D2.2 Architecture Design

Project Acronym	SONATA
Project Title	Service Programing and Orchestration for Virtualized Software Networks
Project Number	671517 (co-funded by the European Commission through Horizon 2020)
Instrument	Collaborative Innovation Action
Start Date	01/07/2015
Duration	30 months
Thematic Priority	ICT-14-2014 Advanced 5G Network Infrastructure for the Future Internet

Deliverable	D2.2 Architecture Design
Workpackage	WP2 Architecture Design
Due Date	November 30, 2015
Submission Date	December 22, 2015
Version	0.4
Status	Final
Editor	Johannes Lessmann, Michael Bredel (NEC)
Contributors	Sharon Mendel-Brin (ALU), Aurora Ramos, Jim Ahtes, Josep Martrat (ATOS), Phil Eardley, Andy Reid (BT), Steven Van Rossem, Wouter Tavernier (iMinds), Shuaib Siddiqui (i2CAT), George Xilouris (NCSRD), Xos Ramn Sousa, Santiago Rodriguez (OPT), Jos Bonnet (PTIn), Panos Trakadas, Sotiris Karachotzitis (SYN), Geoffroy Chollon, Bruno Vidalenc (TCS), Pedro A. Aranda, Diego R. Lopez (TID), Tiago Batista, Ricardo Preto, Tiago Teixeira (UBI), Dario Valocchi, Francesco Tusa, Stuart Clayman, Alex Galis (UCL), Sevil Mehraghdam, Manuel Peuster (UPB)
Reviewer(s)	Holger Karl (UPB)

Keywords:

architecture, service platform, software development kit

	Deliverable Type	
R	Document	**X**
DEM	Demonstrator, pilot, prototype	
DEC	Websites, patent filings, videos, etc.	
OTHER		
	Dissemination Level	
PU	Public	
CO	Confidential, only for members of the consortium (including the Commission Services)	**X**

Executive Summary:

Based on the use-cases and requirements outlined in deliverable D2.1, this document presents the initial overall architecture of the SONATA system. Its aim is to propose an architecture that is able to cover all use-cases and requirements and allows for a flexible implementation with reasonable technical complexity and performance. To this end, it takes state-of-the-art system architectures of NFV MANO systems and state-of-the-art architectures of high performance distributed systems into account.

The overall contributions of D2.2 can be summarized as follows:

- A detailed description of the overall architecture of the SONATA system comprising the various components and their relations. That is, the SONATA service programming and orchestration framework consists of the SONATA software development toolkit (SDK), the SONATA service platform, and different catalogues storing artefacts that can be produced, used, and managed by the SONATA system.
- A general information model describing the most important reference points.
- A detailed specification of the catalogues and repositories used by the service platform as well as the SDK to store, retrieve, and exchange information and data.
- An initial specification of the service development kit, its functions, and components, including a detailed description on how the SDK is integrated with the service platform and how it supports the development, deployment, and operation of network services is provided.
- A description of the high-level workflow and life-cycle management of networks services in the service platform.
- A description of the core components of the service platform including the gatekeeper, an infrastructure abstraction, and the MANO framework.
- The specification of DevOps operations including slicing support as well as recursive installations of the service platform itself.

The architecture deliverable is used as a reference point to kick-off and coordinate the SONATA development. Thus, the overall architecture makes sure that all the different components are integrated efficiently. However, since SONATA is following a modern agile development process, we do not consider the presented architecture as a final results, but expect the architecture to change over time based on findings throughout the project. Besides, many decisions that are directly related to implementation details are left to the development teams of work packages WP3 and WP4.

Contents

List of Figures

List of Tables

1 Introduction

In today's classical networks, service creation and management are crucial. However, service creation can often take several hours or even days, which lowers the quality of experience of customers and so the revenue of service providers. Yet data centres today can set up compute services within minutes, if not seconds. Consequently, there is much interest in network softwarization and network function virtualization, which try and leverage virtualization and cloudification to execute (virtualized) network functions as software on cloud infrastructures. Examples include load balancing, firewalling and deep packet inspection.

Network Function Virtualisation (NFV) creates new challenges with respect to service orchestration and service management. To this end, the European Telecommunications Standards Institute (ETSI) NFV group has defined a standardized Management and Orchestration (MANO) reference architecture [28] that aims at the feasibility of cloud deployments of typical network functions.

In this context, this deliverable proposes a high level overall architecture that comprises all components of the SONATA system. It takes into account the use-cases and scenarios as described in deliverable D2.1, the state-of-the-art system architectures of NFV Management and Orchestration systems, and the state-of-the-art architectures of high performance distributed systems. Our architecture includes : a service platform with a message bus kernel, gatekeeper, repositories and catalogues to store artefacts, and various plugins for various tasks; a software development kit with editors, model checkers and validators, system emulators, and packaging tools; and a service platform, which offers the actual service functionalities. The SONATA architecture is in line with the current ETSI NFV standard, but proposes enhancements wherever needed.

Deliverable D2.2 is the second specification document of the SONATA project, which presents the current outcomes of task 2.3 with regard to the design and specification of the SONATA overall system architecture.

1.1 SONATA in the Wider 5G Context

5G systems will differentiate themselves from fourth generation (4G) systems not only through further evolution in radio performance and capacity but also through greatly increased flexibility end-to-end in all segments of the 5G networks. This end-to-end flexibility will come in large part from the incorporation of softwarization into every component of the network. Well known techniques such as software defined networking, network function virtualization, and cloud computing will together allow unprecedented flexibility in the 5G system. Such flexibility will enable many native and extensive capabilities including unifying and extending network softwarization, network slicing, network function virtualization, multi tenants, network visualisation, network programmability under software control for all segments of the 5G including in the radio access network, front/middle/back haul networks, access networks, aggregation networks, core networks, service networks, mobile edge computing and software defined network clouds. Such native flexibility would:

- Support for on demand composition of network functions and capabilities to levels unattainable in 4G.

- Enforce required capability/capacity/security/elasticity/adaptability/ flexibility where and when needed.
- Enable integrated management and control to be part of the dynamic design of the software architecture.
- Enable orchestration functionality to be part of the dynamic network control.
- Enable services to be executed in one (or more) slices, i.e. a set of virtual machines, or virtual networks.

Network Softwarization is an overall transformation trend for designing, implementing, deploying, managing and maintaining network equipment and/or network components by software programming. It exploits the nature of software such as flexibility and rapidity all along the lifecycle of network equipment and components, for the sake of creating conditions enabling the re-design of network and services architectures, optimizing costs and processes, enabling self-management and bringing added values in network infrastructures. Additional benefits are in enabling global system qualities, including execution qualities, such as usability, modifiability, effectiveness, security and efficiency; and evolution qualities, such as testability, maintainability, reusability, extensibility, portability and scalability. Viable architectures for network softwarization must be carefully engineered to achieve suitable trade-offs between flexibility, performance, security, safety and manageability. *Network softwarization* is also a set of software techniques, methods and processes applicable to a heterogeneous assembly of component networks or an enhanced version of an existing grouping of network components that is operated as a single network. Network softwarization provides abstractions and programmability that are utilized to deliver extreme flexibility in networks to support variety of applications and services, to accelerate service deployment and to facilitate infrastructure self-management.

Network softwarization includes the following:

- *Network virtualization and network slices*, which enables virtualization of network resources.
- *Network programmability* empowers the fast, flexible, and dynamic deployment of new network and management services executed as groups of virtual machines in the data plane, control plane, management plane as service plane in all segments of the network. Programmable networks are networks that allow the functionality of some of their network elements to be programmable dynamically. These networks aim to provide easy introduction of new network services by adding dynamic programmability to network devices such as routers, switches, and applications servers. Dynamic programming refers to executable code that is injected into the execution environments of network elements in order to create the new functionality at run time. The basic approach is to enable trusted third parties, such as end users, operators, and service providers to inject application-specific services into the network. Applications may utilize this network support in terms of optimized network resources and, as such, they are becoming network aware. The behaviour of network resources can be customized and changed through a standardized programming interface for network control, management and servicing functionality. The key question is: how to exploit this potential flexibility for the benefit of both the operator and the end user without jeopardizing the integrity of the network. The answer lies in the promising potential that emerges with the advent of programmable networks in the following aspects:
  - Rapid deployment of large number of new services and applications;

  - Customization of existing service features;
  - Scalability and operational cost reduction in network and service management;
  - Independence of network equipment manufacturer;
  - Information network and service integration;
  - Diversification of services and business opportunities.

- *Software-defined network clouds* - Cloudification of networking and servicing functions, which enables ubiquitous network access to a shared services and shared pool of configurable computing, connectivity and storage resources. It provide users and providers with various capabilities to process and store their data and services in data centres. It relies on sharing of resources to achieve coherence and economies of scale, similar to a utility, like the electricity grid, over a network. It uses virtualization concepts such as abstraction, pooling, and automation to all of the connectivity, compute and storage to achieve network services. It could take also the kind of mobile edge computing architecture where cloud-computing capabilities and an IT service environment are available at the edge of the mobile network or fog architecture that uses one or a collaborative multitude of end-user clients or near-user edge devices to execute a substantial amount of services rather than in cloud data centres, communication rather than routed over the internet backbone, and control, configuration, measurement and management.

Although many of these features are to be expected in future networking specific 5G network softwarization has following additional key characteristics:

- *5G Harmonization and unification of SDN and NFV* - Coordination of the current SDN and NFV technologies for realizing 5G mobile network is required.

- *5G Extensions to the current SDN and NFV* - 5G network needs extreme flexibility in supporting various applications and services with largely deferent requirements. Therefore, 5G specific extensions to the current SDN and NFV, especially pursuing even further and deeper agile software programmability is required. For example, SDN data plane could be enhanced to support deep programmability and NFV MEC needs light-weight management for extreme edge network functions, especially in the area of access network and user equipment or user devices.

- *5G Considerations for applicability of softwarization* - Considering the trade-off between programmability and performance is required. Especially in 5G context, it is important to respect the performance improvement in wireless technologies. Therefore, it is necessary to clearly define the area and criteria for applicability of softwarization in the infrastructure.

- *Application Driven 5G network softwarization*- 5G mobile network is indispensable communication infrastructure for various applications and services such as IoT/M2M and content delivery. Rapid emergence of applications and services enabled in 5G mobile network must be considered in designing and developing the infrastructure.

- *5G network softwarization energy characteristics* - The architecture design, resulting implementation and operation of 5G network softwarization are recommended to minimize their environmental impact, such as explicit energy closed control loops that optimizes energy consumptions and stabilization of the local smart grids at the smart city level.

- *5G network softwarization management characteristics* - The architecture design, resulting implementation and operation of 5G network softwarization are recommended to included uniform and light-weight in-network self-organization, deeper autonomy, and autonomicity as its basic enabling concepts and abstractions applicable to all components of 5Gnetwork.
- *5G network softwarization economic characteristics* - The architecture design, resulting implementation and operation of 5G network softwarization are recommended to consider social and economic issues to reduce significantly the components and systems lifecycle costs in order for them to be deployable and sustainable, to facilitate appropriate return for all actors involved in the networking and servicing ecosystem and to reduce their barriers to entry.

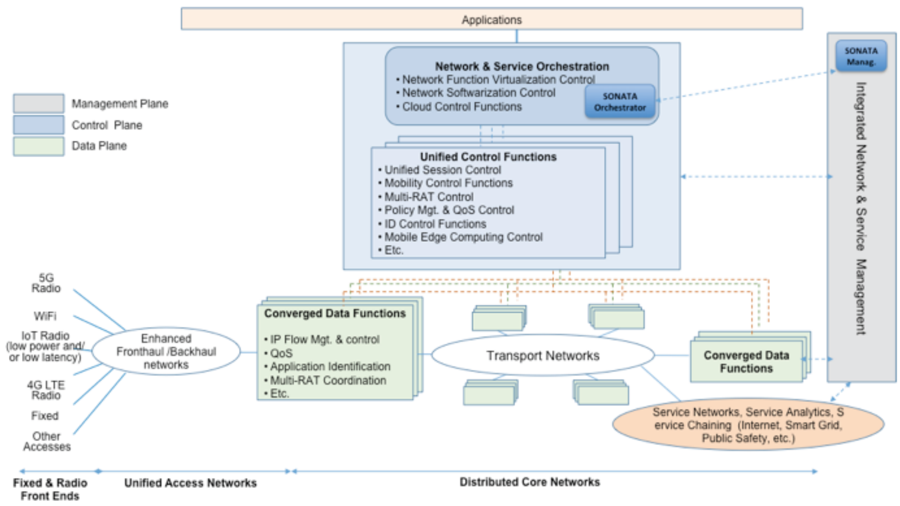

Figure 1.1: High-level 5G network architecture

Figure 1.1 shows the high-level 5G network architecture [30] with the positioning of the SONATA service functionality. A 5G network differentiates itself from legacy networks by further evolution and revolution as well in radio network, front/back-haul networks, and core networks. Multiple various access points, including a new 5G RATs, and e fixed networks, are connected to a converged data plane functions via an integrated access network so that mobile devices can be serviced through an access technology-agnostic network core. The converged data plane functions are distributed to the edges of a 5G network common core network resulting in creating a distributed flat network. The control plane functions, which are responsible for mobility management, QoS control, etc., controls the user traffic to be served agnostically to the access networks to which it is attached.

1.2 SONATA Software Network and Virtualized Network Services

In this scenario, SONATA aims at increasing the flexible programmability of 5G networks with i) a novel Service Development Kit and ii) a modular Service Platform & Orchestrator in order to bridge the gap between telecom business needs and operational management systems.

SONATA is based mainly on the two main Software Networks technologies, SDN and NFV. SDN decouples the network control and forwarding functions and offers abstractions to higher layer applications and network service, and thus the network control becomes programmable. Complementarily, NFV uses the technologies of IT virtualization and cloud computing to decouple network service from proprietary hardware appliances so they can run in software. The virtualized network service utilizes one or more virtual network functions that may be chained together on virtualized infrastructure instead of running on custom hardware appliances. The appropriate integration of SDN and NVF to support next-generation network infrastructures, therefore, is one of the major challenges in the telecom industry today.

Since the advent of SDN and NFV, a common approach was to consider SDN as an enabler for NFV as it provides the flexibility of network infrastructure as needed. Thus, the relation between SDN and NFV is often referred to as 'SDN enables NFV'. But it is more than this. While a software-controlled infrastructure is an essential component to achieve the promises of software networks, a complete integration of the decoupling of capacity and functionality provided by NFV with the fully programmable network control supported by SDN is required. This integration implies a deep understanding of the elements that compose a network service as well as their requirements for an appropriate orchestration. In the following, we briefly elaborate on the basic building blocks that facilitate network softwarization and virtualized network services.

1.2.1 Network Resources

Resources, in general, are the basic components of the Network Function Virtual Infrastructure (NFVI) as defined by the ETSI NFV reference architecture. They support the virtualization and abstraction of key services and facilities available through the infrastructure. Resource orchestration extends the traditional Cloud Infrastructure as a Service to accommodate specialized resources to provision NFVI, including features like Enhanced Platform Awareness to provide a consistent performance across deployments, or support for mechanisms related to WAN SDN control to implement virtual data centre support and the integration of physical network equipment.

These must be complemented with core capabilities related to the management, monitoring and operation of the infrastructure by the network functions using them, directly hosted in the infrastructure itself, taking advantage of the high availability and scalability provided by the virtualized infrastructure:

- The management systems for the virtualized computation and storage services
- The control and management functionalities for the virtualized supporting network infrastructure, such as the infrastructural SDN, say the infrastructure SDN controller itsel), Service Function Chaining (SFC: policy, classification, operations, administration, and management), network topology information, and pooling mechanisms.
- Multi-tenancy support for all infrastructural services
- Data collection facilities able to provide monitoring data from the infrastructure and functions facilitate optimization and fault detection and isolation.

1.2.2 Network Functions

We consider a network function any piece of software that implements a network-related functionality, such as IP forwarding, firewalling, RAN optimization, QoS management, and service fulfilment, over a virtualized infrastructure built as a pool of virtual resources based on hardware,

such as compute, storage, network, and software with common APIs and functions, as discussed above.

Note the definition does not make any assumption on the nature of functions themselves, and they can be implemented in many ways, from the current usual approach of several Virtual Machines running on a hypervisor environment to a single process or thread within a particular operating system or application instance.

Function orchestration must provide a number of basic functional blocks ready to be used by network functions to run, including support for the common VNF Manager, in charge of dealing with the life-cycle events of network functions by executing the appropriate event management callbacks. Other potential functional blocks can even be common network function themselves, including AAA interfaces, database functionalities, or a tenant SDN framework, providing enhanced network information and control facilities coordinated with the infrastructural SDN controller. This way, network functions may be constructed by composing and reusing individual building blocks, allowing their developers to focus on the core aspects of their semantics.

1.2.3 Network Services

If the term *orchestration* is overloaded, we must acknowledge the term *service* is even more. In the context of SONATA, a service is not only the functionality provided to satisfy a certain set of requirements from whatever the user, but we contemplate as well all the elements required to satisfy the requirements of the network end users. That implies services are not only defined by the composition of a certain set of functions and their attachment to network points of presence, but by the information and mechanisms supporting their development, deployment and operation. This requires an integral support for service life-cycle:

- Development, including the free composition and reuse of individual functions and other pre-packaged services according to the service requirements.
- Activation, where the appropriate identity and rights by the actor requesting it must be established in order to perform the necessary orchestration actions.
- Deployment, where the service and its components must be instantiated and configured according to the applicable policies and the deployment environment.
- Runtime, when the service must be monitored to collect appropriate accounting data and apply the necessary corrective actions if required.

Being oriented to the development and operation of software-based constructs, these mechanisms can be integrated into a service development kit supporting service programming, and a service platform facilitating service operations, both interconnected and supporting the DevOps paradigm [32].

1.2.4 The Compositional Continuum

In the two previous sections we have described the compositional creation of both network functions and services, so functions cannot be considered as atomic implementations of a certain set of network functionality, and services are not just limited to a graph connecting a set of network functions. A function will be able to integrate functional blocks that can be considered functions themselves when integrated within a service, and a properly packaged service can be composed with other functions and services to build a higher-layer service.

These recursive composition capabilities allow for what we could call a *compositional continuum* for development and operation, where the same programming tools can be applied at any software layer independently of the complexity of the service or function being addressed, and the orchestration can focus at the two essential layers:

- *Resource orchestration*, dealing with the infrastructure, already addressed by current cloud orchestrators, or at least suitable to be implemented by evolved cloud orchestrators.
- *Service orchestration*, a holistic orchestration of all the components of a service incorporating various constraints expressed, for example, by a service level agreement. This also addresses DevOps operations, like seamless updates, of SONATA network services.

1.3 Structure of this Document

The remainder of the document is structured as follows. First, Chapter 2 provides an overview on the general architecture of the SONATA system: the main components, the main actors, their relations and interactions; the functional and non-functional aspects; and the information model. Chapter 3 presents the basic components and artefacts that are shared by other parts of the architecture, namely the SONATA software development kit and the SONATA service platform. Next, Chapter 4 and Chapter 5 elaborate in detail on the basic workflows, the architecture, and components of the software development kit and the service platform, respectively. Finally, Chapter 6 concludes the document, addresses some open issues that are address by different work packages, and provides an outlook on our future work.

2 SONATA General Architecture

SONATA project main goal is to increase the flexibility and programmability of 5G networks in order to bridge the gap between telecom business needs and operational management systems. In this chapter, we give an overview of the SONATA architecture, including a short description of the main components. As shown in Figure 2.1, the SONATA service programming and orchestration framework consists of the SONATA software development kit, the SONATA service platform, and different catalogues storing artefacts that can be produced, used, and managed by the SONATA system. Services developed and deployed by this system run on top of the underlying infrastructure accessible to the SONATA system via Virtual Infrastructure Managers (VIMs).

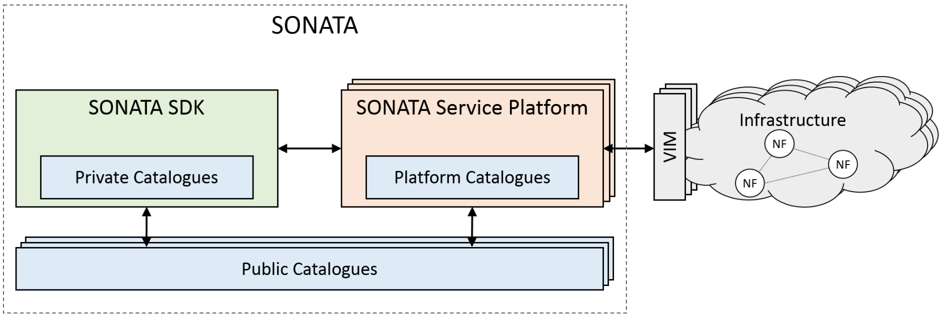

Figure 2.1: Main architecture components

- **Catalogues:** As illustrated in Figure 2.1, the SONATA system includes different catalogues for storing static information regarding network functions and services, like code, executables, configuration data, and specific management requirements and preferences. Contents, location, organization, and implementation of catalogues for different artefacts can vary considerably. However, users of these catalogues need to deal with them in a consistent fashion and the differences across different catalogues need to be harmonized and abstracted away. Different types of catalogues and SONATA's solution for a consistent catalogue access within the SDK are described in more details in Section 4.2 and Section 5.2.2. As a high-level categorization, we foresee the following three types of catalogues in SONATA:
  - Private catalogues of service developers, where they can define, access, reuse, and modify services and service components.
  - Service platform catalogues made available to authorized service developers for reusing existing components in their services, and used for storing services and their components that need to be deployed by the service platform.
  - Public catalogues storing artefacts developed and maintained by third-party developers on arbitrary platforms accessible to service developers and service platform operators.
- **Service Development Kit (SDK):** The SDK supports service developers by providing a service programming model and a development tool-chain. Figure 2.2 shows an overview of

the foreseen SDK components. SONATA's SDK design allows developers to define and test complex services consisting of multiple network functions, with tools that facilitate custom implementations of individual network functions. The implemented artefacts are stored in the developer's private catalogues. Moreover, service components can easily be obtained from external catalogues using the foreseen interfaces. The obtained artefacts can be directly used in a service or after being modified and tested using the SDK development tools. The service components and all the information necessary for deployment and execution of a service are bundled together into a package. The service package can be handed over to a service platform for actual deployment and for testing, debugging, and profiling purposes. The internal structure and the components of the SDK are described in Chapter 4.

