could be directly consumed by the TSP via common HTTP clients, a set of user tools are provided as part of the framework:

- A dashboard providing a GUI for executing operations directly from the browser, without knowing low level details about the API.
- A CLI providing a console-based approach for interacting with the most commonly used NFVO operations.
- A set of SDKs allowing service orchestration programmability using an Object Oriented Programming (OOP) paradigm.

All those tools make use either of the REST API or the message bus for interacting with the NFVO. Through user tools, the TSP can discover and request NFV resources along with the runtime information that are generated during the service orchestration process.

A definition of the REST APIs exposed by the NFVO towards the north domain is provided in Section B.2.4.

5.4 South Domain: NFVI

VNF deployment scenarios, regardless if multidomain or single-domain, is supposed to happen in different sites within the NFV infrastructure of an operator (i.e., central clouds or edges). Based on the design assumptions presented in Section 4.1.2, those sites are typically interconnected via WAN networking technologies. Those sites correspond to NFVI-PoP according to the ETSI NFV specification. Each individual NFVI-PoP provides compute, storage and networking resources via the VIM entity. Although the VIM interface is a central part of the standardization activities conducted by the ETSI NFV specification group, not all existing VIM technologies are currently aligned to such definition. Hence, VIM implementations may expose their own set of northbound and southbound interfaces and can be specialized in handling a specific type of NFVI resource.

Based on the above considerations, the MANO4X framework, and in particular the NFVO, uses the *Resource Orchestration* functional element abstracting the VIM-specific northbound APIs from the *Service Orchestration* logic requiring NFVI resources for deploying VNFC instances. Therefore, the concept of a VIM driver follows a common *Adapter* architectural design pattern, serving as intermediate entity translating requests coming from the *Resource Orchestrator* defined in compliance with the ETSI NFV specification, to a particular VIM technology. The VIM driver approach allows to continuously extend the set of VIM technologies supported, as well as always upgrading the existing ones to the latest releases available, without requiring changes on the orchestration logic.

The VIM driver is a functional element that is uniquely identified in the system by its `'name.type'`, and it is actually activated upon requests coming from the NFVO having as objective the deployment of certain resources on a particular NFVI-PoP. A driver could serve as intermediate between the NFVO and multiple NFVI-PoPs. The translation from an incoming request to the actual ongoing call is VIM

type specific and can ideally be mapped to any kind of VIM, managing either a private or a public NFVI-PoP. Figure 5.2 shows the VIM driver mechanism.

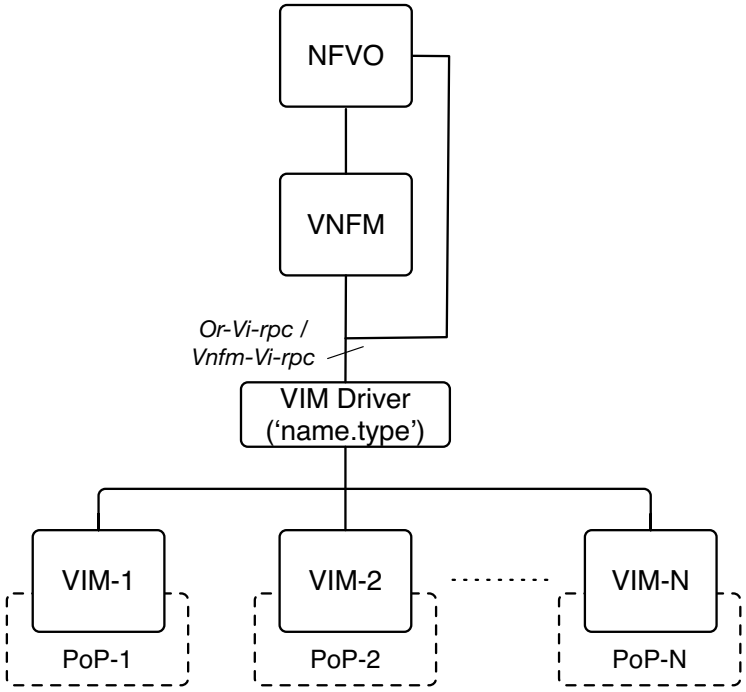

Figure 5.2: VIM Driver Mechanism

As can be noticed, a single VIM driver could be used for interoperating with multiple (N) PoPs of the same type. For scalability reasons, multiple VIM drivers of the same type can also coexist in the same environment.

The VIM driver exposes the *Or-Vi* interface towards the NFVO and VNFM over the message bus using an RPC-based communication mechanism. Each driver creates a queue under the `plugin-exchange` topic having the following syntax: `vim-drivers.plugin-type.plugin-name`, where `plugin-type` corresponds to the VIM type supported by the driver (i.e., in the case of OpenStack, the `plugin-type` may correspond to `'openstack'`) while `plugin-name` may be defined by set by the VIM driver provider to differentiate VIM drivers of the same type.

A definition of the *Or-Vi* interface between NFVO/VNFM and VIM driver is provided in Section B.2.5.

5.4.1 Monitoring integration

Monitoring of the virtualized resources is also a task belonging to the NFVI domain. Typically this is obtained installing a monitoring system based on a server-agent

model (i.e., Zabbix[1], Nagios[2], etc.), or making use of the ones already provided by the NFVI solution (i.e., Telemetry in OpenStack[3]). Those monitoring systems usually collect information from both, the physical and virtualized resources, and could be further customized to retrieve also application-level metrics.

The approach utilized for allowing the different components of the MANO4X framework retrieve monitoring data, is similar to the adapter concept utilized for integrating heterogeneous VIM technologies. A *monitoring plugin* acts as an intermediate entity between the monitoring system and any consumer interested in retrieving monitoring data. It allows any component connected to the message bus to perform operations towards the monitoring system, consuming an interface compliant with the ETSI NFV IFA 006 specification[97]. In particular this interface supports managing performance jobs and alarms.

A definition of the *Vi-Mon* interface between NFVO/VNFM and VIM driver is provided in Section B.2.6.

5.5 West Domain: VNFM

Handling the current state of a VNF is a responsibility solely reserved for its VNFM. Each VNF is associated with one VNFM. One of the major requirements for a MANO framework is to be able to interoperate with multiple VNFMs. This means that the NFVO expects the VNFM to provide a common interface for being orchestrated. The VNFM should provide an interface to the NFVO for receiving requests triggering the execution of each specific life cycle event.

However, considering the large number of existing management and configuration solutions, not all of them expose an interface compliant with the *Or-Vnfm* standard definition. Therefore, the MANO4X framework provides a mechanism to integrate both kinds of VNFMs: Those compliant with the *Or-Vnfm* interface (like the Generic VNFM described in the next section), directly integrated with the NFVO, and those having their own interfaces for being managed, integrated with the NFVO using the VNFM-adapter functional element. Figure 5.3 shows the architectural model of such definition.

In Figure 5.3 there are three different kinds of VNFMs:

- VNFM-A: External VNFM exposing a REST interface compliant with the ETSI NFV definition of the *Or-Vnfm* reference point. It interacts with the NFVO using the *Or-vnfm-rest* APIs provided.

- VNFM-B: MANO4X-compliant VNFM interacting with the NFVO over the message bus. This VNFM uses the *Or-vnfm-amqp* without any intermediate component. An example of such VNFM is the Generic VNFM.

[1] http://www.zabbix.com/
[2] https://www.nagios.org
[3] https://wiki.openstack.org/wiki/Telemetry

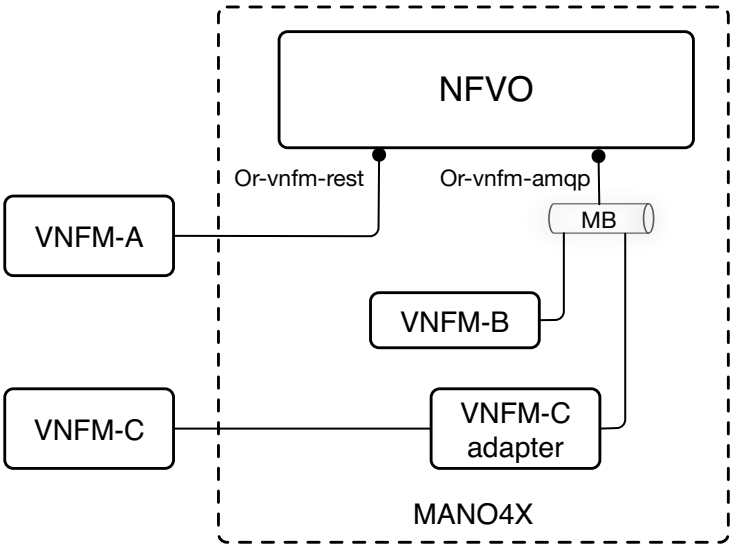

Figure 5.3: VNFM Architectural Models

- VNFM-C: External VNFM integrated within the MANO4X framework using an adapter. This adapter translates incoming requests over the message bus into specific calls to the external VNFM. An example of such VNFM is the Juju VNFM.

The translation from the incoming function call into the actual outgoing management procedure is VNF-specific and can be mapped, for example, to cloud-config[4], Puppet[5] or Chef[6] recipes deployments, or common bash commands executed via Secure Shell (SSH). *Ve-Vnfm-vnf* (or *Ve-Vnfm-em*) reference point is used by the manager to access the VNFC hosting the actual software artifacts (or to contact the EMS), and for executing management operations on it.

A definition of the *Or-Vnfm* interface between NFVO and the VNFM is provided in Section B.2.7.

5.6 East Domain: OSS

Deployed network services are in continuous evolution and are being affected by changing external or internal conditions, defined also as events. Such events could be external in the sense that they are generated by user actions, or internal because they are generated by other functional elements (i.e., OSS/BSS runtime decisions). Loop-based self-organization mechanisms should guarantee the network service operations throughout the complete life cycle.

[4]http://cloudinit.readthedocs.io/en/latest/topics/examples.html
[5]https://puppet.com/
[6]https://www.chef.io/chef/

While analyzing possible mechanisms for realizing the integration of OSS components in the MANO4X it has been considered that i) OSSs should be decoupled from the NFVO, so that they could be optionally used only when needed in a particular scenario or particular VNF, ii) should not follow any implementation schema, so that already existing OSSs could easily be integrated within the framework.

In the context of this research work, two major categories of OSSs have been identified from the perspective of the interactions they are having with the NFVO:

1. *uni-directional*: Comprising systems that, in order to realize the desired state defined in the policies under their control, do not require any interaction with the NFVO, thus do not modify the overall state of the network services and VNFs.

2. *bi-directional*: Comprising systems that, in order to realize the desired state defined in the policies under their control, need to trigger state transitions via the *Or-Oss*, impacting directly on the state of the NSR (e.g., scaling in and out, or switch to standby actions). Typically, those systems apply a control loop[7] for modifying the NSR in order to reach the desired state.

OSSs of both categories need to be aware of the actual NSR instantiated, particularly they need to retrieve either static information set by the TSP (i.e., policies) or runtime information (i.e., IPs, hostnames, etc.) of a particular network service. Hence, the main approach is to provide a mechanism for i) keeping track of existing external OSSs components using a registration process so that they get notified whenever an NSR is modified, ii) allowing those external components to interact with the NFVO in order to apply any kind of modification on existing network services. In some cases, OSSs may expose an independent interface towards the users, for providing more detailed information about the specific operations under their control.

In the next subsections the different OSSs functional elements designed for fulfilling the FCAPS model requirements along the network service life cycle are presented. All the OSS solutions presented use the same approach in order to register and get notified by the NFVO whenever new NSRs are instantiated.

5.6.1 Fault Management System (FMS)

The FMS is one of the fundamental functions in a management framework, as per definition in the FCAPS model (as defined in Section 2.1.1.1), especially because it has strong impacts on the QoE and QoS perceived by end-users. The FMS is part of the *bi-directional* category. With the introduction of virtualization technologies, fault management becomes even more complex, considering that network functions running on VMs are independent from the underlying infrastructure, thus, faults at the physical infrastructure level may consequently cause several faults at the upper

[7] Although control loop is a term typically applied in the context of industrial control systems, it is employed here for defining the overall similar objective addressed by those systems

layers. In NFV, at least three different layers can be identified: Physical layer, virtualization layer, and VNF layer.

Moreover, the multi-layer fault dependency can generate major issues within an NFV environment, especially because of a potential storm of alarm notifications that may happen whenever a fault occurs. Due to a single fault at the physical layer, several fault notifications may be fired. Therefore, a mechanism is needed for properly correlating notifications and adequately propagating the message to the decision maker. Moreover, in order to take the appropriate decision about the fault recovery action to be executed, a sufficient set of information regarding the fault that occurred is required .

The FMS has the overall objective of identifying the root cause of a fault and to take corrective actions aiming at containing the fault and recovering the status of the network service. It is composed of three main functional elements: Monitoring Manager (MM), Fault Correlator (FC), High Availability Manager (HAM).

In the following subsections a short overview is given about the functionality provided by those three functional elements.

5.6.1.1 Monitoring Manager (MM)

The Monitoring Manager communicates with the monitoring system via the monitoring plugin (as presented in Section 5.4.1 through the *Or-Vi* interface), creating performance management jobs and related thresholds, based on the fault management policies defined in the descriptor of the instantiated network service.

When the criteria defined in the policy is met, the monitoring plugin fires an alarm. Such alarms can be of either type *VNF* or *Virtualized Resource* depending on the failure value defined. The *VNF* type relates to the VNF service execution, for instance whenever a particular process stops executing. The *Virtualized Resource* type relates to the virtualized resources, for instance the virtual machine is not reachable anymore or the virtual link is dropping packets.

5.6.1.2 Fault Correlator (FC)

The Fault Correlator is in charge of correlating alarms and identifying the root cause of the issue so that the appropriate recovery action can be executed through the High Availability Manager. The Fault Correlator is policy-based, allowing the user to define specific rules on each individual VNF. The Fault Correlator receives as input alarms from the Monitoring Manager and the NSR from the NFVO. The correlation consists in associating an alarm with the relative VNFC. The Root Cause Analysis (RCA) consists of finding the alarm causing the failure and executing an action through the High Availability Manager. The RCA is necessary since the Fault Correlator could receive several alarms at the same time.

5.6.1.3 High Availability Manager (HAM)

The High Availability Manager is in charge of maintaining the redundancy scheme of the Virtual Deployment Units (VDUs), executing recovery actions. The redundancy scheme is specified in the VDU descriptor and it can be *K:N*, meaning that for *N* VNF active components of such VDU there should be *K* VNF standby components. The standby VNFCs protect the active components against failures. The recovery actions provided are the *healing* and the *switch to standby*. The execution of those actions is triggered by the Fault Correlator. The *healing* action allows to call the heal life cycle event of the VNFM providing the root cause of the fault. The *switch to standby* action activates a VNFC that was in standby mode. This operation typically requires a notification of change in configuration to the dependent VNFs. Subsequently, the High Availability Manager removes the failed VNFC and creates a new standby VNFC reestablishing the desired redundancy scheme.

5.6.1.4 Fault Management Policy

The FMS receives as input from the NFVO the instantiated NSR containing monitoring parameters and fault management policies. The monitoring parameters are used to create performance jobs while the fault management policies are used to create thresholds. Listing 5.1 provides an example of fault management policy in a JavaScript Object Notation (JSON) format.

Listing 5.1: Example of a fault management policy contained in the VNFD

```
{
   "name":"policy name",
   "VNF_failure":"false",
   "criteria": [
     {
         "parameter_ref":"monitoring_paramenter",
         "function":"last()",
         "vnfc_selector":"at_least_one",
         "comparison_operator":"=",
         "threshold":1,
     }],
    "severity":"severity",
    "period":30
}
```

The parameters are explained as follows:

- `name`: name of the fault management policy.
- `VNF_failure`: it defines if it is a VNF failure or not.
- `criteria`: this is the criteria of the fault management policy. Basically it specifies the condition that should be met for firing alarms.

- `parameter_ref`: KPI subject of this criteria.
- `function`: function to be applied to the parameter_ref. For example last() is the last available value.
- `vnfc_selector`: specifies scope of the criteria. Possible values: at_least_one or all.
- `comparison_operator`: the comparison_operator defines how to compare the final measurement result with the threshold. Possible values are: =, >, >=, <, <=, !=.
- `threshold`: the threshold defines the value that is compared with the final measurement.
- `severity`: the severity of the alarm produced when the criteria is met. Possible values: INDETERMINATE, WARNING, MINOR, MAJOR, CRITICAL.
- `period`: this is the criteria updating time. For example, a value of 5 indicates that every 5s the criteria is checked.

5.6.2 Autoscaling Engine System (AES)

The AES provides a policy-based engine for automatically scale in and out VNFs. The AES is part of the *bi-directional* category. Before describing in details the proposed solution, it is important to analyze the autoscaling problem starting with some definitions[8].

The scaling problem can be decomposed into three main functions (or phases): *"detecting the need to scale"*, *"determining the scaling actions"*, and *"executing the scaling actions"*[145].

"Detecting the need to scale" is typically regarded from the resources' and capacities' points of view. Detection methods have already been presented in Section 4.2.3.2 as part of the research work conducted by the author during the initial phase of design. Usually, in threshold-based policies, this operation is the result of a specific measurement being crossed. Several scaling algorithms may be designed and implemented in order to optimize resource utilization. Furthermore, this module may be extended for including concepts coming from the usage of machine learning techniques as also presented in the Section 8.3. Hence, autoscaling policies may become much more complex, and the challenge is to aggregate multiple KPIs ensuring that the required QoS is maintained during scaling operations.

"Determining the scaling actions" is usually realized analyzing the type of KPIs that are being deteriorated in order to counteract and recover the normal situation. Typical scaling actions can be: Adding (scale-out), removing (scale-in), increasing

[8]Concepts and ideas presented as part of the AES are based on the research conducted by the author around autoscaling concepts during the initial and intermediate phases[129][128][13][14]

(scale-up) or decreasing (scale-down) number of VNFCs. Those scaling actions are associated with scaling conditions contained in the autoscaling policies.

"Executing the scaling actions" is the last step that has to be executed. Any action taken implies changes in the NS configuration, therefore, it is required that those modifications are executed by the NFVO that is the component having the overall view of the NS.

Therefore, the AES follows this logical split comprising three major functional elements for detecting the need of scaling (*Detector*), making decisions (*Decision-maker*) and finally executing the required mitigation actions (*Executor*). This modular approach of the AES itself provides the opportunity to replace a functional element with another one fulfilling the requirements of a particular scenario[145].

The following subsections provide an overview of each of the proposed functional elements, focusing on their main functionality provided as well as the interactions with other functional elements of the MANO4X framework.

5.6.2.1 Detector

The detector is in charge of monitoring VNFCs and generating alarms in case conditions defined in the autoscaling policies are met. These conditions are defined as alarms included in the policies included in the NSR received from the NFVO whenever a new NSR reaches the INSTANTIATE_FINISH life cycle event. The detector retrieves metrics related to the virtualized resources instantiated on the NFVI used by the VNFCs via the monitoring plugin over the *Vi-Mon* interface. Therefore, the detector communicates with the monitoring layer to get measurements filtering them based on host names, metrics and the period of time under consideration. Based on those metrics, the detector collects and processes data to detect the need to scale. These measurement results are aggregated and processed, and finally evaluated against a threshold-crossing function as defined in the alarm. Furthermore, it operates a threshold-checking function where previously calculated final measurement results are compared with the threshold defined in the policy. If the threshold is crossed, the alarm is handled internally as fired. Multiple alarms can be combined in a weighted and prioritized way. Finally, if a predefined number of alarms are fired, the detector sends the high-level alarm to the next functional element of the pipeline, the decision-maker.

5.6.2.2 Decision-Maker

The decision-maker is responsible for making decisions based on the types of alarms received from the detector. The decision-maker checks if the execution of the actions defined in the policy is feasible or not, i.e., scaling-in and scaling-out is only executable in case the minimum/maximum limit has not been reached yet. For the feasibility check it may request the NFVO to grant operations and other information that are essential for the decision-making process.

Once decisions are made, the decision-maker submits them to the next functional element of the pipeline, the executor.

5.6.2.3 Executor

Based on the decisions made by the decision-maker, the executor requests either the NFVO or, in case granting mechanism is enabled, the VIM directly to scale-in/scale-out or allocate/release resources. Depending on the approach, the AES communicates with the corresponding VNFM of the VNF using these two different mechanisms:

- In case of an NFVO-centric approach the AES may communicate indirectly with the VNFM by calling scaling function through the NFVO. The NFVO forwards these requests to the corresponding VNFM following the standard procedure for scaling in/out VNFs.
- In the VNFM-centric approach the AES is closely related or even directly integrated into the VNFM and can call respective methods of the VNFM based on the specific needs. Additionally, if the AES is directly integrated, it can act as a part of the VNFM by processing customized tasks[9].

After all the actions are executed, the executor blocks, for a certain period of time defined in the *cooldown* parameter of the autoscaling policy, further incoming scaling requests for the VNF already in scaling state. The reason of introducing the cooldown period is to avoid fluctuations where the load redistribution may cause the triggering of other scaling actions.

5.6.2.4 Autoscaling Policy

The need of scaling and corresponding actions are defined as part of the autoscaling policy contained in the corresponding VNFD. Listing 5.2 shows an example of such policy in JSON format.

Listing 5.2: Example of an autoscaling policy

```
1  {
2     "name":"scale-out",
3     "threshold":100,
4     "period":30,
5     "cooldown":60,
6     "mode":"REACTIVE",
7     "type":"VOTED",
8     "alarms": [
9       {
10          "metric":"item",
11          "statistic":"avg",
12          "comparisonOperator":"<=",
13          "threshold":40,
```

[9]Such approach was utilized in the context of the NUBOMEDIA project and will further be presented in Section 7.1.0.3 – ICT NUBOMEDIA Project

```
14          "weight":1
15      }
16    ],
17    "actions": [
18      {
19          "type":"SCALE_OUT",
20          "value":"2"
21      }
22    ]
23  }
```

The autoscaling policy contains the following parameters:

- `name`: name of the policy.
- `threshold`: a value in percentage indicating how many alarms have to be raised before firing the notification to the Decision-Maker module.
- `period`: the time between different alarm checks.
- `cooldown`: the minimum amount of time between the execution of different scaling actions towards the same VNF. Further scaling actions requested during the cooldown period are rejected.
- `mode`: it defines the way alarms and conditions should be evaluated, like: REACTIVE, PROACTIVE, PREDICTIVE.
- `type`: it defines the way alarms should be processed. Three different types are available: VOTED, WEIGHTED, SIMPLE.
- `alarms`: it defines a list of alarms belonging to the same policy. Each alarm is composed of the following:
  - `metric`: the name of the metric that is considered when checking the alarm (e.g., cpu idle time, memory consumption, network traffic, etc). This metric must be available through the monitoring system.
  - `statistic`: it defines the way the final measurement should be calculated. Possible values are: avg, min, max, sum, count.
  - `comparisonOperator`: it defines how to compare the final measurement result with the threshold. Possible values are: =, >, >=, <, <=, !=.
  - `threshold`: it defines the value that is compared with the final measurement.
  - `weight`: it defines the weight of the alarm and it is used when combining all the alarms to a final indicator that says how many alarms are fired. This way prioritized alarms can be handled with different weights.
- `actions`: a list defining the actions that should be executed once the conditions (alarms) are met.

Each action is composed of a type and a value. The type defines the type of the action to be executed. Possible types of actions are:

- SCALE_OUT: scaling-out a specific number of instances
- SCALE_IN: scaling-in a specific number of instances
- SCALE_TO: scaling-out or scaling-in to a specific number of instances
- SCALE_TO_FLAVOR: scaling-out or scaling-in to specific deployment-flavor

While the value is related to the type of action. SCALE_OUT and SCALE_IN expects a value that defines how many instances should be scaled-out or scaled-in, SCALE_TO expects a number to what the instances should be scaled and SCALE_TO_FLAVOR expects a reference to the which deployment flavor the VNFR should be scaled.

5.6.3 Network Slicing Engine (NSE)

Following the overview given in Section 2.4, network slicing represents a novel concept addressing the need of a logical separation of the physical networking resources into *"slices"* where each slice can offer different capabilities to the end-users. Multi-tenancy can be already considered as one of the features provided by the MANO4X framework for supporting network slicing, however, it is not sufficient as it does not guarantee any specific level of QoS whenever multiple NSs are deployed on the same infrastructure. The NSE is a component that can be plugged into the MANO4X framework for increasing network isolation between NSs sharing the same infrastructure[142]. Therefore, the NFVO handles the whole life cycle of the network service (or slice) while the management of the specific networking requirements of a particular service is delegated to the NSE. For this reason, the NSE is part of the *uni-directional* category.

Basically the NSE extracts the requirements contained in the NSR in terms of required networking capabilities (e.g., bandwidth), and forces them onto the NFVI either using functionalities provided directly by the VIM[10] or via the Connectivity Manager Agent (CMA), a component making use of SDN technologies for steering traffic inside virtualized networks. The CMA provides a simplified API for associating a specific level of QoS required between VNFs composing the NS[11].

The NSE subscribes to the NFVO for receiving the NSR at the end of the INSTANTIATE_FINISH life cycle event. In particular, the NSE subscribes to three events:

- INSTANTIATE_FINISH: published once any NSR is instantiated successfully.

[10] OpenStack neutron supports allocating maximum bandwidth to each individual VM via remote APIs: https://docs.openstack.org/mitaka/networking-guide/config-qos.html

[11] Concepts and ideas presented here as part of the NSE have been based on previous research done during the intermediate phase with Nippon Telegraph and Telephone (NTT)[135]

- SCALED: published whenever a scale in or out operation is executed.
- ERROR: published when the NSR goes into error state.

The event payload, containing the NSR, is then parsed by the NSE to retrieve information about the placement of each VNFC instance (in particular the NFVI-PoP where the VNFC has been deployed), in order to enforce the required network capacities, and the NSE policy (defined in the following subsection) providing QoS requirements for each virtual link. The NSE parses the record and gets an overall vision of logical resource location, checks the slice requirements and sees whether they are feasible with the available network resources.

5.6.3.1 NSE Policy

The NSE includes a policy model that is inspired by the DiffServ[160] mechanism for classifying and managing network traffic. The policies are divided into classes, and each class has its specific slice parameters. These supported parameters are the maximum and minimum bandwidth rates. The model includes by default three classes of slices (*GOLD*, *SILVER* and *BRONZE*), but it can be extended to the bare number of parameters.

The NSE policy needs to be defined at the *vnfd:virtual_link* level inside the NSD, using the `qos` field to set the requirements. Listing 5.3 shows an example of the NSE policy to be included in the virtual link definition in a JSON format.

Listing 5.3: Example of the NSE policy to be included in the virtual link definition

```
1      "virtual_link":[
2         {
3           "name":"private",
4           "qos":[
5             "minimum_bandwidth:GOLD"
6           ]
7         }
8       ],
```

In particular, the `qos` parameter is a list of parameters defining the required network capacity. For instance, the *minimum_bandwidth* is a parameter that indicates the class of quality the particular VNF requires.

5.6.4 Service Function Chain Orchestrator (SFCO)

Chaining NFs is a concept typically applied to mobile core network architectures. In Section 2.4.5.1 an overview was presented about the ongoing standardization work around the concept of dynamic SFC. The SFCO is an additional component, part of the MANO4X framework, capable of instantiating SFCs on the data paths, and supporting the heterogeneity of network requirements such as the SFC forwarding approaches (e.g., Network Service Header (NSH)[161] or MPLS[162])[140][147][134].

The SFCO can update the SFC data paths dynamically and manage the full life cycle ofa SFC including several vertical scenarios (e.g., faulty, overloaded, or hijacked SFs). For this reason, the SFCO belongs to the *uni-directional* category.

Following the approaches taken in other OSSs, one of the main functionalities required by the SFCO is the capability of subscribing to the NFVO for receiving updates whenever a new NS is created or deleted, as well as a particular VNFC instance is healed or scaled. An internal *repository* is used by the SFCO to store SFs, SFCs, Service Function Paths (SFPs) and SFC Classifiers objects so that they can be retrieved anytime by the other modules. The SFCO relies on a southbound interface for interacting with the underneath SDN Controllers/SFC Agents. The Deployment module uses this southbound interface to provide CRUD operations of SFC objects via the REST APIs of the SDN Controller/SFC Agent supporting SFC data plane implementations.

Based on the SFC policy defined in the VNFFG, the SFC driver manages and deploys SFCs by interoperating with the SDN Controller/SFC Agent and the NFVO. It executes CRUD operations on top of the SDN Controller for managing SFCs and SFC rules. Furthermore, it updates the SFP via the SDN Controller in case of faults on one of the SF instances or scaling out/in of an SF. The Driver abstracts the specific SDN Controller technology used. The Path Creation functional element is responsible for creating the path in deployment phase and in runtime phase. Several algorithms can be used: Random, Round Robin Load balancing, Shortest Path based on Dijkstra algorithm and our previously proposed scheme, Trade-off delay and load [134].

Runtime, the monitoring agent module, is responsible for collecting the (traffic/CPU) load statistics for each SF instance via the monitoring plugin and statistics from the SDN controller (OpenFlow Plugin), via a southbound SDN controller interface, about the traffic load per each SFP updating the repository with these live statistics.

5.6.4.1 SFC Policy

In order to instantiate SFCs, an SFC policy has to be defined as part of the VNFFGD. Such policy provides a definition of the chain as well as network requirements (e.g., QoS level, classification rules). The VNFFGD is defined at the level of the NSD (as specified in [28]). SFs are mapped into different VNFDs, and chains are used to define different Network Forwarding Paths (NFPs) between them. Listing 5.4 includes an example of the VNFFGD in a JSON format including the NFP used by the SFC Classifier to create the chain.

Listing 5.4: Example of the SFC policy as part of the VNFFG to be included in the NSD definition

```
1   "vnffgd":[
2      {
3         "symmetrical":false,
4         "dependent_virtual_link":[
```

```
5             {
6                 "name":"sfc-network"
7             }
8         ],
9         "network_forwarding_path":[
10            {
11                "connection":{
12                    "0":"fw-sf",
13                    "1":"http-sf"
14                },
15                "policy":{
16                    "acl_matching_criteria":{
17                        "source_port":0,
18                        "destination_port":5001,
19                        "protocol":17,
20                        "source_ip":"172.0.0.23/32",
21                        "destination_ip":"172.0.0.33/32"
22                    },
23                    "qos_level": "GOLD"
24                }
25            }
26        ]
27    }
28 ]
```

The VNFFGD contains the following parameters:

- `symmetrical`: defines whether the NFP is uni- or bi-directional.
- `dependent_virtual_link`: references the virtual_link that should be used while creating the NFP
- `network_forwarding_path`: defines the NFP that provides the following parameters:
  - `connection`: defines the ordered sequence of SFs in the chain.
  - `policy`: defines the policy to be used by this NFP in terms of:
    - `acl_matching_criteria`: Access Control List (ACL) rule defined using source and destination IP/ports, as well as protocol used.
    - `qos_level`: specifying the QoS level class required for this particular NFP.

5.7 MANO4X High-level Procedures

After having presented functional elements and interfaces composing the MANO4X architectural framework, this section outlines the main procedures driving the life cycle management of a software-based network service from an end-to-end perspective. Figure 5.4 depicts the network service life cycle.

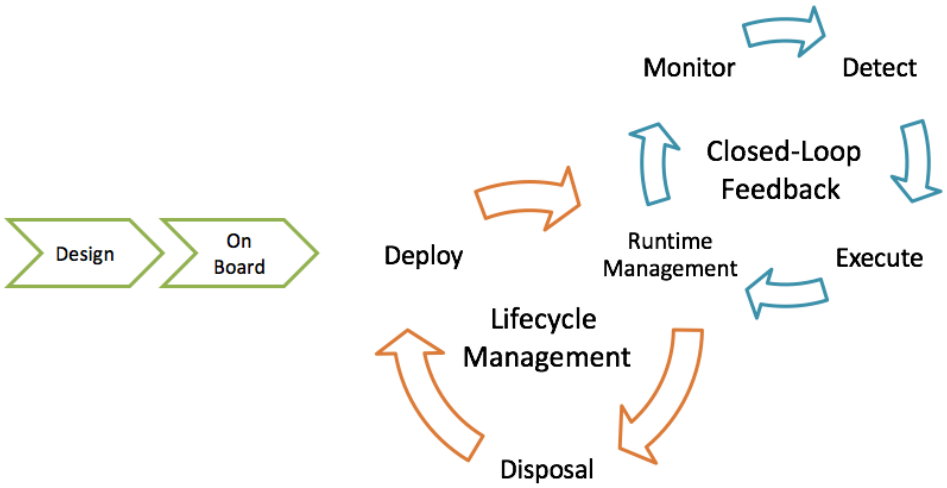

Figure 5.4: Network Service Life Cycle

Before providing a definition of the different phases and procedures executed, a brief overview is given here about the entire process.

Let us consider as a starting point the MANO4X comprising only functional elements of the central and north domain: the NFVO, the message bus and the dashboard. The first two phases (depicted in green in Figure 5.4) are the design and onboarding phases. During these phases, the TSP activates the required functional elements to satisfy a particular scenario, sets up the NFVI provided by the NFVIP and registers it as a PoP to the NFVO. Afterwards the TSP on-boards on the catalog the VNFPs provided by VNFP, and designs the end-to-end network service, based on the required features to provide to its end customers, composing one or more VNFs together. As a result of the design phase the TSP creates the NSD and on-boards it on the catalog.

The second phase is represented by the actual life cycle management of the network service. The TSP selects the NSD from the catalog and triggers its deployment. During this step the NFVO executes several operations involving the VNFMs responsible for the VNFs to be deployed, and the VIMs responsible for the NFVI PoP where the VNFs should be deployed. This process usually requires the deployment of virtualized resources on the NFVI PoP and the configuration of the software artifacts comprising the VNF. At the end of the deployment phase, in case there are no errors, the network service becomes active and a record is stored in the catalog of the NFVO.

The following phase is represented by the runtime management. Typically, this phase consists of different kinds of operations executed by OSSs for maintaining the desired level of reliability following the FCAPS model. Typically, OSSs have a control loop consisting of gathering monitoring data from a monitoring system, making a decision based on policies provided by the TSP and executing actions (in

case of *bi-directional* OSS, see Section 5.6) which may modify the overall configuration of the network service. Those actions are typically requests generated towards the NFVO that is the only component maintaining the overall state of the network service throughout the complete life cycle.

At any time, the TSP could decide to trigger the disposal of the network service because it is not needed anymore. For the sake of clarity, a disposal operation consists only of releasing all the resources that were allocated to a particular record. Therefore, the TSP could decide to again execute a new deployment based on the static descriptors available in the catalog without having to reexecute the design and onboarding phase.

5.7.1 The Virtualized IMS (vIMS) as the Reference Use Case

As already presented in Chapter 2 the ETSI NFV specification presented a set of use cases driving the standardization activities. Although the objective of this thesis is to provide a framework supporting any kinds of software-based NFs, the vIMS use case was selected as the reference use case for exemplifying the whole life cycle process.

Following the definitions given in Section 2.1.2, it is important to clarify that although the 3GPP IMS architecture has been defined in terms of NFs and interfaces, a design, and consequent implementation of such architecture as software, may result in multiple deployment options. For the vIMS use case, the author proposes three different deployment options [121]:

- *vIMS* – an architectural option in which each 3GPP IMS functional entity is mapped 1:1 with a VNF
- *Split-IMS* – an architectural option in which each 3GPP IMS component is split into multiple sub-components in order to be deployed on top of multiple hosts and containers
- *Merge-IMS* – an architectural option in which components are merged into less components enabling a low delay and functional reduced processing for external requests by a single VNF

In this dissertation, the *vIMS* was considered as reference one, as it emphasizes important characteristics of software-based network services. Figure 5.5 depicts the functional architecture of the proposed vIMS deployment option as presented in [121].

In order to better understand the orchestration process of such complex network service it is important to consider the following aspects:

- Individual VNFs can be composed by multiple VNFC instances. Assuming that the selected NFVI site provides standard virtualization technologies, each VNFC corresponds to a virtual machine or container hosting the software components of a particular NF.

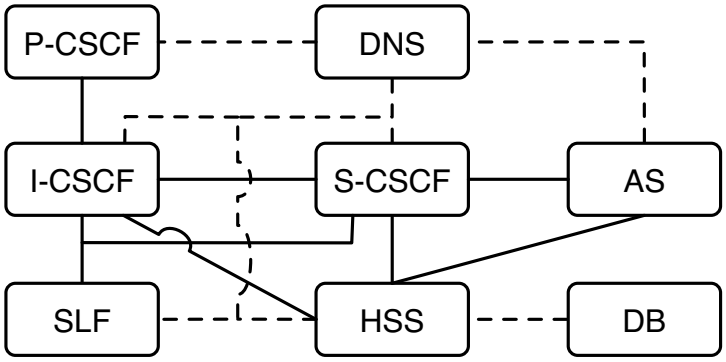

Figure 5.5: vIMS reference use case

- Each line in the previous diagram represents a dependency (sometimes also defined as relationship). A dependency between two VNFs may be needed for exchanging information like IP, ports, configuration parameters, etc, that are needed for creating the end-to-end network service.

- A VNF may scale horizontally increasing/decreasing the number of VNFCs. Each time a new VNFC is added/removed, all the VNFCs of the dependent VNF must be informed so that they can be reconfigured accordingly.

5.7.2 Initial Phase: Design and Onboarding

As briefly mentioned in Section 5.7, the design and onboarding phase consists of preparing the NFVI and designing the network service. The first part involves the preparation of the hardware resources to be used as part of the NFVI: The TSP could decide to build up a private NFVI installing the software components himself, or make use of the NFVI provided by a third-party NFVIP (like public clouds).

The following step is represented by the activation of the MANO4X functional elements required for the specific life cycle of the selected network service. Following the SOA and microservices architectural model, each individual component (being a VIM driver, a VNFM, or an external OSS) must register itself so that the NFVO becomes aware of their availability. Assuming that the central domain functional elements (NFVO and message bus) are already activated, the TSP needs to activate VIM drivers, VNFMs, and OSS components that will be needed during the life cycle and runtime management phases. The registration process differs depending on the type of components. In the following, a brief overview is given about the registration mechanisms designed within the scope of this dissertation. The registration process uses standard pub/sub mechanism over the message bus on a dedicated registration topic.

5.7.2.1 VIM Drivers and Monitoring Plugins

Figure 5.6 shows how a VIM driver is registered and activated within the MANO4X framework. Although the process described here refers to the VIM driver, the same concepts and mechanisms apply in case of a monitoring plugin.

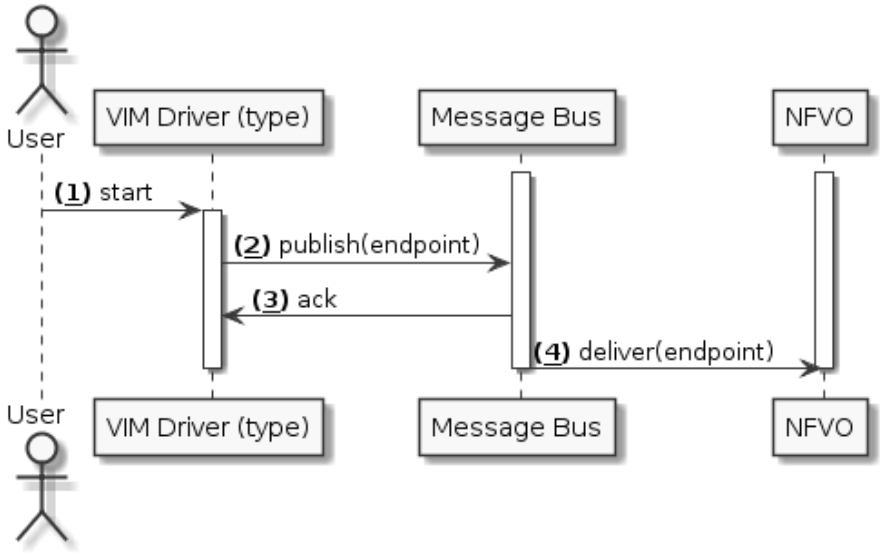

Figure 5.6: VIM Driver Registration Procedure

The first step is represented by the TSP selecting the VIM driver needed in order to deploy VNFs on a certain NFVI PoP. Upon its activation (1) the VIM driver generates an `endpoint` object and publishes (2) it on the message bus registration topic consumed by the NFVO. The `endpoint` object contains the following information:

- `type`: representing the VIM type supported by this driver (i.e., OpenStack, Amazon EC2, etc.).
- `name`: representing a unique identifier for differentiating multiple instances of a driver of the same type (i.e., multiple VIM drivers of type `'openstack'` may receive two different names for differentiating two different implementations).

The NFVO receives (4) the registration requests via the registration topic, and creates a new entry in the database for that VIM driver. Furthermore, the NFVO also keeps information about the state of the driver. A driver (same for a monitoring plugin), can have two different states of type boolean:

- *enabled*: it is a parameter that can be set by the TSP for deciding whether the driver is enabled or not.
- *active*: it is a parameter set by the NFVO itself, constantly checking via a heartbeat mechanism whether the driver is active or not.

5.7.2.2 VNFM Adapters

Considering the heterogeneity of VNFs to be supported by the MANO4X framework, it is expected that different kinds of VNFMs will exist. Thus, it is required to maintain an inventory of those entities, particularly for defining which ones could be supported by a particular VNF. Therefore, the MANO4X framework provides a registration mechanism that should be used by the VNFM to announce itself to the NFVO upon activation. The NFVO requires that the VNFM provides its own *type*, and the *endpoint* where it could be reached at runtime for triggering life cycle management operations. The registration process is depicted in Figure 5.7.

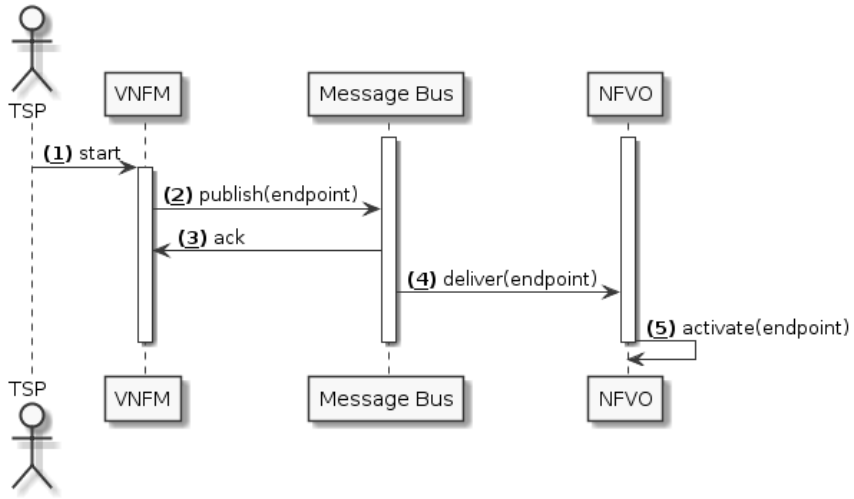

Figure 5.7: VNFM Registration Process

The steps executed for registering a new VNFM are similar to the ones already presented in the previous subsection Section 5.7.2.1 about the VIM driver and monitoring plugin. The approach is the same, however, the content of the `type` parameter represents the VNFM type, and it is used within the MANO4X framework for uniquely identifying the VNFM capabilities provided. In practice, this parameter is referred to the VNFD `endpoint` property so that the NFVO knows about the VNFM responsible for the life cycle management of that particular VNF.

5.7.2.3 Operations Support System

In order to take part on the network service life cycle, in most of the cases during the runtime phase, OSSs shall subscribe to receive notifications about any life cycle events they are interested in. The subscription process usually involves the OSS entity registering an *event endpoint* to the NFVO via its REST APIs.

The first step is represented by the TSP configuring and activating (1) the OSS component needed. Upon activation (2), the OSS entity sends (3) a subscription message to the NFVO including the life cycle event type it is interested in receiving

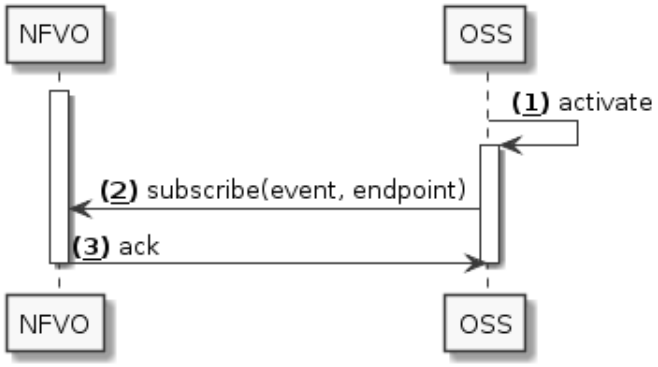

Figure 5.8: OSS Event Endpoint Registration Process

notifications from, and the endpoint that can be used as callback by the NFVO to notify about the event. The NFVO registers (4) the event in its own repository, and acknowledges (5) the OSS about the successful registration of the subscription.

5.7.2.4 PoP Registration

Once all the required components are activated, the TSP should provide details about the NFVI. Considering the multisite requirement, in order to deploy a VNF at a particular site, the NFVO should keep a list of available NFVI PoPs. This is obtained via a registration mechanism where the TSP provides to the NFVO details about the available PoPs. The registration process is depicted in Figure 5.9.

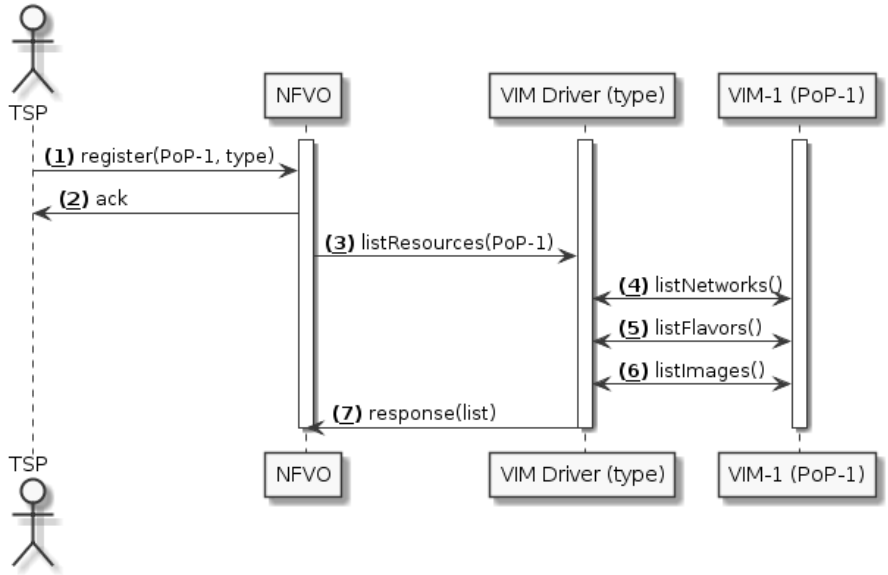

Figure 5.9: PoP Registration Process

The first step is represented by the TSP sending (1) to the NFVO a registration

request of a new PoP of a certain `type`. Listing 5.5 shows an example of the JSON descriptor that should be sent to the NFVO for registering a new PoP.

Listing 5.5: Example of a PoP JSON file

```
{
  "name":"OpenStack-PoP",
  "authUrl":"http://192.168.85.54:5000/v2.0",
  "tenant":"admin",
  "username":"admin",
  "password":"password",
  "keyPair":"key-pair-name",
  "securityGroups": [
    "default"
  ],
  "type":"openstack",
  "location":{
    "name":"Berlin",
    "latitude":"52.525876",
    "longitude":"13.314400"
  }
}
```

In particular, the NFVO expects information like name, credentials, URL endpoints, default security groups, and locations of the PoP being registered, as well as the `type` of PoP that is being registered. As already mentioned, the `type` represents the unique identifier mapping the registered PoP to the responsible VIM driver to use for interacting with it.

After acknowledging (2) the received request, the NFVO starts the registration process requesting (3) the responsible driver (selected by `name.type`) the list of resources available in that particular PoP. The driver requests the list of networks, flavors, and images already available on the PoP calling respectively the *listNetworks* (4), *listFlavors* (5), and *listImages* (6) APIs exposed by the actual VIM. The list of available resources is stored in the NFVO catalog as a PoP object so that it can be retrieved during the onboarding phase for validating the content passed inside VNFDs and NSDs.

5.7.2.5 VNFP and NSD Design and Onboarding

After configuring the required functional elements and registering available PoPs in the multisite NFVI, the MANO4X framework is ready for the onboarding phase. In order to manage a VNF through the MANO4X framework, it is required to build a VNFP. A VNFP is an archive containing all the information required for instantiating and managing a VNF. Listing 5.6 shows an example of a VNFP.

Listing 5.6: Internal structure of a VNFP

```
$ tree scscf-vnf-package/
.
|__ Metadata.yaml
|__ scripts
|   |__bind9_relation_joined.sh
|   |__fhoss_relation_joined.sh
...
|   |__instantiate.sh
|   |__var_scscf.xml
|__ vnfd.json
```

Particularly, it includes a *vnfd.json* file representing the VNFD, the *scripts* folder containing executable files used runtime by the VNFM for managing the VNF life cycle, a *Metadata* file using the YAML format for defining the essential properties of the VNF, and optionally the *image disk* to be used while instantiating it. After onboarding the VNFP, the NFVO stores its content into the different catalogs.

After uploading the VNFPs into the catalog, the TSP can start composing the network service, describing it using the NSD. Figure 5.10 provides an overview of the major elements composing the NSD in a JSON format.

5.7.3 Network Service Life Cycle Management

One of the fundamental functions of the NFVO is to manage and orchestrate the life cycle of network services as composition of one or more VNFs. In most of the cases, network services are composed of multiple dependent VNFs, thus, resolving dependencies among them represent the most crucial functionality that the NFVO has to fulfill as part of the network service orchestration life cycle. In particular, the overall network service life cycle depends upon the individual life cycle of each VNF composing it.

For the individual VNF life cycle, it has been adopted and further adapted the VNF states and transitions as presented in Section 2.3.2.2. Basically, while orchestrating a network service, the individual VNFs composing it move towards different states in which a set of operations are executed. Figure 5.11 shows the adapted VNF state diagram and highlights the most important transitions between different states.

The transitions between one state to the other are triggered by the execution of one or more life cycle events.

5.7.3.1 Network Service Deployment

The *instantiate* life cycle operation is the first step executed during the network service orchestration process, triggered by the TSP request of deploying a network service[12]. The NFVO instantiates a new NSR using the statically defined informa-

[12] A POST request at the URL `/api/v1/ns-records` passing the NSD in the body, or a POST request to the URL `/api/v1/ns-records/nsr-id` specifying the NSD-id already onboarded in the catalog

Figure 5.10: NSD Overview

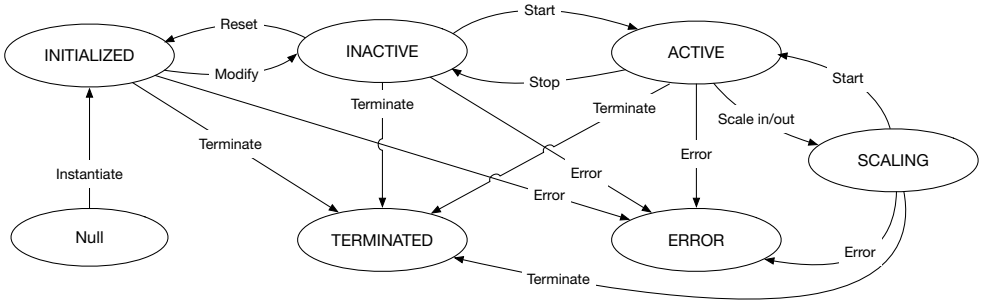

Figure 5.11: VNF States and Transitions

tion contained in the NSD. The initial state of the newly instantiated NSR is set to *NULL* as the VNFRs have been created and stored in the catalog, but the actual instantiate life cycle is under execution. Figure 5.12 provides an activity diagram of the internal execution workflow during the instantiate life cycle.

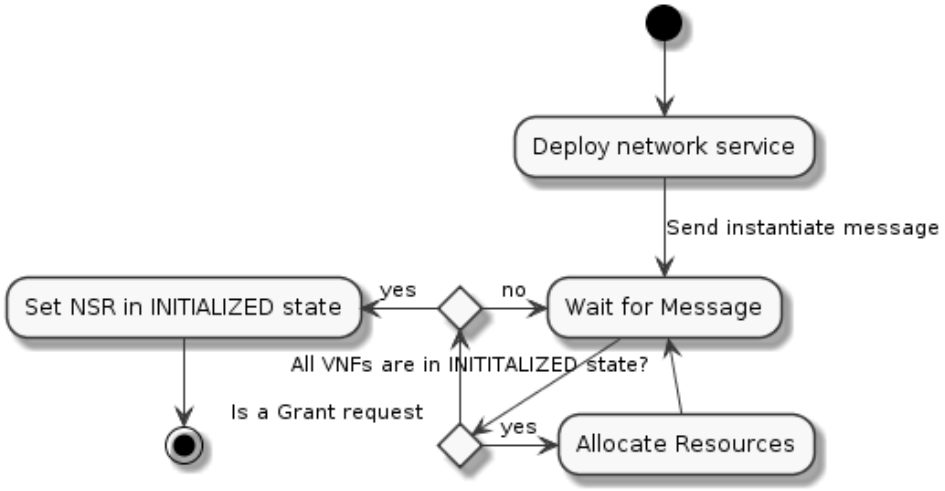

Figure 5.12: Activity Diagram Describing the Instantiate Life Cycle

Figure 5.13 shows the sequence diagram of the operations executed during the *instantiate* life cycle operation. The MANO4X framework, and in particular the NFVO, provides a granting mechanism used for the allocation of the resources on the NFVI. In particular, depending on the approach adopted by the VNFM the resources are either allocated by the NFVO directly or by the VNFM.

Once all the VNFs composing the NS have been instantiated, their records are stored in the catalog of the NFVO. At this point, the NFVO can start executing the next life cycle operation, the *modify*. In the example under consideration, VNF-A provides some information to VNF-B, therefore, their dependency is defined in such a way that VNF-A represents the source providing information to the target

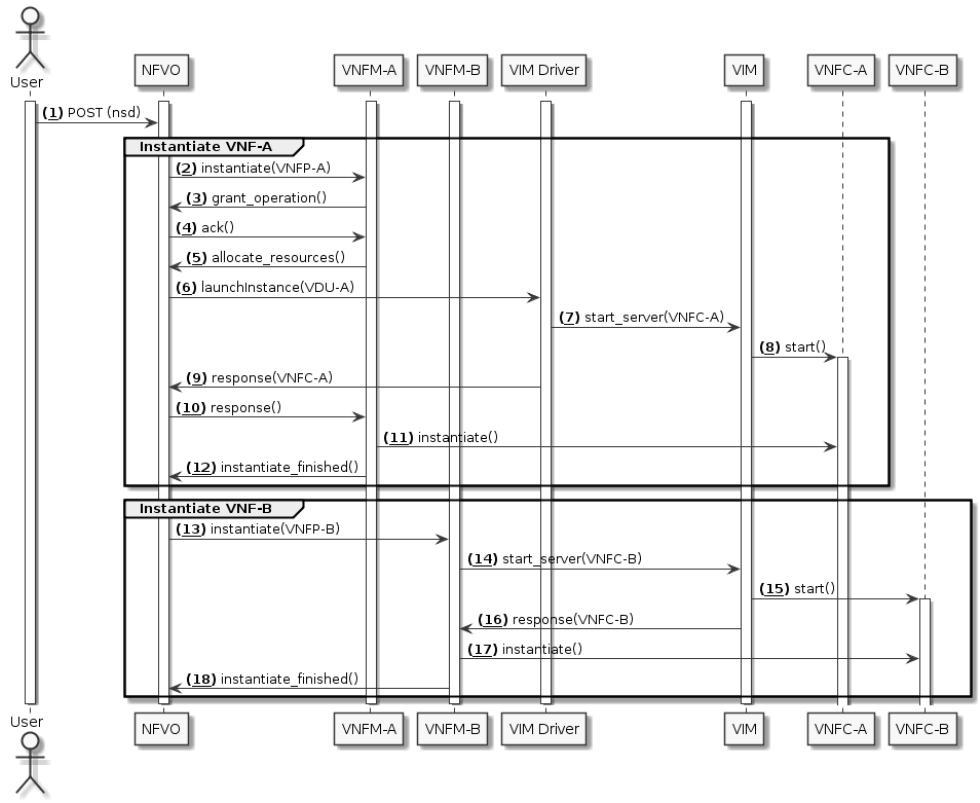

Figure 5.13: Sequence Diagram Showing the Instantiate Life Cycle Operations

VNF-B. A VNF dependency is composed by:

Parameter	Description
`source`	The name of the VNF, as specified in the descriptor, that *provides* one or more parameters
`target`	The name of the VNF, as specified in the descriptor, that *requires* one or more parameters
`parameters`	The name of the parameters that the target requires

Table 5.1: Content of the VNF Dependency

The information passed through the dependency can be of two types:

- *Dynamic*: this type of information includes IP addresses and host names. Their values are generated runtime by the resource orchestrator process, thus set by the NFVO or VNFM directly.
- *Custom*: information provided by the VNFP as part of the *configuration* object

contained in the VNFD. This information can be modified by the TSP either during the onboarding or deployment operation in order to customize the VNF for the specific scenario required.

An alternative approach to define a dependency between VNFs, is to define the *requires* field at the level of the VNFD. This approach allows simplifying the generation of the NSD as the VNF dependency is automatically generated by the NFVO based on the actual requirements of each VNF.

In cases where some VNFs do not depend on any parameters from other VNFs, the *modify* life cycle operation is skipped, and the NFVO immediately triggers the *start* life cycle operation after the instantiation process is finished.

Once all the dependencies have been solved, and the NSR reached the INACTIVE state, all the individual VNFs are basically configured but not yet activated. In order to move the NSR to the ACTIVE state, the NFVO has to execute the *start* life cycle operation. This transition is mainly driven by the *start* primitive call from the NFVO to the VNFMs.

5.7.3.2 Network Service Runtime Management

Reaching the ACTIVE state determines the beginning of the runtime phase of a network service. As already presented in the previous sections, most of the OSS functional elements start their internal control loop upon receiving a notification event from the NFVO about the successful instantiation of the network service. Depending on the category, the OSS may have (*bi-directional*) or (*uni-directional*) interaction with the NFVO. In the following, examples are provided of high-level procedures for both cases.

Figure 5.14 shows the sequence diagram of the interactions between the different functional elements in case of the *uni-directional* case.

In particular, after every state transition, the NFVO checks the list of subscribed entities interested in that particular life cycle event. Thus, it generates an event (1) and publishes it (2) to the destinations via the message bus. The destination entity, in this case the OSS, handles the request (3) extracting the payload of the message providing the NSR containing all the dynamic information generated during the deployment phase. With the content of the NSR the OSS can execute any kind of operation (4-5) towards other components like the VIM.

Figure 5.15 shows the sequence diagram of the interactions between the different functional elements in case of the *bi-directional* case. As already presented, OSSs part of the *bi-directional* category, may need to request state transitions, via the *Or-Oss*, impacting directly on the state of the NSR (e.g., scaling in and out, or switch to standby actions).

Steps [1-3] are the same as the ones presented for the case of the *uni-directional* category. The major difference is represented by steps executed upon reception of the event message. Typically, those systems apply a control loop gathering KPIs from the monitoring system (potentially done using the monitoring plugin as per steps described in [4-7]), detecting (8) whether a particular condition (usually defined

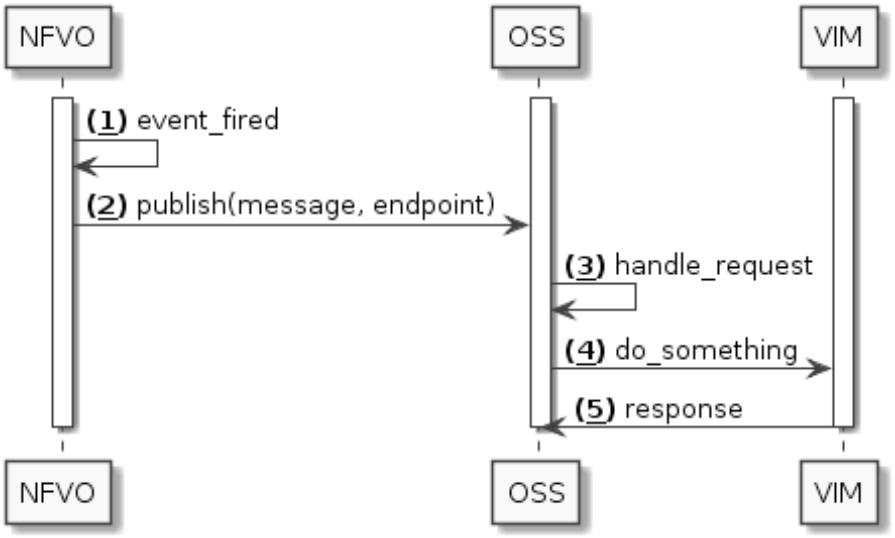

Figure 5.14: Sequence Diagram of the OSS - *Uni-directional* Category

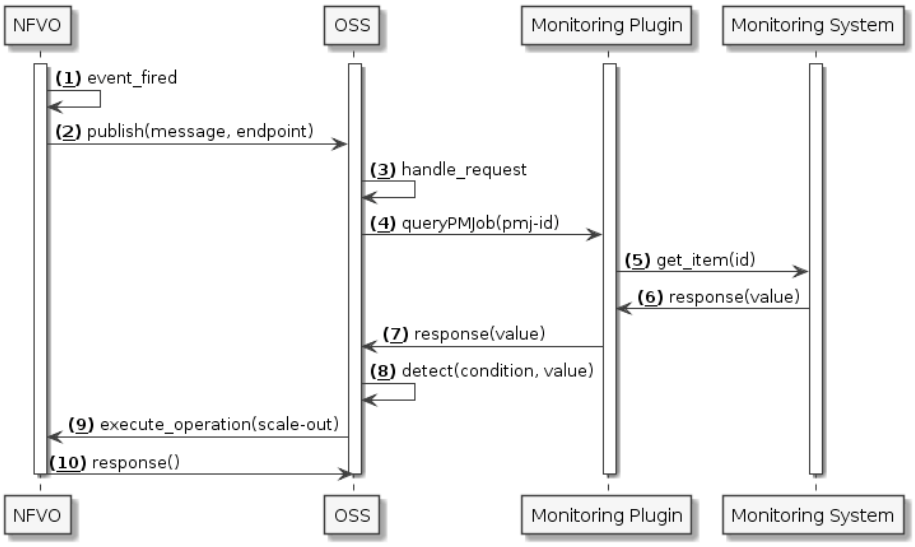

Figure 5.15: Sequence Diagram of the OSS - *Bi-directional* Category

in the received NSR) is met, and executing (9) an action towards the NFVO for modifying the NSR in order to reach the desired state. After receiving the response (10) the control loop can start again.

5.7.3.3 Network Service Disposal

The last stage of the network service life cycle is represented by its termination. This stage involves a set of requests executed by the NFVO towards the VIM drivers and

the VNFMs requesting the termination of the resources under their control. After all resources have been released, the NFVO publishes an event about the termination of the NSR which can be consumed by any OSSs subscribed for consequently releasing any resources under their control.

5.8 Conclusion

The present chapter has introduced the overall MANO4X framework architecture and its specification. Following the Chapter 4 – The Design Evolution of the MANO4X Framework, the first part of the present chapter has specified the final architecture of the proposed framework.

Following SOA, microservices architectural patterns and the EDA design pattern, the proposed framework is based on a central message bus and orchestration system acting as a broker between different elements contributing to the overall network service life cycle. The logical separation in domains allows the integration of any kind of infrastructural resources and configuration management systems.

The last part of the present chapter detailed the high-level procedures executed for managing the entire life cycle of a network service. The information here serves as the foundation for the implementation of the proposed architecture described in the next chapter.

Chapter 6

Implementation of the Open Baton Framework

This chapter follows on exposing details about the software implementation of the proposed MANO4X architectural framework presented in Chapter 4 – The Design Evolution of the MANO4X Framework and Chapter 5 – Specification of the MANO4X Framework. The focus of this chapter is to provide details about the

reference implementation of the final architecture of which specification has been proposed in the previous Chapter 5 – Specification of the MANO4X Framework.

As already presented in Chapter 1 - Introduction, *Open Baton* is the name given to the implementation of the proposed framework, launched publicly in October 2015[163]. *Open Baton* represents the reference implementation of the proposed solution. *"Open"* because of the openness of the solution, considering the major objective of this work to release the source code openly to the community, while *"Baton"* because of the similarities between the music domain and the orchestration domain: As the director needs the baton while managing an orchestra of musicians for playing a particular song, so the administrator needs a tool for managing different VNFs to execute a particular network service.

6.1 The Open Baton Framework

Open Baton is the result of the agile design process previously presented, with the main objective of building a framework capable of orchestrating network services across heterogeneous infrastructural resources. The implementation follows the functional architecture design proposed in the previous chapter, having the NFVO and the message bus as central components, a Generic VNFM as a generic implementation of the VNFM able to manage any kind of VNFs using a ligthweight EMS, one or more specific VNFMs, integrated via VNFM adapters, different drivers for interoperating with external VIMs and monitoring systems, and a FMS, an AES, a SFCO, and a NSE, as exemplary OSS components handling the runtime phase of the network service life cycle. Figure 6.1 depicts the high level architecture of the the Open Baton framework.

A comprehensive list of features supported is provided as follows:

- Installation, deployment and configuration of a large number of VNFs and NSs (i.e. vIMS, vM2M, vEPC, etc.).
- Management of a multi-site NFVI, supporting heterogeneous virtualization and cloud technologies (i.e. OpenStack, Docker, etc.).
- Ensures multi-tenancy at the infrastructure level, making use of SDN technologies (i.e. OpenVSwitch) for ensuring isolation between multiple network services deployed
- Provides a Generic VNFM.
- Integrates with existing VNFMs, which could be easily plugged either directly implementing the *Or-Vnfm* interface exposed by the NFVO, or via SDK available in different programming languages (`Java`, `Python`, and `Go`). The Juju VNFM adapter is an example of integration using the `Python` SDK.
- Supports runtime operations fulfilling the need of the FCAPS model integrating external OSS systems, and providing fault management and autoscaling.

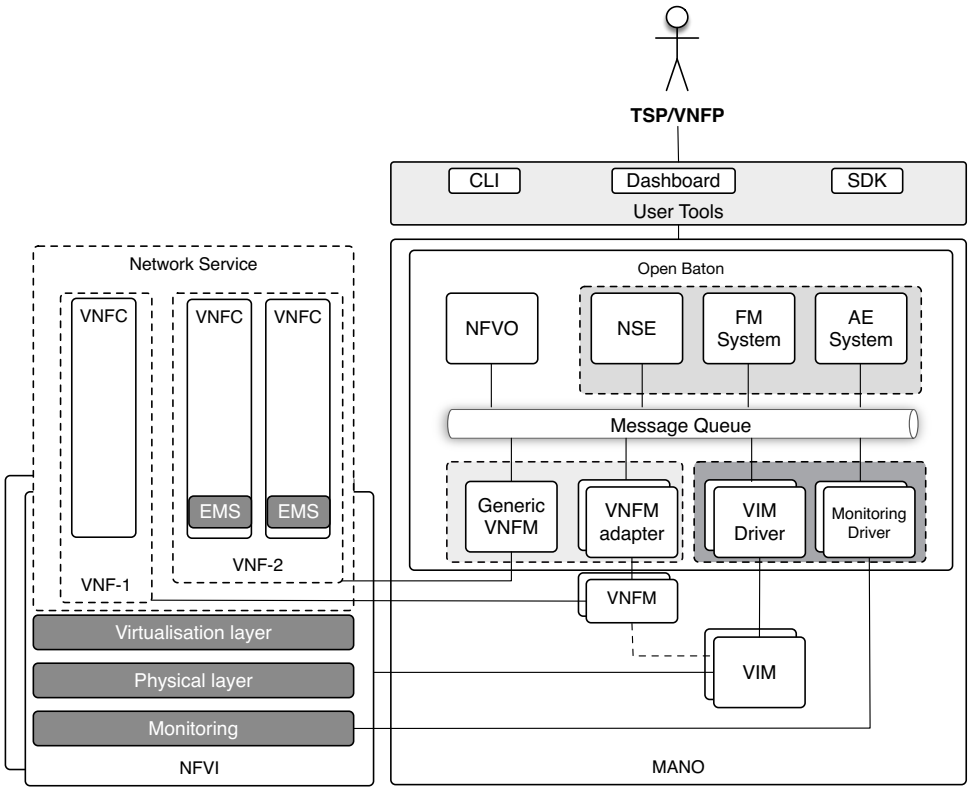

Figure 6.1: Open Baton High Level Architecture

The development of new VIM drivers, monitoring plugins, OSS components, allows extending and customizing the framework for supporting any kind of use cases. Furthermore, the availability of SDKs allows developers to focus on their specific use cases requirements, hiding most of the component management complexity, like registration procedures.

6.1.1 Main Technologies Used

The extreme flexibility required for the MANO4X framework, led to the implementation of a loosely coupled microservice-oriented architecture, allowing each individual component to be implemented with the most suitable programming language. The following Table 6.1 summarizes the different components available as part of the fourth release of Open Baton, and their programming languages.

A large part of the components available have been implemented using `Java`. One of the main reasons is that, when the implementation activities started, `Java` was the programming language supported by most OSs, and therefore increases portability of the final solution between different environments. In particular, the Java Enterprise Edition (Java 2 Enterprise Edition (J2EE)) has been employed as computing platform for the development and deployment of those components.

Project	Programming Language	GitHub Repository Project Name
NFVO	Java	NFVO
Dashboard	Javascript	dashboard
CLI	Java, Python	openbaton-client[1]
Dummy VNFM Advanced Message Queuing Protocol (AMQP)	Java, Python, Go[2]	dummy-vnfm-amqp[3]
Dummy VNFM REST	Java	dummy-vnfm-rest
Generic VNFM	Java	generic-vnfm
Generic EMS	Python	ems
Juju VNFM Adapter	Python	juju-vnfm
Docker VNFM	Go	go-docker-vnfm
OpenStack VIM driver	Java	openstack4j-plugin
Test VIM driver	Java, Go[4]	test-plugin[5]
Docker VIM driver	Go	go-docker-driver
Autoscaling Engine	Java	autoscaling-engine
Fault Management System	Java	fm-system
Network Slicing Engine	Java	network-slicing-engine
Marketplace	Java	marketplace
Zabbix Plugin	Java	zabbix-plugin
Integration Tests	Java	integration-tests
OpenIMSCore Packages	Bash	openimscore-packages
ClearWater Packages	Bash	clearwater-packages

Table 6.1: Different Components, Their Programming Languages, and Their GitHub Project Names

J2EE provides an advanced API for distributed enterprise applications, which extends the basic functionalities available in the Java Standard Edition version. J2EE is based on specifications: basically each functionalities which is exposed via APIs must meet certain requirements in order to be declared J2EE compliant. Such APIs

are typically documented via specification documents. Each specification could be implemented by different technologies. As mentioned, J2EE is an abstract specification which requires applications to be executed on application servers, or make use of supporting libraries implementing the J2EE APIs.

Gradle[6] has been utilized as project management system. Gradle is an open source tool based on the ideas of Apache Ant[7] and Apache Maven[8] introducing a Domain Specific Language (DSL) language based on Groovy[9] instead of the XML one used by Maven for configuring a project. Gradle scripts could be executed directly without having a declaration as it is for Maven. Differently from Maven and Ant, Gradle utilizes a Directed Acyclic Graph (DAG)[10] for determining the order in which processes can be executed.

The message bus represents the central component of the system, allowing loosely coupled communication using topic-based pub/sub mechanisms. RabbitMQ[11] has been selected as message bus implementation. The specific format of the messages exchanged between components is defined by the JSON schema. RabbitMQ has a Mozilla Public License (MPL) license, provides a central broker entity, supporting pub/sub communication mechanisms as well as RPC, including topic-based permissions based on queue and exchanges. The main transport mechanism used is TCP, and supports several programming languages as bindings. Other solutions available, like ActiveMQ[12], ZeroMQ[13], Kafka[14], were evaluated, but not finally considered either because of their licensing model (i.e. ZeroMQ is having a GNU's Not Unix! (GNU) General Public License (GPL) license) or no support of topic-based permission, which is one of the most important requirements for allowing secure-communication between components dealing with confidential information.

6.2 Central Domain: NFVO and RabbitMQ

The NFVO implements almost all the key functionalities required for supporting the requirements and list of features presented. It maintains an overview on the infrastructure, supporting dynamic registration of PoPs. Currently uses the OpenStack as standard de facto VIM. It maintains an inventory of VNFPs including VNF images and VNFDs. Deploys on-demand VNFs on top of a multi-site NFVI. Supports multi tenancy allowing deployment of parallel slices, consisting of one or multiple VNF. Accepts TOSCA CSAR as packaging format for VNFPs and descriptors. It interoperates with the other components of the different domains via the RabbitMQ message bus.

[6] https://gradle.org/
[7] http://ant.apache.org/
[8] https://maven.apache.org/
[9] http://www.groovy-lang.org/
[10] https://en.wikipedia.org/wiki/Directed_acyclic_graph
[11] https://www.rabbitmq.com
[12] http://activemq.apache.org
[13] http://zeromq.org
[14] https://kafka.apache.org

6.2.1 NFV Orchestrator Modules

The NFVO is the central component, basically acting as a workflow executor, dispatching tasks across elements from different domains via the message bus. Considering that the main objective is to manage the life cycle of network services, the NFVO aggregates the different states of each individual resource composing it. Therefore, it maintains an active state diagram of all the resources which are involved in a network service.

The NFVO modules presented in the previous chapter:design have been implemented as separated gradle modules and grouped as part of the NFVO component. The NFVO comprises eight major modules:

- `api`: providing the REST APIs exposed to the different consumers (being a human, or another component, like the Dashboard or the CLI).
- `cli`: Providing an implementation of a simple CLI used as part of the console of the NFVO. This module is different from the external CLI which is utilized by users for interoperating with this component, as it provides methods which are specific for the management of internal part of the NFVO.
- `common`: Including all the common source code across different other modules.
- `core`: Implementing the orchestration logic. This module will be further described in the following subsections.
- `repository`: Including all the Java Entities which are used by other modules for persisting information in the database. This module makes use of the Java Persistence APIs (JPA) specification for abstracting the low level details of the database entity used, providing interfaces to other modules making use of object-oriented programming for storing information.
- `security`: Comprising all the classes which are dealing with authentication and authorization.
- `tosca-parser`: Including all the classes used for parsing TOSCA templates.
- `vnfm`: Providing the interfaces for communicating with available VNFMs.

6.2.2 Information and Data Model

The information and data model implemented follows the one presented in the previous Chapter 4. The information and data model specified by ETSI in MANO v1.1.1[28] has been mapped to different `Java` classes, as JPA entities. All the classes providing the information and data model of the core elements have been packaged as part of the repository module in four different packages.

6.3 North Domain: User Tools

This section provides implementation details about the main user tools which have been developed as part of the Open Baton project. Firstly, it is provided an overview about the dashboard, and in particular about the technologies used while implementing it. Secondly, a set of SDKs are presented. SDKs can be easily imported in other applications (automating the execution of certain operations) interacting with the NFVO via its northbound interface. Last but not least, it is introduced the CLI, allowing users interacting via command line with the framework.

6.3.1 The Dashboard

The dashboard represents a comprehensive web-based GUI exposing a set of web pages allowing end users to manage infrastructure resources and network services. The dashboard has been implemented using Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and `Javascript`. The dashboard source code has been included as git submodule inside the NFVO project, so that after starting the NFVO the dashboard is automatically available[15]. Figure 6.2 shows a screenshot of the overview page of the dashboard available immediately after login.

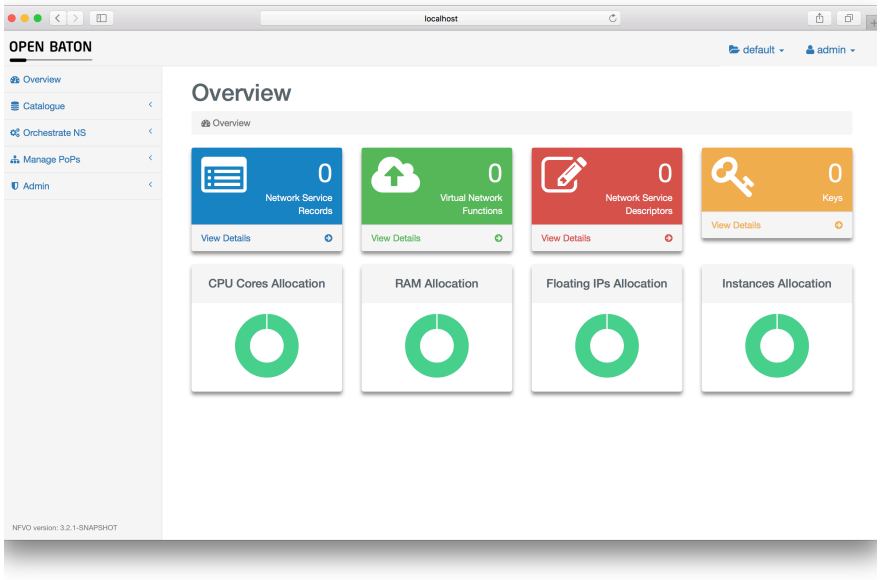

Figure 6.2: Open Baton Dashboard - Overview Page

The dashboard supports authentication and authorization using cookies. Basically, it requests a token to the NFVO (through its REST APIs) using the content

[15] the dashboard is reachable by default on port 8080 exposed by the Tomcat Application Server executing the NFVO process

provided via the login function. The token is stored in a cookie and maintained until the user log out (or until its expiration[16]).

6.3.2 The Sofware Development Kit (SDK)

A set of SDKs have been developed in order to support developers in implementing, packaging and deploying VNFs and network services using the Open Baton framework. SDKs have been implemented in three different languages (`Java`, `Python`, and `Go`) mainly with the objective of providing classes implementing the information and data model following a OOP paradigm.

To further support developers in making use of those libraries in their applications, compiled version of the different SDKs have been uploaded in common central repositories: Sonatype[17] for `Java`, Pypi[18] for `Python`, and GitHub for `Go`.

For instance, the `Java sdk` module provides the `NFVORequestor` class which could be utilized for interacting with the NFVO via its REST APIs. The `sdk` module imports the catalog (`'org.openbaton:catalogue'`) in order to serialize the objects received via the REST APIs. The `NFVORequestor` makes use of a `RequestFactory` implemented as a *Singleton*, providing instances of different *agent* objects implementing a particular API call. The different categories of *agents* follows the same classification done for the NFVO API.

6.3.3 The Open Baton Command Line Interface (CLI)

The first version of the CLI has been implemented in `Java` as part of the *openbaton-client*[19] project.

The CLI, imports the `Java sdk` module, and it instantiates the `NFVORequestor` passing as arguments information needed to the class for authenticating itself with the NFVO.

Listing 6.1: Example of Instantiation of the `NFVORequestor` Class

```
1  ...
2      NFVORequestor nfvo =
3          new NFVORequestor(
4              properties.getProperty("NFVO_USERNAME"),
5              properties.getProperty("NFVO_PASSWORD"),
6              properties.getProperty("NFVO_PROJECT_ID"),
7              Boolean.parseBoolean(properties.getProperty("
                   NFVO_SSL_ENABLED")),
8              properties.getProperty("NFVO_IP"),
9              properties.getProperty("NFVO_PORT"),
10             properties.getProperty("NFVO_API_VERSION"));
```

[16] in this case the user will be logged out and will need to re-authenticate again
[17] http://central.sonatype.org/
[18] https://pypi.python.org/pypi
[19] https://github.com/openbaton/openbaton-client

```
11 ...
```

Since Open Baton release two, a new version of the CLI implemented in `Python` has been provided as part of the *openbaton-cli* project[20]. Compared to the `Java` version, this `Python` version is more lightweight, allowing its execution on any OS without the installation of a Java Virtual Machine (JVM). In terms of functionalities, both versions of the CLI provide the same.

6.4 South Domain: NFVI

This section analyzes the south domain, focusing on providing implementation details about the mechanisms developed for integrating a multi-site NFVI. This domain comprises VIM drivers allowing the seamless integration of heterogeneous infrastructure technologies, controlled via remote APIs exposed by a VIM, as well as monitoring plugins used for integrating existing monitoring systems. Both VIM drivers and monitoring plugins, are based on a plug-and-play mechanism so that they can be added during runtime to the Open Baton framework.

In order to simplify the development of VIM drivers and monitoring plugins a set of SDKs have been provided in three different programming languages:

- `Java`: source code available as part of the *plugin-sdk*[21] project repository. Compiled libraries have been published on the maven central repository and could be imported in any projects using gradle or maven.
- `Python`: source code available under the *python-plugin-sdk*[22] project repository. These python classes have been published directly on pip[23]
- `Go`: source code available under the *go-openbaton*[24] project repository. In order to import them in any project, it is only required to import the github repository URL in the `Go` classes.

6.4.0.1 The OpenStack Driver as the Reference Implementation

The OpenStack driver allows the integration of OpenStack as NFVI-PoP. The driver has been implemented as a standalone standard `Java` application, making use of the `Java` *plugin-sdk*. Considering the large amount of libraries available in many different programming languages[25], the approach taken was to select the most suitable one supporting the functionalities required for querying available resources and instantiating them. Initially, Apache JClouds[26] was chosen because of its support of a large number of cloud providers.

[20] https://github.com/openbaton/openbaton-cli
[21] https://github.com/openbaton/plugin-sdk
[22] https://github.com/openbaton/python-plugin-sdk
[23] https://pypi.python.org/pypi/python-plugin-sdk
[24] https://github.com/openbaton/go-openbaton
[25] https://wiki.openstack.org/wiki/SDKs
[26] https://jclouds.apache.org

Lately, the driver was re-implemented making use of the OpenStack4J[27] library because of its support of the latest version (V3) of the Keystone APIs introduced in recent versions of OpenStack. The latest version of the driver is available as part of the `openstack4j` project[28] and supports Pike as the latest stable OpenStack version available. Table 6.2 provides the mapping between the VIM driver interface exposed towards the NFVO/VNFM and the OpenStack API[29] called.

Function	OpenStack APIs
listImages	`GET /v2/images`
listServer	`GET /servers`
listNetworks	`GET /v2.0/networks`
listFlavors	`GET /flavors`
launchInstanceAndWait	`POST /servers`
deleteServerByIdAndWait	`DELETE /servers/{server_id}`
createNetwork	`POST /v2.0/networks`
getNetworkById	`GET /v2.0/network s/{network_id}`
updateNetwork	`PUT /v2.0/networks/{network_id}`
deleteNetwork	`DELETE /v2.0/networks/{network_id}`
createSubnet	`POST /v2.0/subnets`
updateSubnet	`PUT /v2.0/subnets/{subnet_id}`
deleteSubnet	`DELETE /v2.0/subnets/{subnet_id}`
getSubnetsExtIds	`GET /v2.0/subnets/{subnet_id}`
addFlavor	`POST /flavors`
updateFlavor	NA
deleteFlavor	`DELETE /flavors/{flavor_id}`
addImage	`POST /v2/images`
updateImage	`PATCH /v2/images/{image_id}`
copyImage	NA
deleteImage	`DELETE /v2/images/{image_id}`
getQuota	`GET /os-quota-sets/{tenant_id}`
getType	NA

Table 6.2: Mapping between Or-Vi-rpc/Vnfm-Vi-rpc Interface and OpenStack API Calls

6.4.0.2 The Docker VIM Driver

Another VIM driver available as part of the Open Baton project is the one allowing the integration of docker as VIM, supporting the instantiation of containers instead of classical VMs[139]. Considering that docker containers have a different behavior

[27] http://www.openstack4j.com
[28] https://github.com/openbaton/openstack4j-plugin
[29] https://developer.openstack.org/api-ref/

than VMs (docker containers are practically processes of the VNF software components, see Section 2.2.2 for more details), the role of the docker VIM driver is to ensure that the required networks and container images are available upon request of instantiation of certain network services. In this case, the role of instantiating the actual docker container is delegated to the docker VNFM which will be presented in Section 6.5.3.

6.4.0.3 The Zabbix Monitoring Plugin

Zabbix[30] has been selected as one potential monitoring system[31]. Zabbix is an enterprise-level software designed for monitoring availability and performance of IT infrastructure components. An agent installed on the target resources continuously collects metrics and push them to the central server. Zabbix provides a set of APIs allowing pulling monitoring information regarding monitored resources from the Zabbix server. Moreover, Zabbix provides notifications (i.e. e-mails, Short Messaging Service (SMS), custom alert scripts, etc.) whenever a certain condition (i.e., defined as a threshold on a particular metric) is met.

The Zabbix monitoring plugin has been implemented as a standalone standard `Java` application. It provides an implementation of the *Vi-Mon* interface, allowing creating and deleting items, triggers and actions, as well as retrieving on demand metrics based on the needs of the plugin consumer. The two interfaces exposed by this plugin are: `VirtualisedResourceFaultManagement` and `VirtualisedResourcePerformanceManagement`.

In terms of features, the Zabbix plugin provides: Creation/deletion of metrics, triggers and actions from the Zabbix server; Abstracts the Zabbix APIs providing an NFV compliant interface; Local caching of metrics that are of interest for the plugin's consumers.

The specific mapping between the specified ETSI data model and the Zabbix one is:

- *PerformanceMetric*: zabbix item[32]
- *Threshold*: zabbix trigger[33]

6.5 West Domain: VNF Manager (VNFM)

The west domain comprises VNFMs, and VNFM adapters needed for supporting the life cycle of a VNF. Also in this case, in order to simplify the development of the VNFM a set of libraries have been provided in three different programming languages:

[30] https://www.zabbix.com/

[31] The solution presented can be easily extended for supporting any other type of monitoring system

[32] https://www.zabbix.com/documentation/3.0/manual/config/items

[33] https://www.zabbix.com/documentation/3.0/manual/config/triggers

- `Java`: source code available as part of the *vnfm-sdk*[34] project repository. Compiled libraries have been published on the maven central repository and could be imported in any projects using gradle or maven.

- `Python`: source code available under the *python-vnfm-sdk*[35] project repository. These `Python` classes have been published directly on pip[36]

- `Go`: source code available under the *go-openbaton*[37] project repository. In order to import them in any project, it is only required to import the github repository URL in the `Go` classes.

The three SDKs versions follow a similar approach providing a set of classes that can be used for building the skeleton of a VNFM ready to handle operations triggered by the NFVO. A developer has to focus exclusively on the implementation of the functions executing the life cycle management operations of the VNF. In order to better understand the different mechanisms provided by those three SDKs it is provided an overview of three different VNFMs implemented in three different languages.

6.5.1 Generic VNFM and Element Management System (EMS) as Reference Implementation of a VNFM

The Generic VNFM is the reference VNFM used inside the Open Baton framework. It has been implemented as standalone spring boot `Java` application making use of the `Java` SDK for the marshaling and unmarshaling of the data received over the message bus. It provides a generic mechanism for managing the life cycle of any kind of VNFs based on the remote triggering of execution of scripts inside the VDUs deployed on the NFVI. It allows integrating any kind of VNF making use of a contextualization process managed by a lightweight agent, named Generic EMS, running inside the VMs.