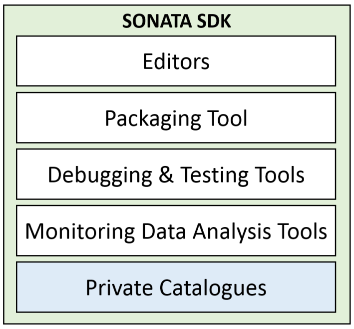

Figure 2.2: SONATA's SDK

- **Service Platform:** As shown in Figure 5.7, a gatekeeper module in the service platform is responsible for processing the incoming and outgoing requests. The service platform receives the service packages implemented and created with the help of SONATA's SDK and is responsible for placing, deploying, provisioning, scaling, and managing the services on existing cloud infrastructures. It can also provide direct feedback about the deployed services to the SDK, for example, monitoring data about a service or its components. SONATA's service platform is designed with full customization possibility, providing flexibility and control to operators and developers at the same time. The service developer can ship the service package to the service platform together with service- or function-specific lifecycle management requirements and preferences, called Service-Specific Managers (SSM) and Function-Specific Managers (FSM), respectively. SSMs and FSMs can influence the Service and VNF lifecycle management operations, e.g., by specifying desired placement or scaling behaviour. By virtue of a modular design in the Management and Orchestration Framework of the service platform, the service platform operator can customize it, e.g., by replacing the conflict resolution or information management modules. SONATA's service platform is described in detail in Chapter 5.

- **Underlying Infrastructure:** The infrastructure needs to host and execute the actual network functions of a service, e.g., as a virtual machine. The service platform sends necessary information and instructions for execution and lifecycle management of services to the infrastructure. The infrastructure may belong to the service platform operator, or to a third-party infrastructure operator. The interaction between the service platform and the infrastructure

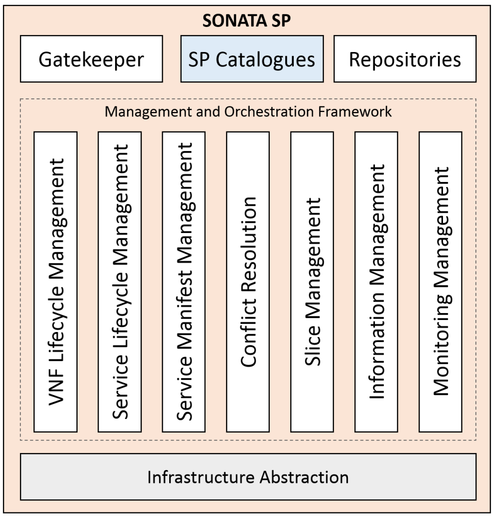

Figure 2.3: SONATA's Service Platform

is done through a VIM, e.g., OpenStack, which provides an abstract view on different infrastructure resources, as described in Section 5.2.4. This description is based on the assumption that a SONATA service platform runs directly on top of an actual infrastructure. However, the SONATA system design also enables a recursive deployment model, where a service platform can act as an abstraction to the underlying infrastructure for another service platform, creating a recursive, hierarchical service platform. More details regarding SONATA's recursive architecture are given in Section 5.5.

SONATA's system design is based on the DevOps workflow, which is supported by the integration between the SDK and the service platform. This workflow implies continuous deployment and continuous integration during service development. The main entity exchanged between the SDK and the service platform is the service package to be deployed and runtime information like monitoring data and performance measurements regarding the service package, which is provided to the service developer during the development phase, as well as the runtime. This information can be used for optimizing, modifying, and debugging the operation and functionality of services.

The general architecture design of the SONATA system complies with and builds upon the ETSI reference architecture for NFV management and orchestration [29]. Lifecycle management operations are divided into service-level and function-level operations in SONATA. These two together, as shown in Figure 2.4, define the elements that build the NFV Orchestrator (NFVO) and VNF Manager (VNFM) functionalities in ETSI's reference architecture. The key reference points of ETSI NFV are preserved (e.g., the Or-Vnfm interface defining the interactions between NFVO and VNFM) and complemented (e.g., Wi-Vnfm interface responsible for WAN infrastructure management) in SONATA.

In the rest of this chapter, we define the main actors of the SONATA system and the way they interact with the system. Afterwards we describe the functional and non-functional aspects of SONATA's architecture, as well as the information model assumed in the architecture design.

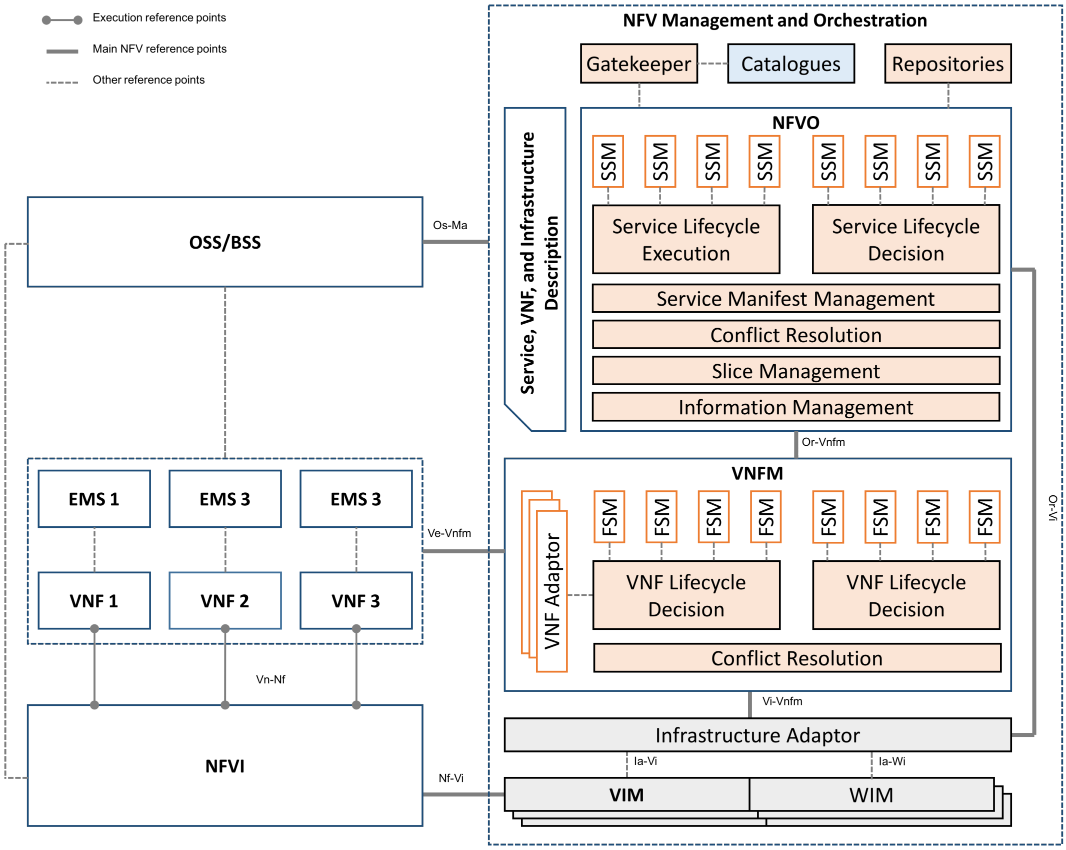

Figure 2.4: Mapping functional architecture of the SONATA system to ETSI reference architecture

2.1 Main Actors And Interactions

There are four main actors in the SONATA ecosystem including, End Users, Developers, Service Platform Operator, and Infrastructure Operator. Figure 2.5 illustrates the interactions between different actors in SONATA ecosystem.

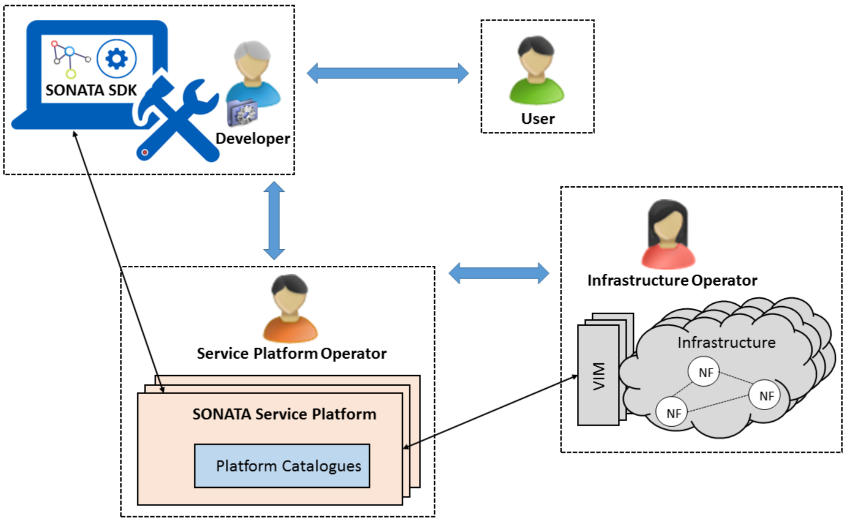

Figure 2.5: Main actor's high-level interaction in SONATA ecosystem

2.1.1 End Users

It is the entity that consumes the network service. Thus, an end user can be a private person or an enterprise, and more interestingly, can also be a network service provider or content providers. In the SONATA ecosystem, the end user requests the required network services from the network service developer. That is, the end user and the network service developer establish a customer-provider relationship which is regulated with service level agreements (SLA).

2.1.2 Developer

It is the developer of a software artefact, for example, a VNF or more specifically, a network service, where a network service can be composed of one or more VNFs. The SONATA SDK shall enable a DevOps environment for the development of network services. The network service developer can use the editor, debugging, and packaging tools in the SONATA SDK along with VNF catalogues to compose a network service which can then be moved to SONATA service platform for deployment and execution. In the SONATA ecosystem, it is assumed that the network service developer also acts as the network service provider/operator for the end user. That is, the network service developer interacts with the end user for providing the network services and also interacts with the SONATA service platform operator for the deployment and operation of those network services.

2.1.3 Service Platform Operator

The service platform operator runs a SONATA platform that manages the execution of network services. The SONATA service platform receives a network service in form of a package which

is validated through series of quality assurance tests prior to their storage in service platform catalogue. In addition to validation, service platform operator also manages the deployment and execution of network services on virtual resources made available by an infrastructure operator. As SONATA service platform supports recursiveness, an operator manages multiple SONATA service platform instances as well. Hence, a handsome responsibility of maintaining smooth operations for a network services , with respect to its corresponding SLA, lies on the service platform operator. On one side, the service platform operator interacts with the SONATA SDK for network services reception and, on the other side, interacts with the infrastructure operator for their deployment and execution.

2.1.4 Infrastructure Operator

It is the entity that actually operates the physical infrastructure, including computation, communication and storage resources. The SONATA ecosystem does not distinguish between infrastructure owner and Infrastructure Operator and treat them as the same. This is because the main focus of SONATA is to reduce time to market for network services by accelerating and facilitating service development, management, and orchestration with DevOps adoption while ensuring features such as multi-tenancy, recursiveness, and virtual network slices. The Infrastructure Operator interacts heavily with service platform operator as the network services are actually executed in the physical infrastructure. This interaction is enabled by the infrastructure abstraction layer in the SONATA service platform. It is worth mentioning that most probably but not necessarily, the Infrastructure Operator will be the same as the Service Platform Operator.

2.2 SONATA Functional and Non-Functional Aspects

The purpose of this section is to present the full SONATA functionality in an implementation free approach. To this end, we wrap up the functional and non-functional aspects that SONATA takes into account.

2.2.1 Orchestration Functionality

The ETSI NFV infrastructure description [54] does not define orchestration explicitly. Its meaning may be inferred from the NFVO definition Network Functions Virtualisation Orchestrator (NFVO): functional block that manages the network yervice lifecycle and coordinates the management of network service lifecycle, VNF lifecycle (supported by the VNFM) and NFVI resources (supported by the VIM) to ensure an optimized allocation of the necessary resources and connectivity. Where lifecycle management is defined as a set of functions required to manage the instantiation, maintenance and termination of a VNF or a network service.

The ETSI NFV orchestration, as described in [54], is seen as a single concentrated functional block, without delegation. The NFV orchestrator may consider resource availability and load when it responds to a new demand, and may rebalance capacity as needed, including creating, deleting, scaling and migrating VNFs.

Although [15] does not formally define (SDN) orchestration, the meaning of the concept is apparent from the SDN controller that is expected to coordinate a number of interrelated resources, often distributed across a number of subordinate platforms, and sometimes to assure transactional integrity as part of the process. This is commonly called orchestration. An orchestrator is sometimes considered to be an SDN controller in its own right, but the reduced scope of a lower level controller does not eliminate the need for the lower level SDN controller to perform orchestration across its own domain of control.

A provisional definition of (SDN) orchestration might be: the continuing process of allocating resources to satisfy contending demands in an optimal manner. The idea of optimal would include at least prioritized customer SLA commitments, and factors such as customer endpoint location, geographic or topological proximity, delay, aggregate or fine-grained load, monetary cost, fate-sharing or affinity. The word continuing incorporates recognition that the environment and the service demands constantly change over the course of time, so that orchestration is a continuous, multi-dimensional optimization feedback loop.

2.2.2 SONATA Functional Specification

The SONATA system is functionally specified by pursuing a dual approach and continuously ensuring the complementarity of these two axes:

- a bottom-up axis, in which functionality is elicited from i) a set of 6 use cases (UC Internet of Things (IoT), UC Virtual CDN (vCDN), UC Guaranteed, resilient and secure service delivery in Industrial Networks (Ind Net), UC Virtual Evolved Packet Core (vEPC), UC Personal Security Service (PSA), UC Separate Client and Hosting Service Providers (SCHProv) [2] (**45 functions**) considering services and systems technologies, will be used to design a system that aims at resolving operators' problems identified in live networks and on existing service/network architectures; ii) previous orchestration framework research ([5], [6]) (**6 functions**) to achieve a coherent set of functionalities that can interwork in a scalable

- a top-down axis, in which functionality is elicited and capitalize on 5G Wider architectural context and from the SONATA description of work [3] (**16 functions**) to achieve a 5G relevant set of functionalities (**5 functions**) that can increase its flexibility and programmability.

2.2.2.1 SONATA Functionality Applicable to all SDK, Service Platform and Service Orchestrator Systems

System Primitives

- Function Name ***Slice Orchestration***: Slice life cycle management, including concatenation of slices in each segment of the infrastructure and vertical slicing of the data pane, the control plane, and the service plane; slice elasticity, placement of virtual machines in slices. It takes over the control of all the virtualized network functions and network programmability functions assigned to the slice, and (re-)configure them as appropriate to provide the end-to-end service. A slice is the collection of virtual network functions connected by links to create an end-to-end networked system. Slices are composed of multiple virtual resources which is isolated from other slices. Slicing allows logically isolated network partitions with a slice being considered as a unit of programmable resources such as network, computation and storage. Considering the wide variety of application domains to be supported by 5G network, it is necessary to extend the concept of slicing targeted by the current SDN/NFV technologies. *(Wider 5G Architectural Context.)*

- Function Name ***Coordination***: It protects the infrastructure from instabilities and side effects due to the presence of many service components running in parallel. It ensures the proper triggering sequence of SC and their stable operation. It Defines conditions/constraints under which service components will be activated, taking into account operator service and network requirements (inclusive of optimize the use of the available network & compute resources and avoid situations that can lead to sub-par performance and even unstable and oscillatory behaviours) *(Wider 5G Architectural Context).*

- Function Name ***Service Platform & SDK Hosting***: 5G Node hosting functions for the service platform and SDK full or partial functionality *(Wider 5G Architectural Context).*

- Function Name ***Recursiveness Enablers***: Virtualization, slicing and orchestration are recursive and involve far more than simply subdividing, aggregating or combining resources. A domain orchestrator sees a set of resources for its exclusive use in satisfying the service request. Recursively within each subordinate/sub domain, the local orchestrator likewise sees and coordinates resources for its own use. Recognizing the generic and recursive nature of virtual resources, the Service Platform may instantiate a VNF of its choice on some available lower-layer container that it knows about in satisfying the service request *(Wider 5G Architectural Context).*

Service Information Model and Enablers

- Function Name ***SP Information Management***: information/knowledge collection, aggregation, storage/registry, knowledge production, distribution and use across all service platform & SDK & infrastructure functions. The importance of the use of uniform information model cannot be overstated as the risk of semantic mismatch is exacerbated if different functions have incompatible information models. This allows purpose-specific APIs to be produced, while enforcing common semantics for the resources of concern. *(Wider 5G Architectural Context)*

- Function Name 'No unnecessary information duplication ': when NS/VNFs and the NFVI are operated by different SPs: Information held by an instance of a SONATA system must not create unnecessary duplication or other dependencies. In particular, a SONATA system offering NFV Infrastructure as a service must not hold specific information about the type of VNFs or network services into which the VNFs are composed. Similarly, a SONATA system of a client SP composing network services hosted on NFV Infrastructure controlled by another service provider must not hold implementation details of the infrastructure. *(UC SCHProv).*

2.2.2.2 Service Development Kit Functionality

- Main Function Name ***SDK: Service Development Kit***: It supports the development of software running inside the actual infrastructure (i.e., both simple virtual network functions and, more importantly, composed services) or running inside the service platform (e.g., decision algorithms about placement or scaling of a service). It also supports the packaging and analysis of services *(DoW).*

SDK Primitives:

- Function Name ***SDK***: Service development kit for fast service specification based on a common and uniform resource abstractions model for connectivity, computing and storage. *(DoW).*

- Function Name ***Invariant specification methods***: invariant specification methods for developers and corresponding verification and debugging tools for invariant checking and reporting during testing and operations. *(DoW).*

- Function Name ***Profiling tools***: profiling tools for monitoring and reporting of service performance and scaling behaviour at run time as feedback to developers. *(DoW).*

- Function Name ***SDK Packaging***: service package that can be handed over to the service platform for execution. This package needs to describe constituting components (individual NFVs, how the composition looks like, scaling properties, certificates for authorization, etc.). We define a package format that describes and encapsulates all such components and that can be processed by the gatekeeper. Support the developer in two ways. One is a set of editors for the various description formats (e.g., for service compositions, for placement optimizations logics). The second is the packaging function, which takes input from the editors, collects all the necessary software artefacts (e.g., required NFV virtual machine images), and produces a package. *(DoW).*

- Function Name ***SDK Catalogue Access***: support the developer in reusing existing NFVs, services, or decision logics. It can interface with, query, or browse existing catalogues, e.g., a private one of the developer, those of the target service platform, or even public marketplaces (like developed by T-Nova or Murano), possibly handling licensing and payment issues. This is akin to dynamically installing libraries in scripting languages from public repositories. *(DoW), (T-NOVA).*

- Function Name ***VNF Catalogue***: The SDK must provide a location to store the different IoT related VNFs. It should be possible for the VNF developer to add update or delete VNFs from that location. Using this list the SONATA service developer can compose a complex service. The system shall offer a VNF catalogue from which the customer can select the desired VNFs. *(UC IoT), (UC Virtual CDN) (T-NOVA).*

- Function Name ***VNF Scaling metadata***: The SDK must allow definition of SLA levels for selected VNFs, other metadata should be possibly be specified as well, such as when and how the SONATA operator should scale the VNF, as well as the scaling strategy (up/down, in/out). This information can be used by the SONATA platform to automatically scale the IoT gateways using appropriate methodologies. The developer should describe in the VNF Descriptor recipes for scaling VNF; VNF composing the network service have to be able to scale up or down depending of the users' demand. SONATA SDK should have the capability to specify different parameters (VM load, BW, latency) to be used by SONATA Orchestrator for scaling (up or down). The developer should describe in the VNF Descriptor recipes for scaling his/her VNF. *(UC vEPC), (UC IoT), (UC Virtual CDN).*

SDK Tools

- Function Name ***VNF SLA Monitor***: SONATA must provide an interface to monitor VNFs SLAs and resource usage. It must highlight VNFs with high and low usage that may need scaling or other kind of manual intervention. The system shall expose service and VNF metrics to the network application. *(UC IoT), (UC Virtual CDN), (UC vEPC).*

- Function Name ***VNF Resource Report***: SONATA must provide an interface to list all resources allocated to a specific service. This service allows the developer or administrator to get an overview of how the service is evolving and what datacenter resources are committed to each service. *(UC IoT).*

- Function Name ***Authorization***: SONATA service must limit operations based on access levels and provide means to create and manage access profiles. This will be used to define the different access levels each user will have to the system, as an example, a VNF developer should be able to deploy a VNF on the catalogue, but should not have the permission to trigger its deployment to a production network. *(UC IoT).*

- Function Name ***VNF Deployment***: SONATA must support placement instructions that express proximity to other entities, e.g, (i) set where the service gateways will be placed on the operator network, (ii) deploy a VNF as near as possible to a specific location, (iii) select were the VNF will be deployed. *(UC IoT).*
- Function Name ***VNF Status Monitor***: SONATA should provide a high level state for each VNF, e.g., (i) deployment, (ii) operating, (iii) error. *(UC IoT).*
- Function Name ***IoT traffic simulator***: Given that there is not yet the amount of IoT traffic this use case is designed to address, there must be a way to simulate IoT sensor traffic with functions like increase or decrease traffic levels per sensor and number of sensors in order to simulate a real IoT sensor environment. *(UC IoT).*
- Function Name ***VNF integration with service***: Sonata must allow new VNFs to be integrated in existing services. It must allow network flow reconfiguration in order to integrate a newly deployed VNF in an existing service graph with minimum or no downtime at all. *(UC IoT).*
- Function Name ***SDK VNF customization***: The SDK must allow the development of custom VNFs with specific algorithms to manipulate IoT traffic, like processing and batching. *(UC IoT).*
- Function Name ***Multiple IoT sensor vendors***: Framework must support traffic from different IoT sensor vendors. Traffic from each sensor should be routed through the appropriate gateway. *(UC IoT).*
- Function Name ***Multiple IoT tenants***: Framework must support multi tenancy, i.e., The infrastructure must support multiple IoT services operating in parallel without any data meant to one operator being routed to another operator's service. *(UC IoT).*
- Function Name ***Support for Service Templates***: The programming model must support service templates. In other words, it must support the inclusion of types of nodes, or at least the notion of cardinalities in inter-node relationships, e.g. in order to define an unspecified number of nodes. Support for corresponding annotations (or primitives) in the service programming model/language. *(UC IndNet).*
- Function Name ***Inter-VNF QoS constraints***: The programming model must support end-to-end QoS properties for inter-VNF communication, such as delay, jitter, reliability (which could be mapped to multi-path transmission by the orchestrator, the developer does not care necessarily), oversubscription. *(UC IndNet).*
- Function Name ***Placement constraints for VNFs***: The programming model must support to specify placement constraints for VNFs, e.g. disjoint placement of active and standby VNF on physically separate machines, pinning a VNF to a specific node or node type (e.g. turbine control must run on a turbine node), hosting nodes must offer certain real-time capabilities or security isolation features, satisfaction of rules of compliance, etc. *(UC IndNet).*
- Function Name ***Security simulation tools***: A common problem in security applications and service is the capability to simulate security incidents. Security simulation tools availability and integration in the SONATA framework are needed for validation and testing, i.e.: DoS traffic generators, or malware traffic samples case. *(UC PSA).*

2.2.2.3 Service Platform Functionality

- Main Function Name: ***Service Platform***: service platform realizes the management functionality to deploy, provision, manage, scale, and place services on the actual infrastructure. It does not execute the services themselves; it rather triggers execution on the actual infrastructure by requiring start-up, migration, shutdown, etc of (virtual) network functions on the actual infrastructure. *(DoW).*