The major task performed by the Generic VNFM is the instantiation of a VNFC. The instantiation is the first operation performed by the Generic VNFM aiming at instantiating all VNFCs of a VNF accordingly to the VNFD and related scripts (contained in the VNFP) received from the NFVO. During the instantiation life cycle operation, the Generic VNFM requests the instantiation of the virtualized resources to the NFVO, passing the user-data[38] containing installation scripts to be performed while booting (including a Universally Unique Identifier (UUID) for uniquely identifying that particular VNFC instance). In practice, the Generic VNFM stores the UUIDs in a local list, and

[34] https://github.com/openbaton/vnfm-sdk
[35] https://github.com/openbaton/python-vnfm-sdk
[36] https://pypi.python.org/pypi/python-vnfm-sdk
[37] https://github.com/openbaton/go-openbaton
[38] Using the cloud-init functionality available in standard cloud image: https://cloudinit.readthedocs.io/en/latest/

Upon activation, the Generic EMS sends a registration message to the Generic VNFM via the message bus. The Generic VNFM extracts the information received, in particular the UUID, and requests the execution of particular scripts through the Generic EMS API. Once the instantiation procedure is finished, the Generic VNFM sends back to the NFVO the VNFR containing all the details about the instantiated resources.

Injection of configuration parameters provided by the TSP via the VNFD, as well as of the runtime information provided through dependency resolution during the modify operation, is done using environment variables. In particular, the Generic EMS exports those environment variables before executing any scripts inside the VM.

In order to make use of those environments variables, VNFP have to follow a particular syntax. All the configuration parameters contained in the parent VNFD of the VNFC are passed directly without modifications. The runtime information (i.e., fixed and floating IPs) are extended by the Generic VNFM including the `network-name` (name contained in the VLD) as prefix. The information provided during the modify lifecycle operations are also extended runtime for uniquely identifying the VNFC instance source of the relation. In particular, the Generic VNFM includes the `source-name` as prefix of the parameter.

6.5.2 The Juju VNFM Adapter

Juju is an open source tool released by Canonical for deploying applications in multi cloud environments. It can be used for deploying applications and even clusters of applications over public or private clouds. Juju supports a large number of public cloud providers including Amazon Amazon Web Services (AWS) and Microsoft Azure, as well as several private cloud open source tools like OpenStack. Applications are defined as *charms* consisting of data and executable files, which are used by Juju for deploying the application on any kind of cloud environment. Those charms are stored in a public Charm Store (or private local repositories), currently containing more than 200 different types of applications' charms.

In the NFV context, Juju is typically associated with a generic VNFM. It consists of a client and a controller. Typically, the client bootstraps a controller on top of the target cloud environment on which a user is willing to deploy charms. One can think of OpenStack as the target cloud environment where the juju client bootstraps a controller in a virtual machine. Once the controller is available, a user can trigger the instantiation of a particular charm, or a set of charms defined as *bundle*, either via the CLI or the GUI.

Juju differentiates between machines and units: machines are the virtual resources (either VMs or containers) on top of which the charm is executed, while units represent instances of the running charms. In practice, it means that when instantiating a charm, Juju creates a new unit and associate it with the machine on which the application actually runs. However, it is possible to associate multiple units (of the same or different types) to run on the same machine. Table 6.3 shows

the mapping between Open Baton and Juju entities.

Open Baton entity	Juju entity
VNFP	Charm
VNFD	Charm
VDU	Machine
NSD	Bundle
NSR	Running application
VNFC	Unit
life cycle operation	action

Table 6.3: Mapping between the Open Baton and Juju Information Model

In order to satisfy dependencies between charms, Juju makes use of the concept of hooks and actions. Those are executable files (part of the charm package) which may run on the deployed machines. Juju uses a strict syntax for hooks names, so that the particular hooks refers to a specific life cycle stage. For instance, the executable file named `install` corresponds to the installation script which is executed right after the machine is created.

The main approach utilized makes use of a VNFM adapter transforming the method calls received by the NFVO into specific calls towards the Juju controller. On the one hand, the Juju VNFM adapter makes use of the Open Baton `Python` SDK to communicate with the NFVO, handling the basic functionalities like registration, handling incoming messages calling appropriate methods, and sending back responses. On the other hand, the Juju VNFM adapter communicates with Juju using the `python-jujuclient`[39], consuming the WebSocket API exposed by the Juju controller.

The Juju VNFM adapter extends the `AbstractVNFM` class implementing a subset of the functions available on the *Or-Vnfm* interface. In particular, it focuses on the main methods required for deploying and scaling a VNF. The `AbstractVNFM` contains a private method `_on_message_(self,body)` being called whenever the message bus publishes a message on the queue of the VNFM type (in this case defined as `''juju''`). Basically this method implements generic operations required before calling the actual methods implementing the business logic of a specific life cycle operation. It provides all the required functions for handling incoming messages, and returning responses to the NFVO, as well as managing the granting mechanisms in a seamless way, so that the developer shall only take care of configuring a parameter in the config file (`allocate = True/False`) for deciding whether the allocation of resources should be done by the NFVO or is done by the juju itself. In particular, the adapter can be configured in order to:

- `allocate = True`: allow Juju allocating infrastructure resources on the NFVI-PoP under its control. In this case, the Juju controller must be bootstrapped on the cloud provider where VNFs should be deployed.

[39] https://github.com/juju/python-libjuju

- `allocate = False`: allow the NFVO to allocate infrastructure resources and let Juju only taking care of the life cycle management of the VNF.

The *instantiate* function is the first one being called by the NFVO. The Juju VNFM adapter transforms the VNFP content into a Juju Charm, deploys the charm according to the VNF structure (i.e. number of VDUs), triggers the instantiate life cycle for the deployed charms, builds the VNFR object based on the information received from Juju.

6.5.3 The Docker VNFM

The Docker VNFM is a component implemented in `Go` capable of managing VNFC instances executing as docker containers. It uses the AMQP library to interact with the remote VIM docker driver over the message bus. The set of libraries provided by the `Go` packages, already implement for the most part a complete VNFM, leaving the developer the only task to implement the functional logic for correctly handling life cycle events issued by the NFVO.

The first operation invoked by the NFVO is the instantiate life cycle event. This operation is invoked after resources have been correctly allocated. The VNFM uses the `Check(id)` method to connect to the VIM and ensure that all VNFCs belonging to the VNF have been correctly instantiated.

The second operation executed is the modify, with the purpose of modifying and updating configuration settings of the instantiated VNFCs. Considering that docker containers are handled differently than VMs, the role of the docker VNFM is to launch a docker container passing all the parameters required for configuring the VNFs processes. In practice, those configurations are injected to the docker containers as environment variables, and are read by the VNF process after booting.

6.6 East Domain: Operations Support System (OSS)

The east domain comprises OSSs cooperating and interacting with the NFVO in order to support the runtime life cycle of network services. In the following are presented implementation details for all the functional elements presented in Chapter 5.

6.6.1 Fault Management System (FMS)

The FMS is implemented in J2EE as a standalone spring boot `Java` application. The monitoring manager communicates with the monitoring plugin via the message bus. The functional flow of the monitoring manager begins when it receives the INSTANTIATE_FINISH event from the NFVO. For any NSR instantiated a thread is launched and executed periodically. Such thread creates the performance jobs and the thresholds accordingly to the fault management policy on any VNFC instance.

The fault correlator has been implemented as a rule-based system. The rules are expressed in Drools rule language and processed by the business rule management

system Drools[40]. Having the fault correlation policy as rules, allow the user to define the logic to cope with the alarms and choose whether or not execute a specific recovery action. Listing 6.2 shows an example of the Drools policy defined for the recovery action switch to standby.

Listing 6.2: Drools Rules for the Recovery Action switch to standby

```
1  rule "Get␣a␣CRITICAL␣Virtualized␣Resource␣Alarm␣and␣switch␣to␣
       standby"
2
3      when
4          a : VRAlarm( hostname : managedObject, alarmState ==
               AlarmState.FIRED,
5          perceivedSeverity == PerceivedSeverity.CRITICAL)
6          not RecoveryAction(status == RecoveryActionStatus.
               IN_PROGRESS)
7      then
8          logger.info("(VIRTUALIZATION␣LAYER)␣A␣CRITICAL␣alarm␣
               is␣received␣regarding␣the␣managedObject:␣" +
               hostname);
9          VNFCInstance failedVnfcInstance = nfvoRequestorWrapper
               .getVNFCInstance(hostname);
10
11         VirtualNetworkFunctionRecord vnfr =
               nfvoRequestorWrapper.
               getVirtualNetworkFunctionRecordFromVNFCHostname(
               hostname);
12
13         VirtualDeploymentUnit vdu = nfvoRequestorWrapper.
               getVDU(vnfr,failedVnfcInstance.getId());
14
15         logger.info("Switch␣to␣standby␣fired!");
16         highAvailabilityManager.switchToStandby(vnfr.getId(),
               failedVnfcInstance.getId());
17
18         RecoveryAction recoveryAction= new RecoveryAction(
               RecoveryActionType.SWITCH_TO_STANDBY,vnfr.
               getEndpoint(),"");
19         recoveryAction.setStatus(RecoveryActionStatus.
               IN_PROGRESS);
20         recoveryAction.setNsrId(vnfr.getParent_ns_id());
21         insert(recoveryAction);
22         logger.debug("Recovery␣action␣in␣progress!␣:"+
               recoveryAction);
23         delete(a);
24 end
```

[40]https://www.drools.org/

The `HighAvailabilityManager` executes the fault recovery actions and maintain the redundancy scheme. The actual execution of the switch to standby function is delegated to the NFVO by the `HighAvailabilityManager` via the `switchToStandby` method provided by the Open Baton SDK, passing the UUID of the failed and the standby VNFC instances. This recovery action is composed by three phases. First, the NFVO activates the standby VNFC instance. Second, it executes the *modify* life cycle operation solving dependencies with other ACTIVE VNFC instances depending from it. Once finished, the NFVO sends a heal message to the dependent VNFC instances so that they can re-execute the *start* operation and re-configure the VNFC software components based on the latest configuration.

The healing action is executed sending a heal request to the NFVO via its REST APIs. The request contains the cause (derived by the `FaultCorrelatorManager`) and the target VNFC where to execute the healing action. The NFVO forwards the healing request to the generic VNFM triggering the execution of the heal life cycle operation.

The solution proposed can also be applied in case of a specific VNFM. In such cases, the FMS identify the potential fault and notifies the VNFM (through the NFVO) about the root-cause of the issue. The specific VNFM receives the heal life cycle operation event and can implement the healing action for solving the issue at the VNF level without having to re-instantiate the complete VNF.

6.6.2 Autoscaling Engine System (AES)

The AES is implemented in J2EE as a standalone spring boot `Java` application. Two different approaches can be taken while instantiating the AES component: NFVO-centric and VNFM-centric.

Typically, the NFVO-centric approach is the most general one, where the AES is instantiated as an external component providing autoscaling mechanisms as an independent component. This case is the most common one, as it can be used for any kind of VNFs that follow a common life cycle execution.

In the VNFM-centric approach the AES is embedded within the VNFM component enabling customizing the autoscaling logic for a certain type of VNF. In such case, the VNFM plays the central role in managing autoscaling policies, deciding when to activate/deactivate detection tasks and providing specific features while executing scaling actions. Depending on the approach taken by the VNFM for allocating resources, instantiation requests of virtual resources towards the VIM maybe directly triggered by the VNFM and AES component without the intervention of the NFVO.

Following the design methodology described in the previous chapter, the AES consists of three main classes: the `Detector`, the `DecisionMaker` and the `Executor`. Those three components correspond to individual spring beans, which compose a single spring boot application. Their implementation is described more in details in the following subsections.

6.6.2.1 Detector

The main functionalities of this component are implemented by the following classes:

- `DetectionManagement`: this class acts as a scheduler and provides methods to start and stop the detection mechanism. Every time the `start()` method is called it creates a new `DetectionTask`. This means a `DetectionTask` is created for each policy contained in the corresponding VNFR.
- `DetectionEngine`: this class provides specific methods used for requesting measurements, calculating final values, checking thresholds and firing events.
- `DetectionTask`: this class is in charge of detecting the need to scale. Therefore, it periodically checks the conditions defined in the policy by interacting with the monitoring system (through the monitoring driver). These measurements are aggregated based on the statistic method defined in the policy and finally checked against a specific condition. When the number of weighted alarms cross the defined threshold percentages, a specific method of the `DecisionMaker` is called.

6.6.2.2 DecisionMaker

The `DecisionMaker` implements the logic for taking decisions about scaling actions. Therefore, it receives alarms from the `Detector` and triggers the execution of certain actions. In general, an alarm sent by the `Detector` identifies the corresponding parent autoscaling policy. In this way the `DecisionMaker` knows what actions might be executed. Based on the specific implementation of this component it can just forward these actions to the `Executor`. A more complex `DecisionMaker` may implement a different logic for checking additional conditions or request operations granted by the NFVO. Additional classes are used by the `DecisionMaker`:

- `DecisionManagement`: This class implements high-level management functionalities and exposes it to other components via its API. One functionality provided by this class is the `decide()` method. This method decides which actions to execute based on inputs received from the *Detector*. For each decision-making it is created an own task that is executed every time the method is called. Another function is the `stop()` function that interrupts the current decision process, either gracefully or interrupting directly after a predefined timeout.
- `DecisionEngine`: This class is used by the `DecisionTask` and allows requesting information to the NFVO, granting operations, or sending decisions to the corresponding `Executor`.
- `DecisionTask`: This class is in charge of taking the final decision about the action to execute based on the alarm received. It makes use of the functionalities provided by the `DecisionEngine` to take into account all the information available for taking the final decision about the scaling actions.

6.6.2.3 Executor

The `Executor` is in charge of executing actions towards the NFVO. This depends on the approach and on the communication flow of the specific AES. If the execution of actions are requested, it may process them either by using the NFVO or the VIM. The `Executor` module comprises three main classes:

- `ExecutionManagement`: This class implements functionalities exposed through external APIs. When the request of executing an action is received it creates an `ExecutionTask` for each request containing a list of actions to be executed.
- `ExecutionEngine`: This class provides functionalities for executing different kind of scaling actions, such as scaling out, scaling in, scaling to a specific number of instances or scaling to a specific deployment flavor.
- `ExecutionTask`: This class is responsible of processing scaling requests by using the capabilities of the `ExecutionEngine`. This class executes actions step-by-step and triggers the cooldown period if at least a scaling action was executed properly. Every time a new scaling request is received, it creates a new `ExecutionTask`. If it processes already a scaling request and another request of executing scaling actions is received for a particular VNF, the second request will be rejected as long as the VNF is still in SCALING state.

As already presented before, the instantiation of virtualized compute resources has strong impacts on the overall instantiation of a VNF. While executing scaling out operations, the time needed for instantiating a new VNFC instance may influence the overall QoE perceived by the end-users consuming that network service. The AES provides a `PoolManager` entity managing a pre-defined pool of already instantiated VNFC instances which could be added directly to the network services. While executing the scale out action the `Executor` requests new instances through the `PoolManager` instead of requesting them to the NFVO/VIM. The `PoolManager` selects one of the available instances compliant with the requirements provided by the `Executor` from the pool, and provides the details to the `Executor`.

6.6.3 Network Slicing Engine (NSE)

The NSE is implemented in J2EE as a standalone spring boot `Java` application. The event subscription has been realized using the SDK consuming the REST APIs exposed by the NFVO. As introduced in the design chapter, the NSE subscribes basically for three events (INSTANTIATE_FINISH, SCALED, and ERROR). During the subscription, the NSE creates three different queues in rabbitmq:

- `core.nse.nsr.create`: used for receiving events of type INSTANTIATE_FINISH
- `core.nse.nsr.scale`: used for receiving events of type SCALED
- `core.nse.nsr.error`: used for receiving events of type ERROR

The NSR management is delegated to a Spring bean, part of the `TemplateProcessing` class, instantiating a different thread for parsing each record received from the NFVO, and collecting all the information related with the QoS policies assigned to the VLD. The subscription expects that the events are delivered through two message queues, one for instantiation and one for termination.

Each class may provide a guaranteed bandwidth as proposed in Table 6.4[41], which is preserved also in case of network congestion, limiting packet loss.

QoS Class	Guaranteed Bandwidth	Maximum Bandwidth
GOLD	250 Mbps	500 Mbps
SILVER	100 Mbps	250 Mbps
BRONZE	1 Mbps	100 Mbps

Table 6.4: Proposed QoS Policies Classes

The `TemplateProcessing` class of the NSE is responsible for receiving NSRs from the NFVO whenever are instantiated. In particular, a `MessageListenerAdapter` is bound to the event INSTANTIATE_FINISH (issued after all resources are allocated and configured) calling the method `receiveNSR`, which starts the parsing process iterating through these VNFR and check if they contain requirements on the network, this means they at least contain one quality of service class inside one of their VLDs.

If none of the VNFRs part of the received NSR contain any NSE policy, the complete NSR is ignored to avoid unnecessary further treatment. In case at least one VNFR included in the NSR contains requirements on the network level, the NSR will be forwarded to the `Core` class, taking care of applying the needed policies on the NFVI. Summarizing the functionality of the `TemplateProcessing` class is to filter out those NSRs which do not need to be handled, and forwarding to the core module the VNFRs requiring dedicated network resources.

In order to extract the minimal information needed to realize the network requirements, the core module first needs to extract from the VNFRs the service class defined in at least one of their VLDs. Afterwards the Core Module should collect the necessary information to contact the related driver responsible for the specific VNFR. Therefore it should poll the NFVO for information how to reach and access the NFVI by using the id of the VIM of the specific VNFR. With the information from the VIM which includes credentials as well as the type of virtualized infrastructure the Core Module can now decide which driver to choose for pushing the network requirements onto the NFVI. Those drivers are used by the Core Module for communicating with the available components at the NFVI able to perform operations for enforcing bandwidth requirements at the physical level.

[41] Specific values provided in the table are just example ones. Different values can be configured while instantiating the NSE component

6.6.3.1 CMA and its Driver

The CMA exposes a well-defined interface towards the NSE and acts as a wrapper for potentially any kind SDN Controller. It is implemented in `Python` and utilizes the Bottle Python web framework[42] to expose northbound REST APIs enabling an independent programming language interaction with any tool that allows enforcement of QoS related configuration capabilities.

In this work, the CMA provides a high-level APIs for enforcing QoS policies and it is composed by three sub-modules, namely:

- The API module that exposes REST endpoints to the NSE to instantiate queues, define flows and retrieve the updated topology of the distribution of virtual resources inside the data center.
- The Core module that acts as broker to forward the requests to the right clients implementation along with the information retrieved.
- The Clients module which locates the implementation of clients used by the Core module to interact with the client-specific endpoint.

The CMA's Core module receives and parses the requests for applying QoS enforcement. In addition, it checks if all the requested resources are available by using the southbound layer to retrieve all the information. In case of feasibility, the same interface is used to configure the QoS parameters on the correct instance(s) among the controllable virtual switch instances. The southbound layer uses the OpenStack python clients to retrieve the data of all VMs that compose the network service and their network topology (such as mapped port to the virtual switch, and port number in the virtual switch).

6.6.3.2 OpenStack Neutron Driver

The OpenStack Neutron Driver acts as an intermediate between the NSE and the OpenStack's network service, Neutron. To be able to push the network requirements such as bandwidth limitations to a VNF using Neutron, a mapping between the information taken from the VNFR and OpenStack data models needs to be done. The necessary QoS policies together with their bandwidth limitation rules can be initialized or reused as well as attached to the virtual link as defined in the VNFR.

6.6.4 Service Function Chain Orchestrator (SFCO)

The SFCO in J2EE as a standalone spring boot `Java` application. It interacts with the NFVO and monitoring plugin over the message bus, and with the OpenDayLight (ODL) SDN Controller through its northbound REST APIs. The SFCO relies on

[42] http://bottlepy.org/docs/dev/index.html

the extended OpenStack version supporting SFC provided as part of the open source OPNFV[43] project[44].

Therefore, assuming that the SDN Controller provides SFC data plane, the role of the SFCO is to interact with it for instantiating specific SFPs. The SFCO uses the `Java` SDK module for registering to specific lifecycle events (INSTANTIATE_FINISH, RELEASE_RESOURCE_FINISH, HEAL, and SCALED).

Once a NS is instantiated, the SFCO receives an event containing its NSR, and extracts the VNFFG required for setting up the SFPs. For deploying the SFC instances the SFCO uses the ODL REST API.

One of the advanced features provided by the SFCO is to work in combination with the FMS and AES in re-establishing a certain SFP whenever a failure occurs, or a VNF is scaled out/in.

For instance, once a VNFC instance failure occurs, the FMS triggers either the heal or the switch to standby operation. In case of heal operation, the NFVO publishes the Heal lifecycle event on the message bus, including the corresponding VNFR payload. The SFCO consumes the event parsing the VNFR and starting searching the SFPs involving the failed VNFC instance in order to update it based on the SF selection algorithm configured at runtime. Then it updates the SFC Classifiers with the updated SFP.

6.7 The Open Baton Bootstrapping CLI

Being mostly implemented with common programming languages and technologies, Open Baton can be easily installed and executed on any kind of OS. Building and starting a component directly using its source code is rather easy using the gradle framework. The only requirement is to have a running instance of RabbitMQ, so that such component can communicate with the NFVO. A typical workflow execution for starting an individual component is shown in 6.3.

Listing 6.3: Example of the workflow for building and starting an individual component (in this case the dummy-vnfm-amqp)

```
1 git clone https://github.com/openbaton/dummy-vnfm-amqp.git
2 cd dummy-vnfm-amqp.git
3 ./gradlew build run
```

In order to simplify the installation of the Open Baton framework on Ubuntu, CentOS, and Debian OSs, a bootstrapping CLI has been provided allowing, with a single command, to perform the installation and complete configuration of an all-in-one Open Baton instance. A minimal installation requires 2 GB of Random Access

[43] OPNFV, online:https://www.opnfv.org

[44] In this work, the OPNFV Apex installer has been selected as it provides support to SFC: https://wiki.opnfv.org/display/apex

Memory (RAM), 2 CPU cores, and around 2 GB of disk space[45]. More information about the boostrap CLI are available in Section B.3.

6.8 Conclusion

This chapter has introduced the Open Baton project as the open source reference implementation of the final version of the MANO4X framework architecture presented in Chapter 5 – Specification of the MANO4X Framework. The implementation followed the same agile approach presented for the design process, including major/minor releases accordingly to the semantic versioning process. Several software components have been implemented and publicly released as part of the Open Baton GitHub organization using an open source licensing model. A single bootstrapping procedure has been provided in order to simplify the setup of the complete environment.

[45] It depends also on the installation version selected

Chapter 7

Validation and Evaluation

The present chapter, first presents an overview of the adoption of the methods and technologies designed and developed during this research work within large scale ICT projects. Next, a set of experimental use cases are presented, including their results indicating how well the proposed solution performed.

Particular attention is given to the analysis of the results based on the list of features presented in Chapter 3 – The MANO4X Requirements and Features Analysis. Finally, the Open Baton framework is compared against other existing similar open source platforms.

7.1 ICT Project Validation and Dissemination

This subsection focuses on the research results generated during the different phases of research presented in The Design Evolution of the MANO4X Framework and contributed towards international ICT research projects.

7.1.0.1 ICT BonFIRE

BonFIRE[1] was a research project funded by the European Commission under the 7th Framework Programme (FP7) Future Internet Research Experimentation (FIRE) initiative. It was launched in 2010, with a duration of 36 months, and ended in 2013. The main objective was to provide a multi-site cloud testing facility, enabling third-party developers to do experimental research across an heterogeneous cloud infrastructure, composed by six sites across Europe. Most of the BonFIRE sites adopted, and further adapted, OpenNebula[2] as IaaS solution, on top of which a set of additional BonFIRE core components were designed in order to provide access as a service to external experimenters. Network connectivity across sites was realised over public internet, as well as VPN connections. Basically VMs could connect across sites either via public IPs or via private IPs part of the internal VPN network. The BonFIRE infrastructural resources were exposed via a REST interface, based on OCCI[127].

During the last year of the project, the author contributed with the work conducted during the initial phase of his doctoral research, particularly on topics related

[1] http://www.bonfire-project.eu/
[2] https://opennebula.org/

with elastic scalability[13][129][128]. In BonFIRE, there have been three possible approaches for supporting scalability: *manual*, *programmed* and *managed*.

The first two approaches, the *manual* and the *programmed* one, were supported by the BonFIRE core components through a web-based portal allowing various performance related metrics to be observed and the BonFIRE Resource Manager's OCCI API through which resources may be created or deleted.

The *managed* approach was achieved integrating the EE, providing *"Elasticity as a Service (EaaS)"*. As already presented in Section 4.2.2, the functionality of the EE is to automatically increase or decrease the number of compute resources based on autoscaling policies. The configuration of the EE autoscaling policies was realized putting some contextualization variables in theOCCI request. Among others, the experimenter could set the number of minimum and maximum instances to be created at runtime during the experiment. Furthermore, the name of the created image and the instance type to be used for deploying compute resources had to be specified. In addition, the experimenter had to configure, using contextualization, the KPI that should be taken into account as alarm condition.

7.1.0.2 ICT Mobile Cloud Networking (MCN) Project

MCN[3] is a large scale project funded by the European Commission under the FP7 ICT initiative. It was launched in 2012 for the period of 36 months, it ended in 2016. The author contributed with the work conducted during the intermediate and final phase of his doctoral research, particularly focusing on the cloudification of the IMS architecture, resulting in the IMS as a Service (IMSaaS) concept[143][131][133][121]. The main objective of the MCN project was to exploit cloud computing technologies for future mobile network deployments and operations[164]. The main assumption of MCN was that future TSP's infrastructures would comprise micro and macro cloud-based datacenters, on top of which network functions will be deployed as a service. MCN proposes an high-level architecture[130] for managing and orchestrating network services across a multi-site cloud-based infrastructure.

One of the most relevant aspects of the MCN architecture is represented by the dynamic service composition, having as main objective an end-to-end mobile core network comprising the RAN and EPC core elements, as well as end-users applications.

The MCN core model is based on OCCI, providing general interoperability among different kind of solutions, paving the way to a fast adoption in different domains and deployment scenarios. The MCN framework solutions are also "functionally" compliant with the ETSI NFV MANO specifications in order to further facilitate its adoption and easy integration. In particular, as better detailed in Section V, our framework covers many of the functional requirements of ETSI NFV MANO, but with a slightly different architecture of solution.

There have three major contributions driven by the author during the MCN activities:

[3]https://github.com/MobileCloudNetworking/

1. Integration of the Open Baton framework as IMS SO.
2. IMS lifecycle management, and particularly autoscaling of Media Gateway (MGW) functional elements.
3. A fault management system for the IMS use case.

7.1.0.3 ICT NUBOMEDIA Project

NUBOMEDIA is a research project funded by the European Commission, launched in 2014 for the period of 36 months. The main objective of the NUBOMEDIA project was to design and develop an open source cloud PaaS for RTC. The author strongly contributed with the work conducted during the intermediate and final phase of his doctoral research, designing the NUBOMEDIA architecture integrating the MANO4X elements for realizing a PaaS for multimedia applications[144][145][149]. NUBOMEDIA represents also one of the catalyst projects of the final phase of research.

In NUBOMEDIA, multimedia applications are decomposed following a three tier model architecture, in which media processing elements are separated from the media control logic. A NUBOMEDIA application comprises one or more containers implementing the application logic using the Kurento APIs and (optionally) one or more supporting services (like Database Management System (DBMS), etc.). The NUBOMEDIA PaaS system designed and developed in the context of the NUBOMEDIA project, provides support for managing and orchestrating independently the two layers of the NUBOMEDIA application, meaning the media services and application services. The NUBOMEDIA system architecture is depicted in Figure 7.1.

This architecture complies with the ETSI NFV specification. At a very high-level perspective, and following a top down approach, these are the core components:

- NUBOMEDIA PaaS being the intermediate level between the infrastructural resources and the users of the platform. Particularly it includes the NUBOMEDIA PaaS Manager exposing an interface to developers for deploying and runtime managing their applications. This layer also holds the NUBOMEDIA PaaS, a system hosting the applications and exposing their capabilities to the end-users.
- NUBOMEDIA Media Plane composed by the media plane capabilities whose lifecycle is managed by the Open Baton framework. Those media service entities are executed on top of virtual containers.Open Baton has been further extended with a specific VNFM, called VNFM-Elastic Media Manager (EMM), able to manage the lifecycle of the media servers and to dynamically allocate sessions on top of available instances.
- NUBOMEDIA IaaS composed by the infrastructure resources in terms of CNs providing virtual compute, storage and networking resources to the upper layers, and the VIM providing the abstracted APIs for controlling their lifecycle.

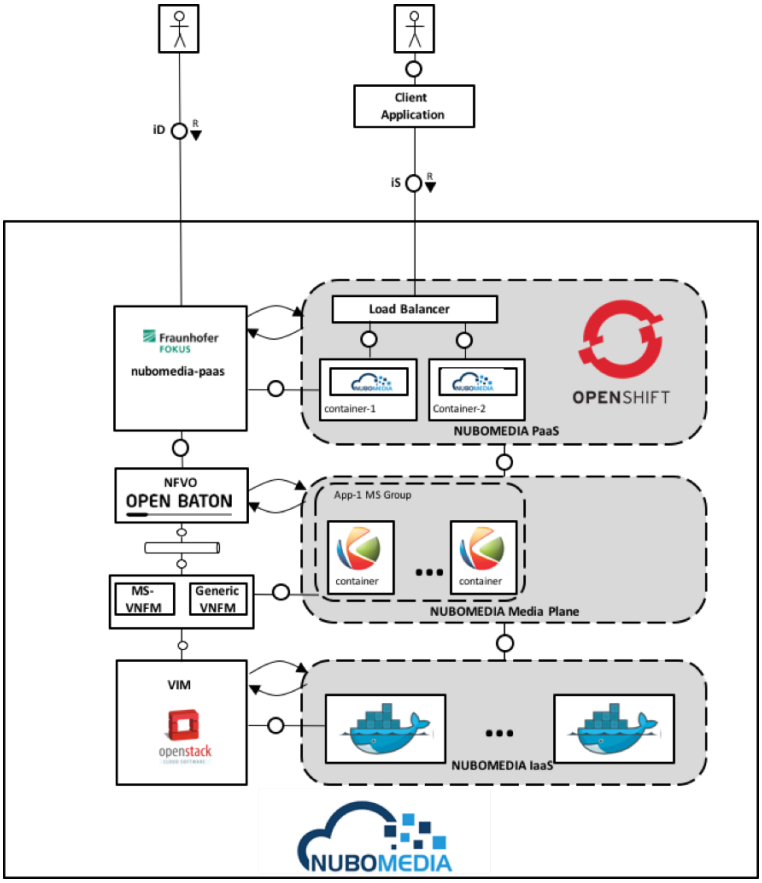

Figure 7.1: NUBOMEDIA PaaS Architecture

The NUBOMEDIA IaaS has been realized using OpenStack extended for fully supporting instantiation of Docker containers on distributed Compute Nodes.

The runtime information related to the available media server instances are provided to the application service layer dynamically. In order to retrieve dynamically information about the media service layer, the PaaS Manager injects runtime information (like the endpoint of the VNFM-EMM and additional unique identifiers) inside the containers executing the application logic as environment variables.

Concepts and methods presented as part of the VNFM domain were partially conceived in NUBOMEDIA. For instance, the VNFM-centric approach has been introduced in this project because of the need of a specific autoscaling mechanism for media applications. A cross-layer interface between the applications running on the PaaS and the VNFM-EMM has been defined: basically the VNFM-EMM is aware of the number of open sessions for a certain application, thus, whenever the capacity (defined as number of sessions per media server) is not enough anymore, it triggers scaling out/in operations.

Furthermore, the introduction of the Pool Management was also required for

speeding up the deployment process of new media server instances anytime scaling out operations were triggered.

7.1.0.4 ICT SoftFIRE Project

SoftFIRE is a research project under the Horizon 2020 (H2020) FIRE+ initiative funded by the European Commission, launched in January 2016. SoftFIRE aims at building a large-scale federated infrastructure across Europe, leveraging SDN, NFV, and 5G technologies, providing access to third parties experimenters via open calls. After an initial design phase, Open Baton has been utilized as central entity for federating, from a NFV perspective, the multi-site NFVI comprising different OpenStack-based PoP in Germany, Italy, and United Kingdom.

The contributions given by the authors to the SoftFIRE project were twofold. First, he provided his knowledge in the MANO domain while designing the SoftFIRE middleware. Second, he provided his technical knowledge in the NFV technology ecosystem integrating the Open Baton framework as the NFV MANO framework of the SoftFIRE infrastructure. The SoftFIRE middleware architecture is shown in Figure 7.2[4].

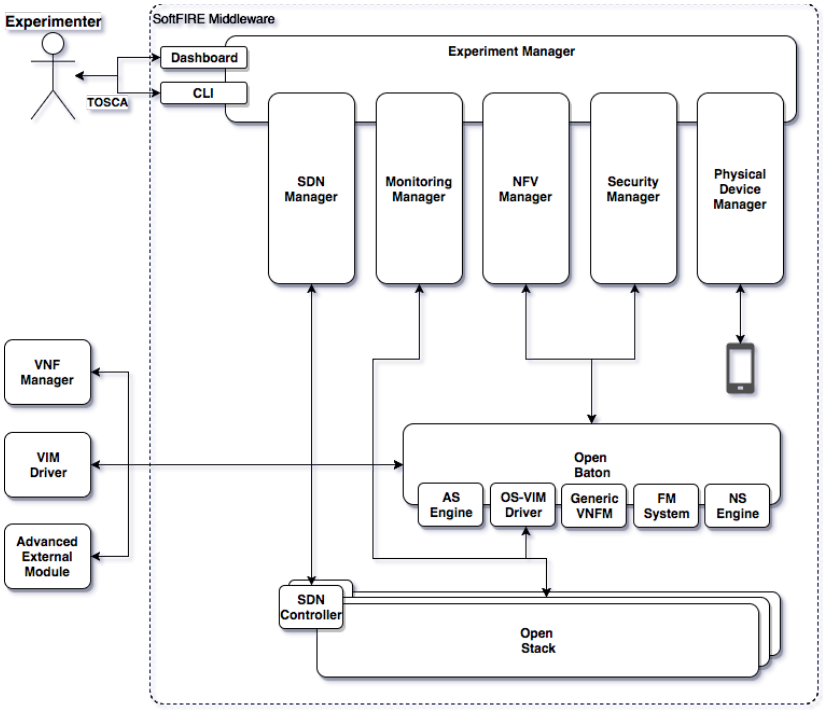

Figure 7.2: SoftFIRE Functional Architecture

The SoftFIRE middleware comprises the experiment manager as the central en-

[4]Additional concepts and ideas have been evaluated on top of the SoftFIRE infrastructure as presented in [139][140]

tity between the experimenters and the SoftFIRE testbeds. The experiment manager exposes a REST API to external experimenters to provision a particular experiment requiring heterogeneous resources at the infrastructure level. TOSCA has been selected as description language for describing resources which needs to be deployed.

7.1.0.5 5G Berlin and the Fraunhofer FOKUS 5G Playground

The Open Baton project has been one of the central components within the Fraunhofer FOKUS 5G Playground[5], providing the MANO functionality required for future 5G based networks. Established in 2015, the 5G Playground represents a state of the art 5G testbed for performing applied research using 5G-ready technologies. Figure 7.3 shows the high-level architecture of the 5G Playground.

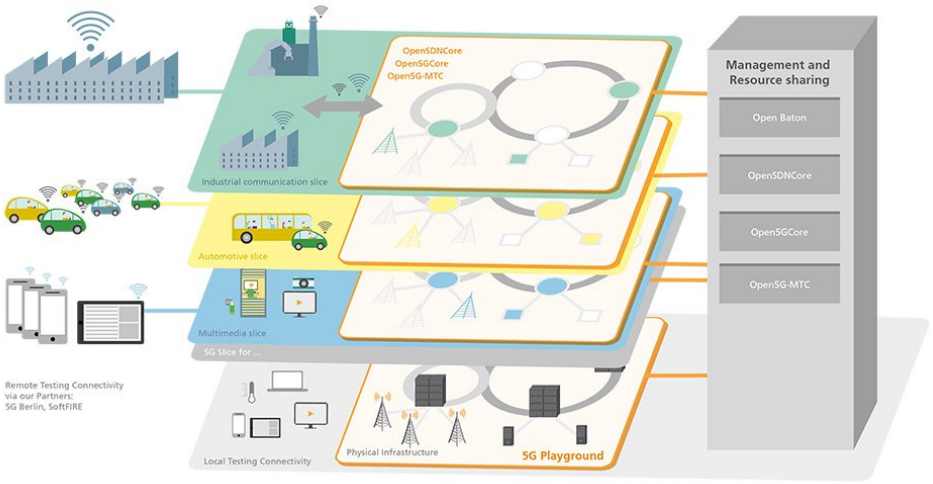

Figure 7.3: The 5G Playground high-level architecture

In addition to the Open Baton framework, the 5G Playground comprises the following set of technologies:

- Open5GCore[6] as reference implementation of the 3GPP EPC architecture.
- OpenIMSCore[7] as reference implementation of the 3GPP IMS architecture[165].
- Open5GMTC[8] extending the Open5GCore set of components for addressing Machine Type Communication (MTC) scenarios.

[5]https://www.fokus.fraunhofer.de/go/en/fokus$_t$*estbeds*/5g$_p$*layground*
[6]https://www.open5gcore.org/
[7]http://openimscore.org/
[8]https://www.open5gmtc.org/

- OpenSDNCore[9] providing an SDN backhaul.

The success gained by the Open Baton framework increased the public visibility of the 5G Playground. Several collaborations with industry as well as academia took place after the launch of the testbed. In particular, a new initiative, named 5G Berlin[10], was launched as a large collaboration between the two Fraunhofer research institutes located in Berlin (FOKUS and Heinrich-Hertz-Institut (HHI)), two major TSP in Germany (Vodafone and Deutsche Telekom), and a local organization, Berlin Partners[11]. 5G Berlin, aims at providing a large scale testbed across the city of Berlin, which can be utilized by third parties (mainly startups and Small and Medium Enterprisess (SMEs)) for experimenting with state-of-the-art technologies in this domain.

7.2 Experimental Use Case Validation

This section presents a set of use cases selected for validating the usability and effectiveness of the methods and concepts outlined in the course of this research work. The scope of this section focuses on validating requirements and features identified in Chapter 3 – The MANO4X Requirements and Features Analysis. Each scenario intentionally focuses on validating only a subset of the features provided, and most of the results presented have been either published as part of scientific publications or technological deliverables of ICT research projects. The following Table 7.1 presents a mapping between the use cases presented, and the set of features under evaluation.

Most of the use cases presented are also addressing the following features: *MANO-1 - Inventory*, *MANO-2 - Lifecycle management*, *MANO-3 - Multi Tenancy*, *MANO-4 - OpenStack support*, *MANO-7 - VNF Placement*, *MANO-8 - Support for heterogeneous VNFs*, *MANO-12 - Manual Scaling support* (indirectly validated using the AES component) *MANO-17 - Integration with existing OSS/BSS components*, and *MANO-18 - User Tools*. Additional validation use cases addressing the remaining features (*MANO-5 - Support for heterogeneous NFVI*, *MANO-6 - Multi-site NFVI*, and *MANO-16 - SFC management support*) presented in Section 3.3 have been presented in scientific papers and international demonstrations as presented in Chapter 8.

Most of the use cases presented follow a similar structure as:

- *introduction*: provides a high-level overview of the selected scenario.
- *testing setup*: describes the testing environment, focusing on the hardware resources and software components selected for executing the experiment.
- *testing scenario*: details the scenario which has been executed in order to validate the proposed functionality.

[9] https://www.opensdncore.org/
[10] http://www.5g-berlin.org/
[11] https://www.berlin-partner.de/en/

Use Case	List of Features
vEPC Deployment	*MANO-8 - Support for heterogeneous VNFs*, *MANO-9 - Generic VNFM*
Network slicing	*MANO-15 - Network Slicing support*, *MANO-13 - Autoscaling support*, *MANO-11 - Monitoring support*
Multimedia applications	*MANO-13 - Autoscaling support*, *MANO-11 - Monitoring support*, *MANO-10 - Support for specific VNFM*
Autoscaling	*MANO-13 - Autoscaling support*, *MANO-11 - Monitoring support*
Fault management in an OpenStack environment	*MANO-14 - Fault Management support*
Juju integration	*MANO-10 - Support for specific VNFM*
Continuous integration	Covering most of the features presented

Table 7.1: Mapping between the Use Cases Presented and the Set of Features under Evaluation

- *functional validation and performance measurements*: presents the functional and non functional results collected while executing the proposed testing scenario in the testing environment.

All the experiments presented in the following subsections have been executed within the Fraunhofer FOKUS laboratories. Here it is provided an overview of the characteristics of the individual testbeds used:

- "orange-box-testbed": As testing environment it has been employed the Tranquil PC V4n box[12], typically known also as the *"Ubuntu Orange box"*[13]. It is equipped with 11 hot-swappable nodes, all with Intel® Core™i5 5300U processors (with 4 cores) and 16GB of DDR3 RAM. Node-0 acts as the jump-host where the OpenStack controller services are installed, and the other nodes act as compute nodes. OpenStack was installed using the OPNFV JOID installer (Colorado release).

- "micro-dc-testbed": comprising one or more Lenovo™ThinkCentre M93p with a 64-bit architecture, an Intel® Core™i7-4785T processor with 2.20 GHz and

[12]https://www.tranquilpc.co/v4n
[13]https://insights.ubuntu.com/wp-content/uploads/DS_The_Orange_Box.pdf

16GB RAM. Typically, one or more hosts were used as NFVI, running an all-in-one OpenStack, installed on a CentOS version 7.2 OS using the RedHat Packstack installer[14]. Another one was used for executing the Open Baton framework, installed using the bootstrap procedure on an Ubuntu 14.04/16.04 OS, the components needed in the specific validation scenario.

7.2.1 Virtualized EPC (vEPC) Deployment

This subsection focuses on validating the effectiveness of the proposed lifecycle management method. The main objective of this use case is to validate the lifecycle management operations executed by the NFVO and Generic VNFM while deploying a complex network service, like the vEPC one.

7.2.1.1 Testing Setup

As testing environment has been used the "micro-dc-testbed".

7.2.1.2 Testing Scenario

The vEPC use case has been selected as testing scenario. The Fraunhofer FOKUS Open5GCore project has been utilized as the reference implementation for developing this scenario. The network service deployed was composed by multiple VNFs having several dependencies among each other. In particular, the network service is composed by a DNS, implemented using the open source Bind9[15], an Mobility Management Entity (MME), an HSS, an emulated eNodeB, a Serving Gateway (SGW) and PGW, an Internet Gateway, an User Equipment/Endpoint (UE) and a Mobility Manager (MM).

It is important to clarify that in this testing scenario it was used a customized *qcow2*[16] disk image including already the software required by the VNFs, mainly because the binaries of the software used were not available for being downloaded on demand during the deployment phase. The NFVO was evaluated calculating the time required for instantiating the network service selecting its NSD published in the NFVO catalog. Six deployments were executed in series and the measurements of multiple steps were collected.

7.2.1.3 Functional Validation and Performance Measurements

From a functional perspective, the lifecycle management operations have been validated using the UE component: a set of emulated attachment and detachment procedures were executed to ensure that the vEPC deployed was functioning correctly. The values shown in Figure 7.4 represent the average of the six consecutive deployment operations.