Service Platform Primitives:

- Function Name ***Infrastructure Abstraction***: assuming no uniform control interface has appeared, a shim layer functionality to hide idiosyncrasies of an infrastructure. This should be lightweight function and hopefully will disappear once standards emerge. *(DoW).*

- Function Name ***Conflict Resolution***: Since service-specific logics are likely selfish, conflicts over resource usage will arise. *(DoW).*

- Function Name ***Service Platform Scalability***: The service platform must be scalable to support a very large number of devices (e.g. sensors), a high traffic load, high dynamics etc. depending on the use case. *(UC IndNet).*

- Function Name ***Service Platform Customizability***: The service platform must be customizable to support large-scale, complex deployments (such as carrier networks) as well as smaller, lightweight deployments (such as enterprises or industrial networks). *(UC IndNet).*

- Function Name ***Capability Discovery in Service Platform***: The service platform, notably the infrastructure abstraction layer, must support the discovery of capabilities of the physical infrastructure, e.g. support for hardware-acceleration for certain functions such as encryption or the availability of a Zigbee interface. That way, it will become possible to optimize function placement and maybe even tune applications that have access to the capability-enriched network model via the service platform's NBI. *(UC IndNet).*

- Function Name ***Isolation constraints for VNFs***: The programming model must support isolation constraints for VNFs. This is in terms of performance, e.g. in order to guarantee min. capacity without being preempted by concurrent services. But it is also in terms of security, e.g. in order to restrict visibility of (virtual or real) infrastructure to a particular service, or to constrain a service to specific time windows (e.g. only between 10am and 11am, or expiry one hour after first use). *(UC IndNet).*

- Function Name ***Multi-tenancy***: Some components can be dedicated to a tenant (hard isolation) and some other can be shared (soft isolation). *(UC vEPC).*

- Function Name ***Security VNF availability***: Security virtual network functions require specific capabilities that are not so common in generic VNF, like Anti DDoS or signature detection of IDS. These functionalities must be present to allow creating a valid use case. SONATA VNF Catalogue must include some Security VNFs in order to support this use case . *(UC PSA).*

- Function Name ***Personalized VNF***: VNF catalogue and management framework in SONATA must support the concept of "personal" in the sense that VNFs are assigned as a non-shareable resource with other users in the platform. Also Users identities in SONATA framework must allow a direct mapping between user and his VNFs case. *(UC PSA).*

Service Platform Tools

- Function Name: ***Catalogue and repositories***: a service (along with its constituting parts: logics, NFVs) is placed into corresponding catalogues inside the service platform, from where both kernel and actual infrastructure can access them. The catalogues hold known entities; they are complemented by repositories, which hold information about running entities as well as other frequently updating data, e.g., monitoring data. *(DoW).*

- Function Name ***GateKeeper***: This service platform function will check whether a service can be executed by the service platform before accepting it. It will check, e.g., authorization of a developer submitting a service, completeness of the service description, or the availability of all NFVs composing the service. *(DoW).*

- Function Name ***Service Monitoring and Monitoring Analysis***: The service monitoring and the monitoring analysis working closely together, will collect and analyse service-level (not just network) performance indicators. *(DoW).*

- Function Name ***NFVI Northbound API***: The SONATA system must be able to support an API which exposes the NFV Infrastructure as a service. *(UC SCHProv).*

- Function Name ***Southbound Plugin to use NFVI API***: The SONATA system must be able to support a southbound interface which can request elements of NFV Infrastructure as a service. *(UC SCHProv).*

- Function Name ***Timely alarms for SLA violation***: The monitoring system must supply alarms for SLA violations (or malfunctioning components) in a timely manner depending on the SLA and type of problem. This means that the failure detection, but also the service platform message bus and notification system must have real-time capabilities. E.g. VNF unavailability for real- time traffic must be signaled as fast as possible, while in the case of best-effort traffic alarm signaling can happen with modest delays. Likewise, urgency of alarms is higher for VNFs with 1000s of users compared to single-user VNFs in the general case. *(UC IndNet).*

- Function Name ***Manage update of components***: Sequence/ strategy of update using DevOps. Sequence for validation and migration. *(UC vEPC).*

- Function Name ***Support different modes of management/control***: EPC can be fully managed by the operator, i.e. EPC fully deployed and managed by the customer (e.g. a MVNO). Or Hybrid were components are managed by the operator (e.g. SGW) and others by the customer (e.g. HSS). Or fully managed by the customer. *(UC vEPC).*

- Function Name ***Support Services with high SLA***: vEPC is a service that is operates under a 5 nines SLA, it can not allow service degradation when scaling / healing / updating / migrating. *(UC vEPC).*

- Function Name ***Support "state-full" services***: vEPC is a "state-full service" - all its components are state-full, they can not loss their state when scaling / healing / updating /migrating. *(UC vEPC).*

- Function Name ***Integration with OSS***: vEPC service operation involves integration with OSS system, SONATA should expose relevant APIs. *(UC vEPC).*

- Function Name ***Distributed NFVI***: ISP and Network Operators architecture requires a geographical distribution of PoP (Point of Presence) where instantiate multiples VNFs as close as possible to user or based on the service demand. One example is when a security attack happens it is preferred to react as close as possible of the source of the attack. As a consequence SONATA orchestration layer should support multiples NFVI and VIM in distributed networks case. *(UC PSA).*

- Function Name ***Open interfaces towards NFV***: NFV components like VIM, NFVI or NFVO could be deployed with multiples providers. Indeed the number of NFV solutions is growing day by day. SONATA orchestration framework must support open or standards interfaces (southbound towards NFVI) to ensure the smooth integration of different NFV providers specially to facilitate the adoption case. *(UC PSA).*

- Function Name ***VNF Real-time Monitoring***: In order to detect and react to security incidents, VNFs will generate in real time information useful for Monitoring and response. SONATA framework must be able to collect, store, process and report in valid time windows to be useful to the ISP or the user case. *(UC PSA).*

- Function Name ***VNF reporting to BSS/OSS and subscriber***: In order to detect and react to security incidents, VNFs will generate in real time information useful for Monitoring and response. SONATA framework must be able to collect, store, process and report in valid time windows to be useful to the ISP or the user case. *(UC PSA).*

- Function Name ***Legacy support***: Any ISP or Network Operators or corporations has today deployed security networks solutions in virtualized or bare metal appliances. The most relevant example is a Firewall device. If SONATA has the aim to offer complex solutions and integrate with existing network environment, then SONATA need to interact and manage with not only VNFs also support legacy NF case – *(UC PSA).*

- Function Name ***Quality of service monitoring***: One of the key method to detect security problems are the deterioration in the QoS. Metrics generation and degradation detection of network traffic, i.e. caused by a overloaded NFVI node or a attack, should be supported and reported case. *(UC PSA).*

- Function Name ***VNF and topology validation***: Based on the principle of providing security service, SONATA service framework by itself, or using third parties, must offer a validation capacity of VNFs when it is deployed in the NFVI. This validation should cover the integrity of the VNF, user attestation and data paths case. *(UC PSA), (T-NOVA).*

2.2.2.4 Service Orchestration Functionality

- Main Function Name ***Orchestrator*:** service orchestrator that maps services to connectivity, computing, and storage resources and that processes the output of the SONATA SDK to generate a resource mapping and composition of a new service from virtualized building blocks, manages the service lifecycle (deployment, operation, modification, termination), supports isolation and policing between different virtual services, and virtual service providers. Uses abstract interfaces for interoperability with different underlying technologies such as OpenStack, Apache Cloudstack, OpenVim, OPNFV, etc.

Service Orchestration Primitives

- Function Name **VNF Placement**: The programmability framework shall allow the customer to deploy VNFs at arbitrary points into the network and set where the components/ gateways will be placed on the operator network. For example, deploy a VNF as near as possible to a specific location or select were the VNF will be deployed. *(UC Virtual CDN), (UC vEPC), (T-NOVA), (UNIFY).*

- Function Name ***Manual Service Function Placement***: It should be possible to manually decide and configure where a service is to be placed. This can be very important for scenarios where the service developer knows that a Service Function has to run in a certain location / on a certain node, but is either unable to or not confident with defining placement constraints in a way that placement can be done by the orchestrator. This may be particularly the case in non-carrier verticals where experience with services may be lacking, deployment scenarios are simple, and ease of use is the primary objective. "(UC *IndNet*) (UNIFY).*'*

- Function Name ***SFC: Service chaining***: the programmability framework shall allow the customer to interconnect VNFs in an arbitrary graph. *(UC Virtual CDN), (T-NOVA).*

- Function Name ***Service Chaining Support Across Wide Area Networks***: The Service Platform must support service function chains that include service functions separated by a wide area network, e.g. across different data centres, or between the core data centre and an edge cloud. "(UC *IndNet*), (T-NOVA).*'*

Resource Orchestration Primitives

- Function Name ***Abstract Programming Model***: abstract programming models for networked services, which enable a high level of automation in service development and deployment processes. Such models refer to the installation of executable code as virtual machines representing application-specific services components into the hosting elements in order to create the new functionality at run time - realizing application-specific service logic, or performing dynamic service provision on demand. *(DoW).*

- Function Name ***Multi NFVI orchestration***: SONATA Orchestrator should be able to orchestrate multiple VNF execution environments (NFVI-PoPs) located in arbitrary places in the operator network topology. The NFVI-PoPs are considered to be controlled and managed by VIMs. *(UC vEPC).*

- Function Name ***Lifecycle Management***: The lifecycle management plugin deal with triggering (groups of) VM start-up, shutdown, ... actions in the actual infrastructure; the service contextualization executive plugin contextualizes VNFs/services via actions provided by the service description. *(DoW) (T-NOVA), (UNIFY).*

- Function Name ***Placement and Scaling***: The placement and scaling logic executive executes algorithms that place and scale a running service, both at start-up and continuously (e.g., when load goes up). While we will provide a fall- back algorithm (based on-going projects), our contribution is the ability to execute service-specific logics, provided in a service's description. Challenges here are e.g. security concerns, which we intend to address by sandboxing approaches. *(DoW), (T-NOVA), (UNIFY).*

- Function Name ***Multi NFVI orchestration***: SONATA Orchestrator should be able to orchestrate multiple VNF execution environments (NFVI-PoPs) located in arbitrary places in the operator network topology. The NFVI-PoPs are considered to be controlled and managed by VIMs. *(UC Virtual CDN), (T-NOVA).*

- Function Name ***Integration with existing VNFs or components***: The programming model and service platform must allow components or VNFs of a new service to be integrated with existing services, VNFs or system components (such as sensors or actuators). *(UC IndNet).*

2.2.2.5 Management System Functionality

- Function Name: ***Automatic re-configuration of running competing services***: mechanisms and functions – both on an algorithmic and on an architectural level – that monitor competing services and dynamically re-configure them when desirable. *(DoW);*

Subfunctions *(DoW):*

- functions to continuously monitor services and infrastructure.
- functions to scale-in and scale-out non-trivial, even state-full services
- functions to ensure stability and integrity of the running of multiple services in the hosting environments and managing conflicts.
- functions for dynamic, interruption-free re-configuration of service components.
- functions for re-configuring service components triggered by monitoring functions for the following KPIs: resource usage, service performance, SLA fulfillment, energy consumption.
- functions for demonstrating the programming service models with the assessment of their effectiveness, performance and flexibility.

2.2.3 Non-functional Characteristics

Non-functional characteristics are meant as implementation attributes and artefacts that a specific SONATA systems should include the followings:

- ***Usability:*** it describes the ease with which a system performing certain functions or features can be adopted and used.
- ***Reliability:*** it describes the degree to which a system must work. Specifications for reliability typically refer to stability, availability, accuracy, and maximum acceptable bugs.
- ***Performance:*** it describes the degree of performances of the system (according to certain predefined metrics, e.g. convergence time).
- ***Supportability:*** it refers to a system's ability to be easily modified or maintained to accommodate usage in typical situations and change scenarios. For instance, how easy should it be to add new blocks and/or subsystems to the support framework.
- ***Security:*** it refers to the ability to prevent and/or forbid access to a system by unauthorized parties.

- ***Safety:*** It refers to conditions of being protected against different types and the consequences of failure, error harm or any other event, which could be considered non-desirable.
- ***Resilience:*** it refers to the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation
- ***Compliance:*** it refers to the conformance to a rule, such as a specification, policy, standard or regulatory.
- ***Extensibility:*** it refers to the ability to extend a system and the level of effort and complexity required to realize an extension. Extensions can be through the addition of new functionality, new characteristics or through modification of existing functionality/characteristics, while minimizing impact to existing system functions.
- ***Inter-operability:*** it refers to the ability of diverse (sub)systems to work together (inter-operate).
- ***Operability:*** it refers to the ability to keep a system in a safe and reliable functioning condition, according to pre-defined operational requirements.
- ***Privacy:*** it refers the ability of system or actor to seclude itself or information about itself and thereby reveal itself selectively.
- ***Scalability:*** it refers to the ability of a system to handle growing amounts of work or usage in a graceful manner and its ability to be enlarged to accommodate that growth.

2.3 Information Model

Functional architectures, information models and data models are all defined prior to actual implementation of a system. They are levels of abstraction, which can help us - the designer - clarify exactly what we want to do (the problem to be solved), what we may want to do in the future (something that can be generalised or changed), and what constraints there are on the solution. The functional architecture and information model define the concepts at a high level of abstraction, independent of any specific implementation, device or protocol used to exchange the information – such details are hidden in order to clearly define the relationships between the managed objects. Whereas a Data Model is defined at a lower level of abstraction and includes many details related to the specific protocol, device or implementation; indeed it is often defined at the same time as the protocol.

A functional architecture is based on functional blocks. Simple examples of a functional block could be "source transfers a packet to a destination", or "monitoring node reports its measurements to data collector" (note that the latter would likely build on the former). The related information models could define the packet (source and destination addresses, the data, etc) and the report (the identity of the monitoring node, the identity of the task that performed the measurement, the timestamp of when it was made, the actual measurement result, etc).

One challenge is how to re-engineer long-standing experience of these abstractions in transport networks into the more challenging world of Sonata which includes processing and storage functions, and virtual functions hosted on processing and storage functions. Our on-going work will tackle this challenge.

An Information Model serves several purposes:

- To help agree what information (at a high-level) the Sonata system needs to support – and then to guide the designer of the specific protocol and data model about what functionality needs to be encoded.
- To ease interoperability between different protocols; if their data models instantiate the same information model, then it should be simpler to translate between them
- To help assess the suitability of existing protocols and data models

Work on these topics has begun and will continue as Sonata progresses.

2.3.1 General Background and Requirements for the Information Model

The essence of any distributed information system is that it is made up of parts and the parts pass information between each other. It is therefore essential for any distributed system that the information sent by one part is correctly understood by the receiving part. Fundamentally, this requires that the sending and receiving parts use the same encoding of the information.

It is normally practical to layer the process of encoding. This carries a number of advantages which principally stem from the decoupling of the systems designed to carry out the logic of the distributed system ('the application') from systems designed to host the distributed system and carry the information between the parts ('the computing and transport'). In general, the objectives are diametrically opposed:

- the intended system wishes to restrict the language to only things are relevant to the distributed system and the language should therefore be specific to the distributed application;
- the hosting system wishes to be as general as possible in order to able to host and carry messages for as wide a variety of different distributed systems as possible.

In the extreme, we see the second objective leads to universal systems of processing and transport. With transport, it is a fundamental principle from information theory that all possible messages can be encoded in binary bits and so the transport of binary messages across a binary transport pipe is truly universal. In a similar way, from the basic theory of computation, we have universal programming languages, which can, at least in principle, encode any computation problem.

The role of the information model is the opposite of universal transport and universal programming languages. The aim of the information model is to describe the specific information elements relevant to the specific distributed system. The information model has two main objectives:

- describe the required information elements as precisely but as minimally as possible
- describe them in such a way that the information can be encoded into any plausible hosting/transport scheme.

As a note, there is a general convention in the industry to refer to *information modelling* and *data modelling*. We try to avoid this convention as it implies that there is only two steps: the information modelling and then encoding into a data model. In practice there can be several layers of encoding and several alternative encoding. This allows for the encoding to be layered. In order to retain clarity, we refer to information modelling and encoding of the model in a language. It should also be noted that the information model itself must be described in some sort of language which is itself an encoding, therefore, strictly, all encoding is really a translation between encodings.

2.3.2 Three Key Principles in Developing the Sonata Information Model

2.3.2.1 Critical Value of Abstraction in the Context of the System

The first key principle is that of abstraction. As we noted above, a good information model is *parsimonious* in that it is as simple as possible consistent with being precise. Ideally, it will be necessary and sufficient.

In many applications, many different entities are treated as equivalent and so in the information model which support the treatment of these entities, great simplification can be achieved if they are all represented as being instances of the same type. Then the information model needs to keep track of far fewer types. Each type has a common behaviour within the system. Such abstraction is a central principle for defining the information model.

This process of defining types according to the needs of a particular system requires taking a suitable abstract view of the information but this will be from the perspective of the needs of context of that system.

This clearly suggests that when the same entity is viewed in different contexts with different purposes, they will want to use different abstractions. Abstraction is therefore unlikely to be *universal* and difficult to define outside the specific context.

It should be noted that this is often a problem for predefined API specifications which are defined to present information to *all comers*. As a result, APIs tend to present all possible information and attempt little or no abstraction. As a result API specifications tend to be very large and complex. A solution to the problem of presenting a simplified abstract view for a specific user of the API is to write a 'stub' which can sit between the API and the user which can present the simplified abstraction needed for the specific context.

2.3.2.2 The Information Model as a View on a Functional Model

The specific information model needed by Sonata is to describe information elements needed to orchestrate the lifecycle of NFV functional components, normally virtual functional functions (VNFs) and their components (VNFCs).

The appropriate starting point, therefore, is the functional entities whose lifecycle is being managed by the Sonata system. The Sonata system will require information elements to represent the functional entities. These information elements will record relevant information about the functional entities including:

- function type
- instance id
- administrative, operational, and alarm state
- relationships to other functional entities

The first and last of these, that is the function type and the relationship of a particular function instance to other function instances will derive directly from the nature of the functional entities themselves (and to some extent alarm state is also dependent on the nature of the functional entity). The appropriate approach to identifying the information elements necessary for the Sonata information model is to have a functional model of the functional entities whose lifecycle is being managed by Sonata.

Developing this functional model is therefore the first step in developing the information model. The process is therefore as follows.

Functional model -> information model -> recursive encoding of information model and data model. This is illustrated in Figure 2.6.

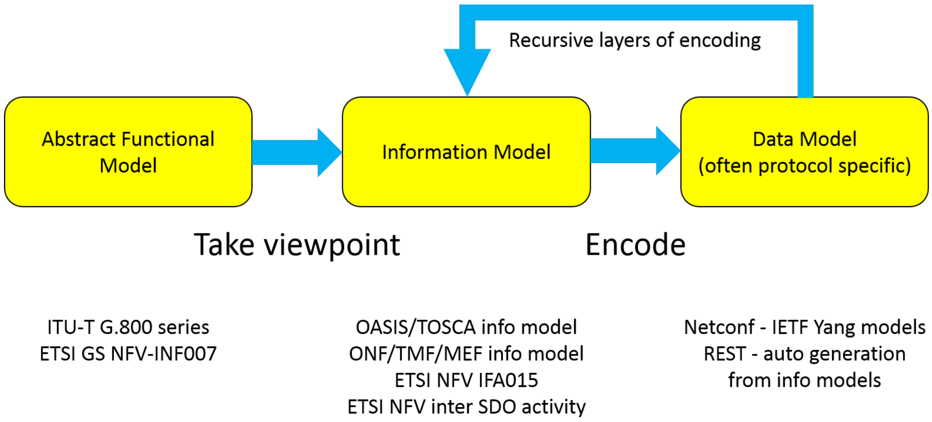

Figure 2.6: Modelling Flow

2.3.2.3 The Sonata Functional Model is a Generalisation of the ITU-T G.800 Series Model

The process of starting with a functional model is already well established for the management of transport networks and is the basis for the interfaces to current OSS systems which manage WDM/OTN networks, Carrier Ethernet, and the SONET/SDH networks before them. More recently, the same functional models have been adopted by ONF for the development of their interfaces.

The functional modelling in all these examples is based on the standards of the ITU-T G.800 series of recommendations. These describe an abstract model of transport networks, including the transport that links together virtual machines and functions hosted on virtual machines. However, it does not including processing and storage functions, nor virtual functions hosted on processing and storage functions.

The functional modelling which is now set out G.800 started with SONET/SDH and was developed from observing the symmetries present in the layering of transport networks and from this observation, the generic, abstract model was asserted. It was a most definite aim of even this earliest work that the model should not just describe SONET/SDH functionality but be extensible to other technologies. Even at the time, around 1990, WDM was on the horizon as a technology as this provided a useful example of future extension.

As the model was successfully extended to cover technologies as they emerged including ATM, MPLS, and Ethernet, those working on the model hypothesised whether the model could be **deduced** rather than simply **asserted** from observation. As a result G.800 in its current form is deduced from five axioms of the model. This means that it is possible to state that the functional model both **necessary** and **sufficient**. In other words, it is possible to state with confidence that any system which conforms with the axioms, can be modelled precisely using the G.800 functional model.

These G.800 axioms are specific to transport networks but otherwise very general. They are (paraphrased)

- a transport network is a system which transports information between geographically separated end points

- the resources of the network are limited and must be shared between different instances of communication
- the end points for communications can be selected dynamically and must have some form of identification scheme
- the transport of the information may be unreliable
- the transport network may be operated by a number of different independent organisations.

In this, the terms **system** and **information** are defined mathematically and very precisely.

The ideal approach would be to generalise the G.800 functional model so that the G.800 model is a strict subset of this new generalised model. This has advantages over an alternative approach which would be to supplement the G.800 model with additional entities to cater for processing and storage.

Work on this has already taken place in Trilogy 2 and this work was adopted by ETSI NFV ISG and published as ETSI GS NFV-INF 007 *Methodology to describe Interfaces and Abstractions*. The functional model presented here is based on this work. This work gives a overall framework, however, there are many aspects where this work is much less mature than that of G.800. A further issue with a common functional model for both the whole scope of NFV which converges the IT compute industry and the networking industry is that there is a general mismatch in the fundamental descriptive paradigms. The networking industry is accustomed to using functional block modelling, while the IT industry basic descriptive paradigm is programming languages, normally universal programming languages.

While the functional model of ETSI GS NFV-INF 007 does successfully unify these different paradigms, there is still a large gap arising from the history of the two different descriptive paradigms. All this means that there is considerable work still required to develop the generic functional modelling whose essence is set out in ETSI GS NFV-INF 007. However, the development of a common information model is likely to be an aid in bridging the gap between the two current descriptive paradigms.

The general relationship between a generic functional model and specific functional models for network and compute (including both processing and storage) together with the corresponding relationships between the corresponding information models is illustrated in Figure 2.7 below.

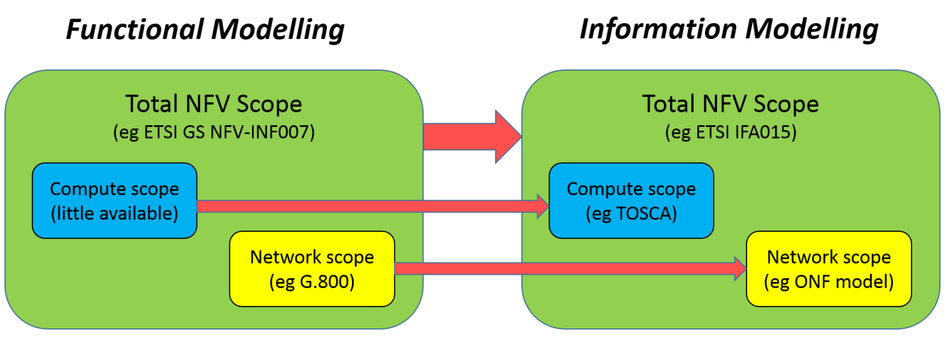

Figure 2.7: Relationahip between functional models and information models at different levels of abstraction

2.3.3 General Abstract Functional Model

ETSI GS NFV-INF 007 generalises the functional model methodology to support virtualization. The idea, as shown in the Figure 2.8, is to divide the functional block between a "host" part and a "virtual" part. From an external perspective, it behaves like the virtual function, whilst from an internal perspective the host implements the virtual function. Thus the virtual function can avoid implementation details, but note that it depends on the host function for its existence. The essential requirement to be able to virtualize is that some of the host's state can be fixed for the lifetime of the virtualised function.

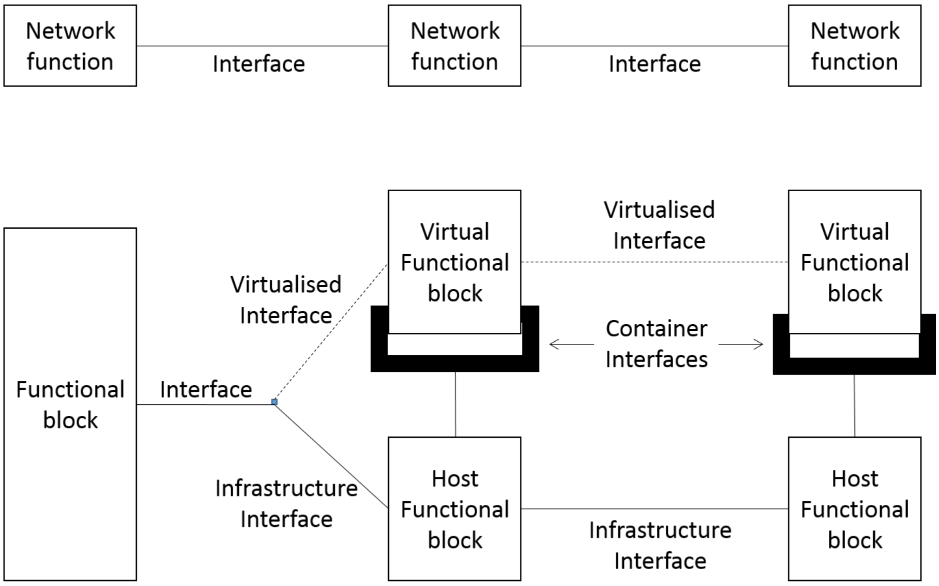

Figure 2.8: Interconnected functional blocks (top), two of which are implemented as virtualized functions on host functions (lower)

We now describe this in more detail. The functional model is based on a fundamental entity which is a hosting functional block. This is a functional block according to the normal conventions of systems engineering and has

- interfaces through which information passes to and from the functional block;
- state which is held in state variables and which can change value over time according to inputs and previous state;
- a transfer function which determines the outputs and the next state depending on the current inputs and the current state.

ETSI GS NFV-INF 007 generalises the functional block methodology to support virtualization. It shows an example where there are three interconnected functional blocks, two of which are implemented as virtualized functions on a host function:

A hosting functional block is a functional block which can be configured or, synonymously, programmed. Once configured, the hosting functional block appears to have the functional block characteristics of new functional block whose behaviour has been determined by the specific configuration. This new functional block is a virtual functional block which is being hosted by the hosting functional block. More precisely, the act of configuring fixes some of the state of the hosting

functional block such that it combines with the host functional block's transfer function to give a new transfer function which is that of the client virtual functional block. This is described in more detail in ETSI GS NFV-INF 007.

This concept of the hosting functional block is very general and covers all form of processing, storage, and networking entities.

We present the Sonata functional model (from which the information model derives) using SysML which is a derivative of UML. There are a number of reasons why SysML is appropriate in these circumstances, and these include a simple point of clarity. The functional model is shown using the functional blocks of SysML while the real information elements of the information model can be shown using UML making the distinction between the functional model and the information model clear.

Figure 2.9 below illustrates this basic hosting functional block. It shows that there are two basic types of host functional block: a physical host functional block and a virtual host functional block. (In SysML diagrams, the triangular-headed arrow indicates 'is a type of'.) A further point to note is that a forwarding domain and the forwarding construct, the central entities of G.800, are each a type of virtual hosting functional block. This is key to the ETSI GS NFV-INF007 functional model being a generalisation of G.800.

The approach reveals that there are two recursive relationships - hosting and composition.

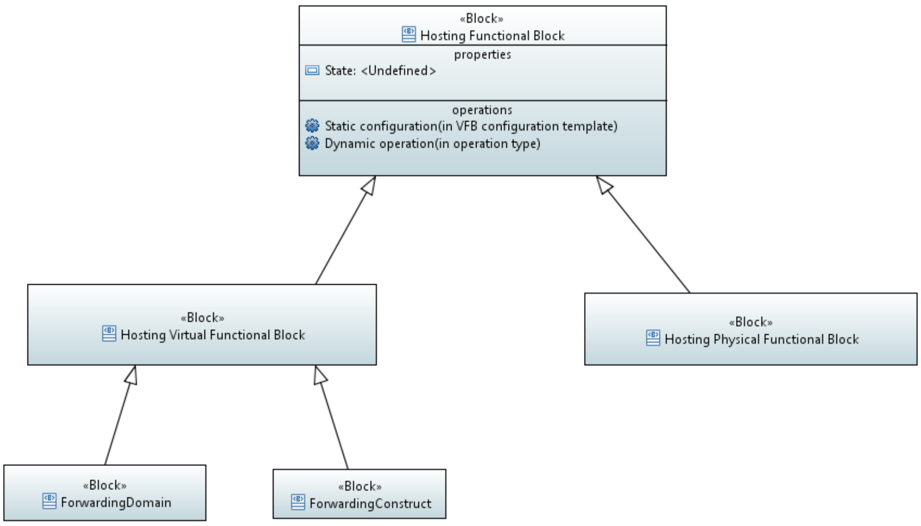

Figure 2.9: Top Level Inheritance

Hosting is the first central recursive relationship and is illustrated in the Figure 2.10. The virtual functional block cannot exist with its hosting functional block (represented by the filled diamond in the SysML diagram). A hosting functional block can host zero, one or more virtual functional blocks (as indicated by the numbers attached to the line in the SysML diagram). The physical hosting functional block is shown as a special case: it can only be defined as on one end of this hosting relationship, since it is at the bottom of the recursion.

The forwarding domain and forwarding construct of G.800 are specializations of hosting virtual functional blocks and that the layering relationship of G.800 is a specialization of the hosting relationship. The forwarding domain is a host function while the forwarding construct is a virtual function hosted on the forwarding domain. A simple example is a network which is a forwarding domain and connections formed across the network are forwarding constructs. The network has hosted the connection. However, the model can also be fully recursive as the connections may be

used as tunnels to form links in a higher layer of network.

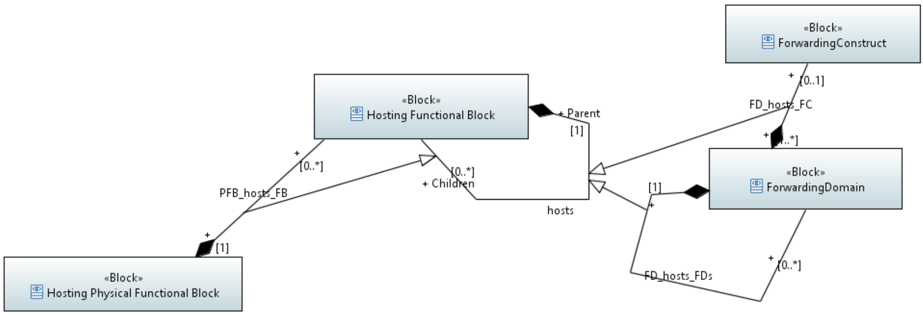

Figure 2.10: Top Level Hosting

The second fundamental recursive relationship is composition. Composition is the well understood property of functional blocks from systems engineering where a set of component functional blocks can be composed together by connecting the outputs of a function to inputs of other functions according to a defined topology of components (nodes of the topology) and bindings between the ports (the branches of the topology). The recursive composition relationship is illustrated in Figure 2.11 below. A component's existence does not depend on its composite, since components can readily exist outside any composite or may be transferred from one composite to another. Therefore this relationship is shows as an open diamond in the SysML diagram.

In addition, the figure shows that the partitioning relationship between forwarding domains of G.800 is a specialization of the of the composition relationship between virtual functional blocks.

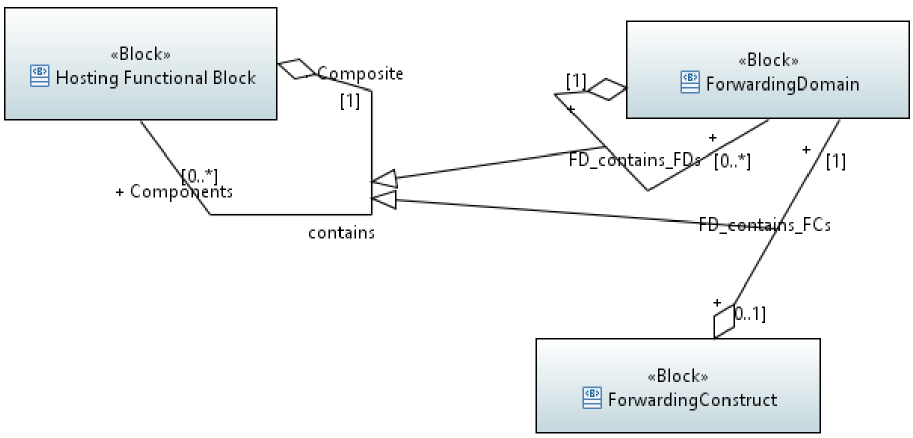

Figure 2.11: Top Level Composition

Hosting and the recursive hosting relationship are the critical additions to the traditional functional block methodology. Without hosting and relying only on composition, there is no mechanism in the functional block description by which functions can be created and this is one of the primary processes for Sonata. The addition of the recursive hosting relationship is critical and profound. In terms of object oriented programming languages, hosting is the mechanism which supports "new". Hosting is the mechanism in the NFV infrastructure which enables Sonata to call "new" to create a new VNFCs, VNFs, and network services.

2.3.4 NFV Abstract Functional Model

2.3.4.1 NFV Specific Entities

The general model described above can be made more specific to the context of NFV. This is achieved by deriving subclasses of the general functional blocks identified in the general model. These subclasses can add detail (which are also restrictions or constraints on the class) which is specific to the context of NFV. The subclasses of the virtual hosting functional block relevant to the Sonata model are shown Figure 2.12.

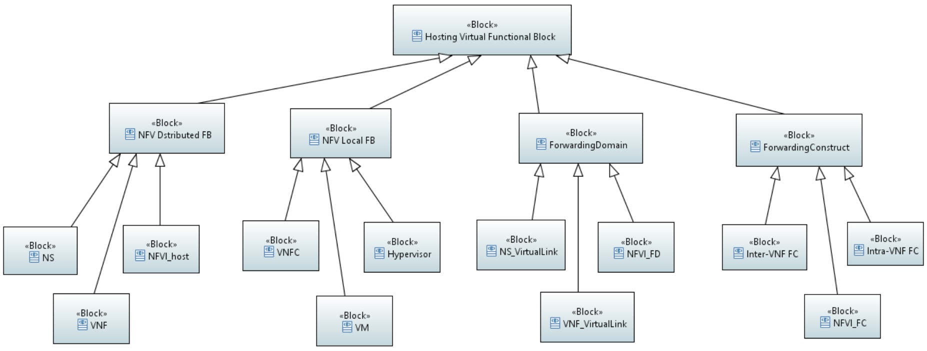

Figure 2.12: NFV Virtual Functional Blocks

In NFV, there are three broad classes of entity, with different properties in terms of distribution and processing/storage:

- general distributed functional blocks which contain both process/storage functionality as well as transport network functionality
- local functional blocks which are confined to a single local processing/storage node
- network transport functional blocks which do not contain general processing/storage functionality but are distributed.

We can define specific subclasses of these broad subclasses:

- examples of "generalized distributed functions" are network services, VNFs, and NFVI hosts (Network Services, Virtual Network Functions and Network Function Virtualization Infrastructure)
- examples of "local processing/storage functions" are VNFCs, VMs, and Hypervisors (Virtual Network Function Component and Virtual Machines)
- examples of "network transport functions" are virtual links between network service, virtual links between VNFCs within a VNF, virtual networks, and other forwarding domains and forwarding constructs within the NFVI

We also identify subclasses of the physical hosting functional block: a server, storage array, forwarding device, and transport medium (for example optical fibre, CAT5/5e/6 cable, etc). The latter two are also forwarding domains, as shown in Figure 2.13.

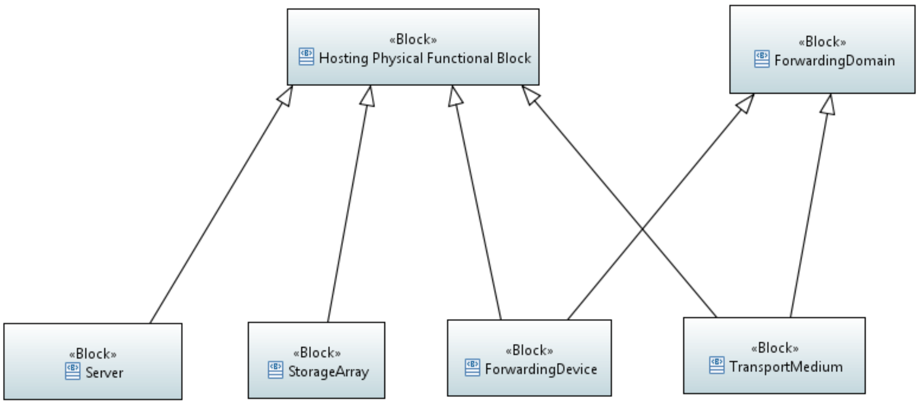

Figure 2.13: NFV Physical Functional Blocks

2.3.4.2 NFV Hosting Relationships

NFV Hosting relationships specialize the generic hosting relationship of a hosting virtual functional block described above.

Within the compute environment the composition of a server and storage can host a hypervisor. Likewise a hypervisor can host a number of virtual machines and in turn these virtual machines can each host VNFCs. These explicit hosting relationships are shown in Figure 2.14 below. While these are the currently defined relationships within NFV, the general model is readily capable of extension to new developments. For example, a virtual machine could be used to host a container environment such as LXC or Docker and a container could then be used to host a VNFC (assuming the terminology of VNFC is extended in this way as well). The likely extension to include containers illustrates the power of the generic recursive hosting relationship.

We can note that hosting in compute is relatively straight forward as the hosting is generally local. The host is normally a single, clearly identifiable hosting functional block. For example, a virtual machine is normally clearly hosted by one single, clearly identifiable hypervisor. This does not need to be the case and the model does not in any way depend on it.

A network, on the other hand, by its very nature, is distributed. We should therefore expect that hosting within the network side needs to reflect this distributed nature and this is indeed the case with the G.800 functional model. The forwarding domain and the forwarding construct are both distributed and the hosting of a forwarding construct by a forwarding domain reflects this. In practice, this often means that the hosting may need to make reference to the composition of the host. For example, with connectionless networks such as IP and Ethernet, the connections formed by TCP or other tunnelling protocols work on an end-to-end principle and the formation of the connection does not need to refer to how the Internet is constructed. The same is not true of connection oriented networks. With connection oriented networks, the formation of a connection does need to involve the composition of the network as forwarding entries specific to the connection need to be added to component forwarding domains, for example switches, of the overall network.

The fundamental NFV hosting relationships on the network side are shown in Figure 2.15 below.

The overall NFV infrastructure (NFVI) is like a network in that it is distributed and the hosting of VNFs and network services often needs to have visibility of the composition of the NFVI. As is the case in the generic model, as a matter of convenience, we show the hosting relationship of the distributed VNFs and network services so that a single NFVI functional block hosts a complete hosted function. The hosting of VNFs and network services by the NFVI is shown in Figure 2.16 below.

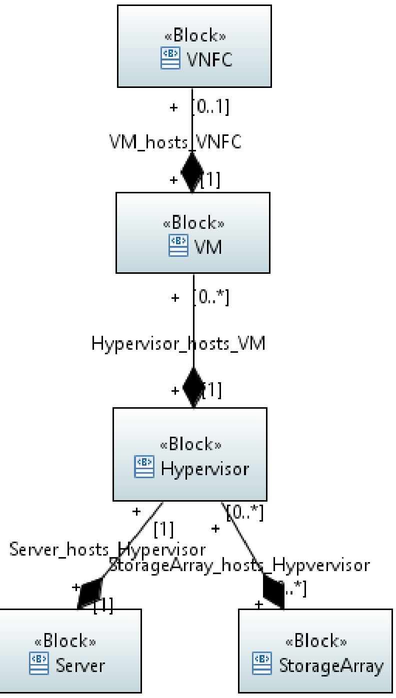

Figure 2.14: Hosting relationships between NFV functional blocks (1)

2.3.4.3 NFV Composition Relationships

Within NFV, composition falls in to two main categories. First there is composition which results in network services and second is composition within the NFVI.

The first of these, the composition of network services, is the composition which the obvious focus of the Sonata orchestration and the primary role of the orchestrator is to form these compositions. The functional model enables the composition to be described and results in the information elements in the information model which can hold the description. The functional model allows for the arbitrary binding of virtual functional blocks, however, in the case of NFV, there are specific restrictions. NFV has decided that the composition of network service should occur in two stages. The first stage is the composing of VNFs from constituent VNFCs and joining the VNFCs together with Intra-VNF Virtual Links (I-VLs). The VNF is therefore a composition of the VNFCs and I-VLs which are bound in a specific topology. The information model therefore needs information elements which holds the list of VNFCs, the I-VLs, and the topology into which they are bound.

Next, the VNFs are composed into network service. This is a conceptually identical composition to the composition of VNFs from VNFCs. The constituent VNFs are joined with Inter-VNF Virtual links (E-VLs) and the network service is therefore a composition of VNFs and E-VLs which are bound in a specific topology. The information model needs information elements which can hold the list of VNFs, the E-VLs, and the topology into which they are bound.

These composition relationships are shown in Figure 2.17 below.

The second category of composition in NFV is the composition of the NFVI. The NFVI is made up from physical hosting functions such as servers, storage arrays, switches/routers, and transmission media such as long distance optical fibre and local patch cables. These are physically composed together by the process of installing the equipment and physically connecting them together with the transmission media. Importantly for Sonata, Sonata assumes ***all*** physical composition has

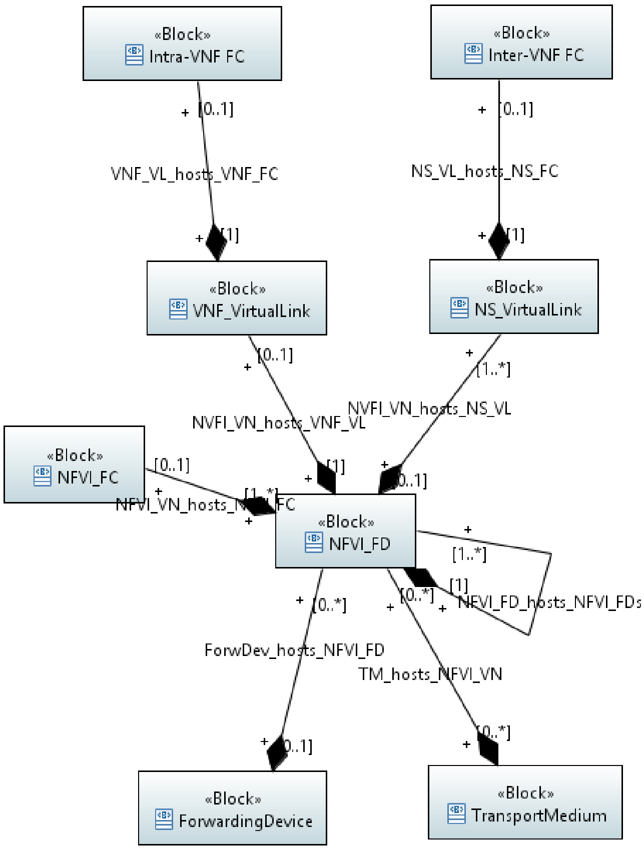

Figure 2.15: Hosting relationships between NFV functional blocks (2)

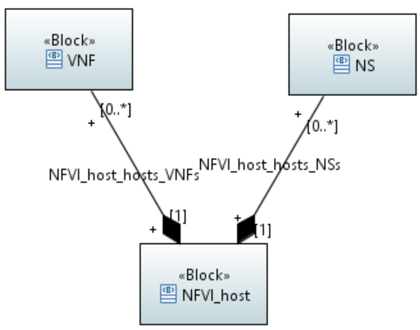

Figure 2.16: Hosting relationships between NFV functional blocks (3)

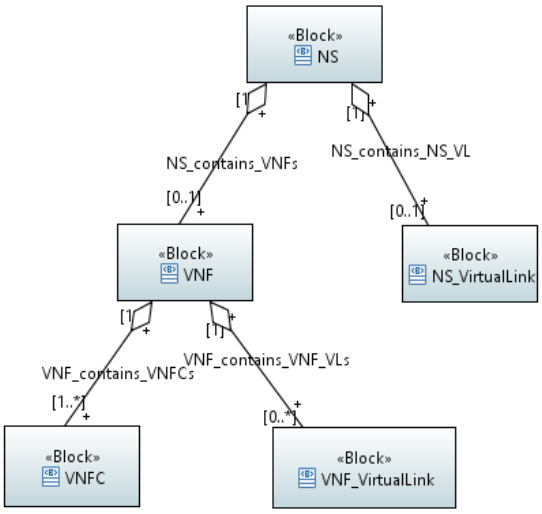

Figure 2.17: Composition relationships between NFV functional blocks to create Network Services

already taken place before any orchestration process commences. Orchestration must apply to a fully physically interconnected NFVI.