[14] https://www.rdoproject.org/install/packstack/
[15] https://wiki.debian.org/Bind9
[16] https://en.wikipedia.org/wiki/Qcow

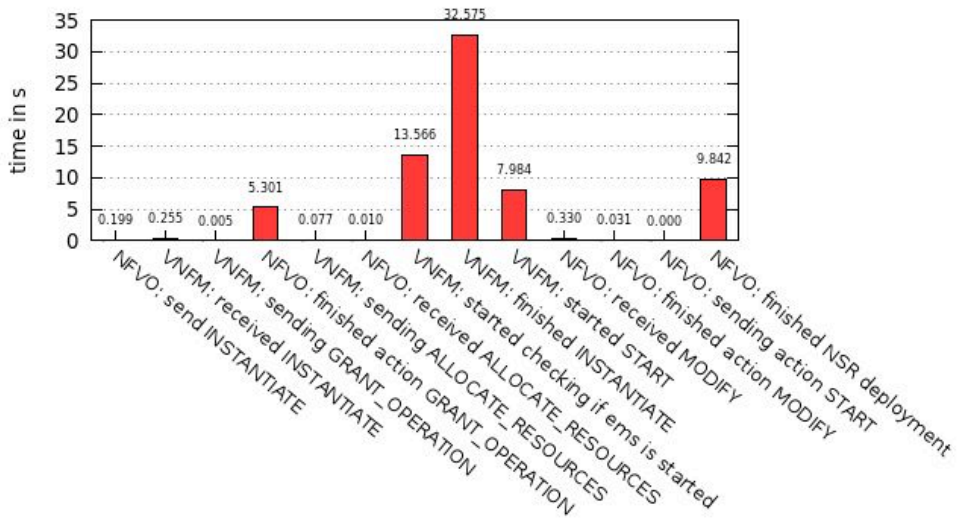

Figure 7.4: Performance Measurements of the vEPC Deployment Scenario

It is important to clarify that the execution of a particular lifecycle event done by a VNFM may differ among different VNFs composing the network service. Results presented in Figure 7.4 for each individual lifecycle event, represent an average of all the VNFs included in the network service.

As it can be noticed the most expensive operation (in terms of time) during the lifecycle management is represented by the INSTANTIATE lifecycle event. This is mainly due to the fact that during this lifecycle event there are two major operations executed by the NFVO and VNFM:

- instantiation of the virtual resources required by the VNF: this time differs between different technologies used at the NFVI level.
- execution of the installation scripts for installing the VNF software inside the virtual container: this time depends primarily on the mechanism used for installing the software. For instance, in case of source code based installation, compilation time may have strong influences on the overall results, while in case of binaries installation, this time can be reduced further, but the overall results are still dependent from the networking performances of the virtualized environment.

As a global result it has been obtained that the time required for deploying a vEPC instance is about 70s.

7.2.2 Network Slicing

In order to evaluate the NSE, it has been realized a scenario very similar to the one foreseen by the 5G white paper about network slicing, presented in Section 2.4:

multiple mobile core networks were deployed in parallel on the same physical infrastructure, but were customized for supporting different verticals' requirements (e.g., mMTC, uRLLC, etc.).

The main objective of this use case is to validate the concepts designed and developed within the context of the NSE component. In particular, the main focus is to validate that the NSE is capable of enforcing networking requirements on the physical infrastructure. Although the main objective of this use case was to evaluate the effectiveness of the NSE capabilities on enforcing networking requirements on the physical infrastructure, autoscaling policies were applied to one of the network slices deployed in order to validate also scenarios in which multiple OSSs are employed together. Similar results obtained with the execution of this scenario have been published as part of [142]. Results obtained during the intermediate phase of research addressing the network slicing topic were published in [135]

7.2.2.1 Testing Setup

The "orange-box-testbed" was configured with five different tenants, for logically separating different slices deployed on the cluster from control functional elements (i.e. Open Baton and Zabbix):

1. *admin*: comprising two VMs. One hosting an Open Baton installation installed using the bootstrap procedure, including the NFVO, the Generic VNFM, the OpenStack VIM driver, the AES, the Zabbix monitoring plugin, and the NSE on a VM with Ubuntu 14.04, 4 Virtual CPUs (vCPUs) and 8 acsGB of RAM. The other one hosting the Zabbix monitoring system installed on a VM with Ubuntu 14.04, 2 vCPUs and 2 GB of RAM.

2. *support-services*: comprising 4 VMs (using a flavour of 1vCPU and 512 MB of RAM), of which 2 hosting emulated eNodeBs and the UE/MM acting as testing tools for the deployed mobile core networks, and the other 2 hosting different traffic endpoints, used by the testing tools as traffic anchors.

3. *slice-low-latency*: used to deploy the so called *"low-latency"* network service, being a customized version of the mobile core network optimized for low-latency use cases. Each VM used for hosting VNFs has a flavour of 1vCPU and 1 GB of RAM.

4. *slice-ultra-reliability*:used to deploy the so called *"ultra-reliability"* network service, being a customized version of the mobile core network optimized for scaling horizontally the data-plane. Each VM used for hosting VNFs has a flavour of 1vCPU and 1 GB of RAM.

5. *slice-iperf*: used to deploy a network service comprising two VNFs implemented using the iperf tool, and configured to act as server and client. Each VM used for hosting VNFs has a flavour of 1vCPU and 1 GB of RAM.

Some of the components of the Open5GCore toolkit were selected as VNFs in the *slice-low-latency* and the *slice-ultra-reliability*. More details about the number of VNFC instances deployed will be provided in the next subsection.

7.2.2.2 Testing Scenario

The NSE was validated deploying three different network services running on the same physical infrastructure, and sharing the same physical resources. A more detailed description of the three network services deployed during this experiment can be found below:

- *slice-low-latency*: this service emulates the mMTC scenario requiring low latency. It comprises a set of VNFs provided by the Open5GCore project, customized by grouping together multiple VNFCs in single VNFs in order to reduce the communication latency.
- *slice-ultra-reliability*: this service emulates the uRLLC scenario where bandwidth and reliability play a central role. Also in this case, a set of VNFs provided by the Open5GCore project have been customized for supporting ultra reliability, achieved by splitting components as much as possible for supporting horizontal scalability. In this scenario, an autoscaling policy was added to the MME VNF (being the component able to maintain an overview about the number of attached subscribers to the core network) so that scale out/in procedures could be applied to the switch VNF (comprising the merged PGW and SGW). A scale out action was executed whenever the number of attached users crossed the value of 60 users per available VNFC instance.
- *slice-iperf*: this service emulates a best effort network service, with the main objective of flooding the physical network resources with TCP/User Datagram Protocol (UDP) traffic in order to validate if the other slices get affected. This service was implemented using the iPerf tool[17], following a client/server approach where the client tries to send a predefined amount of data to the server.

For ensuring traffic isolation, thus validating the NSE capabilities, a QoS policy has been applied to virtual link of the PGW and SGW user network (net-a) as show in tab:qos[18].

Initially, for the *slice-iperf* slice was not applied any particular policy, thus the communication between VNFCs comprising this slice can be considered best effort. For measuring latency between components within a slice, one user from the support services tenant was attached to each slice, and frequently trying to ping the related traffic endpoint VM. The test scenario had a total duration of one hour

[17] https://iperf.fr/

[18] Dimensioning of policies has been done considering the physical limits of the networking environment in the "orange-box-testbed"

Network Service	VNF	Quality	Traffic/User
slice-low-latency	sgwu_pgwu	BRONZE - 1024 Kbps	1,8 Kbps
slice-ultra-reliability	sgwu_pgwu	GOLD - 102400 Kbps	125Kbps

Table 7.2: Virtual Link Quality per Network Service / VNF

and 30 minutes. After the successful deployment of the different slices, a benchmarking tool[166] was employed for emulating different attachment and detachment procedures (240 users each 30 minutes) against the mobile core network deployed in the *slice-low-latency*. After 30 minutes, the benchmarking tool started sending an increasing number (12 users every 5 minutes) of attachments to the mobile core network deployed in the *slice-ultra-reliability*. The autoscaling condition for the *slice-ultra-reliability* was set to a threshold of 12 users.

After 30 minutes a network bandwidth test is started by the VNFCs of the *slice-iperf*. The client starts sending UDP traffic which is increased by 90 Mbps every 5 minutes. Once it reaches 360 Mbps it starts reducing the traffic again in the same steps until it reaches 0 Mbps. As last step of this scenario a QoS policy (SILVER) was applied to the *slice-iperf* and same tests aforementioned were re-executed.

7.2.2.3 Functional Validation and Performance Measurements

Figure 7.5 shows the results gathered executing the testing scenarios described in the previous section. It shows the monitored data of the different network slices each condensed into two graphs displayed in a single row. The first one represents the *slice-low-latency*, the middle one slice shows the *sliceultra-reliability* and the last one presents the *slice-iperf*.

The left graph in the first two rows combines the number of attached users (total and average) to the mobile core network together with the number of switch VNFCs instances. The right graph of the first two rows combines the information of the outgoing network traffic from the mobile core network and incoming in the related traffic endpoint together with the experienced latency for the respective users. The last row's graphs show the network traffic generated within the *slice-iperf*.

As expected, in the time from 11:05 to 12:05 it can be seen that there are no interference among the different slices. The mobile core network deployed in the *slice-low-latency* following the distinct attach-detach pattern, while the one deployed in the *slice-ultra-reliability* attaching the users and scaling the switch VNF.

This situation changes when Iperf NSs, deployed without any network slicing policy, starts generating traffic. The spikes of the network traffic among the Iperf-client VNF and Iperf-server VNFC instances at about 12:07 results in a spike of the experienced user latency in the 5G core designated for ultra reliability. Same situation happens at 12:25 and 12:27 where both slices are affected by the spikes of network traffic among the VNFC instances of the *slice-iperf*. This can be ex-

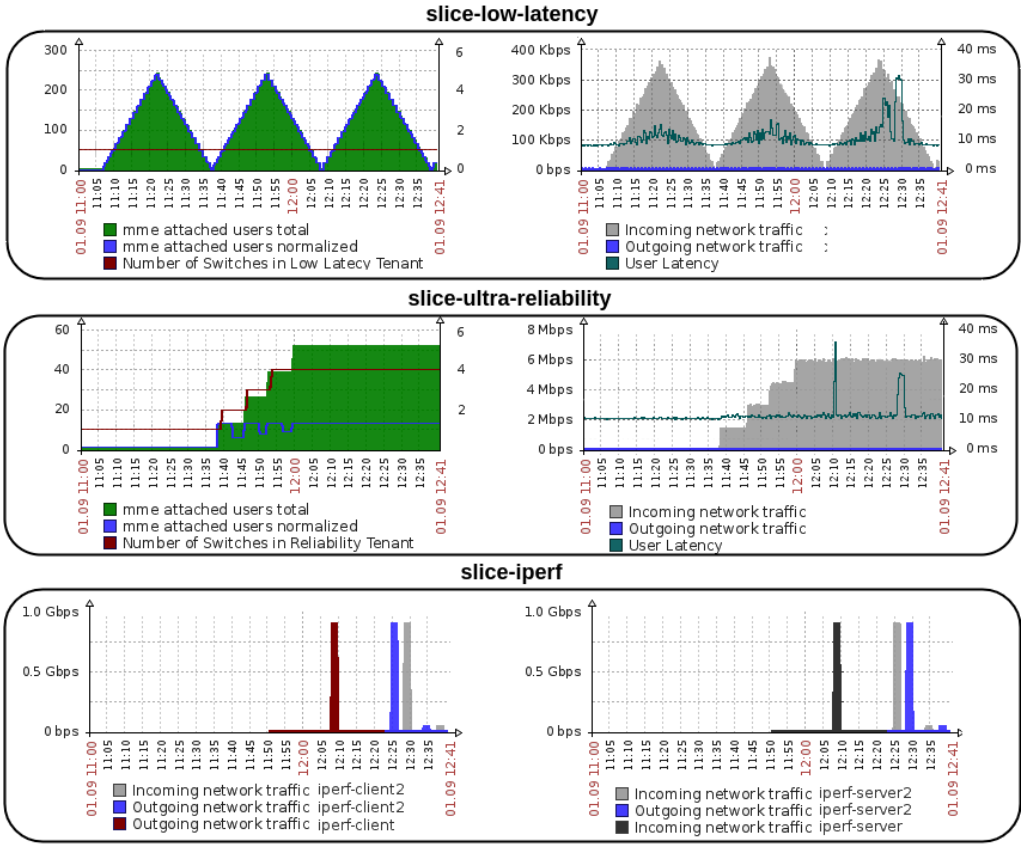

Figure 7.5: Performance Measurements of the Network Slicing Scenario

plained by the fact that the outgoing physical network interface of the compute nodes reach the total available bandwidth limit, thus generating a congestion which delays Internet Control Message Protocol (ICMP) packets and user traffic in the other two slices.

From 12:35 to 12:41 the last experiment is repeated but allocating the SILVER QoS policy to the *slice-iperf*. This time, the generated traffic does not burst the physical limits of the networking elements and there are no side effects on the experienced user latency among other network slices. It can be concluded that the allocation of bandwidth limitations, in compliance with the physical limits of the NFVI, allows the creation of isolated network slices.

7.2.3 Autoscaling

This scenario focuses on validating the methods and related implementation of the AES. In particular, are provided three different scenarios:

- *Emulated scale-in procedure*: focuses on validating the scale-in algorithm proposed in Section 4.2.3.2.

- *Web Service network service*: focuses on validating the effectiveness of the AES implementation with a typical OTT scenario.
- *vIMS network service*: focuses on validating the effectiveness of the AES implementation with the reference vIMS scenario.

Parts of the results presented in the following subsections have been published in [145][167]. In addition, similar other experiments have been conducted over the past years with the objective of validating the autoscaling approaches defined during this dissertation, with results published in [144][131][14][129][133][138].

7.2.3.1 Emulated Scale-in Procedure

In order to evaluate the scale-in algorithm proposed in Section 4.2.3.4 two different scenarios have been emulated using the Matlab[19] multi-paradigm numerical computing environment. Two different scenarios have been simulated considering a different number of VMs (scenario-1 with 10, and scenario-2 with 100) as initial state of the scale-in procedure. Assuming that the load gets equally distributed among VNFC instances comprising a VNF, a decreasing CPU load average in a period of 300 seconds has been taken as input for the algorithm presented in Section 4.2.3.4. The expected behavior is that applying the proposed algorithm, VMs will be removed earlier than using a classical threshold based approach.

Figure 7.6 and Figure 7.7 show on the left graph the average of the CPU utilization, and on the right graph the number of VMs per time for the scenario-1 comparing a common trigger-based approach (blue line) with the scale-in algorithm proposed in this dissertation (red line).

As it can be noticed from the results collected, there is a large optimization obtained by utilizing the proposed scale-in algorithm. In particular, in both cases, VNFC instances (VMs) are terminated earlier in time, thus providing better resource utilization over time.

7.2.3.2 Web Service Network Service

The following tests were done in order to show the reliability, stability and resource efficiency of the AES.

Testing Setup: The "micro-dc-testbed" was utilized as testing environment.

Testing Scenarios: The testing scenarios selected in this case include a network service, shown in Figure 7.8 designed as composition of two VNFs: a load balancer VNF dispatching requests (using Round Robin load balancing algorithm) to the backend VNF, referred to as the elastic-app, instantiated with one or more VNFC instances serving a simple web service. The load balancer has been implemented using the HAProxy[20] load balancer, while the elastic-app has been implemented us-

[19] https://de.mathworks.com/products/matlab.html
[20] HAProxy, online: http://www.haproxy.org/

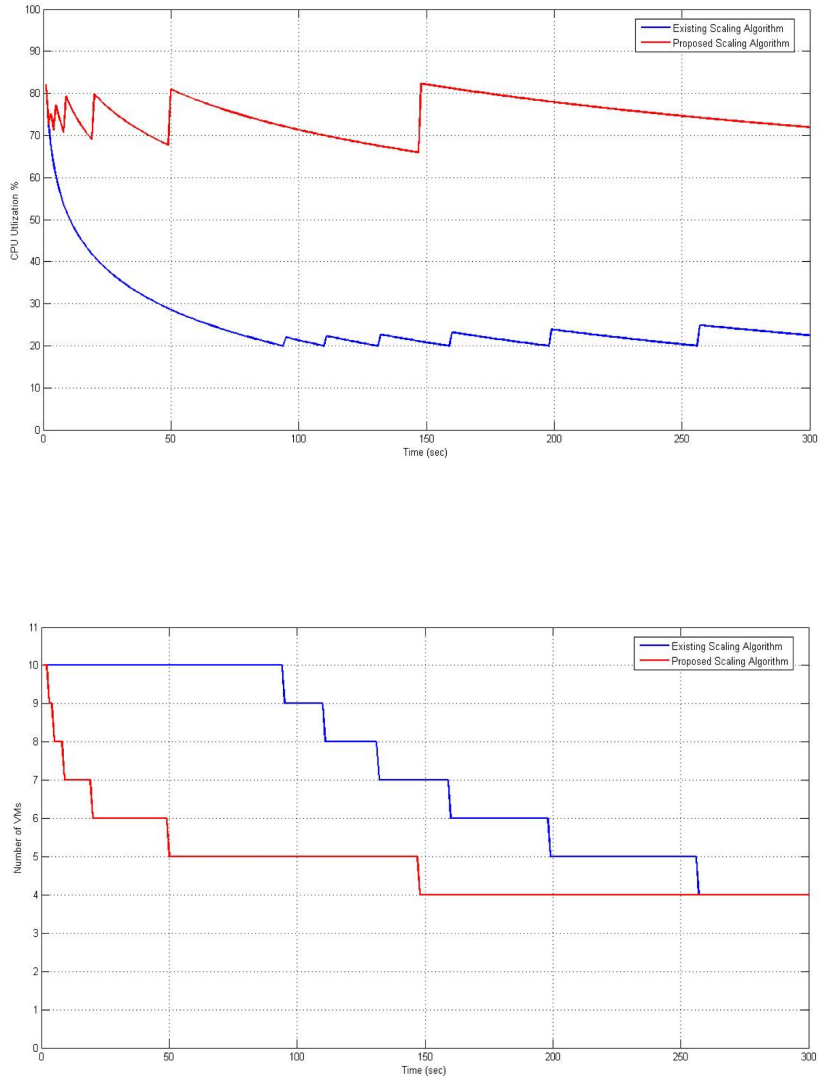

Figure 7.6: Emulated Measurement Results for Scenario-1

ing the apache2 web server exposing a simple web application that for each requests starts a shell process[21] consuming CPU cycles. A load generator (Httperf[22]) was used in order to simulate different traffic patterns emulating user requests.

The AES functionality has been validated by three different testing scenarios:

1. *NFVO-centric*: validating the scale out procedure when using an NFVO-centric approach.

[21] The linux stressapptest application available at: https://github.com/stressapptest/stressapptest
[22] http://www.labs.hpe.com/research/linux/httperf/

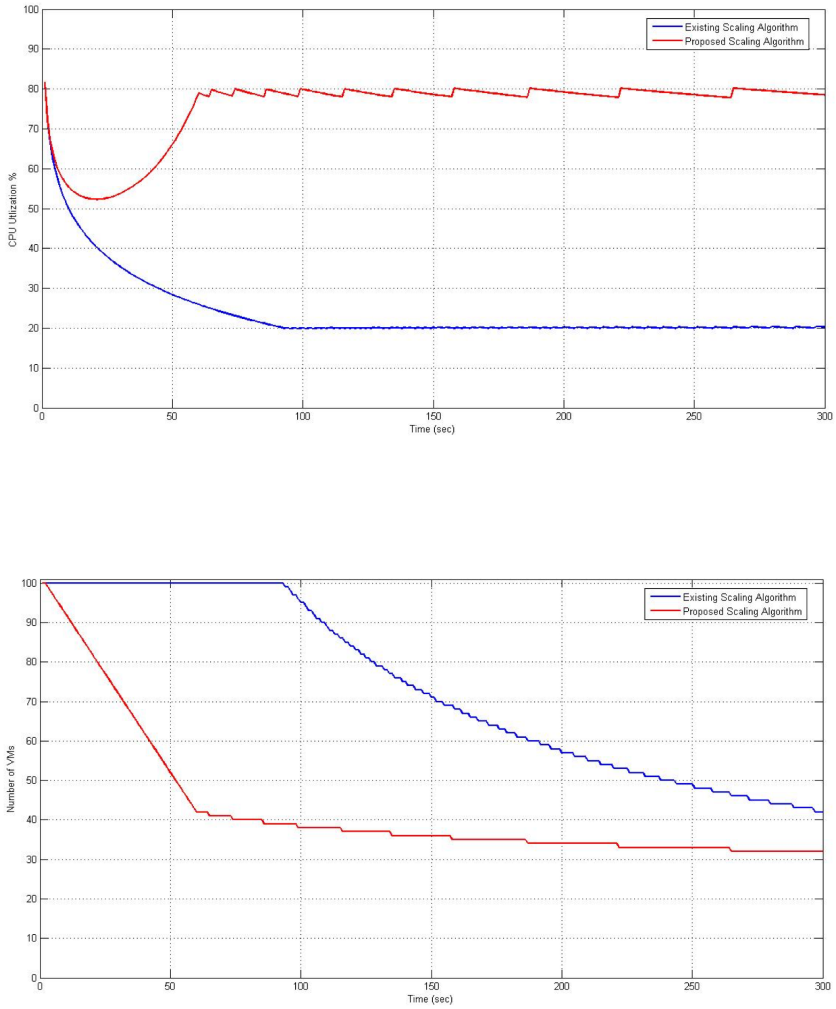

Figure 7.7: Emulated Measurement Results for Scenario-2

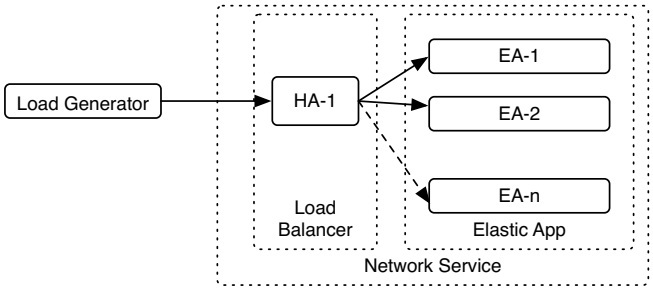

Figure 7.8: Web Service Network Service

2. *VNFM-centric*: validating the scale out procedure when using an VNFM-centric approach.

3. *Long run*: validating the AES on a scenario over a long period of time.

In the first scenario, the AES has been executed as an independent component, while in the second scenario the AES has been integrated within the VNFM. Moreover, the second scenario validates also the pool mechanism proposed as an extension for optimizing scale out procedures.

In both scenarios, tests were repeated ten times in order to gather averaged measurement results. The collection of measurements started from the activation of the detection procedure (measurements fetching and alarm detection), and ended after the scale-out procedure was executed (indicated by the execution of the cooldown procedure). One of the major differences between the first and the second scenario is represented by the different way VNFC are deployed. In the NFVO-centric scenario, it was used the generic VNFM, thus instantiation time is also influenced by the time needed for installing the generic EMS. While in the VNFM-centric scenario, it was employed and further adapted the VNFM-EMM designed and developed in the context of the NUBOMEDIA project, making use of cloud-init functionalities for provisioning VNFC instances. In this last scenario was validated also the possibility of scaling out multiple VNFC instances in a row.

The third testing scenario covered a long run experiment with a duration of eight hours with increasing and decreasing number of requests per minute following an sine-based curve. The conditions and actions for scaling-out and scaling-in were defined in *autoscaling policy* as follows:

- scale-out an instance if the average of all measurements of metric *'CPU idle time'*[23] is less than 40%.

- scale-in an instance if the average of all measurements of metric *'CPU idle time'* is greater than 60%.

Both conditions were checked every 60 seconds, and if scaled successfully, the AES waits for 120 seconds (defined as cooldown period) before accepting the next scaling request. To satisfy the current number of requests the descriptor and corresponding *autoscaling policy* are able to scale in to a VNFC instance whereas scaling-out is limited by eight components. Theoretically, if the number of requests are close to zero, a single VNF component should be enough to handle the load produced. If the number of requests are close to 10000, eight VNFC instances are close to be fully exhausted caused by the load produced. This means more than 10000 requests per minute are not supported by this VNF and leads to an overloaded system.

Performance Measurements: Starting with the first scenario, as shown in Figure 7.9 most of the time is required by the instantiate procedure.

[23] https://en.wikipedia.org/wiki/Idle_(CPU)

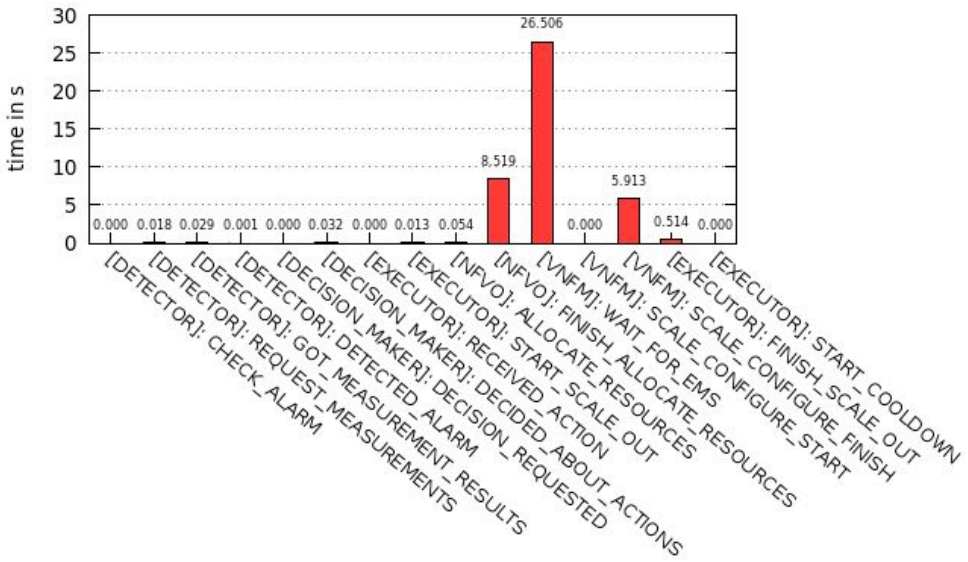

Figure 7.9: Measurement Results of *NFVO-centric* Web Server Scenario

Specifically, it takes 48 ms to detect the need to scale, 32 ms to decide about the scaling actions to execute and then, around 40 seconds to launch and prepare a new VNFC instance. The time to launch and provision the VNFC instance takes around 8 seconds for booting, further 26 seconds until the generic EMS registers to the generic VNFM, and 6 more seconds for executing the configure lifecycle. The last step depends mainly on the VNF software which has to be installed, and may vary from a very short time to a very long time depending on the installation and configuration procedures which have to be executed.

Moving to the second testing scenario, as depicted by figures Figure 7.10 and Figure 7.11, the detection part varied between 56 ms and 78 ms, where the gap, although almost irrelevant, was basically due to the response time of the monitoring system. Scaling-out adding a single VNFC instance took around 17 seconds whereas scaling-out adding five VNFC instances took around 83 seconds. This was mainly due to the fact that scaling out operation were executed sequentially and not in parallel.

In case of scaling-out a single component it takes 99,36% of the overall time from detecting to finish scaling. In the case of scaling-out five components in a row the scaling took around 99,9% of the entire time. As seen, scaling-out five components in a row needs five times more and is caused by scaling-out step-by-step. Obviously, the time needed for scaling-out 5 components can be improved by executing in parallel scaling out operations. As part of the VNFM-centric scenario has been validated also the proposed pool mechanism, useful for optimizing the scale out procedure. Two scenarios were taken into account equally to the two scenarios described before. One test covers the scaling-out of a single VNFC instance and

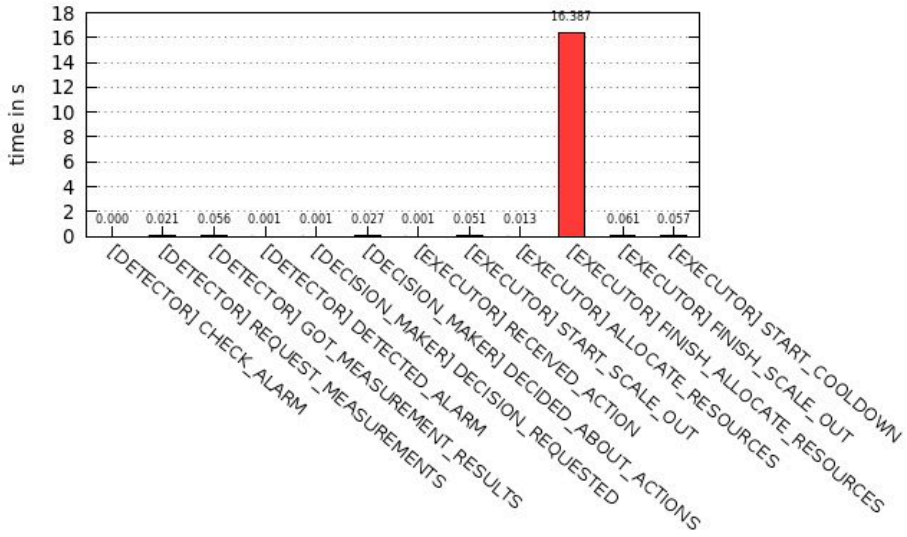

Figure 7.10: Measurement Results of the *VNFM-centric* Web Server Scenario while Scaling Out a single VNFC Instance

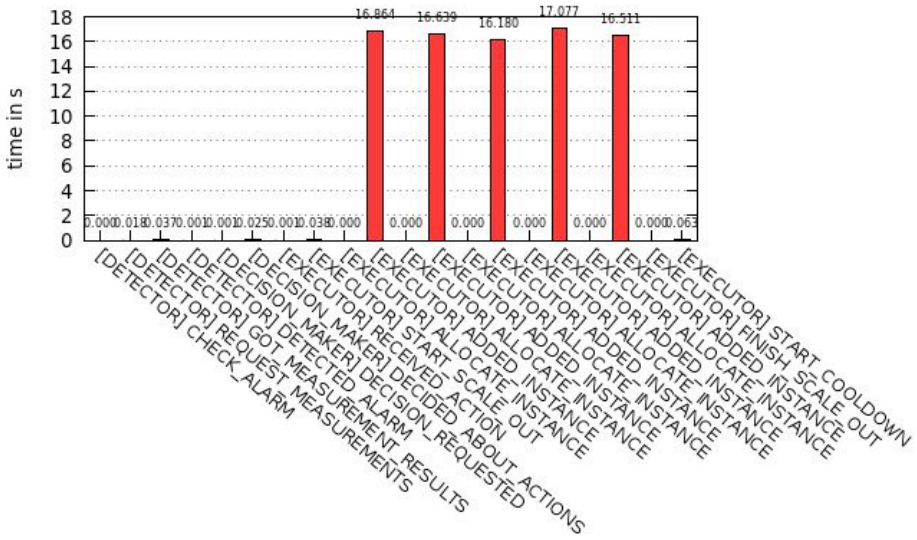

Figure 7.11: Measurement Results of the *VNFM-centric* Web Server Scenario while Scaling Out five VNFC Instances

the other scenario scales out five VNFC instances step-by-step. As presented in the previous section, the time for launching new components takes most of the time of the entire scaling process, and therefore, it decreases the reaction time of the AES dramatically in order to provide new instances on demand.

Figure 7.12 and Figure 7.13 present the measurements obtained while using the

pool mechanism in the case of scaling-out a single or multiple (five in this case) VNFC instances. Similar to the previous case (without the pool mechanism) the detection and decision time had minimal impact (24 ms) on the overall scaling procedure. As expected, differently from the previous case, the execution time required was minimal while requesting the allocation of new VNFC instances to the Pool Manager instead of asking the VIM directly. As depicted in Figure 7.12 in case of adding a single component, the whole scaling operation required around 140 ms. Figure 7.13 reveals that scaling out five components in a row required only 350 ms.

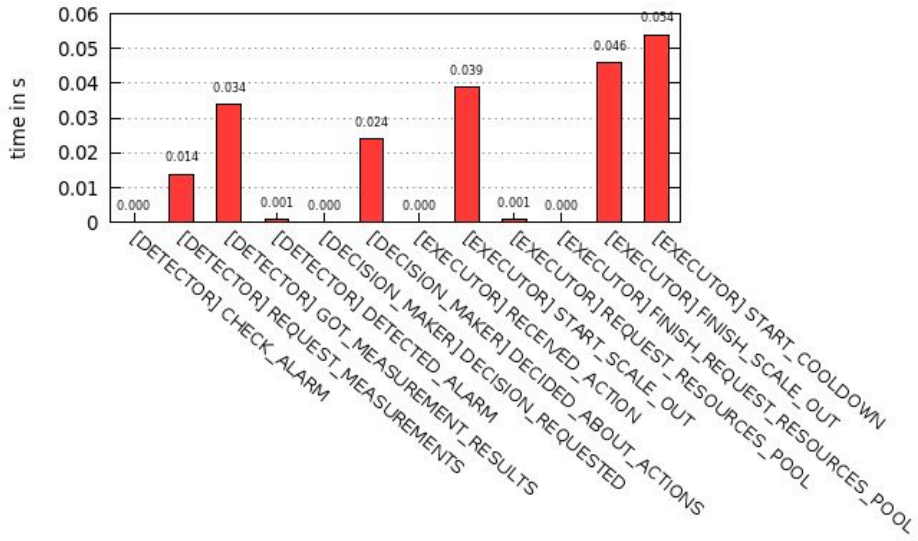

Figure 7.12: Measurement Results of the *VNFM-centric* Web Server Scenario while Scaling Out a single VNFC Instance with the Pool Manager

Compared with the detection time needed, the scaling procedure itself (in case of adding a single component) took 65,7% of the total time, whereas scaling-out five components took 87,93%. Compared with the time needed in the scenarios without the pool mechanism, scaling-out a single component and five components in a row is drastically improved. In fact, the scaling out action to add a new component took in this case 0,85% of the time that was needed without the pool mechanism. Scaling out five components took in this case 0,42% only compared with the time in the previous scenario.

The execution of the third testing scenario provided some performance results of the AES over a longer duration of time. Results of the execution of the third scenario are shown in Figure 7.14, including 4 graphs where each graph covers two hours of the test.

The measurements obtained validate the expected functional behavior of the AES component. In general, as soon as the number of requests (thick blue line) increases, the AES scales out adding new VNFC instances, while whenever the number of requests decreases, the AES scales in removing VNFC instances.

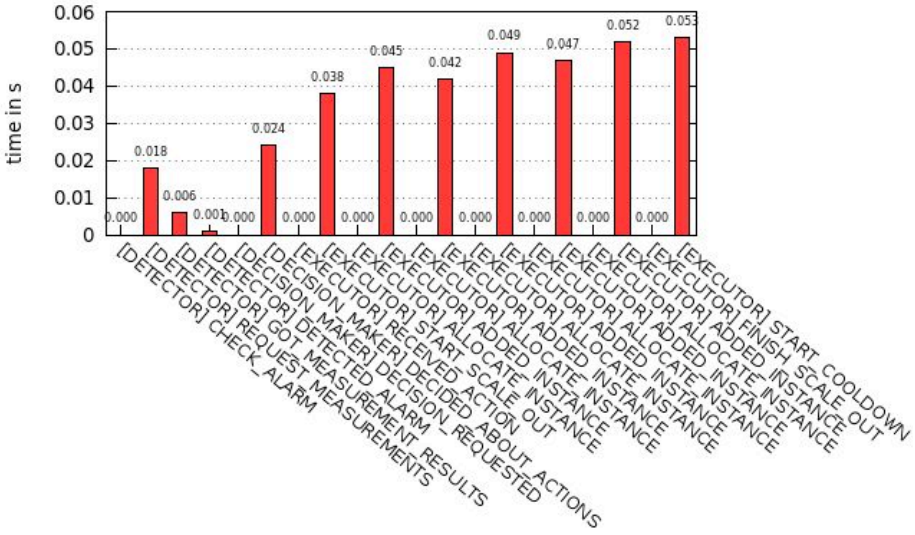

Figure 7.13: Measurement Results of the *VNFM-centric* Web Server Scenario while Scaling Out five VNFC Instances with the Pool Manager

Ideally, the measurement results of each instance should start with 100% *'CPU idle time'*, but unfortunately, the first values shown indicates that the scaled instances received requests before the Monitoring System starts to track this specific metric. Nevertheless, each curve appearing or disappearing in the graphs shows the scaling activities and the load situation of a certain instance in the considered time frame. If a new curve pops up, an VNFC instance was scaled-out and viceversa, if a curve disappears, it means that a VNFC instance was scaled-in.

The bigger red line depicts the number of errors in a certain period of time. The errors appearing during this scenario execution are caused by two particular conditions. The first case happened when the number of requests is greater than 10000. In this case the number of VNFC instances available were not able anymore to handle the load, and the AES cannot scale out anymore as the maximum number of instances was reached (as indicated in the VNFD). In turn, the second case was due to the fact that the load increased too fast and the AES was not able to react quickly. As soon as the AES scaled-out, the number of errors decreases immediately. This depends basically on the autoscaling policy defined in the VNFD and is mainly due to the cooldown period defined. So between two consequent scaling operations the time needed may be more than three minutes. Besides the cooldown parameter, additional time was due to the duration of the instantiate lifecycle while scaling out adding new instances.

7.2.3.3 IMS Network Service

This subsection focuses on validating the proposed AES with the vIMS use case.

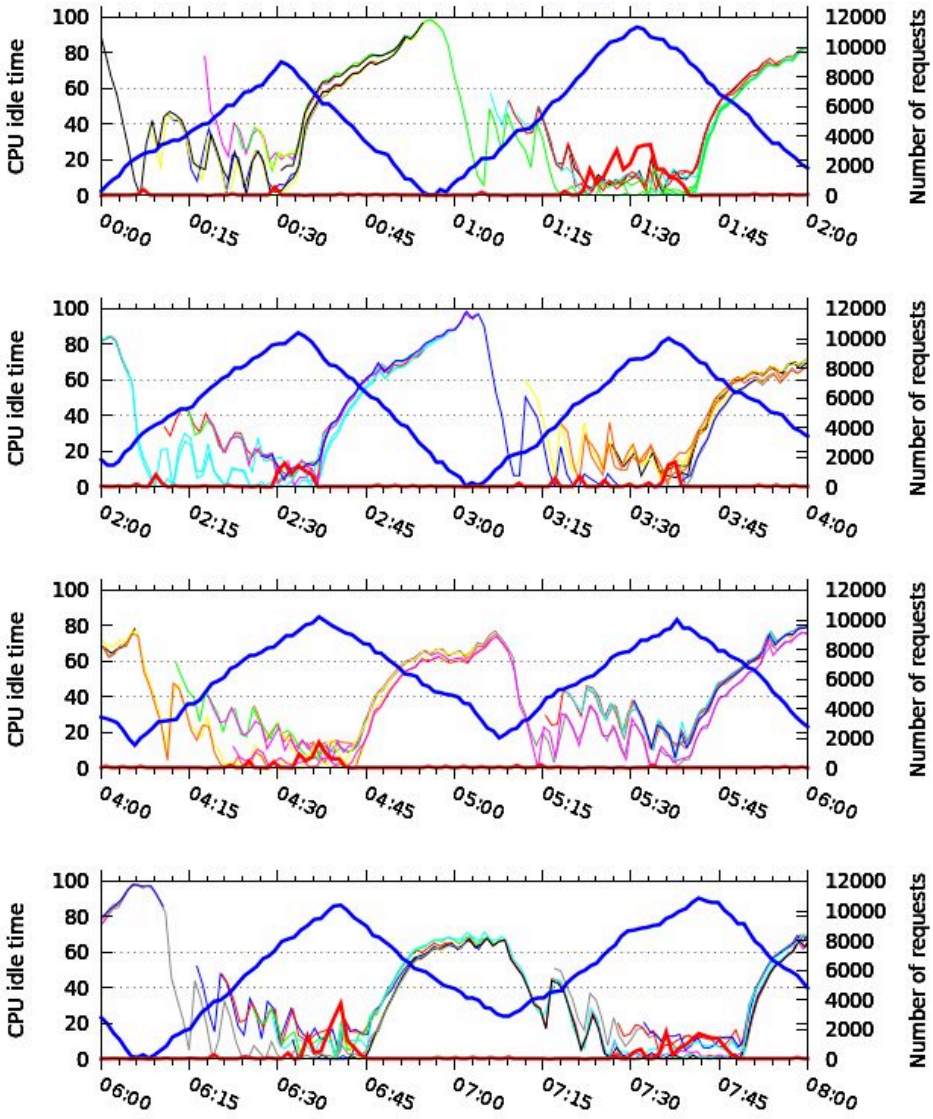

Figure 7.14: Measurement Results of the Third Web Server Scenario

Testing Setup: The "micro-dc-testbed" has been utilized as testing environment.

Testing Scenario: For this scenario it was used the vIMS network service, and enabled High-Availbility for the HSS VNF. This has been achieved making use of the Diameter Routing Agent (DRA) component into the vIMS solution. For this scenario it was employed and further adapted Open Source IMS Core (OpenIMSCore)[24] [165], an open source implementation of the 3GPP IMS standard. The scale out threshold has been configured as 65% *'CPU idle time'* (therefore when the CPU

[24]http://openimscore.org

of the system crosses 35% of utilization). Scaling in should be executed when the measurement results crosses the threshold of 80% in average over both instances, particularly, when the aggregated values are greater than 80%.

Performance Measurements: Performance results obtained by executing the scenario presented in the previous paragraph are shown in Figure 7.15. The load produced by simulated users' requests started at 50 seconds and ended at 1010 seconds. It reflects the situation where a total number of 24000 calls are generated in a time range of 960 seconds, indicating 25 requests per second (req/s) or 125 requests in a 5 second interval as shown in the graph. Since the requests that are calls in the end are remaining for either a shorter or a longer time, the number of successful calls varies from 80 up to 160.

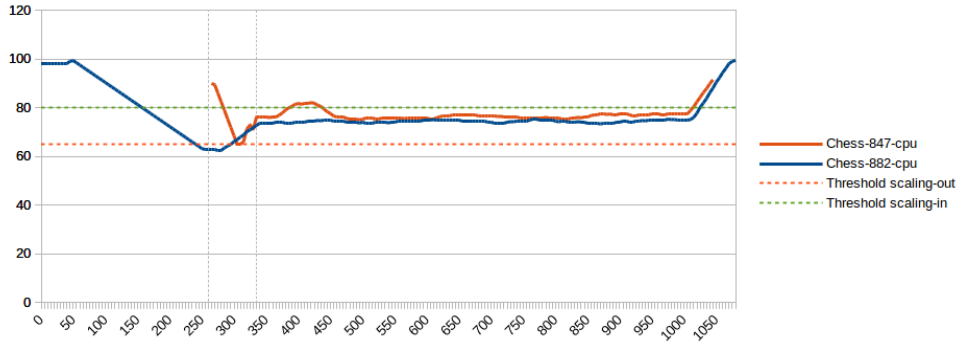

Figure 7.15: *'CPU idle time'* of the two HSS VNFC Instances, and related Scaling Out and In Thresholds

The graph in Figure 7.15 illustrates that the first HSS VNFC instance (line labeled in the graph as `'Chess-882-cpu'`) was idle in the first 50 seconds before it started processing the incoming requests (it can be noticed by the decreasing *'CPU idle time'* and the outgoing network traffic as well). In the following 210 seconds it handled almost the first 5250 incoming requests. At around 260 seconds, the first HSS VNFC instance crossed the threshold of 65% *'CPU idle time'*, defined as the scaling out threshold, thus the AES initiated the scale out procedure. The scaling procedure itself is depicted by the two grey dotted lines, and further analyzed in Figure 7.16.