Normally the NFVI has already gone through levels of virtualization so that the NFVI appears to the VNFs and network services that are hosted on it as a distributed hosting infrastructure of virtual machines and virtual networks which are a specific type of forwarding domain. These may be already bound together within the NFVI but they could also be dynamically created by configuring their underlying hosts. In addition they could be dynamically bound into a required NFVI topology. The creation of VMs and VMs and the binding of them together into a topology is ***always*** achieved by configuring the underlying hosts. This functional model of the NFVI allows a very flexible range of prior allocation of the NFVI resources, importantly, including the creation of slices of the NFVI.

This gives the other primary information elements needed in the information model, that is the mapping of a virtual functional block to its host virtual functional block as well as the other way around, the mapping of a host virtual functional block to the virtual functional blocks that it is hosting. This host/client mapping information is central both the construction of network services and VNFs as well as tracing faults and root cause analysis.

The NFV composition relationships are shown in Figure 2.17 below.

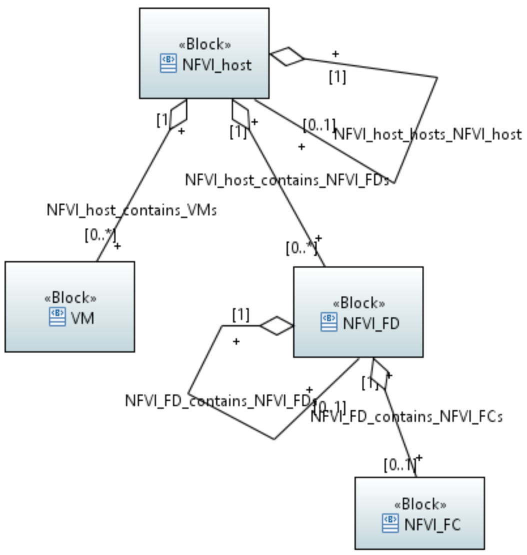

Figure 2.18: Composition relationships between NFV functional blocks within the NFVI

3 Catalogues, Repositories, and Packages

The SONATA system uses a variety of catalogues and repositories to store data and information regarding artefacts such as virtual network function descriptors, virtual machine images, and network service descriptors. That data is usually provided by a SONATA service package that contains all relevant information and data regarding a specific network service. Thus, in the following we elaborate briefly on service packages, catalogues, and repositories.

3.1 Catalogues and Repositories

In SONATA catalogues and repositories are used to store information, which is contained in the service packages, that is used by the service platform in order to manage and operate services. To this end, *catalogues* hold data about which services the system can instantiate, i.e. usually statically defined information provided by the developers, while *repositories* retain data about those instances, i.e. dynamic information about the instantiated - running - services and components.

Catalogues can be present at various parts of the SONATA system. Obviously, the service platform has to maintain catalogues that hold the data to actually operate the platform and run service. Moreover, the SDK maintains private catalogues per developer or per project to store data of reused or newly developed network service locally. This data can be packaged and shipped to a service platform. Finally, we foresee public catalogues that store artefacts developed and maintained by third-party developers on arbitrary platforms accessible to service developers and service platform operators. We point out that, although the different flavours of catalogues might have very different implementations, their interfaces should be unified and standardized. Thus, the usage of the different catalogues can become transparent to service developers, platform operators, and platform users. The API, however, needs to be further detailed by work packages WP3 and WP4.

According to ETSI NFV MANO document [29] the following types of catalogues and repositories are defined:

Network Service Catalogue This catalogue contains all the on-boarded network services, supporting the creation and management of the NS deployment templates namely:

- Network Service Descriptor (NSD) - information on the Network Service composition (referencing VNFs, VL and the forwarding graph)
- VNF Forwarding Graph Descriptor (VNFFGD) - information on the chaining of the VNFs composed in the NS, referencing virtual link elements
- Virtual Link Descriptor (VLD) - information on the virtual links and the connection points participating in each Virtual link

Virtual Network Function Catalogue This catalogue includes the information of all the on-boarded VNF packages, supporting the creation and management of the VNF package (VNFD), software images (or reference to them), manifest files, and further metadata. The VNF catalogue is queried by both NFVO and VNFM when a particular a particular VNF is required.

Virtual Network Function Instances Repository This repository includes information of all VNF instances and network service instances. For each VNF a VNF record is kept. Similarly for each network service a network service record is also kept. The instance information (records) contained in the repository is constantly updated through the network service(and VNF) lifecycle management operations. This allows visibility in the maintenance and management of the entwork service as a whole.

Virtual Network Function Instances Resource Repository This repository holds information about the available/reserved/allocated NFVI resources as abstracted by each VIM operating in each NFVI-PoP domain; thus supporting NFVO Resource Orchestration role. The repository provides the information for sufficient tracking of the reserved/allocated NFVI resources for a particular NS/VNFs combination.

In addition to the above SONATA will defined and implement additional catalogues and their "instantiated" counterparts hosted in the service platform as depicted in Figure 5.7. These components will be meant to host the information related to service specific managers and function specific managers.

SSM/FSM Catalogue This catalogue will be used to hold the information related to the SSM/FSMs are extracted from the package by the Gatekeeper. The information stored will contain location of the SSM/FSM container and other metadata related to its deployment and operation.

SSM/FSM Repository This repository will retain information on the instantiated SSM/FSMs updated according to the SONATA service platform policies. The information will allow monitoring of the instances and also access information retrieval by other components of SONATA platform.

3.2 Service Packages

The information and data stored in the catalogues is extracted from service packages. In the field of computer science, packages are distributions of software and data in archive files. The concept is well known and widely adapted as it provides a lot of advantages. Programming languages, like Java, use packages, say JAR files, to separate the functionality of a program into independent, interchangeable modules, enrich them with meta-data, and ship that as individual components that can be easily reused. Likewise operating system distributions, such as Ubuntu and Gentoo, use packages to handle computer programs in a consistent manner. To this end, package mangers allow for creating, installing, upgrading, configuring, and removing of packages, and therefore support important DevOps operations. Usually, these package managers work closely with software repositories and catalogues to ease the retrieval of new packages and updates. Furthermore, current cloud computing environments, such as OpenStack, support packages to manage virtual machines and services.

To leverage the benefits of packages and a consistent package management, current cloud computing environments adopted the concept as well. Evidently, this allows for a simplified life-cycle management and a simplified migration not only of virtual machines, but complete service that might comprise several virtual instances. The TOSCA Cloud Service Archive (CSAR) [22] for instance, is a container file containing multiple files organized in several subdirectories. Artefacts, such as configuration files, software code, and images, in a CSAR may be sealed. That is, all artefact referred to in such a CSAR are actually contained in the CSAR. A CSAR or a selective artefact within a CSAR may be signed. When signing an artefact of a CSAR the digest of this

artefact as well as the public key of the entity signing the artefact is included in the CSAR together with a corresponding certificate. Likewise, the OpenStack Murano project [41] provides an API which allows to compose and deploy composite environments on the application abstraction level and then manage their life-cycle, so the Murano package contains the environment that allows to deploying a cloud application. It is compressed with .zip, including typically the folders: classes, resources, user interface (UI), manifest, and images.lst. Moreover, Murano packages are TOSCA compatible. Also closely related is JuJu [53]. Using a package-like structure called Charms, which is a collection of YAML configuration and script files, JuJu allows software to be quickly deployed, integrated and scaled on a wide choice of cloud services or servers.

Consequently, SONATA also aims at using packages, named service packages, for the very same purposes and to achieve the very same benefits as described above. To this end, service packages should encapsulate all the components, such as the network service descriptor, references to the virtual network function descriptors including scaling policies descriptions, management scripts, and all the software images or reference to software images, needed to run a given service. In SONATA there will be two types of packages, namely *fat packages* and *thin packages*. Fat packages also contain the images of virtual deployment units, such as virtual machines or Docker containers. Contrary, thin packages contain only references to the images that are already stored in a catalogue, either in the SDK workspace or the service platform. However, since packages have to be transferred to the service platform, and fat packages can get quite big, it is preferable to work with thin packages.

In the following we outline a first draft of the service package content as used by SONATA. The structure of the package basically reflects the structure of the workspace of the SDK as outlined in Section 4.3. However, we want to point out that a complete and coherent information model, which also affects the package content and structure, is still missing and left to further discussions in WP3 and WP4.

- Package-specific meta-data descriptors
- Dependency descriptions to other services
- Incorporated Virtual Network Function Descriptors
- The Network Service Descriptor
- References to the VNF images
- Potentially the VNF images as such
- References to the Service Specific Managers
- Potentially the Service Specific Manger images as such
- Digest of signed content
- The public key or certificate to verify the signed content

Service packages are created by the SONATA software development kit and used by SONATA service platform. To this end, the SDK supports developers by creating a coherent package structure and initial content in a automated or guided way. Moreover, after adapting the content, the SDK allows for automated checks and validation of the package. Finally, the SDK creates the archive file and helps to on-board the package to a service platform or to store it in a (public) catalogue.

4 Software Development Kit

In general, a Software Development Kit (SDK) is a set of tools that allows for the creation of applications and supports developers in implementing, packaging, and deploying the software. SONATA aims at providing a set of tools that helps network service developers to easily implement and deploy new network services on the SONATA platform. To this end, the SONATA SDK comprises

- Editors for generating network function and network service descriptions
- Model checkers and validators for service patterns
- Support of packaging tools and easy catalogue access
- Deployment to SONATA test and production platforms
- Emulators for executing trial runs of services
- A variety of proofing, debugging, and monitoring
- Support for DevOps operations of network services

In order to meet all these features we will reuse existing tools and software as much as possible. For the editors, we rely on exiting editors, such as Sublime Text [47], Eclipse YAML Editor [52], and JSON Mate [9], that can handle YAML [1] and JSON [13] files. Moreover, there are plenty of plugins and libraries that can be integrated into software projects [1]. We expect SONATA to work with TOSCA or TOSCA-like templates. Thus, the SDK will provide editing, verification, proofing, and debugging tools to create and verify the SONATA template files. This might be similar to OpenStacks Heat UI that supports and simplifies the creation of Heat templates in the Web-based Horizon dashboard. Also the web-based GUI used by JuJu [53] for deploying services is open-source and can serve as inspiration. Bearing the more complex network service descriptions in mind, however, the SONATA tools have to provide innovative features in order to provide additional value to the developers.

To further support network service developers, the SONATA SDK connects and interacts directly with the SONATA platform. Thus, the SDK can upload data, save service function packages to the SONATA platform and store it in the catalogues. Moreover, it can retrieve data, such as catalogue information, debugging information, and monitoring data. A variety of integrated managing, debugging, and monitoring tools that are tightly integrated with the service platform allow for remote tests directly on a service platform instance itself. This also supports the DevOps operations, like on the fly updates, as envisaged by SONATA. Likewise, emulators that mimic the functionality of a service platform can be used to test and debug network functions locally, say on the developer's computer, without the need of a fully-fledged service platform.

In the following, we provide a brief description on the service development workflow as envisaged for a network service developer. Based on that workflow we identify and explain the core functionalities and components that are offered by the SDK. Like for the SONATA service platform we assume the components of the SDK to be loosely coupled or even stand-alone. From an implementation point of view, however, it would be beneficial if a lot of code can be shared between

the different components, potentially including the service platform. As throughout the rest of the document, we want to point out that the given architecture represents an early stage and might be revised and improved in an agile way whenever necessary.

4.1 Development Workflow

The SONATA SDK supports the development process of VNFs, SSMs and Services by providing tools for managing the project workspace, packaging, providing access to catalogues, testing and deploying them on the SONATA platform. This section goes into more depth on the involved actors, components and the relationships between them.

As shown in Figure 4.1, the global development and deployment workflow is a continuous, iterative process. Any development is initiated by creating a project workspace where VNFs, SSMs, FSMs and services might be imported from a range of catalogues. Service graph creation tools form the core of the development functionality and will result into a number of files, which may be packaged by the SONATA packaging functionality. Once packaged, tools may be pushed to a Service Platform (or emulator) where it may be deployed. Once deployed, debugging and troubleshooting tools may assist the developer in identifying bugs, triggering re-development, leading to a second iteration of the described development and deployment process.

It has been explained in Chapter 2 that the modular SONATA architecture has the provision for service- and function-specific managers. SSMs and FSMs can influence the Service and VNF Lifecycle Management operations. The SSMs, act on the Service Lifecycle, taking into account all different VNFs and their connections, which are part of the service. FSMs, which act on the lower level of a distinct VNF lifecycle, can however also be of particular interest to a developer. Apart from the default VNF lifecycle functionalities in the platform, the FSMs can, for example, provide a customized way of scaling a VNF vertically, by adding extra resources such as CPU and memory, or horizontally, by cloning extra instances.

4.1.1 Actors and platforms

As extension to the actors described in Section 2.1, we list here the actors relevant in an SDK context:

- **Users/client**: person who consumes communication and cloud services from the Service Platform. Users can be residential or enterprise end users, but also can be other service platforms.
- **Developer**: person who designs, develops, debugs and packages Network Functions, SSMs and Network Services. The SONATA SDK shall support the development directly or indirectly interacting with the Service Platform.
- **SDK**: platform which provides a toolkit to the the developer in developing, debugging and packaging Network Functions, SSMs and Network Services.
- **Service Platform**: offers services to users subject to specific SLAs. The SP make direct use of the SONATA Service Platform using the (potentially virtualized) resources of one or more infrastructure platforms. Service platforms could be in the role of a user with respect to other (recursive) service platforms.

Optionally, an **emulator** platform will be investigated to prototype and test SONATA network services locally on a developer's laptop. It can be considered as a (light-weight) instance of the

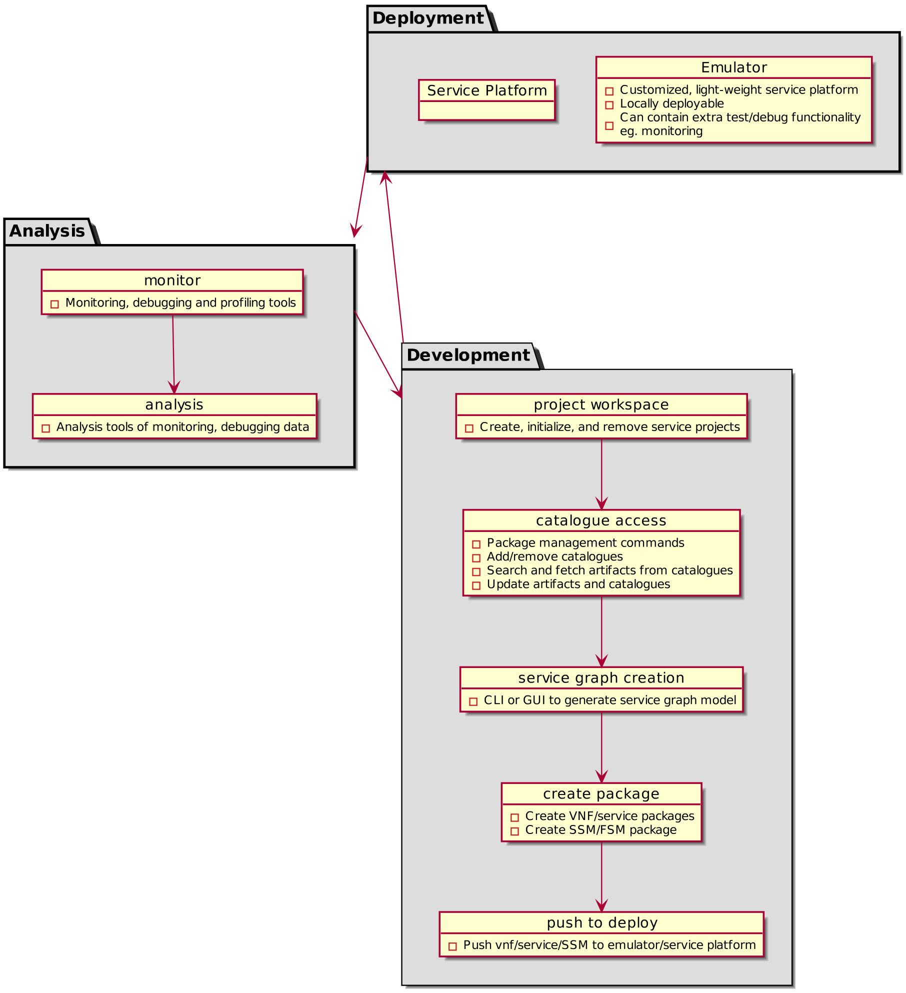

Figure 4.1: SONATA service development and deployment workflow

Service Platform. In the optimal case, the emulator's **infrastructure** behaves like any another VIM and is controlled by a SONATA orchestrator instance that is also executed locally. Having this in place, the emulation can also be used to test automatic scaling/placement strategies in a local environment.

4.1.2 Development of VNFs

VNFs form the atomic components of SONATA services. The development of VNFs might consist of the following parts:

- Development of **VNF source code** providing either data- or control packet-processing or packet generating logic.
- Development of **FSM** modules, controlling any VNF-specific lifecycle events (related to eg. deployment, scaling, shutdown). If no custom event handlers are needed, the VNF could also rely on the default VNF Lifecycle entities in the SONATA platform.
- Writing of **linking instructions** to enable source code compilation in combination with required libraries into executables.
- Production of **performance optimization code** dependent on available platform functionality, either in hardware (e.g., through the use of hardware acceleration facilities), or via software performance enhancements (e.g., using DPDK or netmap improvements to the network stack of operating systems)
- Inclusion of **references to VNFs in catalogues** which might be extended into a new VNF.
- Providing **scripting logic** in order combine (sub-)VNF components into a single VNF
- **Development, execution and packaging of tests** ensuring correct operation of the VNF from the developer's perspective
- Creation of a **VNF package**, which might be deployed on either on an emulator part of the SDK, the SONATA Service Platform, or part of a catalogue. This includes the creation of appropriate manifest files.

The VNF development process is visualized in Figure 4.2.

4.1.3 Development of SSMs and FSMs

In the SONATA platform, the Service and VNF lifecycle entities provide a default functionality handling lifecycle events. FSMs and SSMs provide additional flexibility, allowing a developer to customize any lifecycle event handler, and program tailor-made actions. The development of SSMs and FSMs involves the following tasks:

- Development of **SSM/FSM source code** providing either scaling- or placement logic for execution or decision purposes.
- **Calling appropriate of API calls** within the source code for interacting with the SONATA Service Platform (or SDK emulator) for receiving monitoring-related information for scaling purposes and/or (virtualized) infrastructure related information for placement. These APIs are supposed to be not particularly limited to a particular programming language or environment (e.g., http-based REST API or interacting via the Service Platform messaging system).

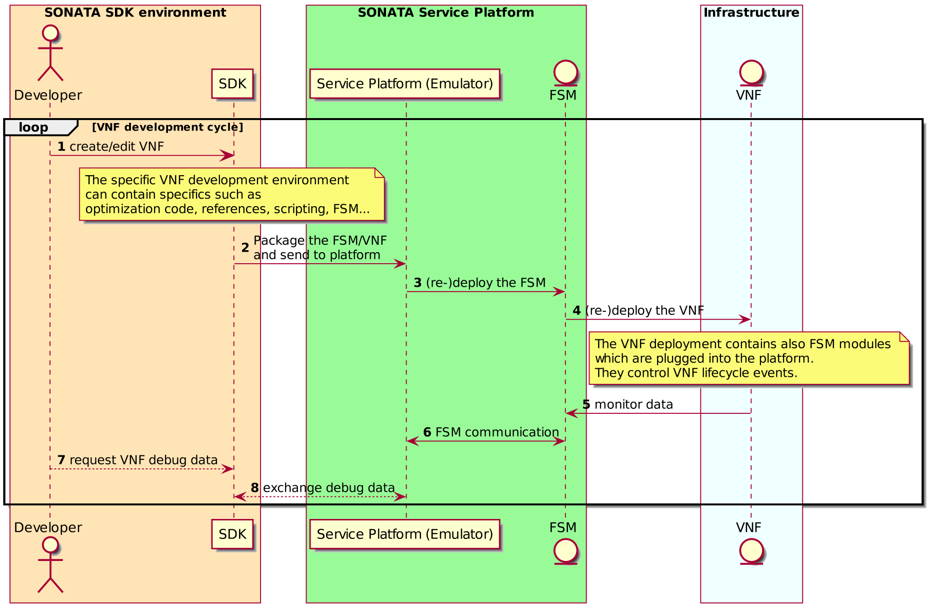

Figure 4.2: VNF development cycle

- **Development, execution and packaging of tests** ensuring correct operation of the SSM/FSM from the developer's perspective
- Creation of a **SSM/FSM package**, which might be deployed on either on an emulator part of the SDK, or on the SONATA Service Platform, or part of a catalogue. This includes the creation of appropriate manifest files.

The development of different SSM/FSM types is illustrated in Figure 4.3. The figure focuses on SSMs, as actions on the Service Lifecycle will probably offer the most advantages for a service developer and have the most complexity. The developer is however to design both FSMs as SSMs in the SONATA SDK. The appropriate editor for editing SSM/FSM source code is available in the SDK. With the appropriate access rights, the developer can also get existing SSMs/FSMs from the platform catalogue or repositories. The high-level functionality of these FSMs/SSMs, and how they can determine VNF and Service Lifecycle Decisions and Executions, is further explained in Section 5.2.3.6. Here, we detail the implementation of these functions in the SDK. By specifying a fixed input and output format, this illustrates how an SDK environment could interact with the customized code of an SSM/FSM. We envision that the service model will contain a graph model describing the VNFs in the service and how they are connected. We distinguish three main categories for the SSM/FSM functionality and highlight their specific input/output parameters.

- **Placement logic**
  - Input: new service graph, which can come from scaling logic, and infrastructure view.
  - Output: mapped service graph.
  - Can take scaling patterns into consideration to adapt the service graph.
- **Scaling logic**
  - Input: deployed service graph and monitoring data including load statistics.

- Output: scaled service graph, including more/less or scaled-up VNFs, connected to each other in a modified topology.
- Can take profiling data into consideration to predict the performance of the scaled graph.
- Can take scaling patterns from the catalogue or create its own new pattern to scale.

- **Lifecycle management logic**
  - Input: monitoring data or trigger coming from another actor such as the service user, platform or infrastructure operator.
  - Output: specific lifecycle actions such as bring up/scaling/tear-down of a VNF/service.
  - The correct execution of the lifecycle action could be tested in the SDK first, say in an emulator environment. The exact form of this action and how it will be communicated in the SONATA platform will be further investigated in work package WP4.