It took around 81 seconds from detecting the need to scale until the second HSS VNFC instance (line labeled in the graph as `'Chess-847-cpu'`) was instantiated and used for receiving requests besides the initial instance. Similarly to the scenarios presented in the previous section, the execution phase took most of the time of the overall process, with 99,8% of the entire time, namely 81.7 seconds. This time was required for launching the new instance and preparing it by installing and configuring the VNFC software component. Moreover, dependent components (e.g. the DRA VNFC instances) are notified about the additional HSS VNFC instance in order to reconfigure related services.

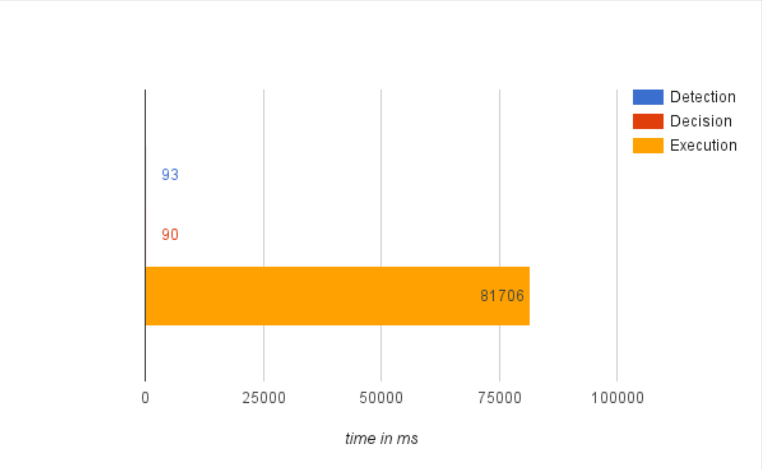

Figure 7.16: Detailed View of the Scale Out Procedure

Looking again at Figure 7.15, once the scaling out process was finished (at 342 seconds) the incoming requests were spread over both instances, thus the CPU load was equally distributed between them. This covers the time interval from 342 seconds to 1010 seconds where the generation of load ended. Afterwards, once all the requests were processed, the number of calls decreases immediately and both HSS VNFC instances got less loaded. As soon as the AES recognized that the CPU idle time of both instances in average crosses the upper threshold of 80%, executes a scale-in action.

Concrete timings of scaling-in operation from the initial check of conditions to finishing the corrective action is show in Figure 7.17. As depicted the scaling-in process itself took around 350 ms leading to a final time of 513 ms for the whole procedure.

The scaling-in process ended at 1045 seconds and the second HSS VNFC instance was removed.

7.2.4 Fault Management in an OpenStack-based NFVI

The test scenario described in this subsection has been performed in order to validate the FMS in scenarios involving failures at both the VNF and the infrastructure level. The main objective is to functionally validate the functions provided by the FMS and to evaluate the performance results obtained by using this additional component. Parts of the results presented as part of this experiment have been published in deliverables of the MCN project [168][169].

7.2.4.1 Testing Setup

The “orange-box-testbed” was utilized as testing environment configured with two tenants:

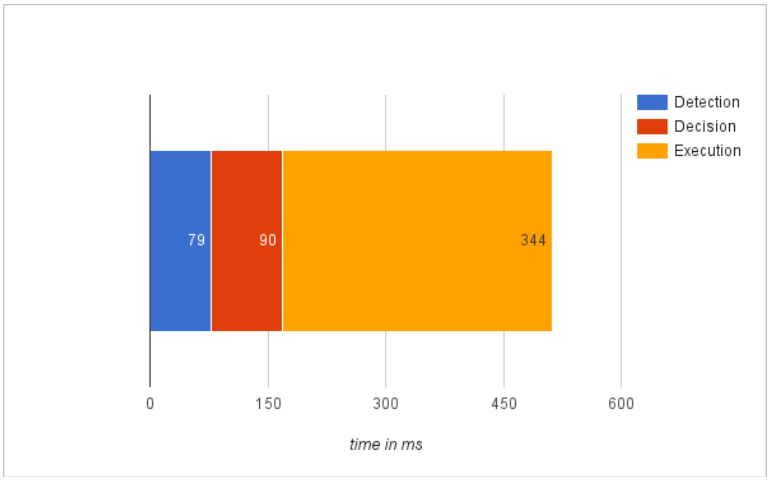

Figure 7.17: Detailed View of the Scale In Procedure

1. tenant *admin*: comprising the Open Baton and Zabbix Server VMs. Open Baton was installed on a VM with ubuntu 16.04 OS, 8 Giga Bytes (1,073,741,824 bytes) (GB) of RAM and 2 vCPUs, including the NFVO, the message bus, the generic VNFM, the OpenStack VIM driver, the FMS, and the Zabbix plugin. While the Zabbix Server was installed on a VM with an ubuntu 14.04 OS.

2. tenant *fm-scenario*: used for deploying the scenarios under tests. The details about the network service used as System under Test (SuT) are explained in the following subsection.

7.2.4.2 Testing Scenario

In order to validate the FMS, it was reused the *Web Service* network service presented in Section 7.2.3.2. Differently from the previous scenario, the elastic-app VNFD included only the fault management policy adopting the 1:N redundancy scheme, and a minimal number of VNFC instances (`scale_in_out` parameter set to 2). the NFVO deployed two VNFC instances (EA1 and EA2), and a third one (EA3) was automatically instantiated by the FMS and put in standby status. Furthermore, the FMS created the required performance jobs and the related thresholds in order to actively monitor the active VNFC instances and the virtual resources needed.

Once the network service was on boarded and instantiated, the load balancer VNFC instance was configured in order to forward all the requests coming from the outside to the elastic-app VNFCs instances available in the backend. The load balancer has been configured for automatically discovering whenever an elastic-app VNFC instance is no more available, so that the number of failed requests was minimized, whenever a fault affected any of the VNFC instances. Therefore, the expected behavior is that as soon as the load balancer detected the unavailability

of one of the elastic-app VNFC instances (e.g. EA1), it will forward all the requests to the other ones active at that point in time (e.g. EA2).

Three different testing scenario have been designed and executed for validating the functionality provided by the FMS:

1. In the first test, it has been simulated a fault at the VNFC level by killing the elastic-app process running in the EA1 instance. This test has been utilized to validate the correct detection of the fault at the VNF level and consequent execution of the the heal lifecycle event.

2. In the second test, it has been simulated a fault at the infrastructure level by manually terminating one of the virtualized compute resources utilized by the elastic-app VNFC instances. In particular, it has been simulated a virtual machine failure in the EA1. This test has been utilized to validate the proper detection of the fault, in particular the root cause analysis executed by the fault correlation function, and consequent execution of the switch to standby action. As already explained in the previous chapter, this action consists in activating the standby VNFC instance (in this case EA3), and re-configuring the load balancer (via the dependency resolution logic executed by the NFVO) in order to remove the failed VNFC (EA1) and add the newly activated VNFC instance (EA3).

3. In the third test, a comprehensive final experiment has been performed in order to validate the FMS injecting repeated failures during one hour. The main objective was to validate the FMS in scenarios with repeated faults.

Although in all the presented testing scenarios the load balancer may represent a potential bottleneck, it is assumed here that the resources allocated to its VNFC instance are enough to serve the load injected by the client emulator.

7.2.4.3 Functional Validation and Performance Measurements

The different testing scenarios presented in the previous subsections have been performed and evaluated separately.

The execution of the first testing scenario provided a functional validation as well as some performance results of the FMS healing functionality. After simulating the fault at the VNFC layer, the FMS required around 25 seconds to recover the VNF to an ACTIVE state. Most of the time required for recovering the VNF while performing the heal functionality was due to the latency of the monitoring system in sending the notification about the fault. In particular, 24 seconds were needed by Zabbix for notifying the FMS (through the zabbix monitoring plugin), while 1 second was the time needed by the FMS to execute the heal recovery function itself. Figure 7.18 shows the workload of the EA2 while the EA1 was experiencing the failure. As can be noticed, all the requests were forwarded to the EA2 because the load balancer automatically detected that EA1 did not respond anymore to incoming requests.

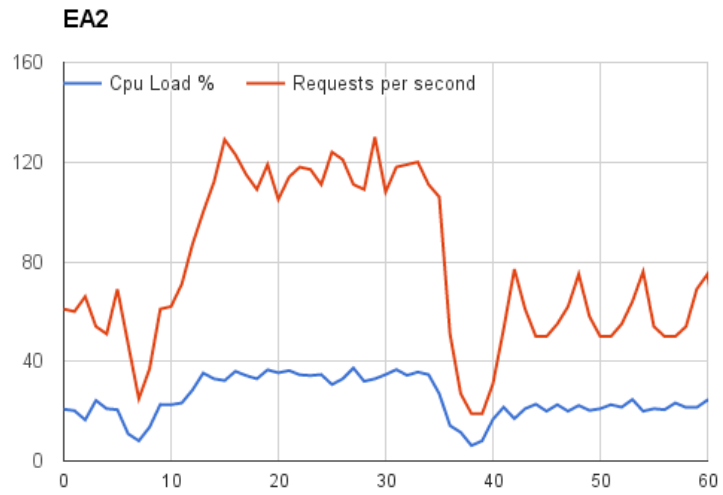

Figure 7.18: Performance Measurements of the First FMS Testing Scenario

The execution of the second testing scenario provided a functional validation as well as some performance results of the FMS switch to standby functionality. Figure 7.19 shows the CPU usage of EA1, EA2 and EA3. The failure was injected at $t_0 = 15(s)$.

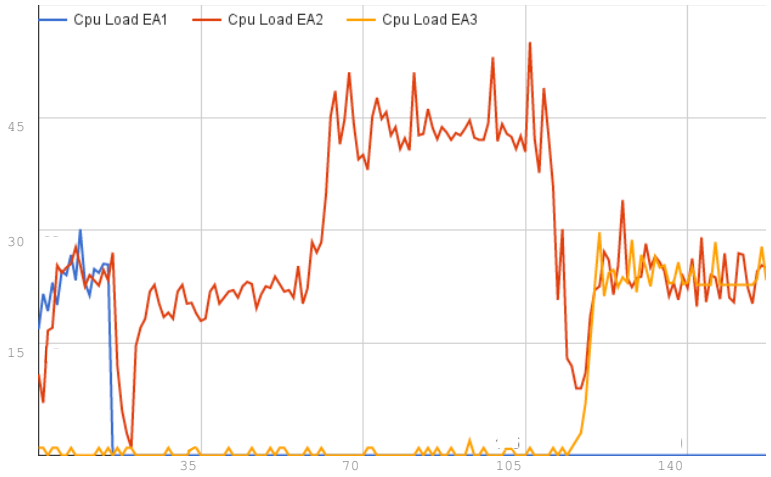

Figure 7.19: Performance Measurements of the Second FMS Testing Scenario

After the failure, the load balancer required about 40 seconds for taking the decision of forwarding all the requests to EA2. This can be noticed in Figure 7.19 as the CPU usage of EA2 increases up to 45%. The switch to standby action is performed at $t_1 = 109(s)$. Considering the network service composition, while executing the switch to standby action triggered by the NFVO executes in sequence the SCALE_IN, MODIFY, and START lifecycle events against the load balancer

VNF. The SCALE_IN is required to remove the failed VNFC instance (EA1) from the backend pool, the MODIFY is used for including the new VNFC instance (EA3) to the pool, while the START is used for re-loading the pool configuration.

Around $t_2 = 115(s)$ can be noticed that the CPU usage of the two instances (EA2 and EA3) reached similar levels, thus the re-configuration of the load balancer was properly executed. It is also important to notice that during this testing scenario the fault correlator properly identified the root cause of the problem: the monitoring system sent several failure notifications (i.e. VM unreachable, VNFC software components not running, etc.), however the system properly identified that the root cause was at the infrastructure level.

The execution of the second testing scenario (very similar to the previous one) provided a functional validation as well as some performance results of the FMS switch to standby functionality during repeated failure events. Within this scenario have been gathered also the errors reported by the client during the execution of the switch to standby functionality.

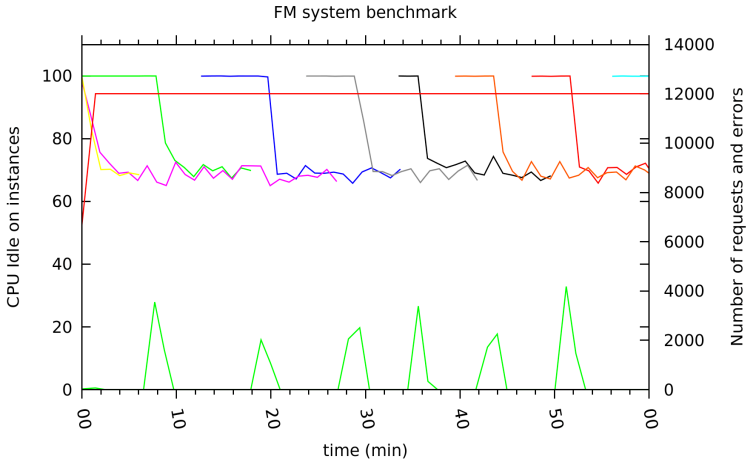

Figure 7.20: Performance Measurements of the Third FMS Testing Scenario

The red horizontal line is the total number of requests performed per minute (12.000 req/min - 200 req/sec). The green line on the bottom represents the number of failed requests. In correspondence of the VM failures the number of failed requests reached approximately 3000 req/min.

The colored lines on the top of the graph represent the averaged metric *'CPU idle time'* of the VM hosting the EA1. Looking closely is possible to note a small decrease of this value (meaning an increase of the CPU usage KPI) as soon as one of the elastic-apps VNFC instances fails. That is due to the load balancer forwarding all the 200 req/sec to the other VNFC instances available. The impact is minimal since the FMS executes in a rather short time the switch to standby recovery action.

The average of the total latency, from the occurrence of the failure until the network service is fully recovered, has been calculated and illustrated in Figure

7.21. The reported results are the average of the six cases of failure emulated in our tests.

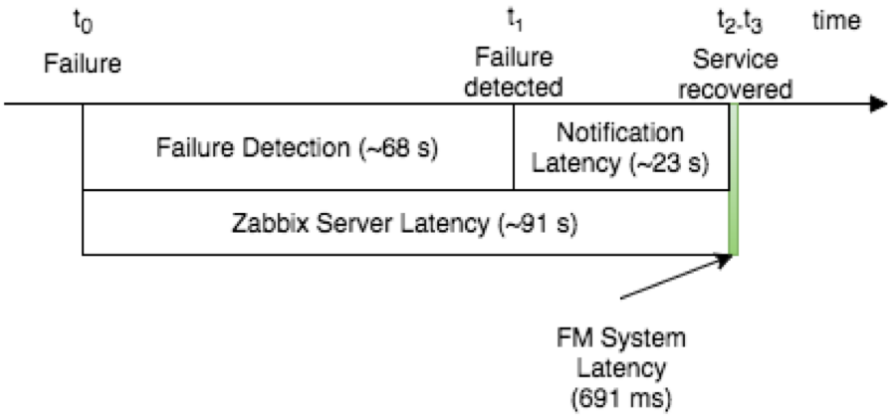

Figure 7.21: Measurements Results of the FMS Latency

The interval $t_0 - t_1$ represents the time between the occurrence of the failure and its detection by the Zabbix Server. Such interval can be optimized (e.g reduced) by tuning the delay of the KPI retrieval via proper configuration of the Zabbix monitoring system. In the considered scenario, the Zabbix Server updated the monitored indicators every 60 seconds, which is considered very usual for many related deployment environments (limited overhead with acceptable latency introduced in management operations).

The interval $t_1 - t_2$ is the latency introduced by the Zabbix Server to send the notification to the monitoring plugin. Ideally, the Zabbix Server should only detect the condition specified in the trigger, and execute the alert script. Although executing the alert script takes few milliseconds, the notification is received by the monitoring driver not before 23 seconds. Such interval could not be reduced since cause by the internal mechanism used by the Zabbix Server to send notifications.

The interval $t_2 - t_3$ is the overhead introduced by the fault restoration procedure. In this interval different components of the Open Baton framework are involved, including the FMS itself.

Although this interval had the lowest impact on the overall procedure, it has been analyzed in details and the specific results obtained are reported in the following tab:fm-perf.

The total overhead was around 691 ms, due to the different actions performed by the different components as succinctly described above. The monitoring plugin introduced an overhead of 28 ms due to the mapping of the Zabbix Server notification to the internal `VirtualizedResourceAlarmNotification`. The FMS introduced an overhead of about 96 ms, in which the major fraction has demonstrated to be caused

Component	Overhead
Monitoring Plugin	28 ms
FMS	96 ms
NFVO & VNFM	567 ms
Total	691 ms

Table 7.3: Overhead Introduced by the FMS while Executing a Switch to Standby Operation

by the execution of the Drools policies.

Last, the NFVO executed the switch to standby action in about 15 ms by sending a message with the cause "switch-to-standby" to the Generic VNFM. The overhead introduced by the VNFM during the activation of the standby VNFC instance is around 220 ms, mostly caused by the execution of the script to start the Apache Web server. After that, the NFVO executed the dependency resolution procedure and sent to the VNFM the SCALE_IN, MODIFY, and START actions in order to reconfigure the load balancer VNFC. Such actions are grouped with an overhead of around 40 ms.

Although the overhead of the FMS depends on the number of the rules to be processed, those evaluation activities demonstrated that the overhead of the FMS is largely acceptable.

7.2.5 Juju Integration

The testing scenario presented in this section has been performed in order to validate the Juju VNFM adapter. The main objective is to functionally validate the integration of an external VNFM within the Open Baton framework, as well as to provide some performance results comparing the Juju VNFM and the Generic VNFM.

7.2.5.1 Testing Setup

For this experiment, it has been employed the “micro-dc-testbed”. In this case, the Open Baton NFVO, the RabbitMQ broker, Juju and the Juju VNFM adapter were executed on a Dell Precision Tower 5810 with a 64-bit architecture, an Intel® Xeon® CPU E5-1650 v3 processor with 3.50 GHz and 14GB RAM running Ubuntu 16.04. Considering that the Open Baton components have been deployed on a different host, OpenStack has been configured with only one tenant, used for deploying the complete network service instance.

Before starting the actual experiments, Juju has been bootstrapped in order to be able to allocate resources on the OpenStack instance. The controller’s name used is `controller-vim`. After configuring and starting the NFVO and Juju VNFM adapter, the Juju VNFM adapter registers to the NFVO following the registration procedure presented in Section 5.7.2.2. A PoP named as the Juju controller was

registered to the NFVO for establishing the reference to the Juju controller created previously.

In addition the Open Baton Generic VNFM was added to the setup by starting it on the same host on which the NFVO, and the Juju VNFM adapter were running.

7.2.5.2 Testing Scenario

For validating the Juju integration, and in particular for collecting performance results of the deployment of a network service using the Juju VNFM, it was selected the vIMS network service, implemented, as in previous cases, by the Fraunhofer FOKUS OpenIMSCore toolkit. In particular, it has been generated a comparison (in terms of execution time) between the deployment of the OpenIMSCore network service using the Juju VNFM and the one making use of Generic VNFM. Therefore, during the OpenIMSCore deployment procedure, the average time of each lifecycle operation was measured.

Three testing scenarios were executed. First, the deployment with the Generic VNFM, second the deployment with the Juju VNFM with resource allocation performed by the NFVO, and lastly the deployment with the Juju VNFM with resource allocation performed by Juju itself. Each of the three cases were executed ten times in order to take comprehensive measurements of the lifecycle operations performed.

After the execution of the three testing scenarios, the following processes were measured and compared. The actual deployment of the network service, by launching the corresponding NSD from the NFVO. The triggering of a scale out operation of the IMS Proxy CSCF (P-CSCF) VNF. The termination of the network service, by deleting the NSR from the NFVO. Considering that individual lifecycle events are executed for each VNFC instance of the network service, the duration of a lifecycle operation was measured as an average of the duration of each individual one.

Table 7.4 shows which lifecycle operations are measured and when the measurement begins and ends for each VNFC instance of the vIMS network service. All the measurements are taken from the NFVO log files.

7.2.5.3 Functional Validation and Performance Measurements

The results of the execution of the aforementioned scenarios are visualized in Figure 7.22. The orange and red bars represents the lifecycle duration when using the Juju-VNFM. In particular, the orange one shows the case when Juju allocated resources on the NFVI, while the red one shows the case in which the allocation was done by the NFVO.

The blue bar represents the time required when the Generic VNFM is used instead of the Juju VNFM. As one can see, the differences between the orange and red bars are minimal indicating that the decision on whether the NFVO allocates the resources or it is done by Juju directly, does not have a great impact on the deployment time.

However, the differences among the orange/red bars and the blue bars are significant. The instantiate operation took about four to five times longer when the

Lifecycle Operation	Start	End
INSTANTIATE	After the NFVO sends the INSTANTIATE message the VNFM	When the NFVO receives the response message
MODIFY	After the NFVO sends the MODIFY message the VNFM	When the NFVO receives the response message
START	After the NFVO sends the START message the VNFM	When the NFVO receives the response message
SCALE_OUT	After the NFVO sends the SCALE_OUT message the VNFM	When the NFVO receives the response message
TERMINATE	After the NFVO sends the TERMINATE message the VNFM	When the NFVO receives the response message

Table 7.4: Measurements Points

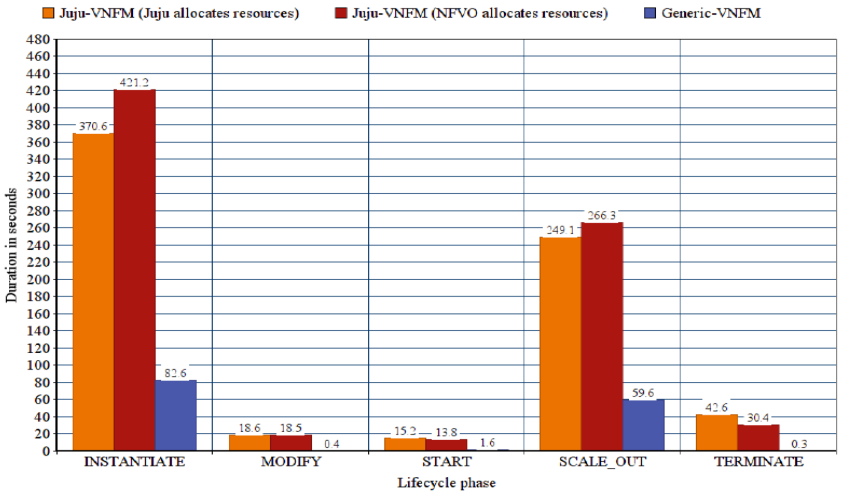

Figure 7.22: Comparison between the Juju VNFMs and the Generic VNFM

Juju VNFM was used. The time that an average modify lifecycle operation required is about 46 times longer when using the Juju VNFM instead of the Generic VNFM and in case of the start lifecycle operation the Generic VNFM is about nine times faster than the Juju VNFM. Similarly to the instantiation lifecycle, the scale out operation of the P-CSCF VNF was four times faster when using the Generic VNFM. Finally, the termination of the network service was even 100 to 140 times faster.

One of the main reasons behind the drastic differences in terms of time required for executing the lifecycle events, is mainly due to the fact that while launching a VNFC instance via Juju, it starts a VM using public ubuntu cloud disk images and triggering the OS upgrade procedures. Thus, a first optimization of the results obtained in the previous experiments, can be obtained downloading the most recent version of the needed disk image from the official ubuntu repository, and disabling the automatic upgrade procedure from the Juju controller. Another important factor which had large impacts on the overall results is the installation of the juju agent inside the VM during the first instantiation. Furthermore, some of the time maybe also due to the internal scheduler utilized by Juju. Infact, while using the Juju via its CLI it has been noticed that it requires some time between receiving the request and triggering the execution of the action requested.

7.2.6 Continuous Integration

This last subsection introduces the CI/CD testing environment, having as main objective the automated functional validation of the framework whenever changes are committed in the source code repositories of the different projects. As already introduced in previous Chapter 2, a CI/CD system allows merging all latest developments, and testing them into a single system in an automated way.

7.2.6.1 Testing Setup

The "micro-dc-testbed" has been utilized as testing environment. In addition to the OpenStack PoP, the setup included one node running the Jenkins server[25].

7.2.6.2 Testing Scenario

The CI/CD system which has been used in the context of this testing environment, make use of Jenkins as a generic continuous integration system allowing setting periodic testing jobs. In particular, it makes use of the Pipeline suite of plugins[26] enabling writing testing scenarios involving multiple stages, including builds and deployments of the SuT. Furthermore, the integration with docker allows ease composition of different components in the SuT, so that particular features can be validated individually.

Considering that the major objective is to test the Open Baton framework itself, several network services (i.e. simple client/server network service, complex network service, etc.) should be on boarded and deployed runtime by the CI/CD system. In order to minimize the effort of writing testing jobs, and in particular for minimizing the effort in updating the testing jobs between different releases of the Open Baton framework, an integration-tests framework has been developed and released on the Open Baton GitHub repository[27]. Such framework specifically addresses the dif-

[25] https://jenkins.io/
[26] https://jenkins.io/doc/book/pipeline/
[27] https://github.com/openbaton/integration-tests

ferent steps of the network service lifecycle. Moreover, a dummy VNFM and VIM driver have been implemented and integrated as part of the SuT in order to validate that the lifecycle management operations are correctly executed by the NFVO.

A set of scenarios[28] have been implemented:

- *scenario-dummy-iperf*: dummy network service composed by eleven VNFs acting as client and server, using the dummy VNFM and VIM driver. Thus, it enables testing the actual dependency resolution between VNFs.

- *scenario-many-dependencies*: similarly to the previous case, dummy network service composed by VNFs having dependency to each other, using the dummy VNFM and VIM driver. Thus, it enables testing the actual dependency resolution between VNFs with complex network services.

- *scenario-real-iperf*: network service composed by two VNFs acting as client and server. The IPerf tool has been used as VNF software, and the expected behavior is that the IPerf client VNFC can connect successfully to the IPerf server VNFC and send traffic. It enables testing the Generic VNFM and the OpenStack VIM driver.

- *scenario-complex-ncat*: complex network service consisting of five VNF each instantiated with only one VNFC instance. Once deployed, the ncat[29] linux command line tool will be executed on the five VNFC instances and act as peers in dependency among each other. The target peer receives messages from the source peer and stores the received IP address so that it is possible to verify which peer has connected to which. In addition, this scenario includes multiple VLs deployments, so that it is also validated the scenario when multiple networks are allocated to deployed VNFC instances.

- *scenario-scaling*: network service composed by two VNFs acting as client and server and tests the execution of several scaling actions. It starts by deploying one VNFC instance per VNF and executes scaling in and out actions checking if new instances of the server and the client are deployed correctly, analyzing if the client instances are provided with the IP addresses of the new server instances, so that they are able to connect to them.

- *error-in-configure, error-in-instantiate, error-in-start, error-in-terminate*: each one deploys a network service using a NSD which contains a failing script in the particular lifecycle event and tests if the NFVO handles it correctly.

- *wrong-lifecycle-event*: it tries to on board a NSD to the NFVO which contains an undefined lifecycle event. The test will pass if the on boarding is not successful.

[28] scenarios and related descriptors can be found at: https://github.com/openbaton/integration-tests/tree/master/src/main/resources

[29] https://nmap.org/ncat/

- *stress-test*: it deploys in parallel a large number of network services using different NSDs. This test uses the dummy VNFM and VIM driver as the major objective is to validate the internal orchestration logic.

7.2.6.3 Functional Validation

The Jenkins server has been configured for executing periodically the scenarios aforementioned. After registering the PoP, each individual scenario is executed. Figure 7.23 shows the Jenkins dashboard providing the results obtained by the execution of the pipeline.

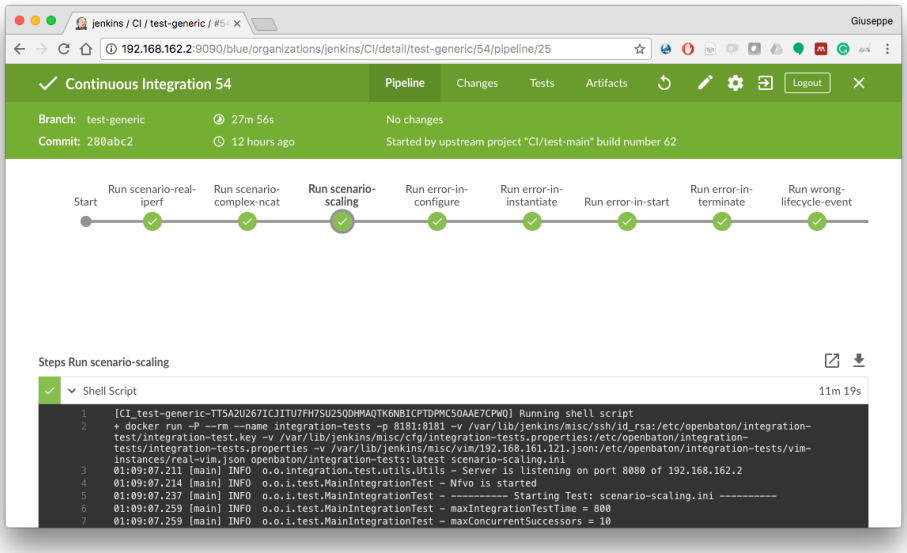

Figure 7.23: Overview of the Results of the Execution of the Jenkins Pipeline

As it can be noticed in Figure 7.23, tests are executed once after the other, and the green status means that they are successfully executed. In case of errors, it is possible to troubleshoot the issues either looking directly at the logs provided underneath the test execution or analyzing the logs of the Open Baton components also collected through the Jenkins server.

7.3 Comparative Evaluation based on the List of Features

This section introduces similar projects addressing the research challenges identified for building future 5G networks, and having as one of the objectives to provide solutions for the management and orchestration of network services in virtualized environments. ICT research projects and other open source initiatives are firstly

presented, and a final comparison overview, based on the list of features identified in Section 3.3, is provided.

7.3.1 ICT Research Projects

During the past years, several large ICT projects addressed research issues and challenges identified as part of this research work. As already presented in Section 7.1 MCN represents one of the initial projects trying to address the cloudification of telecommunication services, in which the author was directly involved.

The FP7 Unifying Cloud and Carrier Networks (UNIFY)[30] started in 2013 with the objective of making use of virtualization and automation technologies across the whole networking and cloud infrastructures[170]. The main focus is unifying carrier networks and cloud technologies in production environments using a set of enablers playing a central role for their overall orchestration[171]. The abstraction model proposed is based on a service creation language for enabling dynamic deployments of network components across the whole infrastructure. Several toolkits have been developed during the project: DoubleDecker[31], RateMon[32], Verigraph[33], UniversalNode[34], and several others. However, most of the technologies developed have been discontinued after the project ended. The author collaborated with the team from the Politecnico of Turin for integrating the UniversalNode as an alternative NFVI in the context of the SoftFIRE project.

T-NOVA[35] (Network Function as a Service over Virtualized infrastructures) is an FP7 project executed between 2014 and 2016 with the main objective of implementing a framework fully compatible with the ETSI NFV architecture allowing the management and orchestration of NFs over the NFVI[172]. In addition, T-NOVA objective was to provide an open marketplace which could be used by operators as a public catalogue. The public GitHub repository[36] lists several projects including management functions as well as VNFs. In particular, TeNOR[37] represents the T-NOVA orchestrator. Ruby was selected as programming language also following a microservice oriented architecture. However, as in the case of the previous project, development activities have been discontinued after the project ended in December 2016.

SONATA[38] is a project from 5G-PPP phase 1[173][174], started in 2015, with the objective of developing a NFV framework for providing a programming model and development toolchain for network services, fully integrated with the proposed DevOps-enabled service orchestration platform. In practice, they released a set of

[30] http://www.fp7-unify.eu/
[31] https://github.com/acreo/doubledecker
[32] https://github.com/nigsics/ramon
[33] https://github.com/netgroup-polito/verigraph
[34] https://github.com/netgroup-polito/un-orchestrator
[35] http://www.t-nova.eu/
[36] https://github.com/T-NOVA
[37] https://github.com/T-NOVA/TeNOR
[38] http://www.sonata-nfv.eu/

SDKs facilitating the generation of network services, a service orchestration platform for orchestrating network services. Although the project is still executing, primary results were already contributed towards other open source communities, like the ETSI OSM one presented in the following subsection.

Selfnet[39] is another project from 5G-PPP phase 1, started also in 2015, having as main objective the design and implementation of a framework able to fully automate the mitigation of certain common networking issues, reducing operational costs and manual intervention. In particular, Selfnet is driven by three major use cases addressing major network management problems: self-protection, for providing capabilities against security attacks, self-healing, for providing capabilities against network failures, and self-optimization, for providing capabilities to automatically re-configure network resources improving performances and enhancing QoE. The Selfnet project made use of some of the Open Baton framework for fulfilling some of the functionalities designed as part of its architecture[175].

5GTANGO[40] is a project from 5G-PPP phase 2 having as main objective the creation of a set of toolkits for enabling programmability of 5G networks[176]. It will develop a set of NFV-enabled SDKs, tools for validating and verifying VNFs and qualifying NSs, and a modular orchestration platform for bridging the gap between the business requirements and OSSs. 5GTANGO will move from static templates to dynamic ones going towards orchestration programmability for scaling and placement of VNFs. 5GTANGO technologies will be most probably based on the ones developed in the context of the SONATA project. Its GitHub repository[41] currently includes only one project[42] containing schema files for the various descriptors used by 5GTANGO.

7.3.2 The Open Source NFV Ecosystem

In the following it is provided a brief overview describing main goals, objectives, and a description about the most relevant open source projects providing a reference implementation of the ETSI NFV MANO specification.

7.3.2.1 ETSI Open Source MANO (OSM)

ETSI OSM is an operator-led ETSI community providing an open source implementation of the ETSI NFV MANO stack. This community has been launched by Intel, Telefonica, BT, Telenor, Canonical and Rift.IO in February 2016 combining existing projects which have already started earlier. As of November 2017, the OSM community counts around 85 companies. OSM official Release ONE has been launched in October 2016. It includes Network Service Orchestrator (NSO), RO and the VNF Configuration Manager. Most of the components are written in Python and their integration is achieved via direct API calls between components. In November 2017,

[39] https://5g-ppp.eu/selfnet/
[40] http://5gtango.eu/
[41] https://github.com/5gtango
[42] https://github.com/5gtango/tango-schema

the OSM community launched release THREE. Figure 7.24 shows its architecture as provided in the OSM White Paper[177].

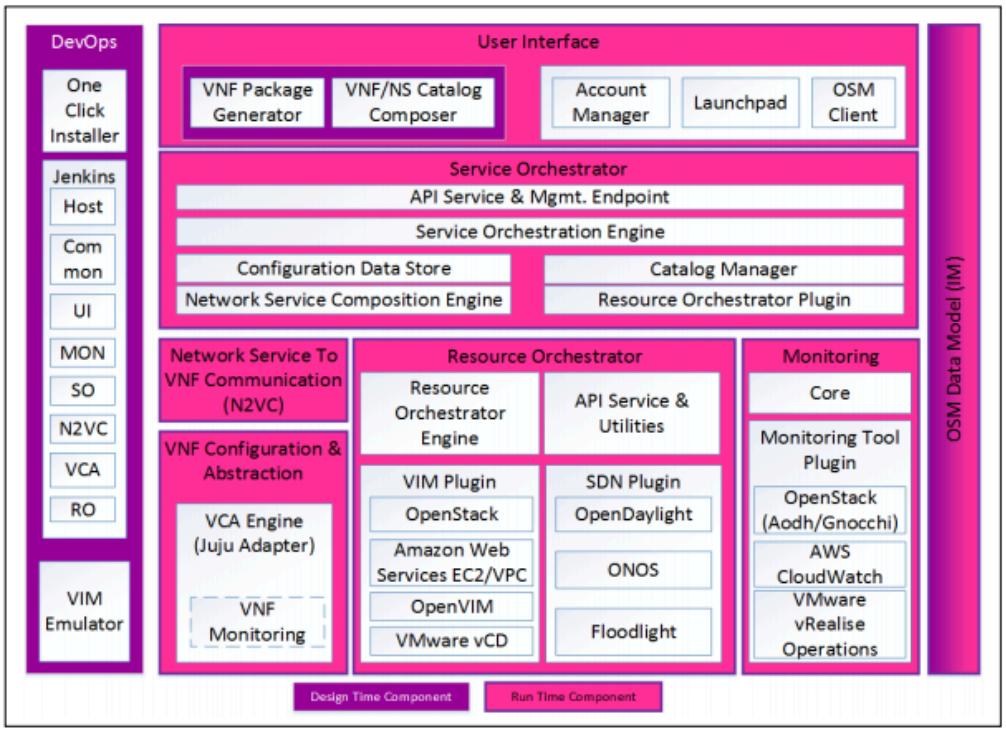

Figure 7.24: OSM High-Level Architecture[177]

OSM provides has three major components:

- The service orchestrator based on the Riftware component initially contributed by Rift.IO
- The resource orchestrator based on the OpenMANO component initially contributed by Telefonica
- The VNF Configuration and Abstraction (VCA) based on the Juju component initially contributed by Canonical

Most of the OSM components have been implemented in `Python` and the communication among components is based on REST APIs or direct python API calls. OSM release THREE integrated also monitoring (still experimental). Basically the service orchestrator plays the central role in the architecture communicating with the resource orchestrator for the allocation of resources across different VIM types (OpenStack, Amazon EC2, OpenVIM, and VMWare), and with the VCA for the configuration and execution of actions towards VNFs. Looking at the runtime phase

support, OSM provides also support for network service scaling, however not autoscaling.

7.3.2.2 OpenStack Tacker

OpenStack Tacker is an official project within the OpenStack foundation[178]. Initially driven by Brocade, Cisco, and Intel, Tacker main objective was to provide an implementation of a VNFM for deploying VNFs on OpenStack. Recently, Tacker extended its scope towards NFV orchestration. It provides an API which can be consumed directly from either the Horizon or the OpenStack CLI, for on boarding VNFPs and NSDs into its catalogue. Tacker follows the TOSCA specification for modeling descriptors, and makes use of the Heat engine to deploy resources on top of OpenStack.

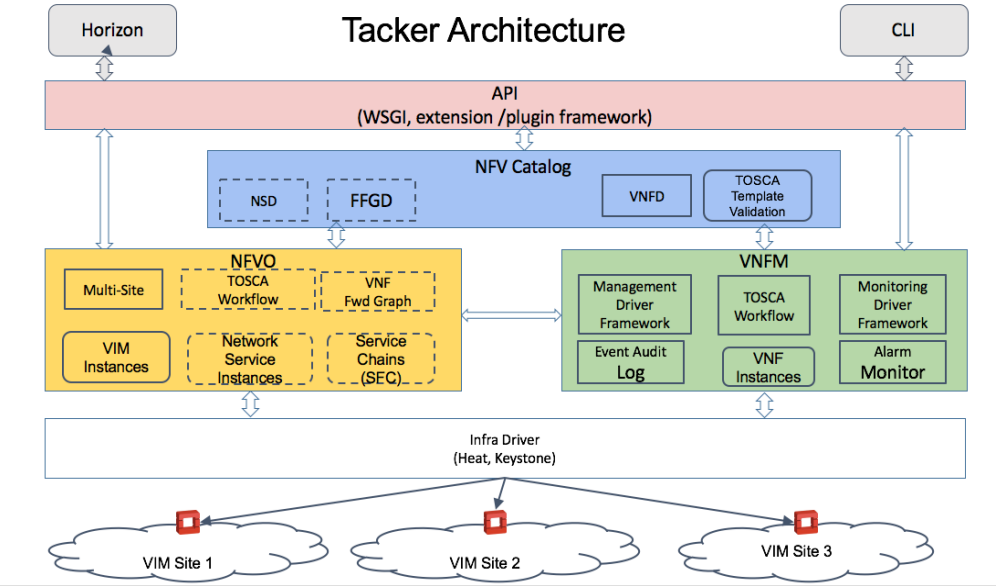

Figure 7.25: OpenStack Tacker High-Level Architecture[178]

7.3.2.3 Open Network Automation Platform (ONAP)

ONAP is a project officially formed in March 2017 by the Linux Foundation[179]. ONAP represents the merge between two other existing individual projects:

- Enhanced Control, Orchestration, Management & Policy (ECOMP) a project developed by AT&T focusing on providing a comprehensive platform for orchestrating and controlling any kind of services[180].

- Open-O a project initially driven by China Mobile, China Telecom and Hong Kong Telecom and later on (in October 2016) moved to the Linux Foundation with a similar scope.

Due to the large similarities between the two projects, the Linux Foundation decided to merge them into a single project. As of October 2017 ONAP counts around 60 members, with a large number of service providers and major vendors. Unfortunately, by the time in which this dissertation was written, ONAP did not have yet an official release publicly available. Thus, the author provides an overview of the project based on the information available on the public wiki page of the project[43].

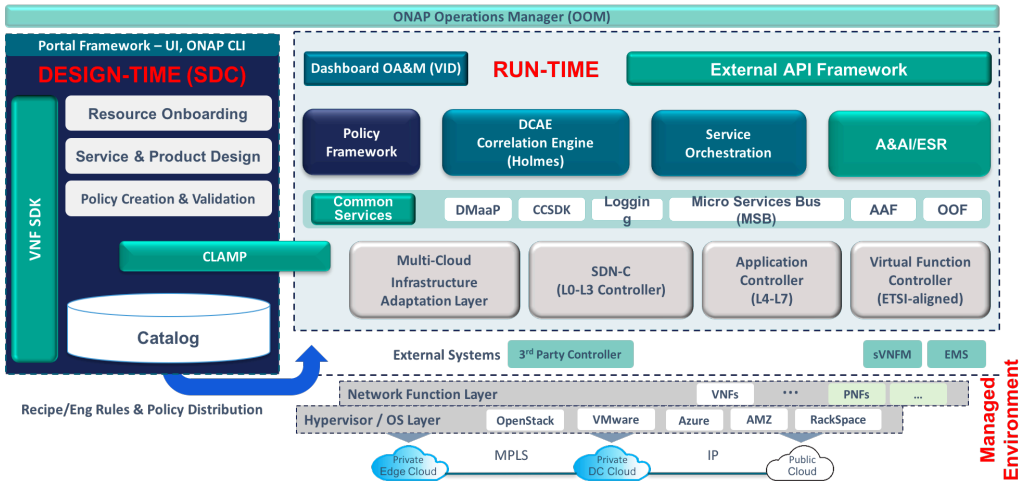

Figure 7.26: ONAP High-Level Architecture[181]

ONAP has definitively a broader scope than just being a reference implementation of the ETSI MANO specification, especially including aspects related with SDN orchestration and cross-domain orchestration. Its architecture consists of two major subsystems: the "design-time environment" and the "execution-time environment". In practice, the design-time environment represents a collection of tools and repositories for defining and describing deployable services. The execution-time environment comprises all the components required for automating the policy-driven service lifecycle and control loops. ONAP follows as well a micro-service oriented architecture in which the Micro Service Bus (MSB) plays the central role providing common functionalities and services across the different components of the architecture. Access to the ONAP subsystems is provided via the portal and CLI components.

[43] https://wiki.onap.org/

7.3.3 Summary and Comparison of Related Solutions

Although several scientific activites as well as ICT research projects addressed the scope of this research work, the final comparison provided in this section focuses only on the three major projects presented in the previous section. This decision is based on the results identified by a recent survey conducted by SDxCentral [182] in which around 150 survey respondents provided an answer to the question "are you considering open-source MANO?". Although a large portion of the respondents said that will consider them once they are more mature, selected platforms included ETSI OSM, and ONAP[44], OpenStack Tacker, and Open Baton solutions, including also the potential usage of commercial ones which are out of scope for the comparison provided here. The comparison provided below is based on the list of features identified by the author and presented in Chapter 3.