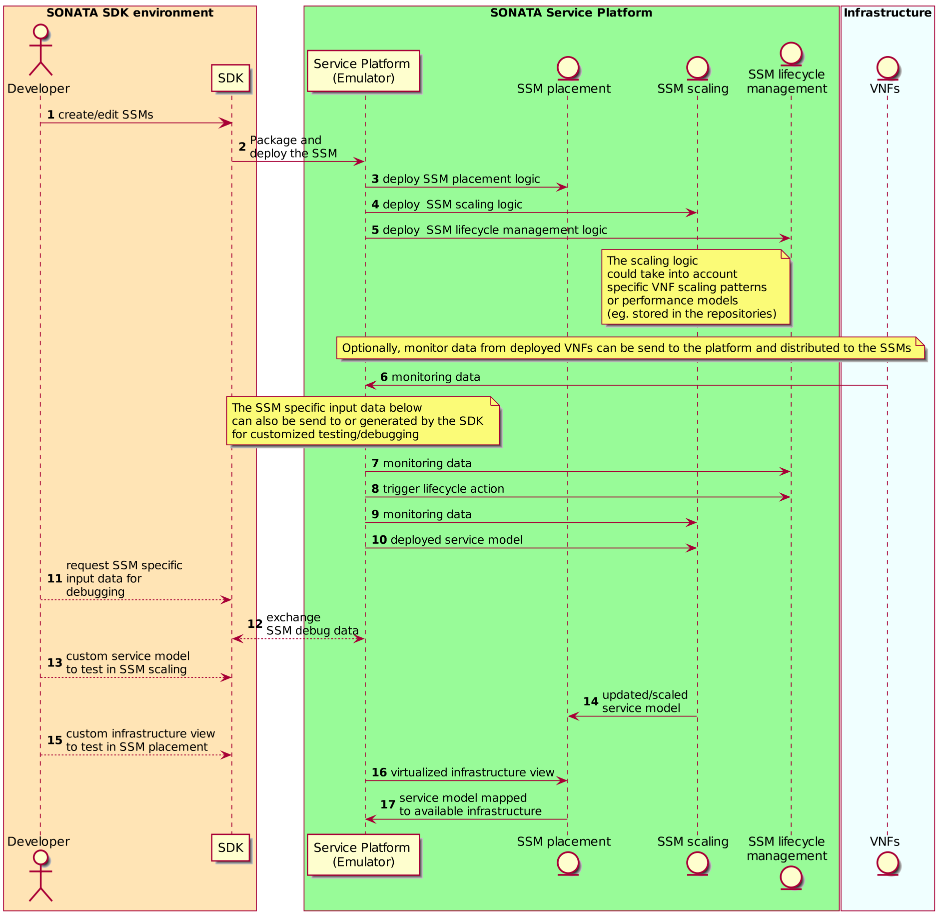

Figure 4.3: SSM development cycle

Inside the SDK, SSM scaling or placement logic could also be verified by checking the output service model with a visualization tool, eliminating the need to really deploy the SSM via the emulator or service platform. These specific SSMs would typically work using the service model as a graph model, which makes the input/output easier to show using a graph visualization which could be incorporated into the SDK. Another SDK feature allowing specific SSM debugging is

the creation of specific service models and infrastructure views. This would allow the service developer to test and debug the scaling and placement logic by tweaking specific input values for the SSMs. As can be seen in Figure 4.3, the developer can send these customized or dummy service models, containing a specific service or infrastructure graph to test. This **SDK mode** is specific characteristic of the SONATA platform, allowing a developer to test SSM code offline in a reliable way. In **Production mode**, when the SSMs are running and the service is really deployed, it is up to the Infrastructure Provider to provide an adequate infrastructure view to the platform and the SSM. With the necessary freedom to choose which nodes are made available to the platform. An additional requirement is that this information needs to pass through the VIM and get translated by the Infrastructure adapter in the SONATA platform . Similarly, the service (platform) provider must expose the necessary monitoring data to the SSMs.

4.1.4 Development of a Service

- Providing **references to used VNFs/SSMs**, either as a reference in a catalogue or by including the full VNF/SSMs package within the service package
- Providing **topological dependencies** or interconnection of involved VNFs, e.g. in the form of a graph or sequence of VNFs
- Providing **forwarding-level information** on top of the topological relationships between VNFs, enabling to guide the traffic steering process in between involved VNFs, e.g. by specifying packet forwarding rules associated to VNFs
- Description of the dependencies of the service and the individual NFs on the **underlying infrastructure**, in terms of resources and potentially in terms of performance
- **Development, execution and packaging of tests** ensuring correct operation of the service from the developer's perspective
- **Creation of a service package**, which might be deployed on either on an emulator part of the SDK, the SONATA Service Platform, or part of a catalogue. This includes the creation of appropriate manifest files.

The development of a service is visualized in Figure 4.4, showing the 3 different domains involved (SDK, Platform and Infrastructure).

4.1.5 Deploying a Service Package

When the developer considers the service package ready to be deployed, it will be submitted to the service platform.

A series of validations will be done in the service platform, before the service package is included in the service platform's catalogue and considered for being instantiated or reused in building other services. As detailed in Section 5.2.1, these validations are:

- syntactic validation of the expected package descriptor format;
- semantic validation of at least some of the simplest parameters;
- license availability and validity for all external dependencies, such as packages, libraries, and services, requiring one;

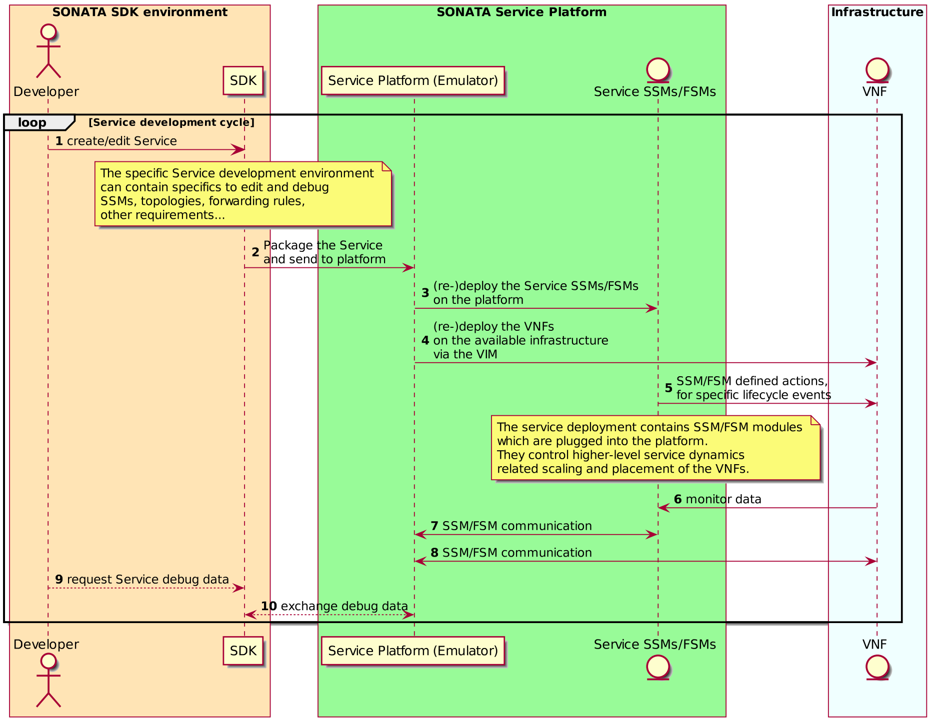

Figure 4.4: Service development cycle

- tests availability to prove the provided service works;
- tests successful execution, where specified tests are executed and its results checked for correctness.

Any inconsistency found in this validation must be returned to the developer, so that it can be corrected and resubmitted when found adequate.

There might be an option for just 'checking' the service package's validity, without storing in the catalogues. This might be useful for the early stages of specification.

4.1.6 Monitoring and Debugging a Service

Once a service is deployed and running on the service platform, the developer might want to perform service specific tests to verify the service is working correctly. To this end, the developer can use the SDK to connect to the service platform to trigger test and gather debugging information. Moreover, the developer can query continuous monitoring data regarding the service using the SDK.

4.1.7 Updating/migrating of a Service Package

Updating a service package means redeploying a new version of the Service Package. The Service Platform's behaviour depends both on:

- the kind of update: for certain kinds of updates, e.g., security related or severe errors, it might be advisable to redeploy or migrate the currently running instances of the service to be updated (i.e., interrupting the currently available instances);

- the kind of service that is being provided: for very strict service level agreements, the package may be updated, but the instances not restarted.

Therefore, the Service Package must be able to express these different kinds or updates, for the Service Platform to be able to take the most adequate behaviour when faced with a new version.

4.2 Components

Based on the development workflow presented above, we present the main components of the SONATA SDK in the following.

4.2.1 Editors

SONATA's SDK includes different editors for facilitating the development process of VNFs, services, and their requirements specified as specific management preferences. The SDK will support the integration of general-purpose editors as well as providing special-purpose editors for different development requirements of service developers.

The SDK will support the VNF development task, for example, by providing editors for functions based on Click or Vagrant. Depending on the selected specification models, VNFs and service descriptor schema, we foresee using simple text-based editors, e.g., for YAML or JSON to specify the requirements of services and the VNFs included in them. For assisting tasks like specifying the service topology, the SDK will also provide graphical editors. For example, using a graphical user interface, a developer can easily see and select existing VNFs in the catalogues and compose a service graph out of them. Developing specific management functionalities for VNFs and services will be supported through generic editors for general-purpose languages based on well-defined APIs.

The output from editors can be stored in the developer's workspace for further testing, debugging, modification, and eventually, creating service packages for actual deployment.

4.2.2 Packager

The packager is a piece of software that takes all the relevant information and data provided in the SONATA workspace to create a service package as described in Chapter 3. The final package can then be uploaded and deployed at the service platform using the catalogues connectors.

4.2.3 Catalogues Connector

As mentioned in Chapter 3, catalogues and repositories are used to store information regarding services, functions, plugins etc. From an SDK point of view, there are three possible types of catalogues and repositories, with different access properties at different locations in the SONATA architecture:

- within the developer's SDK, where the Developer keeps his/her private developments, maybe not in a final, publishable phase;
- within the Service Platform, where only validated services and instances are available to those Developers that are authorized to see, reuse or instantiate them;
- public catalogues, holding other kinds of assets that are public (i.e., whose assets are also available to other developers, even if they're outside the SONATA eco-system) and may be used in the implementation of new services.

The SDK should be able to:

- *search* and *pull from* these multiple catalogues;
- *push to* the SDK (private) and the service platform's catalogues and epositories.

To ease the implementation of such a mechanism, a single API maybe exposed from the service platform for all catalogues and repositories. This API will be further detailed through the work of the dedicated work packages WP3 and WP4.

4.2.4 Debugging and Profiling Tools

To support the development of new VNFs and network services, not only editors are of importance, but also tools that can test and monitor them. These tools enable a shorter development cycle, as they aim to quickly deploy and test a service component so the developer can verify if the performance and functionality is adequate. If this is combined in a SDK environment, the VNF or service developer has a toolset at his/her disposal to not only generate service packages, but to also test and modify them. This is required in a DevOps mindset, which aims at establishing a culture and environment where building, testing, and releasing software, can happen rapidly, frequently, and more reliably.

4.2.4.1 Profiling tools

Profiling tools enable the programmer to estimate resource and performance characteristics of a given VNF or service. They offer interesting functionalities to the developer like workload prediction or resource assessment.

An example architecture of such a tool in the context of SONATA is shown in Figure 4.5.

- Traffic Generator: generates a user definable workload. Packet generating tools such as dpdk's PktGen, iperf, netperf, SIPp, hping3 will be studied and possibly implemented in the SDK. Also traffic streams which have been monitored and stored in the SONATA platform, could be used as test stream in the SDK, like using tcpdump.
- Monitoring Agent: collects different metrics as defined in the test description. Monitored parameters include workload, KPIs and hardware resources. This data will be further processed. Various monitoring tools are described in Section 5.4, additionally other probes like eg. Open-vswitch can also be used to capture or generate certain packets.
- Profiling Process: this process will analyse the monitored data and calculates a performance model. Available mathematical libraries for this kind of task include: Python -statsmodels, for regression analysis or TensorFlow by Google to implement algorithms from the machine learning domain.
- Performance Model: this model describes a relation between the input workload, the KPIs and the used hardware resources that were monitored at the *device under test* during the test. It is the intention that this model can also predict KPI or hardware resource metrics, for workloads that have not been tested.

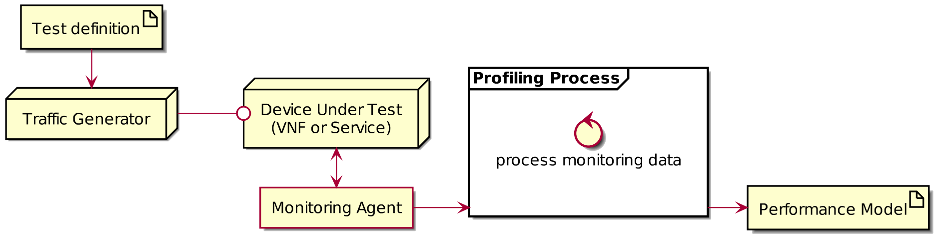

Figure 4.5: SDK Profiling tool

4.2.4.2 Debugging tools

Debugging tools help the programmer to find errors within the written service or VNF. Specific to the SONATA platform, the service developer can program his own SSMs, which include service specific placement or scaling logic. For example, if the service or VNF needs specific scaling logic, this part of the profiling functionality could be made available in the SDK, so the developer can use it in his own SSM. If the SDK provides an editor to develop an SSM, then also debugging tools will be of help. When looking at the typical input and output for this scaling logic, this will be a (partial) service graph, consisting of one or more VNFs. Possible ways of processing such a graph-like structure in the SSM code, include the Python – networkx library or the Neo4J graph database. In a debugger context they can be used to check certain characteristics of the service graph, like check if all nodes are connected, find shortest path. Also regarding the debugging of placement logic, a graph library seems the natural way of working. Eg. by using a visualization of the placement of a service on the infrastructure, a developer could quickly check if the placement algorithm is performing as expected.

Other possible SDK debugging features are these:

- **Proofing tools** enable to validate either syntactical or semantic properties of written service programs at programming time. This might involve parsing and/or model checking functionality. At programming time, syntax errors can be checked in the editor. At deployment time, the deployed VNFs can be checked for connectivity or configuration errors.
- **Runtime debugging tools** enable the programmer to perform fine-grained checks of the running service program in a controlled deployment environment. This usually involves the functionality to inspect state of the program or to execute parts of the program step by step. For example, specific monitoring could be done on the state of the running VNFs. In case of an error, a snapshot could be taken which can be further analysed in the SDK.

4.2.5 Monitoring and Data Analysis Tools

The SDK's service monitoring and data analysis tools interact closely with the SONATA service platform. The tools connect to the gatekeeper and make the service monitoring capabilities of the service platform, as described in Section 5.4, available to the service developer. Thus, they retrieve all wanted and available information that is gathered by the service platform. Service monitoring analysis tools will be capable of retrieving both, real-time alerting and access to historical data, for instance using a RESTful API. While based on its flexible and extensible design, it will collect data from different sources, such as Docker containers, VM instances, OpenFlow switches, and cloud environment, allowing developer to gain service insight, infrastructure owner to be timely informed about service malfunctions and end-user to check SLA violations.

4.3 Workspace

Workspace is a concept used by the SDK to hold development work. It's a structure of directories on the file system the SDK uses to store project's resources like, source code, log files, metadata, settings, and other information. The developer may have multiple workspaces, one for each project, containing specific settings, or one for multiple projects, sharing common settings. In SONATA a single workspace allows many projects with shared configurations, catalogues to reuse other projects and, available platforms for service deployment. With this structure in mind, SONATA workspace is composed by, (i) Projects, (ii) Configuration, (iii) Catalogues and, (iv) service platform operators. Figure 4.6 shows a high level overview of the SONATA workspace. All the names presented in this document are only suggestions and may be adapted over time.

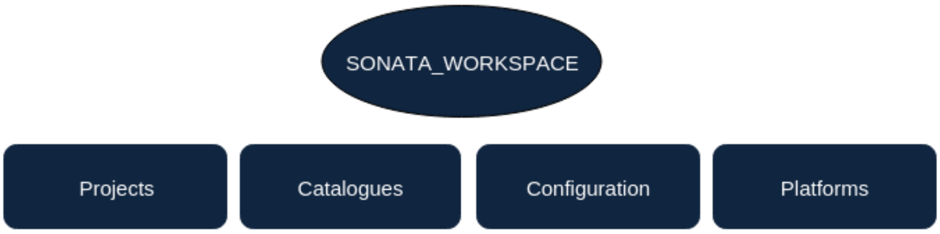

Figure 4.6: SONATA Workspace

4.3.1 Projects

SONATA workspaces aggregate all service development projects inside a single folder, named projects. Developers can easily locate all projects avoiding sparse locations, every project has it's own folder where all the sources, configurations and descriptors are located. Figure 4.7 introduces a complete insight of the project's structure inside the workspace with the expected directories and subdirectories.

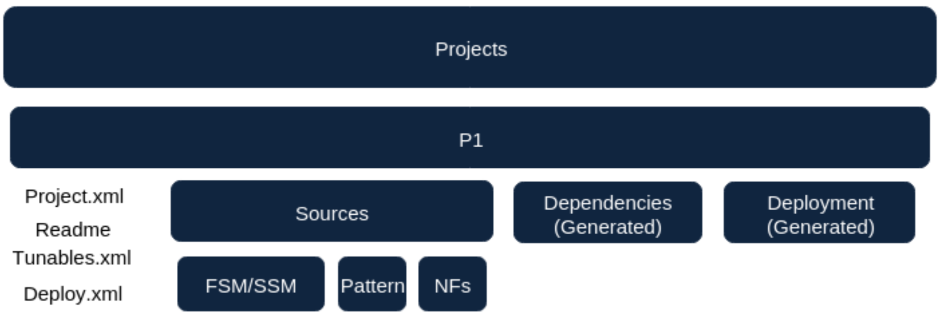

Figure 4.7: SONATA Workspace - Projects

Project configurations are divided in 3, mandatory files, (i) Project.xml, (ii) Readme and, (iii) Tunables.xml, regarding the project a Deploy.xml can be configured or not.

- **Project.xml** - This file contains the, (i) service name, the main identifier for the project, this defines how the project will be known across catalogues, (ii) version, the current project's version that will be deployed, (iii) description, a brief piece of text describing, e.g., the project's main goal, installation process etc., (iv) source description, with all the sources that

must be used in the project and, (v) project dependencies, providing dependency's source and version in order to construct the package correctly.

- **Readme** - Full service description, how to configure the service, Copyright, credits, changelog, and so on. By reading this file a developer, or service platform operator, should be able to understand, at least, the project's main goal.
- **Tunables.xml** - This file contains every variable that must be configured before deployment, this is only related to the local sources. The goal is to define values that must be configured by every developer that wants to use this service as a dependency. These variables are defined by (i) a name, that identifies the the variable, this value should be unique, (ii) type, the type of value that is accepted for this variable, like a number or a string, (iii) description, a brief piece of text describing, e.g., the variable's main goal, the implication of the accepted values, etc., and (iv) a default value, if applicable, which is the value that is assumed if no other provided. Every dependency has its own tunables and all must be gathered, and configured, on the deploy.xml.
- **Deploy.xml** - Every configuration related to the service deployment must be in this file, e.g., billing, credentials, or service platform operator configuration, tunables values, both sources and dependencies, are defined here. This is only needed when the project is intended to be deployed, as is, in a service platform operator.

4.3.1.1 Sources

All service specific development falls under a source folder. The source is divided in 4 distinct parts, (i) FSM, (ii) SSM, (iii) Patterns and, (iv) NFs.

- **SSM/FSM** - Instructions related to VNF lifecycle, as outlined in Section 4.1).
- **Patterns** - The patterns folder contain the files that describe how the service sources and dependencies are connected and interact with each other. Many files may describe smaller patterns and integrated all in an aggregated file. With this file it's possible for a developer, or a service platform operator, to identify the project's workflow and the interaction between its building blocks.
- **NFs** - Developed network functions.

4.3.1.2 Dependencies

This folder contains SDK generated content. Whenever a dependency is added, removed or updated, the content of this directory must be autonomously updated. For every dependency currently associated with the project, a folder must be created containing the dependency name, within the name, other with the version and inside this the Tunables.xml, Project.xml and a link to the artefact associated with the targeted dependency.

4.3.1.3 Deployment

As described in Section 4.1.5, deploying a service will impact the Workspace in terms of **runnable versions**: different service versions may be maintained for different service 'consumers, but always through a SONATA Catalogue. Private service versions, which residing in the developer's private catalogue, are not considered deployed.

The specific way the Workspace will be organized will be the subject of work under work package WP3.

4.3.2 Configuration

The configuration of the whole Workspace will have to take into account things like all the needed *credentials* to reuse existing services, to access the SONATA service platform and deploy the service, etc. Configurations might be *global*, i.e. shared between all projects of the workspace, say the developer's public key, or *specific to a project*. Figure 4.8 shows a possible structure for this directory, with reserved space for the SDK and credentials.

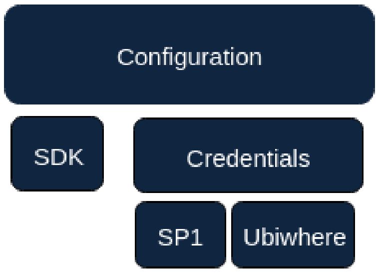

Figure 4.8: SONATA Workspace - Configuration

In scenarios where more than one SONATA service platform can receive the same service, the workspace must be able to support this configuration as well.

4.3.3 Catalogues

Catalogues will have their own share of the workspace, probably a folder, containing every service that was already referenced by the developer as dependency. Within this part of the Workspace, possibly multiple versions of each service may have their own space as well, containing the project file. For instance, suppose SONATA has a public catalogue with project *SONATA Project 1*, that project has two public versions *V1* and *V2*, the catalogues' workspace would have a structure similar to:

- a SONATA space, because it's the name of the public catalogue;
- a SONATA Project 1 space, because it's the dependency name;
- a space for each version.

Figure 4.9 illustrates a possible structure for the workspace' catalogue directory, with local and external catalogues' folder and specific versions.

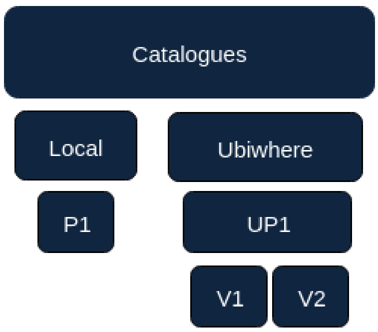

Figure 4.9: SONATA Workspace - Catalogue

The catalogue workspace has also a special folder containing a local catalogue, only available for the developer in the current workspace, that contains each project locally installed, and not yet made public. Projects under development are not available as dependency, only after local or public publishing, that can be used with other projects.

4.3.4 Platforms

The workspace has a specific location to store SONATA platform related information, as depicted in Figure 4.10. The project can only be deployed to known SONATA service platforms.