Feature *MANO-1 - Inventory* is supported by all of them via internal repositories used for collecting descriptors/templates and runtime information. Feature *MANO-2 - Lifecycle management* is also supported by all projects. Tacker and ONAP follow the TOSCA approach allowing TSPs defining management plans as part of the *service template*, while OSM follows a similar approach utilized by Open Baton decoupling the VNFP definition from the network service one. However, Open Baton provides a more comprehensive approach allowing both mechanisms (i.e., TOSCA-based, and descriptor based), and especially allowing different formats for their definitions (TOSCA and JSON). Feature *MANO-3 - Multi Tenancy* is supported by all projects (very recently added in OSM release 3). Feature *MANO-4 - OpenStack support* is also supported by all projects. OpenStack represents the standard de facto VIM, thus most of those projects support it naively. Feature *MANO-5 - Support for heterogeneous NFVI* is supported by all the projects, however the mechanism utilized differs among them. Tacker uses the OpenStack Heat infrastructure orchestrator, thus supports only what heat provides. OSM and ONAP supports public and private clouds as NFVI. In most of these projects, the abstraction layer required for interoperating with different VIMs is directly embedded in the orchestrator components. The clean separation introduced in Open Baton with the external VIM driver approach and communication over the message bus results the most extensible solution. Feature *MANO-6 - Multi-site NFVI* and *MANO-7 - VNF Placement* are consequently supported by all projects, as per *MANO-5 - Support for heterogeneous NFVI*. However, not all of them allow the instantiation of VNFs part of the same network service across multiple sites, and most importantly, not all allow defining VNF instance locations, as it can be done with Open Baton. Open Baton provides also additional capabilities for extending the VNF placement mechanism used just changing the scheduling class.

Looking at the list of features related with lifecycle management, feature *MANO-8 - Support for heterogeneous VNFs* is supported by all platforms as most of them have been validated with very different use cases making use of different VNFs. Feature *MANO-9 - Generic VNFM* is supported by all of them. In particular, Tacker

[44]Open-O at the time of the survey

started as a Generic VNFM project and recently added the NFVO layer. OSM makes use of Juju as Generic VNFM, while ONAP provides the VNFM as part of the Virtual Function Controller. Also in this case, Open Baton provides a Generic VNFM which can be further customized for using any configuration management technology during the instantiation procedures. Feature *MANO-10 - Support for specific VNFM* is supported by ONAP (mentioned on the general architecture, but no specific information available elsewhere in the documentation), is partially supported by OSM using specific charm types in Juju, but not supported by Tacker. Open Baton seems to provide the most comprehensive solution allowing integration of third parties VNFMs via the *Or-Vnfm-rest* and *Or-Vnfm-amqp* interfaces. The process of integrating a specific VNFM is farther simplified by the availability of the SDKs.

Looking at the runtime lifecycle, feature *MANO-11 - Monitoring support* support is provided by all the platforms. Tacker relies on the OpenStack default monitoring system (Telemetry[45]), ONAP does not specify the monitoring tool used, but it provides the Data Collection, Analytics and Events (DCAE) module supporting it, OSM integrates, in an experimental way since latest release THREE, different monitoring systems (OpenStack Aodh/Gnocchi, Amazon CloudWatch, VMWare vRealise Operations). Also in this case, the plug-and-play approach utilized provided by the Open Baton project can be considered the cleanest solution for integrating any kind of external monitoring systems. Feature *MANO-12 - Manual Scaling support* is supported by all projects, but using different approaches. Feature *MANO-13 - Autoscaling support* is supported by all projects. As already mentioned in previous chapters, the most common solution adopted is to allow users defining policies containing threshold-based conditions. Feature *MANO-14 - Fault Management support* is currently not supported by OSM. Tacker follows the same approach used for the previous feature, allowing users defining specific condition as part of the monitoring policies. ONAP provides a solution similar to the Open Baton one, using an analytic engine to correlate faults and taking decisions.

Feature *MANO-15 - Network Slicing support*, in this case intended as the capability of enforcing certain bandwidth requirements on the data plane, seems not to be supported by OSM and Tacker, and no information are available with regards to ONAP. The solution provided by Open Baton allows the smooth integration with the Neutron QoS APIs and SDN Controllers. Feature *MANO-16 - SFC management support* is supported by all projects, and most of them make use of the same technology (OpenDayLight) for the actual creation of the SFCs and SFPs at the networking layer. However, in some cases the SFC orchestration logic is embedded directly in the service orchestrator layer (i.e., Tacker and OSM), complicating its extension in case of different technologies used on the southbound interfaces. The approach designed with the SFCO external component, allow replacing any networking technologies without modifying the lifecycle management operations executed by the NFVO. Feature *MANO-17 - Integration with existing OSS/BSS components*

[45] https://wiki.openstack.org/wiki/Telemetry

is supported by almost all the existing projects since they all expose APIs for allowing the management and orchestration of network services. However, the approach taken with Open Baton framework allows the integration of external existing OSSs also via Pub/Sub mechanisms. Feature *MANO-18 - User Tools* is supported by all tools via classical CLIs and dashboards.

All the projects employed Apache 2.0 as the reference one license model. Looking at the information and data model, ONAP, Tacker, and Open Baton support TOSCA, however at the moment it is not possible to utilize a *service template* from one tool in another, because of the usage of specific type attributes. OSM uses a YANG[183] representation of the ETSI MANO model, which results also rather similar to the JSON representation utilized by Open Baton.

As a summary, all the four projects provide more or less the same features, however, as stated before, the different information and data model adopted limit interoperability between those solutions. Anyway, when the Open Baton project was started ONAP and ETSI OSM were not yet existing, while OpenStack Tacker was only targeting to provide a VNFM solution only for OpenStack-based PoPs. From an installation perspective, Open Baton and OpenStack Tacker have less requirements, while ONAP requires a large amount of resources for being installed[46].

All in all, Open Baton provides a feature-rich set of functionalities for being further extended and customized.

7.4 Summary

This chapter presented the evaluation of the MANO4X framework, based on the employment of its implementation (Open Baton) in different research projects and scenarios. The concepts and methods developed during the course of this research work have been integrated and validated as part of different academic and industry relevant projects as presented in Section 7.1.

The results obtained by the extensive validation of the proposed framework and its implementation have been presented in Section 7.2. Different use cases have been selected for evaluating the different features designed and implemented as part of the Open Baton project.

The last section provides an overview of existing similar projects and provides a comparison between the list of features generated as part of the Chapter 3 and implemented in the MANO4X framework, and the other projects.

[46] http://onap.readthedocs.io/en/latest/guides/onap-developer/settingup/fullonap.htmlfootprint

Chapter 8

Summary & Outlook

This chapter provides an overview of the impacts achieved throughout the research activities towards industry and academia, based on concepts and methods conceived as well as the MANO4X architecture designed and developed. The second part of this chapter provides the final answers to the main research questions identified at the beginning of the dissertation.

8.1 Resulting Impacts

The results obtained by the author and his team around the topics presented in the context of this research work had a huge impact towards the industry as well as academia, with several scientific articles, papers, and presentations published by the author as well as master and bachelor theses directly supervised by the author.

Moreover, the launch of the open source Open Baton project[163] as the first comprehensive reference implementation of the ETSI NFV MANO architecture generated an indirect impact by enabling researchers from all over the world experimenting with such technologies. For a certain period of time being the only available comprehensive open source solution addressing the NFV MANO challenges and pro-

viding a concrete reference implementation of the specification, several researchers started employing the tool with the objective of producing research results making use of state-of-the-art technologies in the NFV context. Several universities were able to employ the tool at practical zero costs, enabling the learning process to happen in a vendor-independent manner. Several scientific results were collected and published by other independent research institutions (most relevant ones[175][184][185][186][187][188][189]).

Open Baton has also been at the core of several European and international projects directly involving the author (as noted in Section 7.1) or just making use of the Open Baton project as a reference NFV MANO framework[175]. Additional results obtained are presented in the following subsections.

8.1.1 Summary of Academic Thesis Contributions

To date this author authored and co-authored 29 scientific peer-reviewed publications that were submitted to international scientific conferences and workshops as listed in Section A.1. In particular, ideas, concepts, and design methods were published since 2012, including practical evaluation scenarios that served for validating the solution outlined in this dissertation. Especially noteworthy are the following most relevant and cited publications:

- [129] in the proceedings of the 1st IEEE International Conference on Cloud Networking (CLOUDNET), representing the author's first publication.
- [121] in the proceedings of the 2014 IEEE Symposium on Computers and Communications (ISCC), discussing the IMS cloudification research challenges and presenting the results obtained within the MCN European research project.
- [135] in the proceedings of the 2015 IEEE International Conference on Communications (ICC), presenting the results obtained within the intermediate phase, about the cross-layer orchestration solution developed during the intermediate phase and further extended as part of the NSE component.
- [145] in the proceedings of the 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), presenting the autoscaling module demonstrating the easy extensibility of the MANO4X framework.
- [142] and [139] published recently in 2017, providing an overview about the latest results obtained with the use of the Open Baton project.

8.1.1.1 Talks and Tutorials

During the research activities, the author gave talks at international scientific conferences and academic workshops. Moreover, several tutorials were presented at scientific and industrial conferences. A comprehensive list is available in Section A.2, while the most relevant ones are the following:

- *"Network Virtualization and Network Slicing for 5G-Ready Networks"*, FUture SEamless COmmunication (FUSECO) Forum[1], November 2017, Berlin
- *"Software Networks in 5G – An Overview from the Core, Management, NFV, SDN, Programmability and Security"*, FUSECO Forum[2], November 2016, Berlin
- *"Open Baton – The First Comprehensive and Standard-Compliant NFV Framework"*, IEEE NFV-SDN, November 2015[3] and 2016[4], San Francisco
- *"5G Foundations and Core Network Evolution: Radio, Convergence Core, Cognitive Management and Virtualization (SDN/NFV)"*, FUSECO Forum 2014, Berlin

8.1.1.2 Demos

Several demonstrations were shown over the years at different scientific events. Following is a list of the most relevant ones:

- *"The NFV MANO Framework for Orchestrating Network Services from the Edge to the Cloud*, IEEE NFV-SDN, November 2017, Berlin.
- *"NUBOMEDIA – The First Open Source PaaS for Developing Multimedia Services"*, IEEE NFV-SDN[5], November 2016, Palo Alto. This demonstration received the best demonstration award at the conference[6].
- *"Deploying a Virtualized Core Network on Top of a Cloud Infrastructure as a Service Using Open Baton"*, KuVS, 2017, Berlin.
- *"Mobile Cloud Networking - Orchestration of a Mobile Core Network on Top of an OpenStack Cloud"*, EUCNC and FIA, 2014.
- *"Mobile Cloud Networking - End-to-end Deployment of a Virtualized Mobile Core Network"*, GlobeCom, 2015, Austin.
- *"Elasticity applied to Real-Time Communication Services"*, NTT R&D Forum, 2013, Tokyo.

8.1.1.3 The IEEE SDN Initiative

The author actively contributed to the IEEE SDN initiative with the results achieved during his research work. First, and most importantly, the author enabled the first version of the publicly available catalog of toolkits and testbeds[7]. The main objective

[1] https://www.fokus.fraunhofer.de/fff2017/day1
[2] https://www.fokus.fraunhofer.de/fff2016/day1
[3] http://nfvsdn2015.ieee-nfvsdn.org/tutorials.htmlT3
[4] http://nfvsdn2016.ieee-nfvsdn.org/program/tutorials/T2
[5] http://nfvsdn2016.ieee-nfvsdn.org/program/demonstrations/
[6] https://twitter.com/nubomedia/status/796647064499343360
[7] https://wiki.sdn.ieee.org/

of this initiative is to collect, in a single place, information related to SDN, NFV, MEC technologies and standards that can be used by the research community for building prototypes boosting the development of future 5G networks. The main idea, as shown in Figure 8.1, was to combine the approach already taken by other open information catalogs, like Wikipedia, with the peer-review approach utilized for scientific publications. Any IEEE member can contribute to the catalog, either providing content or reviewing contributions uploaded by others.

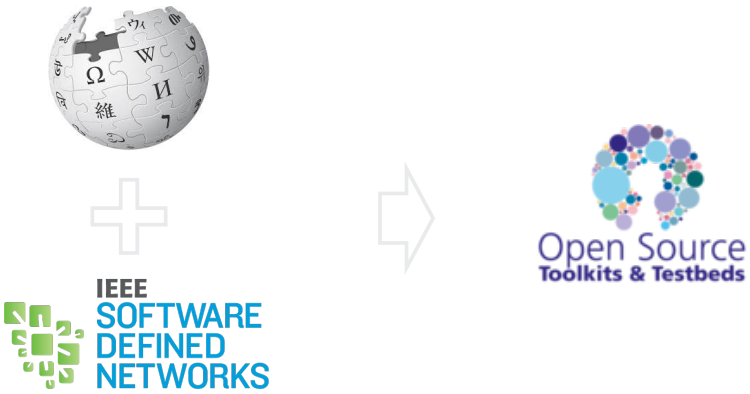

Figure 8.1: The IEEE SDN Catalog of Toolkits and Testbeds

Second, the author also contributed to the IEEE SDN newsletter with articles focusing on the Open Baton open source project:

- [190] providing an overview of the potential employment of the Open Baton framework in MEC scenarios.
- [191] presenting features and functionalities available as part of the Open Baton release 2.

Last but not least, two e-learning modules[8] were generated by the author based on the results obtained during his research activities. Those modules provide an analysis of the state-of-the-art standards and technologies around MANO topics as well as a deep overview of the Open Baton project.

8.1.1.4 Additional Contributions to the Scientific Community

The author organized and chaired several workshops at different editions of the FUSECO Forum. In particular, at the last one titled *"Network Virtualization and Network Slicing for 5G-Ready Networks"* around 15 TSPs shared their lessons learned based on practical experiences using NFV technologies. Especially noteworthy is the fact that several presentations mentioned Open Baton as a NFV MANO

[8] Available at: https://sdn.ieee.org/education/elearningopenbaton

solution, and a few of them also presented their PoCs developed using it[9]. Moreover, the author was involved in several international conferences and workshops as Technical Program Committee (TPC): The Third IEEE International Workshop on Management of 5G Networks (5GMan) in conjunction with the IEEE Network Operations and Management Symposium (NOMS), 2018; the IEEE Global Communications Conference: Next-Generation Networking and Internet (Globecom2017 NGNI), 2017; the Second International Workshop on Software-Driven Flexible and Agile Networking (SWFAN), 2017; The Fourth International Workshop on 5G Architecture (WT01 - 5GArch) in conjunction with the IEEE International Conference on Communications (ICC), 2017.

8.1.1.5 Teaching Contributions

Between 2013 and 2016 the author was a frequent lecturer at the Technical University Berlin for the Chair Architekturen der Vermittlungsknoten (AV) in the Future Internet Technologies (FIT) and Next Generation Network Technologies Services (NGNCourse). Moreover, the author directly supervised numerous Master of Science (MSc) theses and student projects.

8.1.2 Industry Impact of the Author's Work

The Open Baton project source code was published over the GitHub public git repository since its first release. Up to date, Open Baton comprises 38 individual projects, either hosting a particular component or a set of auxiliary scripts or libraries commonly used by other components. As shown in Figure 8.2, the Open Baton public website received a large number of visitors from all over the world, especially from the United States of America (USA) and India.

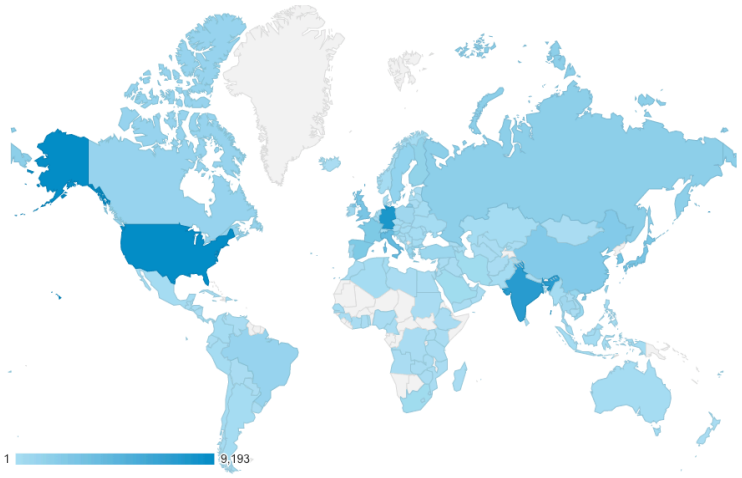

Figure 8.2: Open Baton Website Statistics

[9]https://www.youtube.com/watch?v=lfVy7bKamtU

8.1.2.1 Events Associated with the Launch of Major Open Baton Releases

The first public release of the Open Baton platform was announced and demonstrated at the 6th FUSECO Forum[10], an event organized by Fraunhofer FOKUS and Technische Universität Berlin (TUB), bringing together industry and academia to discuss research challenges in the context of 5G, NFV, and SDN technologies.

The second Open Baton release was presented and demonstrated at the OPNFV Summit, held in Berlin in May 2016. In front of quite a large audience the author presented the numerous features that were added during the second major release of the project, receiving a lot of attraction from both the academic community and industry[11].

The third Open Baton release integrating Juju as VNFM culminated in a joint demonstration at the Canonical booth during the MWC 2017[12]. Figure 8.3 shows the picture of the Open Baton stand at the Canonical booth.

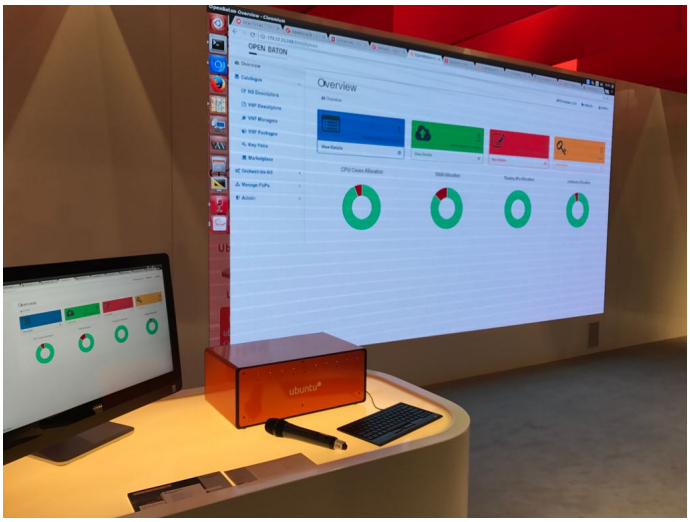

Figure 8.3: Open Baton Stand at the Canonical Booth during the MWC 2017

The demonstration scenario focused on the management and orchestration of the Open Evolved Packet Core (OpenEPC)[192] virtualized mobile core network, provided by Core Network Dynamics[13], using the Juju VNFM. Autoscaling was enabled for showcasing the runtime feature provided by the Open Baton framework for dynamically scaling the user plane based on application-level metrics (i.e., number of subscribers attached to the mobile core network).

[10] https://www.fokus.fraunhofer.de/go/fuseco-forum-2015
[11] https://youtu.be/HKQpCjqrrcY
[12] https://insights.ubuntu.com/2017/03/01/mobile-world-congress-2017-day-3-recap/
[13] https://www.corenetdynamics.com/

8.1.2.2 Presentations at Major Industrial Events

In addition, several talks were given at industrial conferences, especially providing an overview of the work conducted as part of the open source Open Baton project:

- *"Open Baton Overview"*, OPNFV Summit[14], May 2016, Berlin.
- *"OpenSDNCore: a Framework for Prototyping Virtualized Network Function Orchestration in Emerging SDN-based 5G Infrastructures"*, Carrier Network Virtualization, December 2014, Palo Alto.
- *"Smart Communication Platforms for OTT and Telco-integrated Communication Services"*, and *"SDP and Core Network Virtualization"*, 9th SDP Global Summit – Service Delivery Virtualization, September 2013, Rome.

8.1.2.3 Public Blog Posts

Two interviews were also given by the author to SDxCentral[15], a relevant online blog covering topics around *"Software Defined Everything"*[193][194].

8.1.2.4 The OPNFV Orchestra Project

OPNFV[16] is a Linux Foundation open source initiative aiming at creating a reference NFV platform for accelerating the transformation of TSP infrastructures. Since the very beginning, the major objective has been to reduce the gap between what TSPs require for building future network infrastructures and what existing open source projects provide. The scope has been very focused on the NFVI and VIM functional elements of the NFV architecture. For this reason, OPNFV integrates and extends components from upstream projects such as OpenDayLight[17], Open Network Operating System (ONOS)[18], OpenStack, OpenVSwitch, and Linux, directly working with the respective communities providing patches, blueprints, and new code.

In August 2016 the author proposed the creation of the Orchestra project[19] aiming at integrating Open Baton within the OPNFV platform. The initial scope of the project included 1) the integration of the Open Baton bootstrapping procedure in one of the OPNFV installers, 2) the integration of the Open Baton Continuous Integration (CI) system within OPNFV testing projects, and 3) the integration of Open Baton OSS components with existing OPNFV feature projects (like Doctor[20]) for extending the set of use cases supported.

[14] https://wiki.opnfv.org/display/EVNT/Berlin+Design+Summit+Schedule
[15] https://www.sdxcentral.com/
[16] https://www.opnfv.org/
[17] https://www.opendaylight.org/
[18] http://onosproject.org/
[19] https://wiki.opnfv.org/display/PROJ/Orchestra+Home
[20] https://wiki.opnfv.org/display/doctor/Doctor+Home

Objectives 1) and 2) were accomplished and published as part of the Euphrates release. In particular, an Open Baton charm[21] was designed and developed for integrating the bootstrapping procedure within the JOID project[22] as well as two different testing use cases, automatically deploying a vIMS network service via Open Baton, were implemented as part of the FuncTest project[23].

Figure 8.4 shows a snippet[24] of the test results obtained executing some of the Orchestra scenarios using FuncTest.

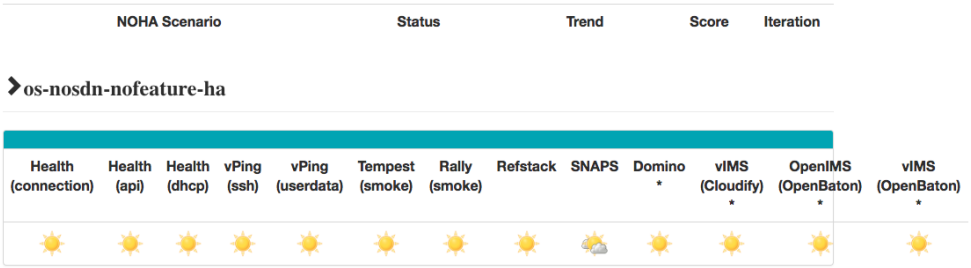

Figure 8.4: OPNFV Test Results Executing Orchestra Use Cases

Since 2016 OPNFV has extended its scope towards the MANO domain, primarily for providing the means for deploying and orchestrating network functions on top of the OPNFV platform comprising heterogeneous technologies. The author has been involved in the MANO working group[25] since its inception, providing his knowledge in the domain for assessing goals and options suitable for integrating upstream MANO projects into OPNFV.

8.1.2.5 The ETSI NFV Plugtests

The first ETSI NFV plugtest took place in January 2017, with the main objective of assessing the level of interoperability between heterogeneous MANO, NFVI, and VNF solutions. A preliminary phase consisted of preparing the infrastructure and installing components participating in the plugtest, as well as defining a set of testing scenarios to be evaluated. Around 160 test sessions combining solutions from different providers were executed during the plugtest, with over 1,500 test results reported[195]. Currently, the author and his team are already actively contributing to the second edition of the ETSI NFV plugtest that will be conducted in Sophia Antipolis in mid-January 2018.

[21] https://github.com/openbaton/juju-charm
[22] https://wiki.opnfv.org/display/joid/JOID+Home
[23] https://wiki.opnfv.org/display/functest/Opnfv+Functional+Testing
[24] The live version of this webpage can be found at the following URL: http://testresults.opnfv.org/reporting/euphrates/functest/status-daisy.html
[25] https://wiki.opnfv.org/display/mano/MANO+Group+Home

8.2 Final Evaluation of Research Questions

At the beginning of this research work (see Section 1.3), the author identified the following main research question to be answered throughout this dissertation:

How to design an extensible and customizable open source Network Function Virtualization Management and Orchestration compliant framework supporting heterogeneous vertical domain requirements on a multisite NFV Infrastructure?

In order to properly design an extensible and customizable NFV MANO compliant framework, an extensive state-of-the-art analysis and consequent requirement analysis were performed and presented in Chapter 2 and Chapter 3, providing an overview about the transition towards software-based networks and network management challenges in future 5G networks. Different ideas, concepts, standards, as well as scientific publications were extensively analyzed and discussed. Based on the analysis conducted by the author, a set of requirements were presented in Chapter 3 while looking at the optimal solution for managing and orchestrating software-based networks.

Extensibility and customizability had been considered throughout the design process. The *"event-driven orchestration"* concept presented in Chapter 4 is the result of the iterative design process of the MANO4X architecture, initiated and managed by the author, and characterized by the parallel evolution of the ETSI NFV standardization work. The final version of the MANO4X architecture, fully compliant with the ETSI NFV MANO one, includes all concepts and methods conceived during the years of research. A functional architecture of the MANO4X framework and different methods required for orchestrating heterogeneous resources were specified in Chapter 5. The final solution proposed can be extended without major changes to its architecture (through domain-driven integration) and with very minimal development efforts (through SDKs provided). Customization is also supported through specific configurations (all components provide several configuration parameters), without requiring modification to the architecture and its implementation.

The open source Open Baton project, launched in 2015, is the result of the implementation activities presented in Chapter 6. Its validation and evaluation was presented in Chapter 7 including results obtained in different research activities performed by the author in the European research ecosystem. As presented in this chapter, today Open Baton represents one out of four globally recognized ETSI NFV MANO frameworks, enabling early 5G prototyping and standardization.

The secondary research questions derived as aspects of the main research question are further explained below, with an overview available in Figure 8.5.

Q1: How to design a framework for end-to-end managing and orchestrating of the whole life cycle of network services?

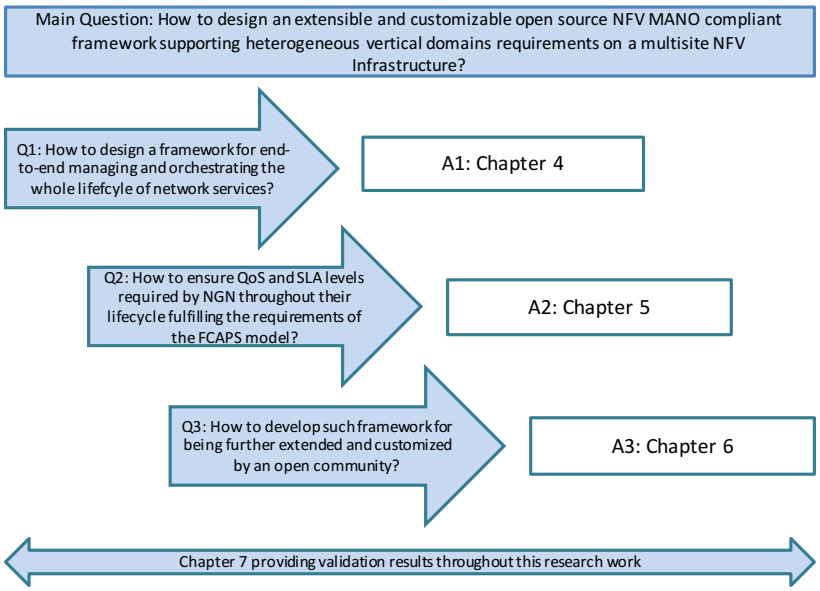

Figure 8.5: Summary Answers to the Key Research Questions of this Dissertation

The research question Q1 was extensively covered in Chapter 4. The design process followed an agile approach, starting from simple scenarios targeting the deployment of individual NF on cloud environments, moving towards more complex scenarios targeting the deployment of complex network services comprising heterogeneous NFs. The separation of the design process in different macro phases allowed the author to identify limitations encountered with the different proposed solutions. Microservices architectures and cloud-native principles influenced the final design decisions made. Following the *Domain Driven Design* the final architecture proposed comprises different domains whose functional elements interact using an event-based approach, enabling end-to-end life cycle orchestration.

Q2: How to ensure Quality of Service and Service Level Agreement levels required by Next Generation Network throughout their life cycle fulfill the requirements of the FCAPS model?

The research question Q2 was extensively covered in Chapter 5, with the specification of the MANO4X framework. One of the critical aspects for software-based networks is that overloaded services cannot maintain the expected satisfactory QoS causing an immediate QoE deterioration. Thus, methods for elastic scalability and recovery, as well as guaranteed QoS were considered and defined as part of the final MANO4X architecture. In particular, dynamic scalability of the deployed service instances[13][129] and QoS management were addressed as part of the east domain (Section 6.6) where OSSs were introduced for extending the MANO4X framework towards runtime critical operations during the runtime life cycle.

Q3: How to develop such framework for being further extended and customized by an open community?

The research question Q3 was extensively covered in Chapter 6, with the implementation of the open source Open Baton framework. The validation proof comes from the different evaluation scenarios presented in Chapter 7 where the author directly contributed, as well as from the usage of the Open Baton framework in several testbeds worldwide, as presented in Section 8.1, allowing researchers from all over the world to experiment with state-of-the-art technologies in the NFV context.

8.3 Outlook

After five years since the ETSI NFV ISG published the first version of the White Paper, the research work around NFV topics starts consolidating, with major tier-1 operators deploying virtualized solutions in production environments.

Looking at the Gartner hype cycle from July 2017[196], there are complementary technologies that could also be considered in the context of this research domain. In particular, edge computing and machine learning technologies are currently in the *"Peak of Inflated Expectations"*, and research challenges related with those technologies have started appearing also as part of new industry-driven initiatives that are extending the work conducted primarily by ETSI NFV so far. ETSI MEC and the Open Edge Computing (OEC) are just a few initiatives around edge computing technologies, while ETSI Zero Touch (under elaboration) and TMF Zero-touch Orchestration, Operations and Management (ZOOM) (started in 2015) are related with machine learning techniques.

8.3.1 MANO4X as the Enabler for Edge Computing Orchestration

Nowadays edge computing starts consolidating, with infrastructure providers deploying computational resources at the edge of the network that can be exploited also for executing VNFs. Those VNFs are in fact better suited for being executed at the edge of the network, possibly even in the Customer Premises Equipment (CPE) residing in the users' homes (sometimes defined also as FOG devices).

The ETSI MEC ISG proposed a set of use cases[197] (and consequently also requirements[198]) for exploiting edge computing technologies in the telecommunication domain. On the one hand, moving network services to the edge is particularly important in scenarios where latency and networking capacity are important factors for the end-user QoE. On the other hand, the complexity of such environment will increase, especially because the edge will be comprised of a set of heterogeneous technologies and hardware devices (e.g., resource-constrained home gateways able to execute lightweight linux containers, servers located near the base station providing common virtualization technologies, etc.) that will need to be administered via a centralized system.

Although some of the features required for supporting edge computing technologies, like multisite and VNF placement, were already covered by this research work[190], MEC requires some alternative solutions for managing in real-time and in a more efficient way VNFs at the edge of the network.

One approach could be to decouple even further the orchestration logic, placing local orchestrators at the edge of the network. Those local entities would be in charge of managing the resources of a particular edge node, executing operations that do not require the intervention of the central orchestration entity.

In this decentralized (asynchronous) approach, the local orchestrator entity could make faster decisions compared to the centralized (synchronous) approach. Extensions to the orchestration logic of the MANO4X framework are required in order to provide asynchronous orchestration at the edge of the network.

8.3.2 Towards Zero Touch Orchestration

Ensuring carrier grade QoS while providing cost efficiency motivates the need for improving MANO operations reducing manual interventions. Although life cycle management operations can be automated with the support of frameworks like the MANO4X, current approaches are based on reactive methods where the system makes decisions based on predefined rules set by the user. Using predefined rules can have an adverse outcome in cases where the user does not have good knowledge of the expected system load and how it changes from one step to the next.

What is actually needed is a framework able to autonomously determine operations that should be executed, making use of predictive machine learning techniques. Time series predictions and reinforcement learning can be beneficial when applied to the MANO domain.

Extending the MANO4X framework for achieving zero touch orchestration would require an adaptation of the external OSS elements in order to make use of machine learning techniques for detecting the need to scale and correlating alarms. One approach could be the replacement of the detection systems already presented in the context of the AES and FMS entities, with a component that uses proactive techniques for detecting conditions and making decisions about next steps to be executed.

Bibliography

[1] Haleplidis, E, K Pentikousis, S Denazis, J Hadi Salim, D Meyer, and O Koufopavlou: *Software-Defined Networking (SDN): Layers and Architecture Terminology*. RFC 7426 (Informational), jan 2015. http://www.ietf.org/rfc/rfc7426.txt. (Cited on page 1.)

[2] Feamster, Nick, Jennifer Rexford, and Ellen Zegura: *The road to SDN*. ACM SIGCOMM Computer Communication Review, 44(2):87–98, apr 2014, ISSN 01464833. http://dl.acm.org/citation.cfm?doid=2602204.2602219. (Cited on page 1.)

[3] *The latest tidbits on Sun deals and product news - SunWorld - February 1997*. http://sunsite.uakom.sk/sunworldonline/swol-02-1997/swol-02-sunspots.html. (Cited on page 1.)

[4] ETSI NFV: *ETSI GS NFV 003 v1.1.1 (2013-10) - Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV Group Specification*. Technical report, 2013. http://www.etsi.org/deliver/etsi{_}gs/NFV/001{_}099/003/01.01.01{_}60/gs{_}NFV003v010101p.pdf. (Cited on page 1.)

[5] Psounis, Konstantinos: *Active networks: Applications, security, safety, and architectures*. IEEE Communications Surveys & Tutorials, 2(1):2–16, 1999, ISSN 1553-877X. http://ieeexplore.ieee.org/document/5340509/. (Cited on page 1.)

[6] Magedanz, Thomas. and R. Popescu-Zeletin: *Intelligent networks*. International Thomson Computer Press, London, 1996, ISBN 9781850322931. (Cited on page 1.)

[7] Erickson, David: *Open Networking Foundation Formed to Speed Network Innovation*, 2011. http://archive.openflow.org/wp/2011/03/open-networking-foundation-formed-to-speed-network-innovation/, visited on 2016-01-15. (Cited on page 1.)

[8] Daniels, Jeff and Jeff: *Server virtualization architecture and implementation*. Crossroads, 16(1):8–12, sep 2009, ISSN 15284972. http://portal.acm.org/citation.cfm?doid=1618588.1618592. (Cited on page 2.)

[9] *Cisco VNI Mobile Forecast (2015 – 2020) – Cisco - Cisco*. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html, visited on 2017-01-15. (Cited on page 2.)

[10] Mijumbi, Rashid, Joan Serrat, Juan Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba: *Network Function Virtualization: State-of-the-Art and Research Challenges*. IEEE Communications Surveys & Tutorials, 18(1):236–262, 2016, ISSN 1553-877X. http://ieeexplore.ieee.org/document/7243304/. (Cited on pages 2 and 8.)

[11] Iyer, Sundar: *Virtualisation of network functions and the SDN: Improving the economics of the network*. Australian Journal of Telecommunications and the Digital Economy, 2(2), jun 2014, ISSN 22031693. http://telsoc.org/ajtde/2014-06-v2-n2/a41. (Cited on page 3.)

[12] Lopez, D R: *Network functions virtualization: Beyond carrier-grade clouds*. In *OFC 2014*, pages 1–18, mar 2014. (Cited on page 3.)

[13] Bellavista, Paolo, Giuseppe Carella, Luca Foschini, Thomas Magedanz, Florian Schreiner, and Konrad Campowsky: *QoS-aware elastic cloud brokering for IMS infrastructures*. In *Proceedings - IEEE Symposium on Computers and Communications*, pages 000157–000160, 2012. (Cited on pages 3, 4, 59, 77, 80, 83, 120, 169 and 222.)

[14] Carella, Giuseppe, Thomas Magedanz, Konrad Campowsky, and Florian Schreiner: *Elasticity as a service for federated cloud testbeds*. In *2013 IEEE International Conference on Communications Workshops (ICC)*, pages 256–260. IEEE, jun 2013, ISBN 978-1-4673-5753-1. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6649239. (Cited on pages 3, 77, 80, 120 and 182.)

[15] *Building NFV Business Benefits*. https://www.sdxcentral.com/cisco/service-provider/info/analysis/building-nfv-business-benefits/, visited on 2017-01-15. (Cited on page 3.)

[16] Wang, Lizhe, Gregor Von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu: *Cloud computing: a perspective study*. New Generation Computing, 28(2):137–146, 2010. (Cited on pages 3 and 18.)

[17] Chowdhury, N.M.M.K. and R. Boutaba: *Network virtualization: state of the art and research challenges*. IEEE Communications Magazine, 47(7):20–26, jul 2009, ISSN 0163-6804. http://ieeexplore.ieee.org/document/5183468/. (Cited on page 4.)

[18] Wei Quan, Jun Wu, Xiaosu Zhan, Xiaohong Huang, and Yan Ma: *Research of presence service testbed on cloud-computing environment*. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pages 865–869. IEEE, oct 2010, ISBN 978-1-4244-6769-3. http://ieeexplore.ieee.org/document/5705213/. (Cited on page 4.)

[19] Jebalia, Maha, Asma Ben Letaifa, and Sami Tabbane: *Emerging Technologies of the Modern Services Industry*. In *2010 Second International Conference*

on Network Applications, Protocols and Services, pages 110–113. IEEE, sep 2010, ISBN 978-1-4244-8048-7. http://ieeexplore.ieee.org/document/5635657/. (Cited on page 4.)

[20] Chen, J L, S L Wuy, Y T Larosa, P J Yang, and Y F Li: *IMS cloud computing architecture for high-quality multimedia applications*. In *2011 7th International Wireless Communications and Mobile Computing Conference*, pages 1463–1468, jul 2011. (Cited on pages 4 and 80.)

[21] Gouveia, Fabricio, Sebastian Wahle, Niklas Blum, and Thomas Magedanz: *Cloud computing and EPC / IMS integration: new value-added services on demand*. In *Proceedings of the 5th International Mobile Multimedia Communications Conference*, page 51. ICST, sep 2009, ISBN 978-963-9799-62-2. http://dl.acm.org/citation.cfm?id=1653543.1653604. (Cited on page 4.)

[22] *Network Function Virtualization WhitePaper*, 2012. https://portal.etsi.org/nfv/nfv{_}white{_}paper.pdf. (Cited on page 4.)

[23] Hu, Yun Chao, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young: *Mobile edge computing—A key technology towards 5G*. ETSI White Paper, 11, 2015. (Cited on page 4.)

[24] Han, Bo, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee: *Network function virtualization: Challenges and opportunities for innovations*. IEEE Communications Magazine, 53(2):90–97, feb 2015. http://ieeexplore.ieee.org/document/7045396/. (Cited on page 4.)

[25] Saydam, Tuncay and Thomas Magedanz: *From networks and network management into service and service management*. Journal of Network and Systems Management, 4(4):345–348, 1996, ISSN 1573-7705. http://dx.doi.org/10.1007/BF02283158. (Cited on pages 6 and 19.)

[26] *White Paper on NFV priorities for 5G*. \unskip\space, feb 2017. https://portal.etsi.org/NFV/NFV{_}White{_}Paper{_}5G.pdf. (Cited on pages 7 and 62.)

[27] ITU-T: *M.3400 : TMN management functions*, feb 2000. https://www.itu.int/rec/T-REC-M.3400/en. (Cited on pages 7, 21 and 22.)

[28] *ETSI GS NFV-MAN 001 V1.1.1 - Network Functions Virtualisation (NFV); Management and Orchestration*. Technical report, 2014. (Cited on pages 8, 52, 53, 54, 97, 110, 126, 148 and 265.)

[29] *Gigaom | SDN, NFV, and open source: the operator's view*. https://gigaom.com/report/sdn-nfv-and-open-source-the-operators-view/. (Cited on page 8.)

[30] *Taming the NFV 'Orchestration Zoo' | Light Reading.* http://www.lightreading.com/nfv/nfv-mano/taming-the-nfv-orchestration-zoo/d/d-id/716244, visited on 2016-06-04. (Cited on page 8.)

[31] Schwaber, Ken and Mike Beedle: *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002. (Cited on pages 13 and 76.)

[32] Newcomer, Eric and Greg Lomow: *Understanding SOA with Web services.* Addison-Wesley, 2005. (Cited on page 18.)

[33] Wood, Timothy, K K Ramakrishnan, Jinho Hwang, Grace Liu, and Wei Zhang: *Toward a software-based network: integrating software defined networking and network function virtualization.* IEEE Network, 29(3):36–41, may 2015, ISSN 0890-8044. http://ieeexplore.ieee.org/document/7113223/. (Cited on page 18.)

[34] Subramanian, Mani: *Network management: principles and practice.* Pearson Education India, 2010. (Cited on page 20.)

[35] Fedor, Mark, James R Davin, Martin Lee Schoffstall, and Dr. Jeff D Case: *Simple Network Management Protocol (SNMP).* RFC 1157, may 1990. https://rfc-editor.org/rfc/rfc1157.txt. (Cited on page 21.)

[36] Presuhn, Randy: *Management Information Base (MIB) for the Simple Network Management Protocol (SNMP).* RFC 3418, dec 2002. https://rfc-editor.org/rfc/rfc3418.txt. (Cited on page 21.)

[37] *SNMP, SNMPv2 and CMIP — the technologies for multivendor network management.* Computer Communications, 20(2):73–88, mar 1997, ISSN 0140-3664. http://www.sciencedirect.com/science/article/pii/S0140366496011723. (Cited on page 21.)

[38] ITU-T: *M.3000 : Overview of TMN Recommendations*, feb 2000. http://www.itu.int/rec/T-REC-X.200/en. (Cited on page 21.)

[39] *ITU-T: M.3010 - Principles for a telecommunications management network.* Technical report. https://www.itu.int/rec/T-REC-M.3010. (Cited on page 21.)

[40] ITU-T: *X.700 : Management framework for Open Systems Interconnection (OSI) for CCITT applications*, sep 1992. https://www.itu.int/rec/T-REC-X.700/en. (Cited on page 21.)

[41] ITU-T: *M.3200 : TMN management services and telecommunications managed areas: overview*, apr 1997. https://www.itu.int/rec/T-REC-M.3200/en. (Cited on page 21.)

[42] Raman, Lakshmi G: *{OSI} Systems and Network Management*. 36(3):46–53, 1998. http://faculty.wiu.edu/Y-Kim2/OSISysNetMagt.pdfhttp://pubs.comsoc.org/ci1. (Cited on page 22.)

[43] Yemini, Y: *The OSI network management model*. IEEE Communications Magazine, 31(5):20–29, may 1993, ISSN 0163-6804. (Cited on page 22.)

[44] Knightson, Keith, Naotaka Morita, and Thomas Towle: *NGN architecture: generic principles, functional architecture, and implementation*. IEEE Communications Magazine, 43(10):49–56, 2005. (Cited on page 22.)

[45] ITU-T: *ITU-T Recommendation Y.2001 (12/2004) - General overview of NGN*, dec 2004. http://www.itu.int/rec/T-REC-Y.2001. (Cited on pages xiii, 22 and 23.)

[46] Hill, Goff: *The Cable and Telecommunications Professionals' Reference: PSTN, IP and cellular networks, and mathematical techniques*, volume 1. Taylor & Francis, 2007. (Cited on page 23.)

[47] 3GPP: *TS 23.228 - IP Multimedia Subsystem (IMS); Stage 2*. http://www.3gpp.org/ftp/Specs/html-info/23228.htm. (Cited on page 24.)

[48] Li, Mo and K Sandrasegaran: *Network management challenges for next generation networks*. In *The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)l*, pages 6 pp.–598, nov 2005. (Cited on page 24.)

[49] Ray, Pradeep Kumar: *Integrated Management from E-Business Perspective*. Network and Systems Management. Springer US, Boston, MA, 2003, ISBN 978-1-4613-4918-1. http://link.springer.com/10.1007/978-1-4615-0089-6. (Cited on page 24.)

[50] Salus, Peter H., Peter H./Foreword By-Cerf, and Vinton G.: *Casting the net : from ARPANET to Internet and beyond*. Addison-Wesley Pub. Co, 1995, ISBN 0201876744. http://dl.acm.org/citation.cfm?id=545810. (Cited on page 26.)

[51] Strassner, John, Ian Foster, Dr. Reagan W Moore, Benjamin R Teitelbaum, Dr. Clifford A Lynch, Brian E Carpenter, Joe Mambretti, and Robert J Aiken: *Network Policy and Services: A Report of a Workshop on Middleware*. RFC 2768, feb 2000. https://rfc-editor.org/rfc/rfc2768.txt. (Cited on page 26.)

[52] Vinoski, Steve: *CORBA: integrating diverse applications within distributed heterogeneous environments*. IEEE Communications magazine, 35(2):46–55, 1997. (Cited on page 27.)

[53] Michael Hogan and Annie Sokol: *NIST Cloud Computing Standards Roadmap*. https://www.nist.gov/sites/default/files/documents/itl/

cloud/NIST{_}SP-500-291{_}Version-2{_}2013{_}June18{_}FINAL.pdf. (Cited on page 27.)

[54] MIT Technology Review: *The Cloud Imperative*. https://www.technologyreview.com/s/425623/the-cloud-imperative/. (Cited on page 27.)

[55] Mastorakis, George: *Resource Management of Mobile Cloud Computing Networks and Environments*. IGI Global, Hershey, PA, USA, 1st edition, 2015, ISBN 1466682256, 9781466682252. (Cited on page 27.)

[56] Jennings, Brendan and Rolf Stadler: *Resource Management in Clouds: Survey and Research Challenges*. Journal of Network and Systems Management, 23(3):567–619, jul 2015, ISSN 1064-7570. http://link.springer.com/10.1007/s10922-014-9307-7. (Cited on page 27.)

[57] Heckel, Philipp C: *Hybrid clouds: comparing cloud toolkits*. In *Seminar Paper Business Informatics, University of Mannheim*. Citeseer, 2010. (Cited on pages 28 and 31.)

[58] Graziano, Charles: *A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project*. Graduate Theses and Dissertations, 2011. http://lib.dr.iastate.edu/etd/12215. (Cited on page 28.)

[59] Chiueh, Susanta Nanda Tzi cker and Stony Brook: *A survey on virtualization technologies*. RPE Report, pages 1–42, 2005. (Cited on pages 28 and 29.)

[60] Carapinha, Jorge and Javier Jimenez: *Network virtualization*. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures - VISA '09*, page 73, New York, New York, USA, 2009. ACM Press, ISBN 9781605585956. http://portal.acm.org/citation.cfm?doid=1592648.1592660. (Cited on page 28.)

[61] Berl, Andreas, Andreas Fischer, and Hermann de Meer: *Using system virtualization to create virtualized networks*. Electronic Communications of the EASST, 17, 2009. (Cited on page 28.)

[62] Qian, Ling, Zhiguo Luo, Yujian Du, and Leitao Guo: *Cloud computing: An overview*. Cloud computing, pages 626–631, 2009. (Cited on page 29.)

[63] Daniels, Jeff and Jeff: *Server virtualization architecture and implementation*. Crossroads, 16(1):8–12, sep 2009. http://portal.acm.org/citation.cfm?doid=1618588.1618592. (Cited on page 29.)

[64] Matthews, Jeanna N, Eli M Dow, Todd Deshane, Wenjin Hu, Jeremy Bongio, Patrick F Wilbur, and Brendan Johnson: *Running Xen: a hands-on guide to the art of virtualization*. Prentice Hall PTR, 2008. (Cited on page 30.)

[65] Habib, Irfan: *Virtualization with kvm.* Linux Journal, 2008(166):8, 2008. (Cited on page 30.)

[66] Bellard, Fabrice: *QEMU, a fast and portable dynamic translator.* In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005. (Cited on page 30.)

[67] Xavier, M G, M V Neves, F D Rossi, T C Ferreto, T Lange, and C A F De Rose: *Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments.* In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, feb 2013. (Cited on page 30.)

[68] Mell, Peter and Tim Grance: *The NIST definition of cloud computing.* 2011. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf. (Cited on page 31.)

[69] Edmonds, Andy, Thijs Metsch, Alexander Papaspyrou, and Alexis Richardson: *Toward an open cloud standard.* IEEE Internet Computing, 16(4):15–25, jul 2012, ISSN 10897801. http://ieeexplore.ieee.org/document/6200250/. (Cited on pages xiii, 33 and 34.)

[70] Manvi, Sunilkumar S. and Gopal Krishna Shyam: *Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey.* Journal of Network and Computer Applications, 41:424–440, may 2014, ISSN 10848045. http://www.sciencedirect.com/science/article/pii/S1084804513002099http://linkinghub.elsevier.com/retrieve/pii/S1084804513002099. (Cited on page 35.)

[71] Yadav, Sonali: *Comparative study on open source software for cloud computing platform: Eucalyptus, openstack and opennebula.* International Journal Of Engineering And Science, 3(10):51–54, 2013. (Cited on page 35.)

[72] Pfaff, Ben, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, and Others: *The Design and Implementation of Open vSwitch.* In *NSDI*, pages 117–130, 2015. (Cited on page 36.)

[73] Erl, Thomas: *Service-oriented architecture: concepts, technology, and design.* Pearson Education India, 2005. (Cited on page 37.)

[74] MacKenzie, CM, K Laskey, F McCabe, and PF Brown: *Reference model for service oriented architecture 1.0.* OASIS, 2006. https://www.oasis-open.org/committees/download.php/18486/pr2changes.pdf. (Cited on page 37.)

[75] Herbst, Nikolas Roman, Samuel Kounev, and Ralf Reussner: *Elasticity in Cloud Computing: What It Is, and What It Is Not.* In

Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), pages 23–27, San Jose, CA, 2013. USENIX, ISBN 978-1-931971-02-7. https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst. (Cited on page 38.)

[76] Balalaie, Armin, Abbas Heydarnoori, and Pooyan Jamshidi: *Microservices architecture enables DevOps: migration to a cloud-native architecture*. IEEE Software, 33(3):42–52, 2016. (Cited on page 38.)

[77] Bonvin, Nicolas, Thanasis G. Papaioannou, and Karl Aberer: *A self-organized, fault-tolerant and scalable replication scheme for cloud storage*. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, page 205, New York, New York, USA, 2010. ACM Press, ISBN 9781450300360. http://portal.acm.org/citation.cfm?doid=1807128.1807162. (Cited on page 38.)

[78] Gill, Phillipa, Navendu Jain, Nachiappan Nagappan, Phillipa Gill, Navendu Jain, and Nachiappan Nagappan: *Understanding network failures in data centers*. In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM - SIGCOMM '11*, volume 41, page 350, New York, New York, USA, 2011. ACM Press, ISBN 9781450307970. http://dl.acm.org/citation.cfm?doid=2018436.2018477. (Cited on page 38.)

[79] Nygard, Michael: *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf, 2007. (Cited on page 38.)

[80] *Cloud Design Patterns | Microsoft Docs*. https://docs.microsoft.com/en-us/azure/architecture/patterns/. (Cited on page 38.)

[81] *Microservices, SOA, and APIs: Friends or enemies?* https://www.ibm.com/developerworks/websphere/library/techarticles/1601{_}clark-trs/1601{_}clark.html. (Cited on page 38.)

[82] Abbott, Martin L and Michael T Fisher: *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009. (Cited on pages xiii, 38 and 39.)

[83] Httermann, Michael: *DevOps for developers*. Apress, 2012. (Cited on page 41.)

[84] Loukides, Mike: *What is DevOps?* " O'Reilly Media, Inc.", 2012. (Cited on pages 41 and 42.)

[85] Talwar, V., D. Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung: *Approaches for Service Deployment*. IEEE Internet Computing, 9(2):70–80, mar 2005, ISSN 1089-7801. http://ieeexplore.ieee.org/document/1405978/. (Cited on page 42.)

[86] Endres, Christian: *Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications*. In *Conference on Pervasive Patterns and Applications*, pages 22–27, 2017. (Cited on page 42.)

[87] Breitenb?cher, Uwe, Tobias Binz, Oliver Kopp, Frank Leymann, and Johannes Wettinger: *Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies*. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 130–148. Springer, 2013. http://link.springer.com/10.1007/978-3-642-41030-7{_}9. (Cited on page 42.)

[88] Breitenbücher, Uwe, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger: *Combining declarative and imperative cloud application provisioning based on TOSCA*. In *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, pages 87–96. IEEE, mar 2014, ISBN 9781479937660. http://ieeexplore.ieee.org/document/6903461/. (Cited on page 43.)

[89] OASIS: *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. Technical report, 2013. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf. (Cited on pages xiii, 43 and 44.)

[90] Binz, Tobias, Gerd Breiter, Frank Leyman, and Thomas Spatzier: *Portable Cloud Services Using TOSCA*. IEEE Internet Computing, 16(3):80–85, may 2012, ISSN 1089-7801. http://ieeexplore.ieee.org/document/6188582/. (Cited on page 43.)

[91] OASIS: *TOSCA Simple Profile in YAML Version 1.0*. Technical report, 2016. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/os/TOSCA-Simple-Profile-YAML-v1.0-os.pdf. (Cited on page 44.)

[92] ETSI, NFVISG: *Network functions virtualization, white paper*, 2012. (Cited on page 44.)

[93] ETSI NFV: *ETSI GS NFV 002 V1.1.1 (2013-10) - Network Functions Virtualisation (NFV); Architectural Framework*. Technical report, 2013. http://www.etsi.org/deliver/etsi{_}gs/nfv/001{_}099/002/01.01.01{_}60/gs{_}nfv002v010101p.pdf. (Cited on pages xiii, 46, 47 and 79.)

[94] ETSI NFV: *ETSI GS NFV-SWA 001 V1.1.1 (2014-12) - Network Functions Virtualisation (NFV); Virtual Network Functions Architecture*. Technical report, 2014. http://www.etsi.org/deliver/etsi{_}gs/NFV-SWA/001{_}099/001/01.01.01{_}60/gs{_}NFV-SWA001v010101p.pdf. (Cited on pages xiii, 50 and 51.)

[95] ETSI NFV: *ETSI GS NFV-IFA 001 V1.1.1 (2015-12) - Network Functions Virtualisation (NFV); Acceleration Technologies; Report on Acceleration Technologies & Use Cases*. Technical report, 2015. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/001/01.01.01{_}60/gs{_}NFV-IFA001v010101p.pdf. (Cited on page 56.)

[96] ETSI NFV: *ETSI GS NFV-IFA 005 V2.1.1 (2016-04) - Network Functions Virtualisation (NFV); Management and Orchestration; Or-Vi reference point - Interface and Information Model Specification* . Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/005/02.01.01{_}60/gs{_}NFV-IFA005v020101p.pdf. (Cited on page 56.)

[97] ETSI NFV: *ETSI GS NFV-IFA 006 V2.1.1 (2016-04) - Network Functions Virtualisation (NFV); Management and Orchestration; Vi-Vnfm reference point - Interface and Information Model Specification.* Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/005/02.01.01{_}60/gs{_}NFV-IFA005v020101p.pdf. (Cited on pages 56 and 115.)

[98] ETSI NFV: *ETSI GS NFV-IFA 007 V2.1.1 (2016-10) - Network Functions Virtualisation (NFV); Management and Orchestration; Or-Vnfm reference point - Interface and Information Model Specification.* Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/007/02.01.01{_}60/gs{_}NFV-IFA007v020101p.pdf. (Cited on page 56.)

[99] ETSI NFV: *ETSI GS NFV-IFA 008 V2.1.1 (2016-10) - Network Functions Virtualisation (NFV); Management and Orchestration; Ve-Vnfm reference point - Interface and Information Model Specification* . Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/008/02.01.01{_}60/gs{_}NFV-IFA008v020101p.pdf. (Cited on page 56.)

[100] ETSI NFV: *ETSI GS NFV-IFA 011 V2.1.1 (2016-10) - Network Functions Virtualisation (NFV); Management and Orchestration; VNF Packaging Specification* . Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/011/02.01.01{_}60/gs{_}NFV-IFA011v020101p.pdf. (Cited on page 56.)

[101] ETSI NFV: *ETSI GS NFV-IFA 013 V2.1.1 (2016-10) - Network Functions Virtualisation (NFV); Management and Orchestration; Os-Ma-Nfvo reference point - Interface and Information Model Specification* . Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/013/02.01.01{_}60/gs{_}NFV-IFA013v020101p.pdf. (Cited on page 56.)

[102] ETSI NFV: *ETSI GS NFV-IFA 014 V2.1.1 (2016-10) - Network Functions Virtualisation (NFV); Management and Orchestration; Network Service Templates Specification* . Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/014/02.01.01{_}60/gs{_}NFV-IFA014v020101p.pdf. (Cited on page 56.)

[103] *5G White Paper*. \unskip\space, NGMN Alliance, feb 2015. https://www.ngmn.org. (Cited on pages xiii and 57.)

[104] *5GMF White Paper - 5G Mobile Communications Systems for 2020 and beyond*. 2017. http://5gmf.jp/wp/wp-content/uploads/2017/10/5GMF-White-Paper-v1{_}1-All.pdf. (Cited on pages xiii and 58.)

[105] 3GPP: *Feasibility Study on New Services and Markets Technology Enablers*. Tr 22.891, 3rd Generation Partnership Project (3GPP), sep 2016. (Cited on page 58.)

[106] 3GPP: *Study on Architecture for Next Generation System*. Tr 23.799, 3rd Generation Partnership Project (3GPP), dec 2016. (Cited on page 59.)

[107] *5G PPP Architecture Working Group - View on 5G Architecture*. 2016. https://5g-ppp.eu/wp-content/uploads/2014/02/5G-PPP-5G-Architecture-WP-July-2016.pdf. (Cited on pages xiii and 59.)

[108] Zúñiga, Juan Carlos, Carlos Jésus Bernardos, Akbar Rahman, Luis M Contreras, Pierre Lynch, and Pedro Andres Aranda: *Network Virtualization Research Challenges*. Internet-draft draft-irtf-nfvrg-gaps-network-virtualization-05, Internet Engineering Task Force, mar 2017. https://datatracker.ietf.org/doc/html/draft-irtf-nfvrg-gaps-network-virtualization-05. (Cited on page 60.)

[109] Bernardos, Carlos Jésus, Luis M Contreras, Ishan Vaishnavi, and Robert Szabo: *Multi-domain Network Virtualization*. Internet-draft draft-bernardos-nfvrg-multidomain-02, Internet Engineering Task Force, mar 2017. https://datatracker.ietf.org/doc/html/draft-bernardos-nfvrg-multidomain-02. (Cited on page 60.)

[110] Shin, Myung Ki, Ki Hyuk Nam, Sangheon Pack, Seungik Lee, and Ram (Ramki) Krishnan: *Verification of NFV Services : Problem Statement and Challenges*. Internet-draft draft-irtf-nfvrg-service-verification-03, Internet Engineering Task Force, mar 2017. https://datatracker.ietf.org/doc/html/draft-irtf-nfvrg-service-verification-03. (Cited on page 60.)

[111] Galis, Alex: *Network Slicing - Revised Problem Statement*. Internet-draft draft-galis-netslices-revised-problem-statement-00, Internet Engineering Task Force, jun 2017. https://datatracker.ietf.org/doc/html/draft-galis-netslices-revised-problem-statement-00. (Cited on page 60.)

[112] Kiran.makhijani@huawei.com, Jun Qin, Ravi Ravindran, Liang Geng, Li Qiang, Shuping Peng, Xavier de Foy, Akbar Rahman, and Alex

Galis: *Network Slicing Use Cases: Network Customization and Differentiated Services*. Internet-draft draft-netslices-usecases-00, Internet Engineering Task Force, jun 2017. https://datatracker.ietf.org/doc/html/draft-netslices-usecases-00. (Cited on page 60.)

[113] Rahman, Akbar and Xavier de Foy: *Network Slicing - 3GPP Use Case*. Internet-draft draft-defoy-netslices-3gpp-network-slicing-01, Internet Engineering Task Force, apr 2017. https://datatracker.ietf.org/doc/html/draft-defoy-netslices-3gpp-network-slicing-01. (Cited on page 60.)

[114] Geng, Liang, Stewart Bryant, Jie Dong, Kiran.makhijani@huawei.com, Alex Galis, Xavier de Foy, and Slawomir Kuklinski: *Network Slicing Architecture*. Internet-draft draft-geng-netslices-architecture-01, Internet Engineering Task Force, jun 2017. https://datatracker.ietf.org/doc/html/draft-geng-netslices-architecture-01. (Cited on page 60.)

[115] Qiang, Li, Pedro Martinez-Julia, Liang Geng, Jie Dong, Kiran.makhijani@huawei.com, Alex Galis, Susan Hares, and Slawomir: *Gap Analysis for Network Slicing*. Internet-draft draft-qiang-netslices-gap-analysis-00, Internet Engineering Task Force, jun 2017. https://datatracker.ietf.org/doc/html/draft-qiang-netslices-gap-analysis-00. (Cited on page 60.)

[116] Kreutz, Diego, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig: *Software-Defined Networking: A Comprehensive Survey*. Proceedings of the IEEE, 103(1):14–76, jan 2015, ISSN 0018-9219. http://ieeexplore.ieee.org/document/6994333/. (Cited on page 60.)

[117] Shenker, Scott: *The Future of Networking, and the Past of Protocols*. http://docs.huihoo.com/open-networking-summit/2011/the-future-of-networking-and-the-past-of-protocols.pdf. (Cited on page 61.)

[118] Mckeown, Nick: *SDN and Streamlining the Plumbing*. http://www.comsnets.org/archive/2014/doc/NickMcKeownsSlides.pdf. (Cited on page 61.)

[119] McKeown, Nick, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner: *OpenFlow: enabling innovation in campus networks*. ACM SIGCOMM Computer Communication Review, 38(2):69–74, 2008. (Cited on page 61.)

[120] Medhat, Ahmed M, Tarik Taleb, Asma Elmangoush, Giuseppe A Carella, Stefan Covaci, and Thomas Magedanz: *Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges*. IEEE Communications Magazine, 55(2):216–223, feb 2017, ISSN 0163-6804. http://ieeexplore.ieee.org/document/7593430/. (Cited on pages 62 and 78.)

[121] Carella, Giuseppe, Marius Corici, Paolo Crosta, Paolo Comi, Thomas Michael Bohnert, Andreea Ancuta Corici, Dragos Vingarzan, and Thomas Magedanz: *Cloudified IP Multimedia Subsystem (IMS) for Network Function Virtualization (NFV)-based architectures*. In *2014 IEEE Symposium on Computers and Communications (ISCC)*, volume Workshops, pages 1–6. IEEE, jun 2014, ISBN 978-1-4799-4277-0. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6912647. (Cited on pages 65, 77, 87, 129, 169 and 214.)

[122] Magedanz, Thomas, Giuseppe Carella, Marius Corici, Julius Mueller, and Andreas Weber: *Prototyping new concepts beyond 4G – The Fraunhofer Open5GCore*. it - Information Technology, 57(5), jan 2015, ISSN 1611-2776. https://www.degruyter.com/view/j/itit.2015.57.issue-5/itit-2015-0021/itit-2015-0021.xml. (Cited on pages 65, 77 and 87.)

[123] Corici, Andreea Ancuta, Ranjan Shrestha, Giuseppe Carella, Asma Elmangoush, Ronald Steinke, and Thomas Magedanz: *A solution for provisioning reliable M2M infrastructures using SDN and device management*. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, pages 81–86. IEEE, may 2015, ISBN 978-1-4799-7752-9. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7231401. (Cited on pages 65, 77 and 87.)

[124] ETSI NFV: *ETSI GS NFV-IFA 010 V2.1.1 (2016-04) - Network Functions Virtualisation (NFV); Management and Orchestration; Functional requirements specification* . Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/NFV-IFA/001{_}099/010/02.01.01{_}60/gs{_}nfv-ifa010v020101p.pdf. (Cited on page 65.)

[125] Cohn, Mike: *User stories applied: For agile software development*. Addison-Wesley Professional, 2004. (Cited on page 68.)

[126] Pras, A. and J. Schoenwaelder: *On the difference between information models and data models*, jan 2003. https://www.rfc-editor.org/info/rfc3444http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:On+the+Difference+between+Information+Models+and+Data+Models{#}0. (Cited on page 76.)

[127] Kavoussanakis, Konstantinos, Alastair Hume, Josep Martrat, Carmelo Ragusa, Michael Gienger, Konrad Campowsky, Gregory Van Seghbroeck, Constantino Vazquez, Celia Velayos, Frederic Gittler, Philip Inglesant, Giuseppe Carella, Vegard Engen, Michal Giertych, Giada Landi, and David Margery: *BonFIRE: The Clouds and Services Testbed*. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 321–326. IEEE, dec 2013, ISBN 978-0-7695-5095-4. http://ieeexplore.ieee.

org/lpdocs/epic03/wrapper.htm?arnumber=6735444. (Cited on pages 77, 82 and 168.)

[128] Campowsky, Konrad, Giuseppe Carella, Thomas Magedanz, and Florian Schreiner: *Optimization of Elastic Cloud Brokerage Mechanisms for Future Telecommunication Service Environments*, 2012. ISSN 0930-5157. (Cited on pages 77, 80, 83, 120 and 169.)

[129] Carella, Giuseppe, Thomas Magedanz, Konrad Campowsky, and Florian Schreiner: *Network-aware Cloud Brokerage for telecommunication services*. In *2012 1st IEEE International Conference on Cloud Networking, CLOUDNET 2012 - Proceedings*, pages 131–136, 2012. (Cited on pages 77, 80, 83, 120, 169, 182, 214 and 222.)

[130] Sousa, Bruno, Luis Cordeiro, Paulo Simoes, Andy Edmonds, Santiago Ruiz, Giuseppe A. Carella, Marius Corici, Navid Nikaein, Andre S. Gomes, Eryk Schiller, Torsten Braun, and Thomas Michael Bohnert: *Toward a Fully Cloudified Mobile Network Infrastructure*. IEEE Transactions on Network and Service Management, 13(3):547–563, sep 2016, ISSN 1932-4537. http://ieeexplore.ieee.org/document/7534870/. (Cited on pages 77, 89 and 169.)

[131] Edmonds, Andy, Giuseppe Carella, Faqir Zarrar Yousaf, Carlos Goncalves, Thomas Michael Bohnert, Thijs Metsch, Paolo Bellavista, and Luca Foschini: *An OCCI-compliant framework for fine-grained resource-aware management in Mobile Cloud Networking*. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 1306–1313. IEEE, jun 2016, ISBN 978-1-5090-0679-3. http://ieeexplore.ieee.org/document/7543918/. (Cited on pages 77, 89, 169 and 182.)

[132] Cau, Eleonora, Marius Corici, Paolo Bellavista, Luca Foschini, Giuseppe Carella, Andy Edmonds, and Thomas Michael Bohnert: *Efficient Exploitation of Mobile Edge Computing for Virtualized 5G in EPC Architectures*. In *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 100–109. IEEE, mar 2016, ISBN 978-1-5090-1754-6. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7474417. (Cited on page 77.)

[133] Carella, Giuseppe, Luca Foschini, Alessandro Pernafini, Paolo Bellavista, Antonio Corradi, Marius Corici, Florian Schreiner, and Thomas Magedanz: *Quality Audit and Resource Brokering for Network Functions Virtualization (NFV) Orchestration in Hybrid Clouds*. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, dec 2015, ISBN 978-1-4799-5952-5. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7417385. (Cited on pages 77, 169 and 182.)

[134] Medhat, Ahmed M., Giuseppe Carella, Christian Luck, Marius Iulian Corici, and Thomas Magedanz: *Near optimal service function path instantiation*

in a multi-datacenter environment. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 336–341. IEEE, nov 2015, ISBN 978-3-9018-8277-7. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7367379. (Cited on pages 77, 125 and 126.)

[135] Carella, Giuseppe, Junnosuke Yamada, Niklas Blum, Christian Luck, Naoyoshi Kanamaru, Naoki Uchida, and Thomas Magedanz: *Cross-layer service to network orchestration.* In *2015 IEEE International Conference on Communications (ICC)*, pages 6829–6835, London, United Kingdom, jun 2015. IEEE, ISBN 978-1-4673-6432-4. http://ieeexplore.ieee.org/document/7249414/. (Cited on pages 77, 89, 124, 178 and 214.)

[136] Mwangama, Joyce, Neco Ventura, Alexander Willner, Yahya Al-Hazmi, Giuseppe Carella, and Thomas Magedanz: *Towards Mobile Federated Network Operators.* In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6, London, United Kingdom, apr 2015. IEEE, ISBN 978-1-4799-7899-1. http://ieeexplore.ieee.org/document/7116187/. (Cited on page 77.)

[137] Hassan, Ahmed Mohamed Medhat, Joyce Mwangama, Giuseppe Carella, and Neco Ventura: *Multi-tenancy for Virtualized Network Functions.* In *IEEE NetSoft WS 2015: MISSION 2015 (MISSION 2015)*, London, United Kingdom, apr 2015. (Cited on page 77.)

[138] Yamada, Junnosuke, Niklas Blum, Giuseppe Carella, Naoyoshi Kanamaru, Naoki Uchida, and Thomas Magedanz: *A Platform for Converged, Feature-based Real-time Communications.* In *2015 18th International Conference on Intelligence in Next Generation Networks (ICIN 2015)*, pages 200–207, Paris, France, feb 2015. (Cited on pages 77 and 182.)

[139] Carella, Giuseppe A, Michael Pauls, Thomas Magedanz, Marco Cilloni, Paolo Bellavista, and Luca Foschini: *Prototyping nfv-based multi-access edge computing in 5G ready networks with open baton.* In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–4. IEEE, jul 2017, ISBN 978-1-5090-6008-5. http://ieeexplore.ieee.org/document/8004237/. (Cited on pages 78, 152, 172 and 214.)

[140] Medhat, Ahmed M, Giuseppe A. Carella, Michael Pauls, and Thomas Magedanz: *Orchestrating scalable service function chains in a NFV environment.* In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–5. IEEE, jul 2017, ISBN 978-1-5090-6008-5. http://ieeexplore.ieee.org/document/8004207/. (Cited on pages 78, 125 and 172.)

[141] Mukudu, Nyasha and Steinke, Ronald and Carella, Giuseppe and Mwangama, Joyce and Corici, Andreea and Ventura, Neco and Willner, Alexander and Magedanz, Thomas and Barros, Maria and Gavras, Anastasius: *TRESCIMO: Towards Software-Based Federated Internet of Things*

Testbeds across Europe and South Africa to Enable FIRE Smart City Experimentation. In Serrano, Martin, Nikolaos Isaris, Hans Schaffers, John Dominigue, Michael Boniface, and Thanasis Korakis (editors): *Building the Future Internet through FIRE*, chapter 30, pages 1–794. River Publishers, jun 2017, ISBN 978-87-93519-12-1. http://riverpublishers.com/dissertations{_}xml/9788793519114/9788793519114.xml. (Cited on page 78.)

[142] Pauls, Michael, Giuseppe Carella, Ahmed M Medhat, Lars Grebe, and Thomas Magedanz: *A Network Function Virtualization framework for Network Slicing of 5G Networks.* In *22 VDE-ITG-Fachtagung Mobilkommunikation (MKT'17)*, Osnabrueck, Germany, may 2017. (Cited on pages 78, 124, 178 and 214.)

[143] Bellavista, Paolo, Luca Foschini, Riccardo Venanzi, and Giuseppe Carella: *Extensible Orchestration of Elastic IP Multimedia Subsystem as a Service Using Open Baton.* In *2017 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 88–95. IEEE, apr 2017, ISBN 978-1-5090-6325-3. http://ieeexplore.ieee.org/document/7944877/. (Cited on pages 78 and 169.)

[144] Garcia, Boni, Micael Gallego, Luis Lopez, Giuseppe Antonio Carella, and Alice Cheambe: *NUBOMEDIA: An Elastic PaaS Enabling the Convergence of Real-Time and Big Data Multimedia.* In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 45–56. IEEE, nov 2016, ISBN 978-1-5090-5263-9. http://ieeexplore.ieee.org/document/7796153/. (Cited on pages 78, 170 and 182.)

[145] Carella, Giuseppe Antonio, Michael Pauls, Lars Grebe, and Thomas Magedanz: *An extensible Autoscaling Engine (AE) for Software-based Network Functions.* In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 219–225. IEEE, nov 2016, ISBN 978-1-5090-0933-6. http://ieeexplore.ieee.org/document/7919501/. (Cited on pages 78, 120, 121, 170, 182 and 214.)

[146] Wantamanee, Apichart, Sorawee Watarakitpaisarn, Giuseppe Carella, Chaodit Aswakul, and Thomas Magedanz: *Virtualising machine to machine (M2M) application using open Baton as NFV-compliant framework for building energy management system.* In *2016 11th International Conference on Computer Science & Education (ICCSE)*, pages 199–204. IEEE, aug 2016, ISBN 978-1-5090-2218-2. http://ieeexplore.ieee.org/document/7581580/. (Cited on page 78.)

[147] Medhat, Ahmed M, Quang Thanh Tran, Giuseppe Carella, Stefan Covaci, and Thomas Magedanz: *Orchestrating Service Function Chaining in Cloud Environments.* In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE) (IEEE ICCE 2016)*, Ha Long Bay, Vietnam, jul 2016. (Cited on pages 78 and 125.)

[148] Tran, Quang Thanh, Ahmed M Medhat, Asma Elmangoush, Giuseppe Carella, Alexander Willner, Stefan Covaci, and Thomas Magedanz: *Enabling Future Internet Testbeds with Open Source Software.* In *European Conference on Networks and Communications 2016: Posters (EuCNC2016-Posters)*, Athens, Greece, jun 2016. (Cited on page 78.)

[149] Cheambe, Alice, Maiorano Pasquale, Flavio Murgia, Boni Garcia, Micael Gallego Carrillo, Giuseppe Carella, Lorenzo Tomasini, Alin Calinciuc, and Cristian Spoiala: *Design and Implementation of a High Performant PaaS Platform for Creating Novel Real-Time Communication Paradigms.* In *19th conference on Innovations in Clouds, Internet and Networks (ICIN 2016)*, pages 157–163, Paris, France, feb 2016. (Cited on pages 78 and 170.)

[150] Magedanz, Thomas and Florian Schreiner: *QoS-aware multi-cloud brokering for NGN services: Tangible benefits of elastic resource allocation mechanisms.* In *2014 IEEE Fifth International Conference on Communications and Electronics (ICCE)*, pages 168–173. IEEE, jul 2014, ISBN 978-1-4799-5051-5. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6916698. (Cited on page 80.)

[151] Iqbal, Waheed, Matthew Dailey, and David Carrera: *SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud*, pages 243–253. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ISBN 978-3-642-10665-1. http://dx.doi.org/10.1007/978-3-642-10665-1{_}22. (Cited on page 80.)

[152] Galante, Guilherme and Luis Carlos E de Bona: *A survey on cloud computing elasticity.* In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 263–270. IEEE, 2012. (Cited on page 80.)

[153] Giri, Mr Manish, Sachin Waghmare, Balaji Bandhu, Akshay Sawwashere, and Atul Khaire: *Migration of Mobicents SIP Servlets on Cloud Platform.* (Cited on page 80.)

[154] Cardellini, V, M Colajanni, and P S Yu: *Dynamic load balancing on Web-server systems.* IEEE Internet Computing, 3(3):28–39, may 1999, ISSN 1089-7801. (Cited on page 84.)

[155] *ETSI - ETSI Network Functions Virtualisation completes first phase of work*, 2015. http://www.etsi.org/news-events/news/864-2015-01. (Cited on page 97.)

[156] *PoC Details - NFVwiki.* https://nfvwiki.etsi.org/index.php?title=PoC{_}Details. (Cited on page 99.)

[157] *Microservices Architecture and Design Principles | Ness Digital Engineering.* https://www.ness.com/

microservices-architecture-and-design-principles-2/. (Cited on page 100.)

[158] Sefraoui, Omar, Mohammed Aissaoui, and Mohsine Eleuldj: *OpenStack: Toward an Open-Source Solution for Cloud Computing*. International Journal of Computer Applications, 55(03):975–8887, 2012. (Cited on page 102.)

[159] KAVANAGH, ALAN: *OpenStack as the API framework for NFV: the benefits, and the extensions needed*. Ericsson Review, 2, 2015. (Cited on page 102.)

[160] Grossman, Daniel B: *New Terminology and Clarifications for Diffserv*. RFC 3260, apr 2002. https://rfc-editor.org/rfc/rfc3260.txt. (Cited on page 125.)

[161] Quinn, Paul and Uri Elzur: *Network Service Header*. Internet-draft draft-ietf-sfc-nsh-25, Internet Engineering Task Force, feb 2017. https://datatracker.ietf.org/doc/html/draft-ietf-sfc-nsh-25. (Cited on page 125.)

[162] R. Fernando et. al.: *Service Chaining using Virtual Networks with BGP VPNs*. Technical report, BGP Enabled Services (bess), 2015. https://tools.ietf.org/pdf/draft-fm-bess-service-chaining-02.pdf. (Cited on page 125.)

[163] *Fraunhofer FOKUS | OpenBaton: the new open-source platform for virtualization of network functions*. https://www.fokus.fraunhofer.de/en/fokus/news/openbaton{_}2015{_}10. (Cited on pages 144 and 213.)

[164] Karagiannis, Georgios, Almerima Jamakovic, Andy Edmonds, Carlos Parada, Thijs Metsch, Dominique Pichon, Marius Corici, Simone Ruffino, Andre Gomes, Paolo Secondo Crosta, and Thomas Michael Bohnert: *Mobile Cloud Networking: Virtualisation of cellular networks*. In *2014 21st International Conference on Telecommunications (ICT)*, pages 410–415. IEEE, may 2014, ISBN 978-1-4799-5141-3. http://ieeexplore.ieee.org/document/6845149/. (Cited on page 169.)

[165] Vingarzan, Dragos, Peter Weik, and Thomas Magedanz: *Development of an open source IMS core for emerging IMS testbeds, the academia and beyond*. Journal of Mobile Multimedia, 3(2):131–149, 2007, ISSN 1550-4646. https://dl.acm.org/citation.cfm?id=2010540.2010544. (Cited on pages 173 and 190.)

[166] Corici, Marius, Ilie Gheorghe-Pop, Eleonora Cau, Andreea Ancuta Corici, and Thomas Magedanz: *A benchmarking methodology for virtualized packet core implementations*. In *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 1–6. IEEE, oct 2016, ISBN 978-1-5090-3862-6. http://ieeexplore.ieee.org/document/7785156/. (Cited on page 180.)

[167] NUBOMEDIA: *D3.3 Cloud Platform v3*. Technical report, 2017. https://www.nubomedia.eu/sites/default/deliverables/WP3/D3.

3{_}Cloud{_}Platform{_}R9{_}V1.1-04-05-2017{_}FINAL-PC{_}rev1.pdf. (Cited on page 182.)

[168] MCN: *D5.5 Additions to Mobile Platform, IMSaaS and DSN*. Technical report, 2017. (Cited on page 192.)

[169] MCN: *D6.5 Final Report on Testbeds, Experimentation, and Evaluation*. Technical report, 2016. (Cited on page 192.)

[170] Csaszar, Andras, Wolfgang John, Mario Kind, Catalin Meirosu, Gergely Pongracz, Dimitri Staessens, Attila Takacs, and Fritz Joachim Westphal: *Unifying Cloud and Carrier Network: EU FP7 Project UNIFY*. In *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, pages 452–457. IEEE, dec 2013, ISBN 978-0-7695-5152-4. http://ieeexplore.ieee.org/document/6809448/. (Cited on page 204.)

[171] *Unifying cloud and carrier networks | Ericsson Research Blog*. Technical report, 2015. https://www.ericsson.com/research-blog/unifying-cloud-and-carrier-networks/. (Cited on page 204.)

[172] Kourtis, Michail Alexandros, Michael J. McGrath, Georgios Gardikis, Georgios Xilouris, Vincenzo Riccobene, Panagiotis Papadimitriou, Eleni Trouva, Francesco Liberati, Marco Trubian, Josep Batalle, Harilaos Koumaras, David Dietrich, Aurora Ramos, Jordi Ferrer Riera, Jose Bonnet, Antonio Pietrabissa, Alberto Ceselli, and Alessandro Petrini: *T-NOVA: An Open-Source MANO Stack for NFV Infrastructures*. IEEE Transactions on Network and Service Management, 14(3):586–602, sep 2017, ISSN 1932-4537. http://ieeexplore.ieee.org/document/7997799/. (Cited on page 204.)

[173] Dräxler, Sevil, Manuel Peuster, Holger Karl, Michael Bredel, Johannes Lessmann, Thomas Soenen, Wouter Tavernier, Sharon Mendel-Brin, and George Xilouris: *SONATA: Service Programming and Orchestration for Virtualized Software Networks*. may 2016. http://arxiv.org/abs/1605.05850. (Cited on page 204.)

[174] Karl, Holger, Sevil Dräxler, Manuel Peuster, Alex Galis, Michael Bredel, Aurora Ramos, Josep Martrat, Muhammad Shuaib Siddiqui, Steven van Rossem, Wouter Tavernier, and George Xilouris: *DevOps for network function virtualisation: an architectural approach*. Transactions on Emerging Telecommunications Technologies, 27(9):1206–1215, sep 2016, ISSN 21613915. http://doi.wiley.com/10.1002/ett.3084. (Cited on page 204.)

[175] Bernini, G, E Kraja, G Carrozzo, G Landi, and N Ciulli: *SELFNET Virtual Network Functions Manager: A Common Approach for Lifecycle Management of NFV Applications (Short Paper)*. In *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, pages 150–153, oct 2016. (Cited on pages 205 and 214.)

[176] Martrat, J, S Castro, M Bredel, and R Vilalta: *5G Development and Validation Platform for global Industry-specific Network Services and Apps*. jun 2017. http://www.cttc.es/publication/5g-development-and-validation-platform. (Cited on page 205.)

[177] *An ETSI OSM Community White Paper*. 2017. https://osm.etsi.org/images/OSM-Whitepaper-TechContent-ReleaseTHREE-FINAL.PDF. (Cited on pages xv and 206.)

[178] *Tacker - OpenStack*. https://wiki.openstack.org/wiki/Tacker. (Cited on pages xv and 207.)

[179] *Open Network Automation Platform (ONAP) Architecture White Paper*. https://www.onap.org/wp-content/uploads/sites/20/2017/11/ONAP{_}CaseSolution{_}Architecture{_}FNL.pdf. (Cited on page 207.)

[180] AT&T Inc.: *ECOMP (Enhanced Control, Orchestration, Management & Policy) Architecture White Paper*. http://about.att.com/content/dam/snrdocs/ecomp.pdf. (Cited on page 207.)

[181] *Architecture - Developer Wiki - Confluence*. https://wiki.onap.org/display/DW/Architecture. (Cited on pages xv and 208.)

[182] *SDxCentral Survey: 26% of Users Will Not Consider Open Source MAN*. https://www.sdxcentral.com/articles/news/sdxcentral-survey-26-users-will-not-consider-open-source-mano/2017/05/. (Cited on page 209.)

[183] Bjorklund, Martin: *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020, oct 2010. https://rfc-editor.org/rfc/rfc6020.txt. (Cited on page 211.)

[184] Tsietsi, M, T Chindeka, and A Terzoli: *An IMS subscriber location function for OpenBaton - A standards based MANO environment*. In *2017 IEEE AFRICON*, pages 894–899, sep 2017. (Cited on page 214.)

[185] Mechtri, M, C Ghribi, O Soualah, and D Zeghlache: *NFV Orchestration Framework Addressing SFC Challenges*. IEEE Communications Magazine, 55(6):16–23, 2017, ISSN 0163-6804. (Cited on page 214.)

[186] Hoyos, L C and C E Rothenberg: *NOn: Network function virtualization ontology towards semantic service implementation*. In *2016 8th IEEE Latin-American Conference on Communications (LATINCOM)*, pages 1–6, nov 2016. (Cited on page 214.)

[187] Francescon, A, G Baggio, R Fedrizzi, E Orsini, and R Riggio: *X-MANO: An open-source platform for cross-domain management and orchestration*. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6, jul 2017. (Cited on page 214.)

[188] Lake, D, G Foster, S Vural, Y Rahulan, B H Oh, N Wang, and R Tafazolli: *Virtualising and orchestrating a 5G evolved packet core network*. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–5, jul 2017. (Cited on page 214.)

[189] González, Sergio, Antonio de la Oliva, Xavier Costa-Pérez, Andrea Di Giglio, Fabio Cavaliere, Thomas Deiß, Xi Li, and Alain Mourad: *5G-Crosshaul: An SDN/NFV control and data plane architecture for the 5G integrated Fronthaul/Backhaul*. Transactions on Emerging Telecommunications Technologies, 27(9):1196–1205, 2016, ISSN 2161-3915. http://dx.doi.org/10.1002/ett.3066. (Cited on page 214.)

[190] *MEC Enablement by Means of an Open Source ETSI MANO Orchestrator*. https://sdn.ieee.org/newsletter/march-2016/mec-enablement-by-means-of-an-open-source-etsi-mano-orchestrator. (Cited on pages 216 and 224.)

[191] *Open Baton: A Framework for Virtual Network Function Management and Orchestration for Emerging Software-Based 5G Networks*. https://sdn.ieee.org/newsletter/july-2016/open-baton. (Cited on page 216.)

[192] Corici, Marius, Fabricio Gouveia, Thomas Magedanz, and Dragos Vingarzan: *OpenEPC: A Technical Infrastructure for Early Prototyping of NGMN Testbeds*. pages 166–175. Springer, Berlin, Heidelberg, 2011. http://link.springer.com/10.1007/978-3-642-17851-1{_}13. (Cited on page 218.)

[193] *Berlin Becomes Hotbed for 5G, SDN, NFV, and MEC Events*. https://www.sdxcentral.com/articles/news/berlin-plays-host-5g-sdn-nfv-mec-events-next-week/2017/11/. (Cited on page 219.)

[194] *The Open Baton MANO Group Pre-Dates Both OSM and Open-O*. https://www.sdxcentral.com/articles/news/open-baton-mano-group-pre-dates-osm-open-o/2017/01/. (Cited on page 219.)

[195] ETSI: *ETSI Plugtests Report V1.0.0 (2017-03); 1st ETSI NFV Plugtests*. Technical report, 2017. https://portal.etsi.org/Portals/0/TBpages/CTI/Docs/1st{_}ETSI{_}NFV{_}Plugtests{_}Report{_}v1.0.0.pdf. (Cited on page 220.)

[196] *Top Trends in the Gartner Hype Cycle for Emerging Technologies, 2017 - Smarter With Gartner*. https://www.gartner.com/smarterwithgartner/top-trends-in-the-gartner-hype-cycle-for-emerging-technologies-2017/. (Cited on page 223.)

[197] ETSI: *ETSI GS MEC-IEG 004 V1.1.1 (2015-11);Mobile-Edge Computing (MEC); Service Scenarios*. Technical report, 2015. http:

//www.etsi.org/deliver/etsi{_}gs/MEC-IEG/001{_}099/004/01.01.01{_}60/gs{_}MEC-IEG004v010101p.pdf. (Cited on page 223.)