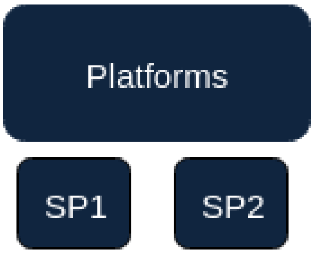

Figure 4.10: SONATA Workspace - Platforms

5 Service Platform

This chapter introduces SONATA's service platform that is able to manage complex network services throughout their entire lifecycle. To this end, the platform contains a highly flexible MANO framework that can be customized on a per-service and per-function level. First Section 5.1 introduces the high-level workflow of a network service developer interacting with the service platform. Second Section 5.2 introduces the platform's components. Next Section 5.3 describes how the platform manages the lifecycle of network services and VNFs. Then the monitoring system as well as the recursiveness of SONATA's service platform architecture are described in Section 5.4 and Section 5.5 respectively. After that Section 5.6 describes the slicing support of the platform. Finally, the usage of the platform for the DevOps is discussed in Section 5.7.

5.1 High-level Workflow

This subsection presents high-level workflow, for different scenarios, for SONATA service platform from the perspective of the entity interested in deploying a service on it. The scenarios include deployment/starting a new service, receive monitoring feedback for a deployed service, manually manipulate the scaling of the service, pause/suspend a deployed service, restore a suspended service back to production, and termination of a deployed service.

5.1.1 Upload/Start a service package

This scenario refers to the situation when the network service operator wants to deploy and start a particular network service for the first time. The uploading of the corresponding service package by the SDK serves as a prerequisite to this scenario. Prior to the deployment request, the network service operator has to authenticate itself with the service platform. On receiving the deployment request, the service platform fetches the respective service package and forwards the network service deployment request to the Infrastructure, as shown in Figure 5.1. After the successful deployment of the network service, the operator can request the service platform to start it.

5.1.2 Receive Monitoring Feedback

In light of efficient and incident free operation of a network service, the monitoring feedback plays a vital role. Moreover, a network service operator will require continuous monitoring data for its deployed network service in order to ensure SLAs it agreed with the end user. The network service operator can retrieve the monitoring information for a particular network service that it owns by requesting the service platform. As a first step, the service platform checks if the network service operator is authorized to retrieve monitoring data for that particular network service. If the authorization is valid, then the monitoring data is fetched and pushed to the network service operator, otherwise, the request is denied. It is worth mentioning that the service platform continuously updates itself with the required monitoring data for every network service it runs, as shown by the loop statement in Figure 5.2.

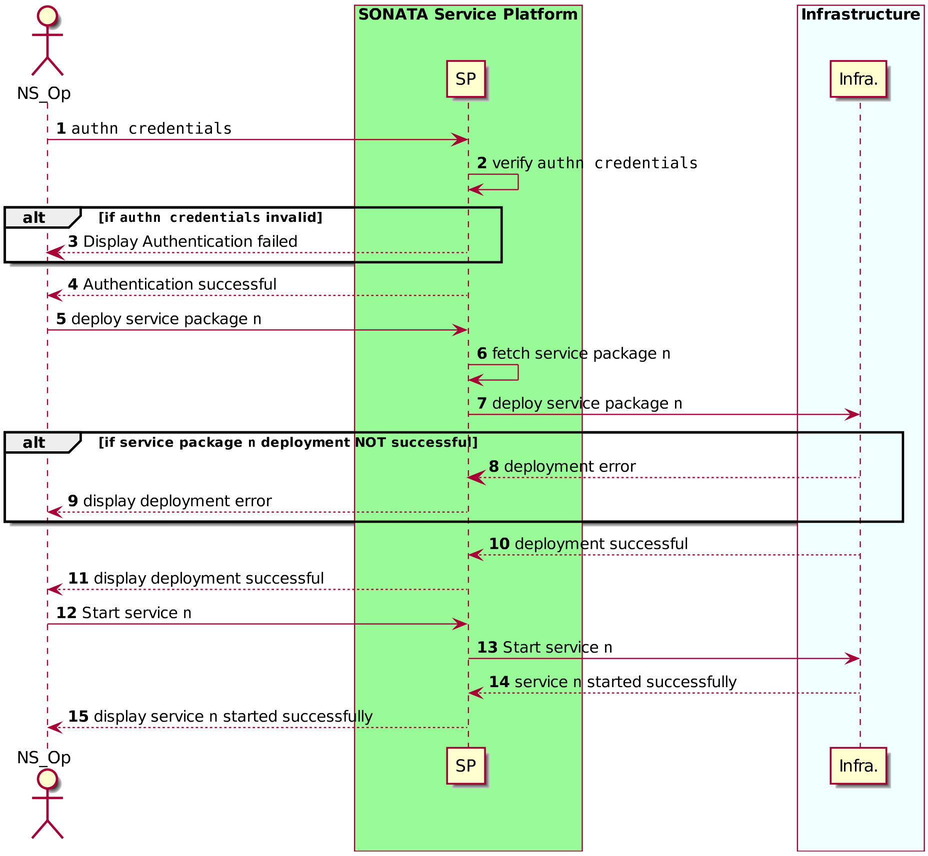

Figure 5.1: Upload/Start a service package on the service platform

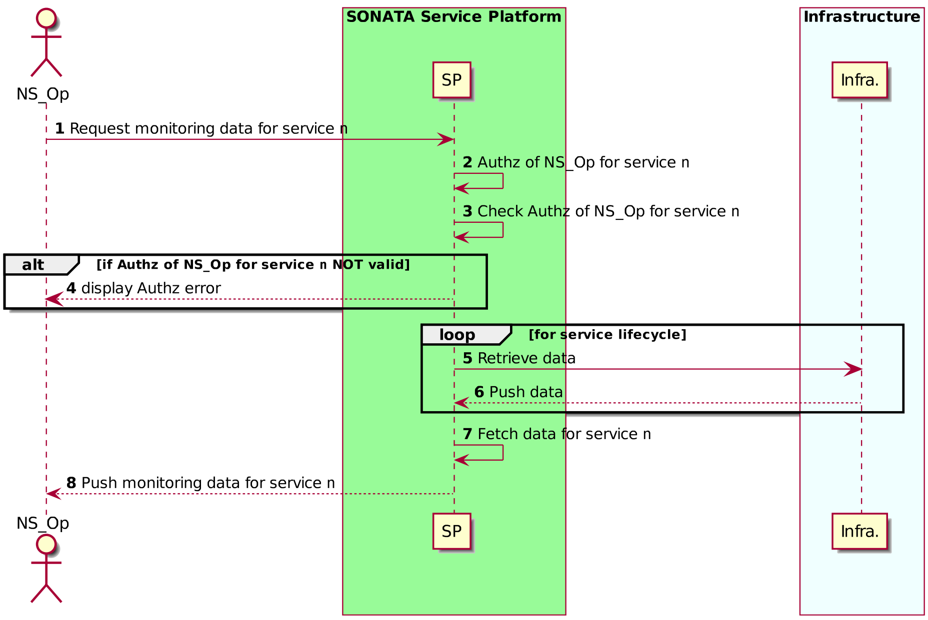

Figure 5.2: Receive monitoring feedback from a executed service

5.1.3 Manual service scaling

A network service operator may need to scale the network service manually, based on monitoring data analysis, to improve the network service. The network service operator can send the instructions for the manual scaling of the network service once the service platform verifies if the network service operator is authorize to do so. The service platform also validates the scaling instructions in light of the SLA and corresponding NSD and VNFDs. If the scaling instruction pass the validation, only then the scaling request is forwarded to the Infrastructure. The workflow for a manual service scaling is shown in Figure 5.3.

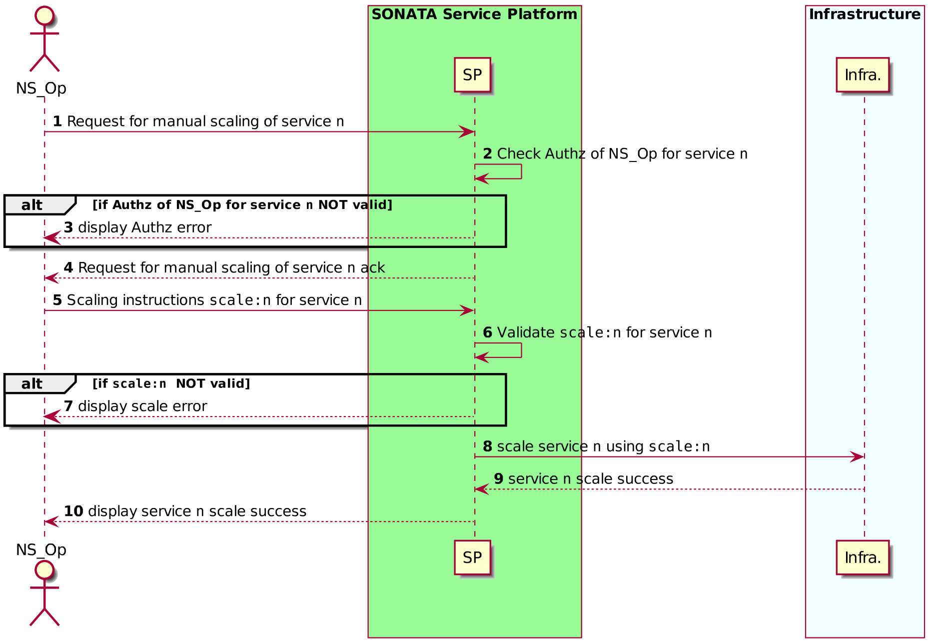

Figure 5.3: Receive monitoring feedback from a executed service

5.1.4 Pause a service

The network service operator may need to pause the network service , i.e., suspend it operationally, for maintenance or other purposes. For a network service suspension request, the service platform carries out two basic checks, firstly, if the network service is deployed or not, and secondly, if the network service operator is authorized to carry out this action. Based on the outcome of the checks, the service platform can forward the network service suspension request to the Infrastructure or deny the request. The workflow for a network service suspension is illustrated in Figure 5.4.

5.1.5 Restore a service to production

The workflow for restoring a network service is shown in Figure 5.5. In order to put a network service back in to operation, upon the request of the network service operator, the service platform checks if the request is valid and the network service operator is authorized to do so. If the checks are affirmative, the service platform forwards the request to the Infrastructure and informs the network service operator accordingly.

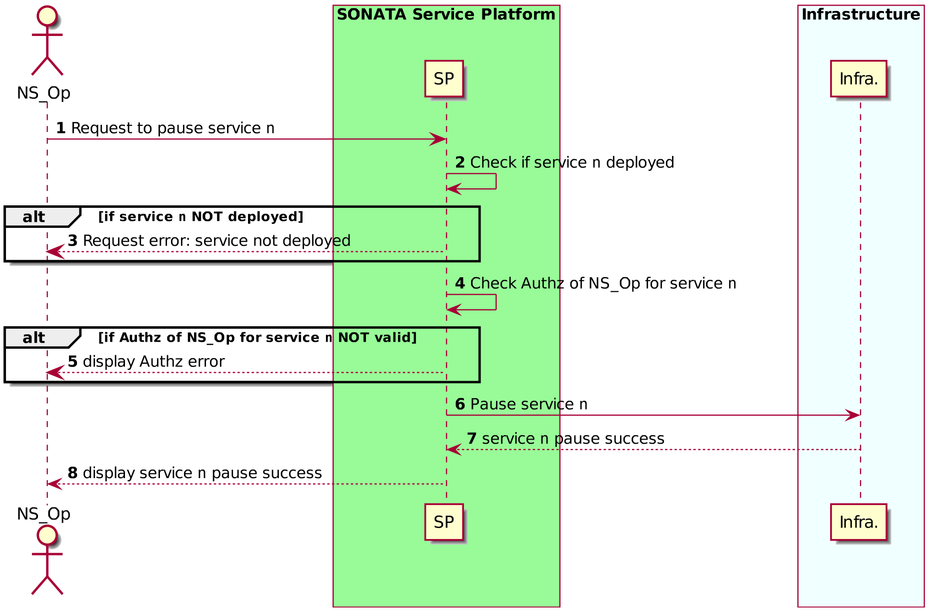

Figure 5.4: Pause/Suspend a service

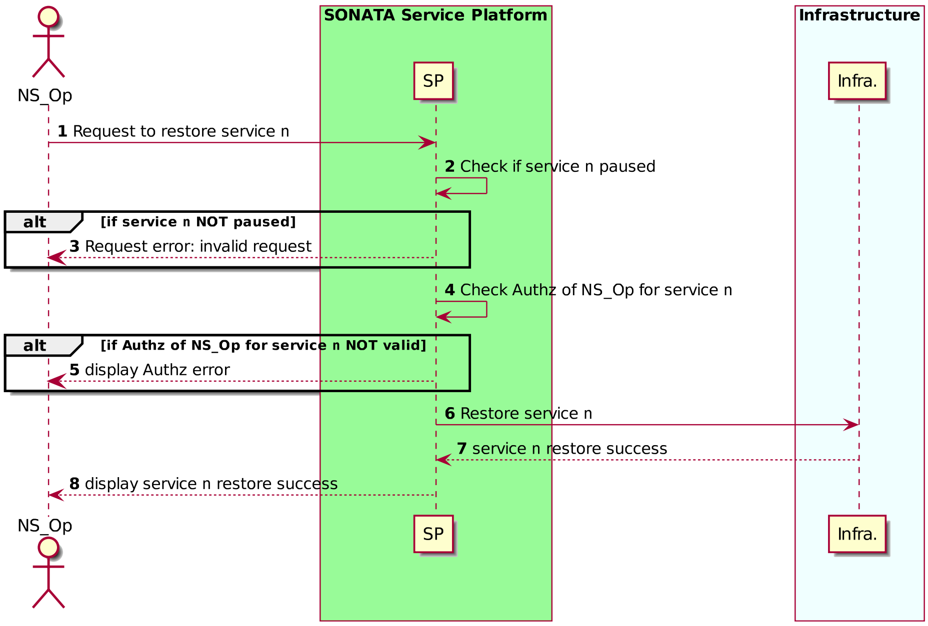

Figure 5.5: Restore a service to production

5.1.6 Terminate a service

The termination of a network service refers to shutting down the network service completely and release the assigned resources. Similar to Pause and Restore action workflow, the service platform verifies the status of the network service and authorization of the network service operator for carrying out this action. In addition, the service platform also updates its available resource database after the resources assigned to the terminated network service are freed. The workflow for termination of a network service is shown in Figure 5.6.

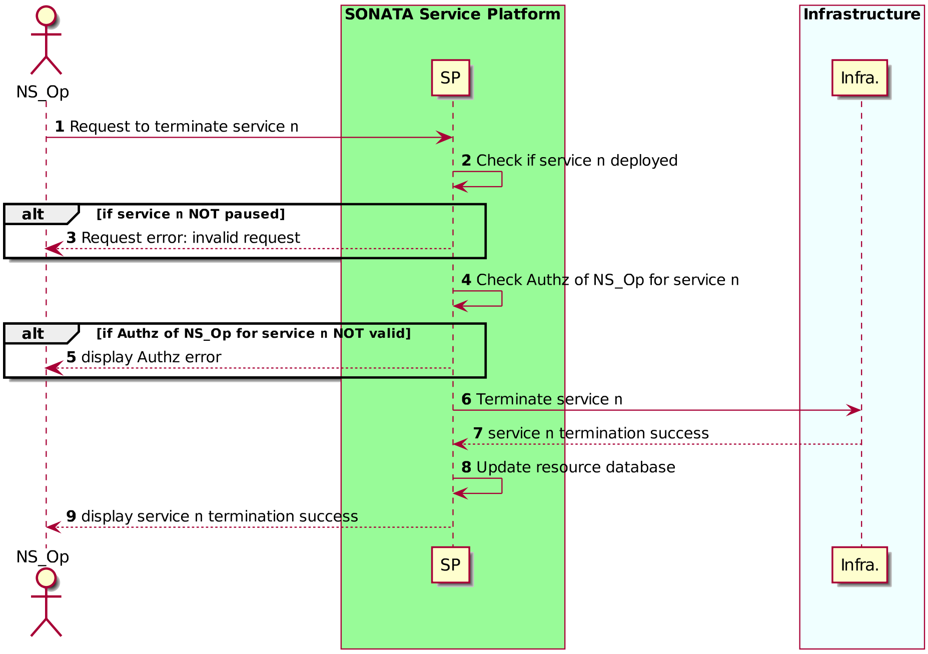

Figure 5.6: Terminate a service

5.2 Core Components

SONATA's service platform consists of four high-level components shown in Figure 5.7. The first component is the gatekeeper module that is the main entry point of the service platform. It implements API endpoints for SDK tools, like the packaging tool, and allows service developers to manually control and manage services deployed on the platform.

The gatekeeper directly interfaces with the second platform component that is a platform-specific catalogue, which stores service artifacts uploaded to the platform. The third component contains repositories used to store meta data of running services, e.g., monitoring data or placement results. Further information on the aforementioned components is provided in Section 5.2.2.

The last and main component is SONATA's extensible management and orchestration (MANO) framework which interfaces with other components and implements all management and orchestration functionalities of the service platform. This framework is the central entity of the platform and uses a message broker system as flexible communication backend. Such a message broker is a software artefact that can be used where software components communicate by exchanging formally-defined messages. Message brokers typically provide functionalities like message translation between sender and receiver, authentication, or different messaging patterns. We describe our requirements for such a broker component in Section 5.2.3.1 in more detail.

The framework implements service management as well as VNF management functionalities using a set of loosely coupled functional blocks, which are called *MANO plugins*. This design allows platform operators to easily integrate new functionalities into the platform by adding new MANO plugins to it. The behavior of some of these plugins can be customized by service developers with so-called *service or function specific managers* which are bundled with a service package and uploaded to the platform. This allows service developers, for example, to define service specific scaling and placement algorithms. The following sections describe these components and their functionalities in more detail.

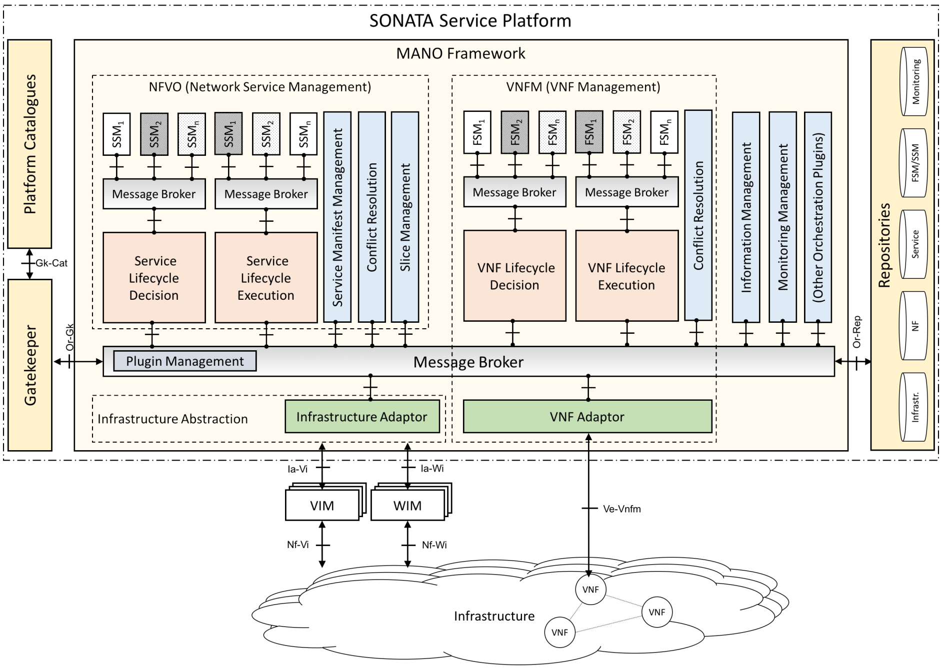

Figure 5.7: Detailed ar.chitecture of SONATA service platform

5.2.1 Gatekeeper

The Gatekeeper is the module of the SONATA's Service Platform who validates the services submitted to that platform. This section presents a preliminary design of this module, which will be further detailed in the context of work package WP4, *Resource Orchestration and Operations repositories*.

5.2.1.1 User Management

The User Management component of the Gatekeeper will allow the SONATA service platform owner to control who can do what in the platform. The project will try to **reuse** existing services or components, either from other EU projects or open-sourced ones, to implement this feature, since it is not its focus.

This feature is particularly important in the recursive scenario, on which we may have a chain of SONATA service platforms interacting for the implementation of an end-to-end service.

Authentication

The most basic feature of any user management component will be to know *who* is the user, a feature that is usually called *authentication*. Authentication in the SONATA service platform will be done by a user registration and email confirmation. An extra step of validating the user might be needed, after the user has confirmed her/his email, if only restricted users are supposed to register.

In the scenarios where the SONATA service platform will be used, developers are usually more used to security policies, say an authentication based on a public-private key approach, that are considered more secured than the basic authentication based on username and password. Keys are generated by the developer in pairs, one being his/her *private* key, which must be kept secret, and the other being his/her *public* key, which can be public and therefore stored, say, on the service platform. By providing his/her public key to the platform and encrypting his/her requests with the private key, the platform can use the public key to decrypt the request and be sure it came from who claims to have sent it.

In more sophisticated scenarios, *suspending* a user, usually for misbehaviour, might as well be useful. For administrative purposes, *deleting* a user might also be useful.

A scheme of lists of users and processes to automatically migrate them between those lists might as well become handy. For example, users that have failed to present adequate login credentials for a pre-defined number of times may put on a *grey* list for a certain amount of time. After a number of presences in the grey list, that user may be put in a *black* list, thus forbidding him/her to login.

Authorization

The definition of *what* each (known) user can do is usually called *authorization*.

The most common approach nowadays to authorization is called *role-based*, in which each user is assigned one a (or more) role(s) and different roles have different permissions. This extra level of indirection, that is users to roles and roles to permissions, simplifies the overall maintenance of the system, when compared to a more direct scheme, like users permissions.

Specially when accessing external APIs, it is common to issue *temporary keys* (then usually called *tokens*) which enable temporary access to those APIs. Real keys therefore do not leave the realm on which they're valid and useful, thus increasing the overall level of security.

Secured (internal) interfaces

At least part of this Gatekeeper component may have to be promoted into a module of the Service Platform, in case the project decides to secure the internal interfaces between several modules.

As stated before, the project will try to *reuse* already available components for this.

5.2.1.2 Package Management

The Gatekeeper module receives the software to be validated in the form of *packages*. Package management is mostly about accepting and validating new or updated packages.

Package descriptor

The metadata describing such packages is called *package descriptor*, which exact format is being defined in Section 3.2.

On-boarding packages

Only known (i.e., successfully authenticated) and authorized users will be able to submit new or revised services to the SONATA Service Platform.

On-boarding of a package can only be considered successful when package *validation and attestation* is successful. Only then the (new version of) the package will be part of the *catalogue*.

It makes sense that on-boarding requests are processed in a *first come, first served* way, otherwise contradictory requests may jeopardize the whole system. The usual solution for this problem is to use a *queue* mechanism that guarantees this sequence.