[198] ETSI: *ETSI GS MEC 002 V1.1.1 (2016-03); Mobile-Edge Computing (MEC); Technical Requirements*. Technical report, 2016. http://www.etsi.org/deliver/etsi{_}gs/MEC/001{_}099/002/01.01.01{_}60/gs{_}MEC002v010101p.pdf. (Cited on page 223.)

List of Acronyms

3G	3rd Generation Mobile Telecommunications (UMTS, CDMA2000)
4G	4th Generation Mobile Telecommunications (LTE, WiMAX)
5G	5th Generation Mobile Telecommunications
5G-PPP	5G Infrastructure Public Private Partnership
5GMF	The 5th Generation Mobile Communication Promotion Forum
3GPP	3rd Generation Partnership Project
ACL	Access Control List
AES	Autoscaling Engine System
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AS	Application Server
AV	Architekturen der Vermittlungsknoten
AWS	Amazon Web Services
BSS	Business Support System
CAPEX	Capital Expenditures
CDMA	Code Division Mutliple Access
CDMA2000	CDMA 2000 - first 3G technology deployed
CI	Continuous Integration
CI/CD	Continuous Integration / Continuous Development
CLI	Command Line Interface
CM	Cloud Manager
CMA	Connectivity Manager Agent
CMS	Cloud Management System
CN	Compute Node
CORBA	Common Object Request Broker Architecture
COTS	Commercial off-the-shelf
CPE	Customer Premises Equipment
CPU	Central Processing Unit
CP	Connection Point
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
CSAR	Cloud Service Archive
CSCF	Call Session Control Function
DAG	Directed Acyclic Graph
DBMS	Database Management System
DevOps	Development and Operations
DNS	Domain Name System
DPI	Deep Packet Inspection
DRA	Diameter Routing Agent
DSL	Domain Specific Language

EE	Elasticity Engine
ECOMP	Enhanced Control, Orchestration, Management & Policy
EDA	Event-Driven Architecture
EM	Element Manager
EMM	Elastic Media Manager
eMBB	enhancecd Mobile BroadBand
EMS	Element Management System
EPC	Evolved Packet Core
ETSI	European Telecommunications Standards Institute
FCAPS	Fault-management, Configuration, Accounting, Performance, and Security
FIT	Future Internet Technologies
FIRE	Future Internet Research Experimentation
FMS	Fault Management System
FOKUS	Fraunhofer Institute for Open Communication Systems
FP7	7th Framework Programme
FUSECO	FUture SEamless COmmunication
FW	Firewall
GB	Giga Bytes (1,073,741,824 bytes)
GNU	GNU's Not Unix!
GPL	GNU General Public License
GS	Group Specification
GUI	Graphical User Interface
H2020	Horizon 2020
HHI	Heinrich-Hertz-Institut
HSS	Home Subscriber Server
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IaaS	Infrastructure as a Service
ICT	Information and Communication Technology
ICC	International Conference on Communications
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IFA	Interfaces and Architecture
IMS	IP-Multimedia Subsystem
IMSaaS	IMS as a Service
IoT	Internet of Things
IP	Internet Protocol
ISG	Industry Specification Group
ISO	International Organization for Standardization
IT	Information Technology
ITU	International Telecommunication Union

ITU-T ITU - Telecommunication Standardization Sector
J2EE Java 2 Enterprise Edition
JPA Java Persistence APIs
JSON JavaScript Object Notation
JVM Java Virtual Machine
Kbps Kilo Bits per Second (1,000 bits / second)
KPI Key Performance Indicator
LAN Local Area Network
LTE Long Term Evolution
LXC Linux Containers
M2M Machine-to-Machine
MA Monitoring Aggregator
MANO Management and Orchestration
MANO4X Management and Orchestration for Everything
MB Mega Bytes (1,048,576 bytes)
Mbps Mega Bits per Second (1,000,000 bits / second)
MCN Mobile Cloud Networking
MEC Mobile Edge Computing
MGW Media Gateway
MIB Management Information Base
MME Mobility Management Entity
MIT Massachusetts Institute of Technology
MM Mobility Manager
MPL Mozilla Public License
MPLS Multiprotocol Label Switching
MSB Micro Service Bus
MSc Master of Science
mMTC massive MTC
MTC Machine Type Communication
MWC Mobile World Congress
MVC Model View Control
NASA National Aeronautics and Space Administration
NAT Network Address Translation
NF Network Function
NFFG Network Function Forwarding Graph
NFP Network Forwarding Path
NFV Network Function Virtualization
NFVI NFV Infrastructure
NFVIP NFVI Provider
NFVO NFV Orchestrator
NFVRG Network Function Virtualization Research Group
NGMN Next Generation Mobile Network
NGN Next Generation Network
NGNCourse Next Generation Network Technologies Services

NGMN Next Generation Mobile Network
NGOSS New Generation OSS
NOMS Network Operations and Management Symposium
NIC Network Interface Controller
NIST National Institute of Standards and Technology
NMS Network Management System
NS Network Service
NSH Network Service Header
NSD Network Service Descriptor
NSE Network Slicing Engine
NSO Network Service Orchestrator
NSR Network Service Record
NTT Nippon Telegraph and Telephone
OASIS Organization for the Advancement of Structured Information Standards
OCCI Open Cloud Computing Interface
ODL OpenDayLight
OEC Open Edge Computing
OGF Open Grid Forum
OMA Open Mobile Alliance
ONAP Open Network Automation Platform
ONF Open Networking Foundation
ONOS Open Network Operating System
OOP Object Oriented Programming
OpenEPC Open Evolved Packet Core
OpenIMSCore Open Source IMS Core
OpenSDNCore Open Software Defined Network Core
OPEX Operating Expenditure
OPNFV Open Platform for NFV
OS Operating System
OSM Open Source MANO
OSS Operations Support System
OTT Over-The-Top
OvS OpenVSwitch
PaaS Platform as a Service
PC Personal Computer
P-CSCF Proxy CSCF
PDN Packet Data Network
PGW Packet Data Network Gateway
PM Performance Management
PNF Physical Network Function
PoC Proof of Concept
PoP Point of Presence
QoE Quality of Experience

QoS	Quality of Service
RAN	Radio Access Network
RAM	Random Access Memory
REST	Representational State Transfer
RCA	Root Cause Analysis
R&D	Research and Development
RAM	Random Access Memory
RE	Rules Engine
RMON1	Remote Network MONitoring
RO	Resource Orchestrator
ROI	Return on Investment
RPC	Remote Procedure Call
RTC	Real-time Communication
SA	Service Adapter
SaaS	Software as a Service
S-CSCF	Serving CSCF
SDK	Sofware Development Kit
SDN	Software-Defined Networking
SDP	Service Delivery Platform
SDP	Session Description Protocol
SDO	Standards Developing Organization
SF	Service Function
SFC	Service Function Chain
SFCO	Service Function Chain Orchestrator
SFP	Service Function Path
SGW	Serving Gateway
SIP	Session Initiation Protocol
SLA	Service Level Agreement
SLF	Subscriber Locator Function
SLM	Service Lifecycle Management
SME	Small and Medium Enterprises
SMS	Short Messaging Service
SNMP	Simple Network Management Protocol
SO	Service Orchestrator
SOA	Service-Oriented Architecture
SP	Service Provider
SSH	Secure Shell
SuT	System under Test
TCP	Transmission Control Protocol
TMA	Telecommunications Managed Areas
TMF	TeleManagement Forum
TMN	Telecommunications Management Network
TOSCA	Topology and Orchestration Specification for Cloud Applications
TP	Total Power

TPC	Technical Program Committee
TSP	Telecommunication Service Provider
TUB	Technische Universität Berlin
UDP	User Datagram Protocol
UE	User Equipment/Endpoint
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
UNIFY	Unifying Cloud and Carrier Networks
UO	Unit Orchestrator
URL	Uniform Resource Locator
USA	United States of America
uRLLC	Ultra Reliable and Low Latency Communications
UUID	Universally Unique Identifier
VCA	VNF Configuration and Abstraction
vCPE	Virtualized CPE
vCPU	Virtual CPU
VDU	Virtual Deployment Unit
VIM	Virtualized Infrastructure Manager
vIMS	Virtualized IMS
vEPC	Virtualized EPC
vM2M	Virtualized M2M
VL	Virtual Link
VLD	Virtual Link Descriptor
VM	Virtual Machine
VMM	Virtual Machine Manager
VNF	Virtual Network Function
VNFC	Virtual Network Function Component
VNFD	Virtual Network Function Descriptor
VNFFG	Virtual NFFG
VNFFGD	VNFFG Descriptor
VNFM	VNF Manager
VNFP	VNF Package
VNFP	VNF Function Provider
VNFR	VNF Record
VPN	Virtual Private Network
VoIP	Voice over IP
ZOOM	Zero-touch Orchestration, Operations and Management
WAN	Wide Area Network
WiMAX	Worldwide Interoperability for Microwave Access
WG	Working Group
XML	Extensible Markup Language
YAML	Yet Another Markup Language

Appendix A

Author's Publications, and Presentations to International Events

The present appendix provides additional support for Section 8.1.1 – Summary of Academic Thesis Contributions listing all of the author's academic contributions around topics addressed by this dissertation, with comprehensive bibliographical referencing.

Although some of the listed elements have been directly referenced in the bibliography section starting on page 225, here it is presented the list in a consistent style and reverse chronological order.

A.1 Author's Publications

[1] Carella, G. A., M. Pauls, T. Magedanz, M. Cilloni, P. Bellavista, and L. Foschini: *Prototyping NFV-based Multi-Access Edge Computing in 5G ready Networks with Open Baton.* In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–4, July 2017. http://ieeexplore.ieee.org/document/8004237/.

[2] Medhat, A. M., G. A. Carella, M. Pauls, and T. Magedanz: *Orchestrating Scalable Service Function Chains in a NFV Environment.* In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–5, July 2017. http://ieeexplore.ieee.org/document/8004207/.

[3] Mukudu, Nyasha and Steinke, Ronald and Carella, Giuseppe and Mwangama, Joyce and Corici, Andreea and Ventura, Neco and Willner, Alexander and Magedanz, Thomas and Barros, Maria and Gavras, Anastasius: *TRESCIMO: Towards Software-Based Federated Internet of Things Testbeds across Europe and South Africa to Enable FIRE Smart City Experimentation.* In Serrano, Martin, Nikolaos Isaris, Hans Schaffers, John Dominigue, Michael Boniface, and Thanasis Korakis (editors): *Building the Future Internet through FIRE*, chapter 30, pages 1–794. River Publishers, June 2017, ISBN 978-87-93519-12-1. http://riverpublishers.com/dissertations{_}xml/9788793519114/9788793519114.xml.

[4] Pauls, Michael, Giuseppe Carella, Ahmed M. Medhat, Lars Grebe, and Thomas Magedanz: *A Network Function Virtualization framework for Network Slicing of 5G Networks.* In *22 VDE-ITG-Fachtagung Mobilkommunikation (MKT'17)*, Osnabrueck, Germany, May 2017.

[5] Bellavista, P., L. Foschini, R. Venanzi, and G. Carella: *Extensible Orchestration of Elastic IP Multimedia Subsystem as a Service Using Open Baton.* In *2017 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 88–95, April 2017. http://ieeexplore.ieee.org/document/7944877/.

[6] Medhat, A. M., T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci, and T. Magedanz: *Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges.* IEEE Communications Magazine, 55(2):216–223, February 2017, ISSN 0163-6804. http://ieeexplore.ieee.org/document/7593430/.

[7] Garcia, Boni, Micael Gallego, Luis Lopez, Giuseppe Antonio Carella, and Alice Cheambe: *NUBOMEDIA: An Elastic PaaS Enabling the Convergence of Real-Time and Big Data Multimedia.* In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 45–56. IEEE, November 2016, ISBN 978-1-5090-5263-9. http://ieeexplore.ieee.org/document/7796153/.

[8] Carella, G. A., M. Pauls, L. Grebe, and T. Magedanz: *An extensible Autoscaling Engine (AE) for Software-based Network Functions.* In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 219–225, November 2016. http://ieeexplore.ieee.org/document/7919501/.

[9] Sousa, Bruno, Luis Cordeiro, Paulo Simoes, Andy Edmonds, Santiago Ruiz, Giuseppe A. Carella, Marius Corici, Navid Nikaein, Andre S. Gomes, Eryk Schiller, Torsten Braun, and Thomas Michael Bohnert: *Toward a Fully Cloudified Mobile Network Infrastructure.* IEEE Transactions on Network and Service Management, 13(3):547–563, September 2016, ISSN 1932-4537. http://ieeexplore.ieee.org/document/7534870/.

[10] Wantamanee, A., S. Watarakitpaisarn, G. Carella, C. Aswakul, and T. Magedanz: *Virtualising Machine to Machine (M2M) Application using Open Baton as NFV-compliant Framework for Building Energy Management System.* In *2016 11th International Conference on Computer Science Education (ICCSE)*, pages 199–204, August 2016.

[11] Medhat, Ahmed M, Quang Thanh Tran, Giuseppe Carella, Stefan Covaci, and Thomas Magedanz: *Orchestrating Service Function Chaining in Cloud Environments.* In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE) (IEEE ICCE 2016)*, Ha Long Bay, Vietnam, July 2016.

[12] Edmonds, Andy, Giuseppe Carella, Faqir Zarrar Yousaf, Carlos Goncalves, Thomas Michael Bohnert, Thijs Metsch, Paolo Bellavista, and Luca Foschini: *An OCCI-compliant Framework for Fine-grained Resource-aware Management in Mobile Cloud Networking.* In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 1306–1313. IEEE, June 2016, ISBN 978-1-5090-0679-3. http://ieeexplore.ieee.org/document/7543918/.

[13] Tran, Quang Thanh, Ahmed M Medhat, Asma Elmangoush, Giuseppe Carella, Alexander Willner, Stefan Covaci, and Thomas Magedanz: *Enabling Future Internet Testbeds with Open Source Software.* In *European Conference on Networks and Communications 2016: Posters (EuCNC2016-Posters)*, Athens, Greece, June 2016.

[14] Cau, Eleonora, Marius Corici, Paolo Bellavista, Luca Foschini, Giuseppe Carella, Andy Edmonds, and Thomas Michael Bohnert: *Efficient Exploitation of Mobile Edge Computing for Virtualized 5G in EPC Architectures.* In *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 100–109. IEEE, March 2016, ISBN 978-1-5090-1754-6. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7474417.

[15] Cheambe, Alice, Maiorano Pasquale, Flavio Murgia, Boni Garcia, Micael Gallego Carrillo, Giuseppe Carella, Lorenzo Tomasini, Alin Calinciuc, and Cristian Spoiala: *Design and Implementation of a High Performant PaaS Platform for Creating Novel Real-Time Communication Paradigms.* In *19th conference on Innovations in Clouds, Internet and Networks (ICIN 2016)*, pages 157–163, Paris, France, February 2016. http://dl.ifip.org/db/conf/icin/icin2016/1570230514.pdf.

[16] Carella, Giuseppe, Luca Foschini, Alessandro Pernafini, Paolo Bellavista, Antonio Corradi, Marius Corici, Florian Schreiner, and Thomas Magedanz: *Quality Audit and Resource Brokering for Network Functions Virtualization (NFV) Orchestration in Hybrid Clouds.* In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, December 2015, ISBN 978-1-4799-5952-5. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7417385.

[17] Medhat, Ahmed M., Giuseppe Carella, Christian Luck, Marius Iulian Corici, and Thomas Magedanz: *Near Optimal Service Function Path Instantiation in a Multi-Datacenter Environment.* In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 336–341. IEEE, November 2015, ISBN 978-3-9018-8277-7. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7367379.

[18] Carella, Giuseppe, Junnosuke Yamada, Niklas Blum, Christian Luck, Naoyoshi Kanamaru, Naoki Uchida, and Thomas Magedanz: *Cross-layer Service to Network Orchestration.* In *2015 IEEE International Conference on Communications (ICC)*, pages 6829–6835, London, United Kingdom, June 2015. IEEE, ISBN 978-1-4673-6432-4. http://ieeexplore.ieee.org/document/7249414/.

[19] Corici, Andreea Ancuta, Ranjan Shrestha, Giuseppe Carella, Asma Elmangoush, Ronald Steinke, and Thomas Magedanz: *A Solution for Provisioning Reliable M2M Infrastructures using SDN and Device Management.* In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, pages 81–86. IEEE, May 2015, ISBN 978-1-4799-7752-9. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7231401.

[20] Willner, Alexander, Thomas Magedanz, Yahya Al-Hazmi, Giuseppe Carella, Joyce Mwangama, and Neco Ventura: *Towards Mobile Federated Network Operators.* In *IEEE NetSoft WS 2015: Soft5G 2015 (Soft5G 2015)*, London, United Kingdom, April 2015. http://ieeexplore.ieee.org/document/7116187/.

[21] Hassan, Ahmed Mohamed Medhat, Joyce Mwangama, Giuseppe Carella, and Neco Ventura: *Multi-tenancy for Virtualized Network Functions.* In *IEEE NetSoft WS 2015: MISSION 2015 (MISSION 2015)*, London, United Kingdom, April 2015. http://ieeexplore.ieee.org/document/7116177/.

[22] Yamada, Junnosuke, Niklas Blum, Giuseppe Carella, Naoyoshi Kanamaru, Naoki Uchida, and Thomas Magedanz: *A Platform for Converged, Feature-based Real-time Communications.* In *2015 18th International Conference on Intelligence in Next Generation Networks (ICIN 2015)*, pages 200–207, Paris, France, February 2015. http://ieeexplore.ieee.org/document/7073832/.

[23] Magedanz, Thomas, Giuseppe Carella, Marius Corici, Julius Mueller, and Andreas Weber: *Prototyping new Concepts Beyond 4G – The Fraunhofer Open5GCore.* it - Information Technology, 57(5), January 2015, ISSN 1611-2776. https://www.degruyter.com/view/j/itit.2015.57.issue-5/itit-2015-0021/itit-2015-0021.xml.

[24] Carella, Giuseppe, Marius Corici, Paolo Crosta, Paolo Comi, Thomas Michael Bohnert, Andreea Ancuta Corici, Dragos Vingarzan, and Thomas Magedanz: *Cloudified IP Multimedia Subsystem (IMS) for Network Function Virtualization (NFV)-based architectures.* In *2014 IEEE Symposium on Computers and Communications (ISCC)*, volume Workshops, pages 1–6. IEEE, June 2014, ISBN 978-1-4799-4277-0. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6912647.

[25] Kavoussanakis, Konstantinos, Alastair Hume, Josep Martrat, Carmelo Ragusa, Michael Gienger, Konrad Campowsky, Gregory Van Seghbroeck, Constantino Vazquez, Celia Velayos, Frederic Gittler, Philip Inglesant, Giuseppe Carella, Vegard Engen, Michal Giertych, Giada Landi, and David Margery: *BonFIRE: The Clouds and Services Testbed.* In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 321–326. IEEE, December 2013, ISBN 978-0-7695-5095-4. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6735444.

[26] Carella, Giuseppe, Thomas Magedanz, Konrad Campowsky, and Florian Schreiner: *Elasticity as a service for federated cloud testbeds.* In *2013 IEEE International Conference on Communications Workshops (ICC)*, pages 256–260. IEEE, June 2013, ISBN 978-1-4673-5753-1. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6649239.

[27] Bellavista, Paolo, Giuseppe Carella, Luca Foschini, Thomas Magedanz, Florian Schreiner, and Konrad Campowsky: *QoS-aware elastic cloud brokering for IMS infrastructures.* In *Proceedings - IEEE Symposium on Computers and Communications*, pages 000157–000160, 2012. http://ieeexplore.ieee.org/document/6249285/.

[28] Campowsky, Konrad, Giuseppe Carella, Thomas Magedanz, and Florian Schreiner: *Optimization of Elastic Cloud Brokerage Mechanisms for Future Telecommunication Service Environments*, 2012. ISSN 0930-5157. https://www.degruyter.com/view/j/piko.2012.35.issue-3/pik-2012-0036/pik-2012-0036.xml.

[29] Carella, Giuseppe, Thomas Magedanz, Konrad Campowsky, and Florian Schreiner: *Network-aware Cloud Brokerage for telecommunication services*. In *2012 1st IEEE International Conference on Cloud Networking, CLOUDNET 2012 - Proceedings*, pages 131–136, 2012. http://ieeexplore.ieee.org/document/6483667/.

A.2 Presentations to International Events

[1] Giuseppe Carella, Michael Pauls: *Understanding the Network Function Virtualization transformation based on practical experiences.* In *8th FOKUS International FUSECO Forum.* 9th-10th of November, Berlin, Germany, 2017. https://www.fokus.fraunhofer.de/fuseco-forum-2017.

[2] Dragos Vingarzan, Giuseppe Carella: *Meshing and Virtualizing of IMS and EPC with Kamailio.* In *5th Kamailio World Conference.* 8th-10th of May, Berlin, Germany, 2017. https://www.youtube.com/watch?v=1_rX76l0muI.

[3] Marius Corici, Giuseppe Carella: *Software Networks in 5G – an overview from the core, management, NFV, SDN, programmability and security.* In *7th FOKUS International FUSECO Forum.* 3rd-4th of November, Berlin, Germany, 2016. https://www.fokus.fraunhofer.de/fuseco-forum-2016.

[4] Carella, Giuseppe: *Software Networks in 5G – an overview from the core, management, NFV, SDN, programmability and security.* In *7th FOKUS International FUSECO Forum.* 3rd-4th , Berlin, Germany, 2016.

[5] Giuseppe Carella, Lorenzo Tomasini: *Software Networks in 5G – an overview from the core, management, NFV, SDN, programmability and security.* In *7th FOKUS International FUSECO Forum.* 3rd-4th of November, Berlin, Germany, 2016. https://www.fokus.fraunhofer.de/fuseco-forum-2016.

[6] Marius Corici, Giuseppe Carella: *Network Evolution towards the 5G Environment: an Overview of Use Cases and Technology (Radio, Core, Management, NFV and SDN).* In *6th FOKUS International FUSECO Forum.* 5th-6th of November, Berlin, Germany, 2015. https://www.fokus.fraunhofer.de/go/fuseco-forum-2015.

[7] Marius Corici, Giuseppe Carella: *5G Foundations and Core Network Evolution: Radio, Convergence Core, Cognitive Management and Virtualisation (SDN/NFV).* In *5th FOKUS International FUSECO Forum.* 13th-14th of November, Berlin, Germany, 2014. http://www.fokus.fraunhofer.de/en/fokus_events/ngni/ims_ws_06/index.html.

[8] Giuseppe Carella, Thomas Magedanz: *"[U+FFFC] OpenSDNCore: a Framework for prototyping virtualized Network Function Orchestration in emerging SDN-based 5G Infrastructures".* In *Carrier Network Virtualization 2014.* 9th-11th of December, Palo Alto, USA, 2014. https://tmt.knect365.com/nfv-and-carrier-sdn/.

[9] Giuseppe Carella, Thomas Magedanz: *"SDP and Core Network Virtualization".* In *9th SDP Global Summit – "Service Delivery Virtualization".* 17th of September, Rome, Italy, 2013. https://tmt.knect365.com/nfv-and-carrier-sdn/.

Appendix B

Relevant Information

The present appendix provides additional support for Chapter 5, and Chapter 6. In particular, it provides a complete example of a NSD and the definition of the main interfaces exposed by the NFVO.

B.1 Complete Example of a Network Service Descriptor (NSD)

This section provides the extended version of the Network Service Descriptor (NSD) presented in Figure 5.10 of Section 5.7.2.5.

Listing B.1: Extended Version of the NSD presented in Figure 5.10 of Section 5.7.2.5

```
{
   "name":"nsd",
   "vendor":"vendor",
   "version":"version",
   "vld":[
      {
         "name":"private"
      }
   ],
   "vnfd":[
      {
         "name":"vnf-a",
         "vendor":"vendor",
         "version":"version",
```

```
15          "lifecycle_event":[
16             {
17                "event":"INSTANTIATE",
18                "lifecycle_events":[
19                   "install.sh"
20                ]
21             },
22             {
23                "event":"CONFIGURE",
24                "lifecycle_events":[
25                   "configure.sh"
26                ]
27             }
28          ],
29          "vdu":[
30             {
31                "vm_image":[
32                   "image-name"
33                ],
34                "scale_in_out":5,
35                "vnfc":[
36                   {
37                      "connection_point":[
38                         {
39                            "floatingIp":"random",
40                            "virtual_link_reference":"private",
41                            "interfaceId":0
42                         }
43                      ]
44                   }
45                ],
46                "high_availability":{
47                   "resiliencyLevel":"ACTIVE_STANDBY_STATELESS"
                        ,
48                   "redundancyScheme":"1:N"
49                },
50                "monitoring_parameter":[
51                   "agent.ping"
52                ],
53                "fault_management_policy":[
54                   {
55                      "name":"Sipp␣Server␣not␣available",
56                      "criteria":[
57                         {
58                            "parameter_ref":"agent.ping",
59                            "function":"nodata(1m)",
60                            "vnfc_selector":"at_least_one",
61                            "comparison_operator":"=",
62                            "threshold":"1"
```

```
                        }
                    ],
                    "severity":"CRITICAL",
                    "period":60
                }
            ],
            "vimInstanceName":[

            ]
        }
    ],
    "configurations":{
        "configurationParameters":[
            {
                "confKey":"PARAM-1",
                "value":"0",
                "description":"Controls␣the␣length␣(in␣
                    milliseconds)␣of␣calls.␣More␣precisely,␣
                    this␣controls␣the␣duration␣of␣'pause'␣
                    instructions␣in␣the␣scenario,␣if␣they␣do␣
                    not␣have␣a␣'milliseconds'␣section.␣
                    Default␣value␣is␣0."
            }
        ],
        "name":"configuration"
    },
    "virtual_link":[
        {
            "name":"private",
            "qos":[
                "minimum_bandwith:BRONZE"
            ]
        }
    ],
    "deployment_flavour":[
        {
            "flavour_key":"m1.small"
        }
    ],
    "auto_scale_policy":[
        {
            "name":"scale-out",
            "threshold":100,
            "comparisonOperator":">=",
            "period":30,
            "cooldown":60,
            "mode":"REACTIVE",
            "type":"VOTED",
            "alarms":[
```

```
107                     {
108                         "metric":"system.cpu.util[,idle]",
109                         "statistic":"avg",
110                         "comparisonOperator":"<=",
111                         "threshold":40,
112                         "weight":1
113                     }
114                 ],
115                 "actions":[
116                     {
117                         "type":"SCALE_OUT",
118                         "value":"1"
119                     }
120                 ]
121             },
122             {
123                 "name":"scale-in",
124                 "threshold":100,
125                 "comparisonOperator":">=",
126                 "period":30,
127                 "cooldown":60,
128                 "mode":"REACTIVE",
129                 "type":"VOTED",
130                 "alarms":[
131                     {
132                         "metric":"system.cpu.util[,idle]",
133                         "statistic":"avg",
134                         "comparisonOperator":">=",
135                         "threshold":60,
136                         "weight":1
137                     }
138                 ],
139                 "actions":[
140                     {
141                         "type":"SCALE_IN",
142                         "value":"1"
143                     }
144                 ]
145             }
146         ],
147         "type":"client",
148         "endpoint":"generic",
149         "vnfPackageLocation":"https://github.com/openbaton/
                vnf-scripts.git"
150     },
151     {
152         "name":"vnf-b",
153         "vendor":"vendor",
154         "version":"version",
```

```
155         "lifecycle_event":[
156            {
157               "event":"INSTANTIATE",
158               "lifecycle_events":[
159                  "install.sh"
160               ]
161            },
162            {
163               "event":"CONFIGURE",
164               "lifecycle_events":[
165                  "configure.sh"
166               ]
167            }
168         ],
169         "vdu":[
170            {
171               "vm_image":[
172                  "image-name"
173               ],
174               "scale_in_out":5,
175               "vnfc":[
176                  {
177                     "connection_point":[
178                        {
179                           "floatingIp":"random",
180                           "virtual_link_reference":"private",
181                           "interfaceId":0
182                        }
183                     ]
184                  }
185               ],
186               "vimInstanceName":[
187
188               ]
189            }
190         ],
191         "configurations":{
192            "configurationParameters":[
193
194            ],
195            "name":"configuration"
196         },
197         "virtual_link":[
198            {
199               "name":"private",
200               "qos":[
201                  "minimum_bandwith:BRONZE"
202               ]
203            }
```

```
204           ],
205           "deployment_flavour":[
206              {
207                 "flavour_key":"m1.small"
208              }
209           ],
210           "auto_scale_policy":[
211
212           ],
213           "type":"vnf-b",
214           "endpoint":"generic",
215           "vnfPackageLocation":"https://github.com/openbaton/
                 vnf-scripts.git"
216        }
217     ],
218     "vnffgd":[
219        {
220           "symmetrical":false,
221           "dependent_virtual_link":[
222              {
223                 "name":"sfc-network"
224              }
225           ],
226           "network_forwarding_path":[
227              {
228                 "connection":{
229                    "0":"vnf-a",
230                    "1":"vnf-b"
231                 },
232                 "policy":{
233                    "acl_matching_criteria":{
234                       "source_port":0,
235                       "destination_port":5001,
236                       "protocol":17,
237                       "source_ip":"172.0.0.42/32",
238                       "destination_ip":"172.0.0.43/32"
239                    },
240                    "qos_level":"GOLD"
241                 }
242              }
243           ]
244        }
245     ],
246     "vnf_dependency":[
247        {
248           "source":{
249              "name":"vnf-a"
250           },
251           "target":{
```

```
252             "name":"vnf-b"
253          },
254          "parameters":[
255             "private"
256          ]
257       }
258    ]
259 }
```

B.2 Definition of the Main NFVO Interfaces

This section provides the definition of the interfaces exposed towards functional elements of other domains. The interface naming convention follows the one proposed by the ETSI NFV MANO specification[28].

B.2.1 NS Catalog

The following Table B.1 provides the REST API definition of the NSD catalog. Those APIs are exposed at the URL: `/api/v1/ns-descriptors` and supports standard CRUD operations over those resources.

Endpoint	Method	Description	Returned Code
/	GET	Returns the list of NSDs	200 OK
/	POST	On boards and validates an NSD	200 OK
/nsd-id	GET	Returns the NSD identified by nsd-id	200 OK
/nsd-id	PUT	Updates an NSD identified by nsd-id	202 Accepted
/nsd-id	DELETE	Removes an NSD identified by nsd-id	204 No Content
/nsd-id/ vnfdescriptors	POST	Adds a new VNFD to the NSD identified by nsd-id	200 OK
/nsd-id/ vnfdescriptors	GET	Returns the list of VNFDs part of a NSD identified by nsd-id	200 OK
/nsd-id/ vnfdescriptors/ vnfd-id	GET	Returns the VNFDs identified by vnfd-id part of a NSD identified by nsd-id	200 OK
/nsd-id/ vnfdescriptors/ vnfd-id	PUT	Updates the VNFD identified by vnfd-id contained by NSD identified by nsd-id	202 Accepted
/nsd-id/ vnfdescriptors/ vnfd-id	DELETE	Deletes a VNFDs identified by vnfd-id part of a NSD identified by nsd-id	204 No Content
/nsd-id/ vnfdependencies	POST	Adds a new VNF dependency to the NSD identified bynsd-id	204 No Content

Table B.1: REST APIs Exposed by the NFVO Interface

B.2.2 VNF Catalog

The following tables list the REST API definition of the VNF catalog. The APIs listed in Table B.2 are exposed at the URL: `/api/v1/vnf-descriptors`.

Endpoint	Method	Description	Returned Code
`/`	GET	This operation returns the list of VNFDs	200 OK
`/`	POST	This operation allows on boarding and validating a VNFD	200 OK
`/vnfd-id`	GET	This operation returns the VNFD identified by vnfd-id	200 OK
`/vnfd-id`	PUT	This operation updates a NSD identified by vnfd-id	202 Accepted
`/vnfd-id`	DELETE	This operation is used to remove a VNFD identified by vnfd-id	204 No Content

Table B.2: VNFD REST APIs Exposed by the NFVO

The APIs listed in Table B.3 are exposed at the URL: `/api/v1/vnf-packages`.

Endpoint	Method	Description	Returned Code
/	GET	This operation returns the list of VNFDs	200 OK
/	POST	This operation allows on boarding and validating a VNF Package	200 OK
/vnfp-id	GET	This operation returns the VNF Package identified by vnfp-id	200 OK
/vnfp-id	PUT	This operation updates a VNF Package identified by vnfp-id	202 Accepted
/vnfp-id	DELETE	This operation is used to remove a VNF Package identified by vnfp-id	204 No Content

Table B.3: REST APIs Exposed by the NFVO

B.2.3 Security REST APIs

The APIs listed in Table B.4 are exposed at the URL: `/api/v1/users`.

Endpoint	Method	Description	Returned Code
/	GET	This operation returns the list of registered users	200 OK
/	POST	This operation allows registering a new user	200 OK
/username	GET	This operation returns the user identified by username	200 OK
/username	PUT	This operation allows updating a user	202 Accepted
/user-id	DELETE	This operation allows to remove a user	204 No Content

Table B.4: REST APIs exposed by the NFVO

The APIs listed in Table B.5 are exposed at the URL: `/api/v1/projects`.

Endpoint	Method	Description	Returned Code
/	GET	This operation returns the list of projects	200 OK
/	POST	This operation allows creating a new project	200 OK
/project-id	GET	This operation returns the project selected by project-id	200 OK
/project-id	PUT	This operation allows updating a project	202 Accepted
/project-id	DELETE	This operation allows to remove a project	204 No Content

Table B.5: REST APIs Exposed by the NFVO

B.2.4 *Or-Oss*

The *Or-Oss* interface correspond to the one exposed by the NFVO and consumed either by the TSP via user tools, or by other functional elements of the MANO4X framework, especially the ones in the OSS and user tools domain.

The APIs listed in Table B.6 are exposed at the URL: `/api/v1/ns-records`.

Endpoint	Method	Description	Returned Code
/	GET	This operation returns the list of NSRs	200 OK
/	POST	This operation allows instantiating a NSR from the NSD passed in input	201 Created
/nsr-id	GET	This operation returns the NSR identified by nsr-id	200 OK
/nsd-id	POST	This operation allows instantiating a NSR using the NSD identified by nsd-id	201 Created
/nsr-id	PUT	This operation updates a NSR identified by nsr-id	202 Accepted
/nsr-id	DELETE	This operation is used to remove a NSD identified by nsd-id	204 No Content
/nsr-id/ vnfrecords	GET	Returns the list of VNFRs part of a NSR identified by nsr-id	200 OK
/nsr-id/ vnfrecords	POST	This operation allows adding a new VNFR to the NSR identified by nsr-id	200 OK
/nsr-id/ vnfrecords/ vnfr-id/ vdunits/ vnfcinstances	POST	This operation allows adding a new VNFC instance to the VNFR. It corresponds to a scale out operation.	200 OK
/nsr-id/ vnfrecords/ vnfr-id/ vdunits/ vnfcinstances	DELETE	This operation allows removing a new VNFC instance from the VNFR. It corresponds to a scale in operation.	200 OK
/nsr-id/ vnfrecords/ vnfr-id/ vdunits/vdu-id/ vnfcinstances	POST	This operation allows adding a new VNFC instance to the VNFR specifying the specific VDU to be used via the vdu-id. It corresponds to a scale out operation.	200 OK

/nsr-id/ vnfrecords/ vnfr-id/ vdunits/vdu-id/ vnfcinstances	DELETE	This operation allows removing a new VNFC instance from the VNFR specifying the specific VDU to be used via the vdu-id. It corresponds to a scale in operation.	200 OK
/nsr-id/ vnfrecords/ vnfr-id/ vdunits/vdu-id/ vnfcinstances/vnfc-id/ start	POST	This operation allows starting a VNFC instance part of the VNFR.	200 OK
/nsr-id/ vnfrecords/ vnfr-id/ vdunits/vdu-id/ vnfcinstances/vnfc-id/ stop	POST	This operation allows stopping a VNFC instance part of the VNFR.	200 OK
/nsr-id/ vnfrecords/ vnfr-id/ vdunits/vdu-id/ vnfcinstances/vnfc-id/ standby	POST	This operation allows adding a new VNFC instance in standby state instance to the VNFC instance.	200 OK
/nsr-id/ vnfrecords/ vnfr-id/ vdunits/vdu-id/ vnfcinstances/vnfc-id/ switchtostandby	POST	This operation allows executing the switch to standby operation removing the VNFC instance.	200 OK

Table B.6: REST APIs Exposed by the NFVO over the *Or-Oss* Reference Point

B.2.5 *Or-Vi* and *Vi-Vnfm*

The *Or-Vi* and *Vi-Vnfm* interfaces correspond to the ones exposed by the VIM driver and consumed either by the NFVO or by the VNFM. This abstraction allows the implementation of the orchestration logic to be agnostic to the selected VIM technology and be reusable with different implementations.

Table B.7 provides the definition of the *Or-Vi* interface as a list of methods exposed by the VIM driver over the message bus.

B.2.6 *Vi-Mon*

Table B.8 provides the *Vi-Mon* interface definition.

Whenever one of those operations is called, the monitoring plugin creates the performance metric in the specific monitoring system. Once the PM job is created,

Function	Description
launchInstanceAndWait	Creates an instance of a Server, waiting for it to start if necessary
listServer	Returns a list of the server under control of the VIM
deleteServerByIdAndWait	Deletes a Server with a given ID, waiting for it to shut down if necessary
createNetwork	Creates a new network
listNetworks	Returns all of the known networks
getNetworkById	Returns the network having the given ID
updateNetwork	Updates a network
deleteNetwork	Deletes a network
createSubnet	Creates a new subnet
updateSubnet	Updates a subnet
deleteSubnet	Deletes a subnet
getSubnetsExtIds	Returns the list of the external IDs of subnets
addFlavor	Adds a new flavour
listFlavors	Returns a list of available flavours
updateFlavor	Updates a flavour
deleteFlavor	Deletes a given flavour
addImage	Adds a new NFV Image
listImages	Queries the VIM for a list of the available NFV Images
updateImage	Updates an NFV Image
copyImage	Copies a NFV Image to a new one
deleteImage	Deletes an NFV Image
getQuota	Returns the resource Quota of the VIM
getType	Returns the type supported by the VIM driver

Table B.7: Reference Point Or-Vi-rpc/Vnfm-Vi-rpc

it is possible to create a certain *threshold* condition so that the monitoring system can notify the consumer whenever the threshold is crossed. Such notifications are later on delivered to the consumers in form of alarms.

B.2.7 *Or-Vnfm*

The interface *Or-Vnfm* is the interface produced by the VNFM and consumed by the NFVO. This interface allows performing lifecycle operations on VNFs in conformance with the content of the VNF Package. Thus, in most of the operations, the payload provided is the content (or a subset of its content) of the VNF Package.

Function	Description
`createPMJob`	This operation allows creating a *Performance Management* job, setting specific metrics required, and enabling the consumer to specify a resource (or a set of resources), under the responsibility of the VIM.
`deletePMJob`	This operation allows the consumer deleting one or more Performance Management (PM) job(s).
`queryPMJob`	This operation allows the consumer to retrieve the actual values of one or more PM job(s).
`createThreshold`	This operation allows the consumer to create a threshold to specifying levels on specified performance metrics associated to certain resources. Whenever the threshold is crossed a notification is generated. Creating a threshold does not trigger collection of metrics. In order for the threshold to be active, there should be a PM job collecting the needed metric for the selected entities.
`deleteThreshold`	This operation allows the consumer to delete existing thresholds.
`subscribeForFault`	This operation enables the consumer to subscribe for notifications related to a particular threshold and their state changes resulting from the virtualized resources faults. This also enables the consumer to specify the scope of the subscription in terms of the specific alarms for the virtualized resources to be reported by the VIM using a filter as the input.
`unsubscribeForFault`	This operation allows to unsubscribe to any fault notifications
`getAlarmList`	This operation enables the consumer to query for active alarms.

Table B.8: Reference Point Vi-Mon

Table B.9 provides the *or-vnfm-amqp* interface definition, as well as the ETSI NFV corresponding operation.

B.2.8 *Or-vnfm-rest*

The APIs exposed in Table B.10 allow integrating external VNFM which are not communicating over the message bus. Those APIs are typically consumed by the VNFM to communicate to the NFVO the end of a certain lifecycle operation. The APIs listed in Table B.10 are exposed by the NFVO at the URL: `/api/v1/admin`.

Function	Description	ETSI NFV Operation
instantiate	Requests the instantiation of a VNFR	*Instantiate VNF*
query	Retrieves the state of a VNF instance	*Query VNF*
scale	Scales a VNF (in/out, up/down)	*Scale VNF*
checkInstantiationFeasibility	Checks if a VNF can be instantiated	*Check VNF instantiation feasibility*
heal	Handles a failed VNF instance, to support healing capabilities	*Heal VNF*
updateSoftware	Applies a very limited software update	*Update VNF software*
modify	Instructs the VNFM to make structural changes to a VNF instance	*Modify VNF*
upgradeSoftware	Applies a new software release to a VNF instance	*Upgrade VNF software*
terminate	Manages the termination of a VNF instance	*Terminate VNF*
handleError	Handles an NFVO error, in response to a previous action	NA
start	Starts a previously created VNF instance	NA
stop	Stops a previously started VNF instance	NA
startVNFCInstance	Starts a VNFC	NA
stopVNFCInstance	Stops a VNFC	NA
configure	Configures a VNF	NA
resume	Resumes a VNF	NA
notifyChange	Provides notifications about the state changes of a VNF instance	*Notify*

Table B.9: Operations Exposed over the *Or-Vnfm* Interface

Endpoint	Method	Description	Returned Code
/vnfm-core-actions-reply	POST	This API is called whenever	200 OK
/vnfm-core-actions	POST	This operation allows	200 OK
/vnfm-core-grant	POST	This operation allows	202 Accepted
/vnfm-core-allocate	POST	This operation allows	202 Accepted
/vnfm-core-scale	POST	This operation allows	202 Accepted

Table B.10: Operations Exposed over the *Or-Vnfm-rest* Interface

B.3 The Boostrap CLI

The bootstrap CLI has been written in bash, and it allows installing three different versions of Open Baton:

- release version: binary installation of the latest stable version
- nightly version: binary installation of the latest nightly build
- develop version: source code installation of the latest sources contained in the develop branches of each individual repository

It accepts some arguments as following:

- `-openbaton-bootstrap-version`: allowing selecting a particular version of Open Baton to be installed
- `-config-file`: allowing passing the path of the config file which could be used as input instead of interactively passing them while executing the script. This mechanism is typically used for non interactive installation

The following B.2 shows the single command needed for performing the installation of the Open Baton framework on top of a standard Ubuntu 16.04 OS in a non interactive mode.

Listing B.2: Bootstrap command needed for installing Open Baton on a standard Ubuntu 16.04 OS

```
1 sh <(curl -s http://get.openbaton.org/bootstrap) release \
2         --config-file=./config.cfg
```

In order to perform a binary installation on the OSx OS a brew[1] formula has been provided for the NFVO [2] and generic VNFM[3].

Last but not least, the installation has been also automated for virtualized environments like Docker and Vagrant. For Docker, a docker compose file has been provided[4], so that each individual component can be executed as individual container. Using the docker approach each component may be installed on a different host, and can be easily scaled in/out based on the requirements of the use cases executed. While for Vagrant, a Vagrantfile[5] has been provided which automate the deployment of an Ubuntu based virtual machine and executes the non-interactive bootstrap procedure.

[1] https://brew.sh
[2] http://get.openbaton.org/homebrew/openbaton-nfvo.rb
[3] http://get.openbaton.org/homebrew/openbaton-vnfm-generic.rb
[4] https://github.com/openbaton/bootstrap/tree/master/distributions/docker
[5] http://get.openbaton.org/vagrant/Vagrantfile