Validation and attestation

A package descriptor is validated in several ways:

- **Syntax:** validation against the expected package descriptor format;
- **Semantics:** validation of at least some of the simplest parameters. The exact semantic aspects to be validated depend on the content and format chosen for the package descriptor;
- **Licensing:** all external dependencies (i.e., packages, libraries or services) have to have their licenses checked before being used;
- **Tests existence:** although this might be seen as part of the syntactic/semantic correction, there must be a set of tests that can be executed (see below) when validating the package. Depending of the scope and complexity of the Service, these tests may be a subset of the unit tests or a more elaborate suit of integration tests;
- **Tests execution:** besides providing a suit of tests, these have to be successfully executed. This execution may (usually will) imply the creation and initialization of at least one test environment. When the package under test depends on other packages, libraries or services, those too should be taken into account in the execution of the package tests.

If no clear trust exist between developers and the orchestration framework, the service package must include some signatures that allows validate it and VNFs. A potential model could be a hash signed with public certificates of relevant info:

- VNFs images and/or specific configurations deployed as part of the VNF
- The set of descriptors and topologies (VNFD, NSD,..)

Life-cycle management

Requests for a change in the life-cycle of a package must be validated. This might be a simple authorization configuration. The actual life-cycle management may be split with the life-cycle plugin of the SONATA Service Platform, specially the ones more related with the run-time of the instance of the service.

- **Deployment:** Valid packages, available at the Repository, may receive request for deployment. This package deployment implies the creation of all the environments and connections needed for the package and its dependencies to work and of an instance of that package;
- **Instance (re)-configuration:** A deployed package instance may need to be configured. A special kind of configuration might be, for packages supporting multi-tenancy, adding a new tenant. The package may have 'open parameters' that can only be closed upon instantiation (e.g., an IP address). If a Package upgrade happens, a reconfiguration of the instance must also be made;

- **Instance (re-)start:** When, e.g., configuration changes;
- **Instance monitoring:** This is not strictly a 'change' in the life-cycle.
- **Instance stop:** includes soft-stop (i.e., not accepting new requests and letting currently running request reach their end of life normally, with a pre-defined time-out) and hard-stop (i.e., a sudden stop, with requests still being answered by the service);
- **Instance termination:** Frees any resource(s) that were being used. Beware of dependencies.
- **Removal:** Beware of currently running instances and dependencies.

5.2.1.3 Key Performance Indicators

Key Performance Indicators (KPIs) specific to every module or plugin must be collected, calculated, stored and displayed upon request, possibly in a *dashboard* like a Graphical User Interface (GUI). This is not necessarily part of monitoring, but may be implemented by extracting some of the values needed for the calculation of those KPIs from monitoring.

5.2.1.4 Interfaces

This section lists the different kinds of interfaces the Gatekeeper module will have. The GUI and the Application Programming Interface will provide services to the SDK and eventually other SONATA Service Platform(s). Other interfaces will mostly be internal and will be detailed later.

Graphical User Interface

As stated above for User Management, a Graphical User Interface (GUI) is not a key feature of the SONATA's platform, so we propose to choose a framework from the many available that will allow us to quickly build such a GUI and easily maintain it. Ease of maintenance is key here, since we're adopting an agile development approach, on which we embrace changes. In principle, this GUI will be for internal use only, so the project does not intend to invest a lot of resources into the overall user experience of this GUI.

SONATA expects also to provide some information (e.g., the KPIs mentioned above for users with the management role, or the results of monitoring the execution of the services) in graphical form (i.e., in a *dashboard* like).

Application Programming Interface

Application Programming Interfaces is what makes the module (and any module) programmable. APIs have different demanding in terms of traffic going through them. For example, monitoring related APIs will be subject to many messages of small size, while package related APIs will mostly support a low number of messages but with some significant size. APIs, specially the external ones, will have to be secured, avoiding eavesdropping and other security threats.

5.2.2 Catalogues and Repositories

The service platform implements all the catalogues and repositories that are described in Chapter 3. External access, say using the SDK catalogues connector, is only possible via the gatekeeper. However, the SONATA service platform itself can act as a client to external catalogues, like public catalogues, in order to retrieve missing service components, when needed.

5.2.3 MANO Framework

The core of SONATA's service platform is the highly flexible management and orchestration framework that provides all functionalities to manage complex network services throughout their entire lifecycle. The framework consists of a set of loosely coupled components that use a message broker to communicate. This section describes this framework, its components, and the communication between them.

5.2.3.1 Extensible Structure: Leveraging a Message Broker Approach

Figure 5.8 shows the extensible MANO framework consisting of a set of loosely coupled components, called MANO plugins. Each of these plugins implements a limited, well-defined part of the overall management and orchestration functionality. This design has the advantage that functional parts of the system can easily be replaced, which provides a high degree of flexibility. With this, the platform operator who owns the service platform is able to customize the platform's functionality or to add new features by adding additional plugins.

In our design, all components are connected to an asynchronous message broker used for inter-component communication. This makes the implementation and integration of new orchestration plugins much simpler than it is in architectures with classical plugin APIs. The only requirement a MANO plugin has to fulfil is the ability to communicate with the used messaging system, such as RabbitMQ and ActiveMQ. We do not enforce a programming language for implementing plugins. Similarly, we do not prescribe how the plugins are executed. For example, plugins may be executed as operating system-level processes, containers, like Docker, or separate virtual machines within the platform operator's domain. It is also possible to allow remote execution of MANO plugins, which implies the availability of a secure channel between the remote site and the messaging system, since all MANO plugins are expected to be executed in an environment trusted by the platform operator.

Another key feature of the MANO framework is the ability to customize the management and orchestration behaviour for single services and network functions. This is realized with service-specific managers and function-specific managers. Such a specific manager is a small management program implemented by a service/function developer with the help of SONATA's SDK. These specific managers are shipped together with a network service or function inside the package uploaded to the service platform. They connect to specialized MANO plugins, which provide an interface to customize their default behaviour. Such plugins are called executive plugins. Typical examples for such specific managers are custom service scaling algorithms or function lifecycle managers, like it is shown in Figure 5.7.

5.2.3.2 Topic-based Communication

The MANO framework's internal communication is based on a message broker and uses a topic-based publish/subscribe pattern. This enables each component to talk to all other components without the need to configure or announce API endpoints among them. An advantage of this communication approach is that the set of connected components, and thus the overall behaviour, can easily be customized by the platform operator. It is also possible to introduce additional components that are integrated into the workflow of existing ones without changing the component's implementation. This can either be done by reconfiguring existing components to emit messages to which the new component subscribes or by re-routing the messages on the message broker, e.g., by rewriting message topics. All components that want to connect to the system have to register themselves to the message system that controls which messages are allowed to be sent and received by the component. This can be implemented by a topic-based permission system defining read/write permissions for each connected component.

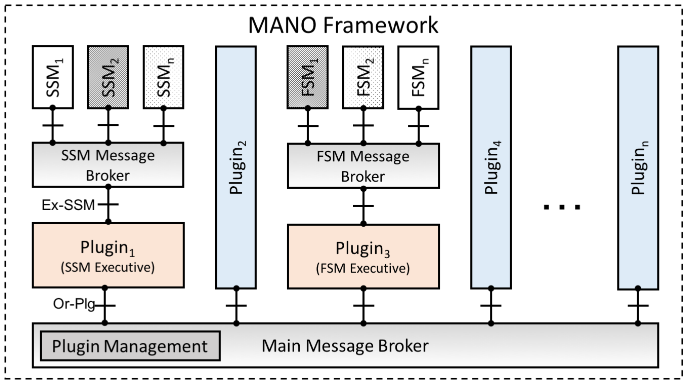

Figure 5.8: MANO framework with MANO plugins and service-/function-specific management components

The publish/subscribe communication pattern comes with the advantage that a single message can directly be delivered to multiple receivers, which helps distributing global information inside the management and orchestration system. A hierarchical topic structure allows components to have fine-grained control over the information they want to receive. For example, in case of a change in the underlying infrastructure, e.g., a link failure between two data centres, a failure might be detected by the infrastructure abstraction layer and announced to all components that are interested in reacting to the event, e.g., a placement component or a service lifecycle manager. Topic subscriptions can be either *specific*, which means that a component subscribes to exactly one particular topic, or they can be *generic* by adding wildcard symbols to the subscription topic.

Some interactions between components require a request/reply-based communication behaviour, e.g., allocation requests of compute resources. These interactions can also be done on top of the used publish/subscribe pattern and implemented with two distinct topics to which requests and the corresponding replies are published. As a result, the communication of the system is completely *asynchronous* which is an advantage because requests, like resource allocations, typically need some time to generate a reply. The underling publish/subscribe pattern also allows to receive replies from multiple components at the same time. For example, the information management can easily keep track of allocated resources by listening to the corresponding message topics. Examples for the used topic hierarchy with three top level topics are given in Table 5.1. A more detailed definition of the topic hierarchy can be found in Appendix B.

Table 5.1: Examples for hierarchical message topics

Topic	Example(s)
service.*	service.management.lifecycle.onboard.*
	service.inventory.instances.create.*
	service.monitoring.perfomance.SLA.*
function.*	function.management.scale.*
	function.management.placement.*
	function.management.placement.*
infrastructure.*	infrastructure.management.image.*
	infrastructure.management.compute.*
	infrastructure.monitoring.storage.*

Besides defining the topic hierarchy used for the communication in the MANO framework, a detailed description of the format and content of each message sent or received by MANO plugins is needed. Message broker systems do typically not fix these formats and leave their definition to the clients connected to the broker. This is commonly done with definition languages like XML schema, JSON schema or YAML schema which describe the data fields of a message. Each of the components connected to the MANO framework will have such descriptors to define their interfaces

publish/subscribe messaging pattern. Additional requirements are as follows:

Topic-based message routing: The used message broker must support message routing and topics. A client must be able to subscribe to single topics and also to multiple topics using wildcard symbols in the subscription request.

Scalability: The selected system should be able to handle a high number of messages exchanged between different components without becoming a bottleneck to the system. The current system architecture expects a couple of dozen components to be connected to the main message broker and a couple of components for each deployed service connected to the service-/function-specific executive message brokers.

Reliable message delivery: The broker must provide basic delivery guarantees to the connected components. This includes reliable message delivery, in-order message delivery and message persistence for easy debugging and failure recovery.

Authentication and Authorization: The broker should offer authentication and authorization mechanisms for connected components, which is especially important if MANO plugins are executed on remote sites.

Transport mechanisms: The available transport mechanism should rely on standard protocols, like TCP for easy deployment of the service platform.

Topic based permission system: A broker should provide a topic-based permission system to offer fine-grained control over the topics to which each component can subscribe or publish messages.

Table B.7 in the Appendix gives an overview of candidate message broker implementations which can be considered for the implementation of SONATA's service platform. All of them provide publish/subscribe functionality but not all of them fulfil the complete set of requirements. The final selection of the used message broker is not part of this document and will be done during the first prototype iteration.

Table 5.2: Candidate message broker systems

	RabbitMQ	ActiveMQ	Apollo	OpenAMQ	ZeroMQ	Kafka	HornetQ
License	MPL	Apache 2.0	Apache 2.0	GPL	GPL	Apache 2.0	Apache 2.0
Broker	central	central	central	decentral	no broker	decentral	central
Pub/sub	yes	yes	yes	yes	yes	yes	yes
Topic w. wildcards	yes	yes	yes	yes	yes	no	yes
Guarantees	reliable, in order, persistent	reliable, in order, persistent	reliable, in order, persistent	reliable, in order, persistent	none	persistent	reliable, in order, persistent
Auth.	yes	yes	yes	no	no	yes	yes

	RabbitMQ	ActiveMQ	Apollo	OpenAMQ	ZeroMQ	Kafka	HornetQ
Transport mechanisms	TCP	TCP, UDP, IPC, multicast	TCP, UPD, websockets	TCP	IPC, TCP, TICP, multicast	TCP	Netty, TCP, HTTP
Topic-based permissions	queue/exchange based	no	no	no	no	no	no
Bindings	C, C++, Java, Python, Ruby, Erlang, ...	C, C++, Java, Python, Ruby, Erlang, ...	C, C++, Java, Python, Ruby, ...	C, C++, Java, Python, Ruby, ...	C, C++, Java, Python, Ruby, ...	C, C++, Java, Python, Ruby, Erlang, ...	C, C++, Java, Python, Ruby, ...
Reference	[49]	[17]	[19]	[25]	[26]	[18]	[44]

5.2.3.4 MANO Plugin Management and Coordination

MANO plugins are the main components of the MANO framework and are under the control of the platform operator. They can be executables, containers, or VMs which typically run inside the domain of the platform operator and connect to the main message broker. To keep track of the connected MANO plugins, the system uses a *plugin management* component, which can either be an extension of the used message broker system or an external component that interacts with the configuration interface of the message broker. This component does not provide functionalities related to service management or orchestration tasks and is only responsible for managing service platform-related components. It implements the following functionalities:

Authentication: Plugins that want to connect to the message broker system have to be authenticated. This authentication procedure can either be implemented in the plugin management component or directly be based on the message brokers default authentication mechanisms which is the favoured solution, since it is already provided by almost every message broker implementation.

Plugin registration: After a plugin is authenticated to the broker, it is allowed to send messages to and receive messages from it. Additionally, it has to register itself to the plugin manager so that the system becomes aware of its presence. This registration procedure is shown in Figure 5.9 and uses the standard publish/subscribe mechanism on a dedicated *registration* topic over which each authenticated plugin can publish and the plugin manager needs to subscribe. A plugin also announces its capabilities and functionalities to the plugin manager during the registration procedure. It announces, for example, that it implements the lifecycle management logic for a certain type of VNF.

Topic permission management: The plugin manager is also responsible to control the topic permissions that specify which plugins are allowed to publish or subscribe to a certain topic. To do so, plugins have to request write and read permissions for each topic they are interested in during the registration procedure as outlined in Figure 5.9 (8-11).

Bookkeeping: An additional functionality of the plugin manager is maintaining a dictionary of active plugins and all functionalities registered by them. Each plugin can query this dictionary to obtain information about available functionalities in the system. A service manifest

manager can, for example, query for a plugin that is able to parse a specific service description format.

Watchdog functionality: All plugins are executed as independent entities that can fail. The plugin manager is responsible to monitor the system and check for failed plugins that do not respond anymore. If it detects a failed plugin, it can generate events to inform other components about the issue and it can restart the failed plugin.

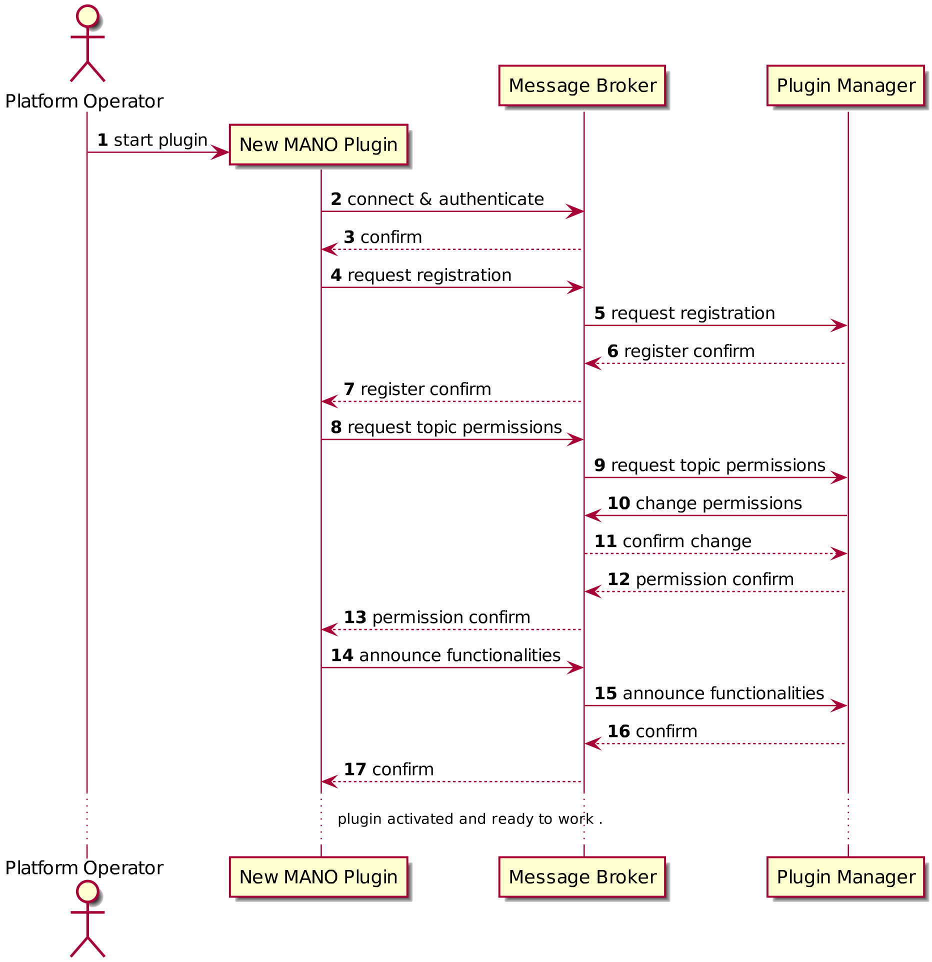

Figure 5.9: MANO plugin registration

Plugin Lifecycle

Figure 5.10 shows the lifecycle of a MANO plugin. A plugin connects and authenticates over the message broker after it has been started. Using this initial connection, it is able to register itself to the plugin manager. A registered plugin can either be in *active*, *pause* or *standby* state . A plugin in standby state does still consume messages from the broker and might process them internally but it does not publish new messages to the system. Plugins in pause state do not receive any messages nor do they perform any actions.

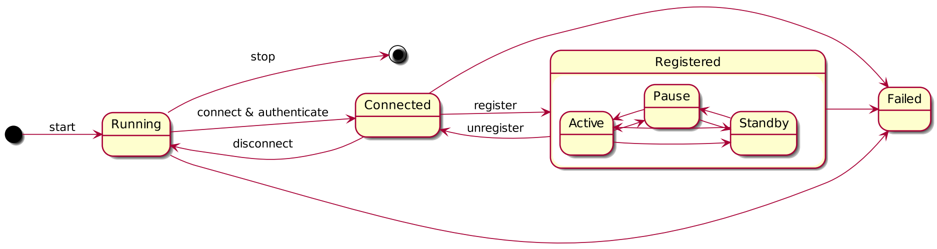

Figure 5.10: MANO plugin lifecycle

Replacing MANO Plugins at Runtime

A typical use case for a platform operator is replacing a running plugin with a newer version. This action can be done without restarting any other component of the system as long as the plugin implementation fulfils certain requirements. First, the plugin has to be able to buffer incoming messages for later processing when it is in the *standby* state. Second, the old and the new version of the plugin have to be able to migrate their internal state, e.g., the old plugin writes the state to a repository and the new plugin fetches and imports this state. If these functionalities are implemented, a plugin can be replaced by performing the following steps.

1. Start the new plugin, register it to the platform, subscribe it to the needed topics and put it into *standby* state so that it receives and buffers all message which are also received by the old plugin.
2. Pause the old plugin and migrate its state to the new plugin (incoming messages are still buffered in the new plugin during this).
3. Set new plugin to *active* state and process buffered messages.
4. Unregister, disconnect and stop the old plugin.

Using this procedure, plugins can be hot-swapped without touching other components of the system. However, there is a small timespan in which neither the old nor the new plugin is active which can cause additional delay in the system. This delay depends mostly on the time needed for the state migration between old and new plugin.

5.2.3.5 Function and Service Specific Management and Orchestration

Function- and service-specific managers are a unique feature of the presented MANO framework. To this end, the framework has to allow the integration and activation of these highly customizable components on-the-fly whenever a new service package is deployed to the platform. The design has to ensure that the platform operator always keeps control over its own platform even if parts of its functionality are customized by service developers.

FSM/SSM Integration

To allow function- and service-specific behaviour customizations, each MANO plugin of the platform can declare itself as FSM-/SSM-capable. Such a customizable plugin is called *executive plugin* and must offer a northbound interface to interact with FSMs/SSMs. An example for this is a placement and scaling plugin which wants to allow the replacement of its default scaling algorithm.

Figure 5.11 shows an executive plugin with SSMs connected through a dedicated message broker. Each of these SSMs is responsible to manage one or more services belonging to exactly one platform customer. The figure also shows that every executive plugin has to offer a default FSM/SSM that should be used if a function or service is not bundled with its own FSM/SSM. Here, once more a message broker-based design replicates the ideas of the MANO framework and its loosely coupled MANO plugins. As a result, FSMs/SSMs have a similar lifecycle like MANO plugins, including separated connection and registration procedures. The difference to MANO plugins is that FSMs/SSMs are provided by function/service developers and not by the platform operator. This makes their management and coordination more complicated.

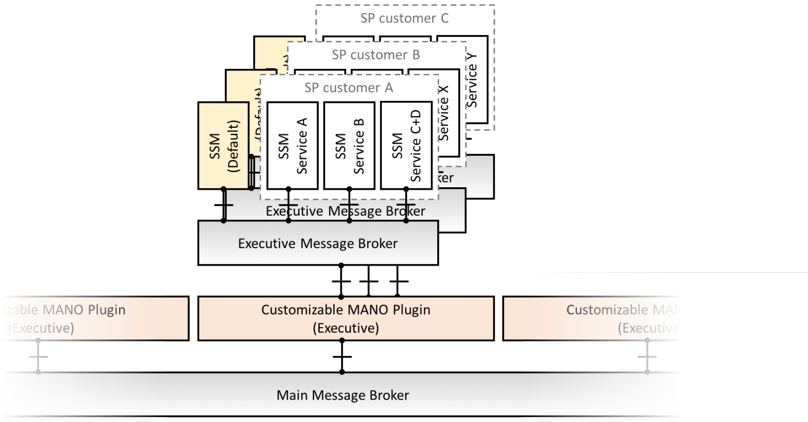

Figure 5.11: Customizable MANO plugin (executive) with active SSMs managing different services deployed by different platform customors

Executive plugins provide a clear separation between FSMs/SSMs and the rest of the management and orchestration system. An advantage of this is that each executive has fine-grained control over messages sent to and received from a FSM/SSM. A placement executive can, for example, filter and change topology information used as input to customized placement SSMs. The reason for this can be that the platform operator would not want to share the original network topology with the platform customers. Furthermore, executive plugins are responsible to check all outputs of FSMs/SSMs to detect misbehaviour, e.g., a scaling SSM may not be allowed to request scale-out operations with hundreds of new VNF instances. In such cases, outputs of customized SSMs are discarded by the executive and the results of the default FSM/SSM can be used. The interface between executive plugins and FSM/SSM is defined by the executive plugin developer and tailored to the tasks it should provide. Only the FSM/SSM boarding and management procedures are pre-specified as shown in the next section.

FSM/SSM Management and Coordination

FSMs/SSMs are bundled into service packages and have to be on-boarded and activated whenever a service is uploaded and deployed. Just like any other artefact contained in a service package, FSMs/SSMs are statically checked by the gatekeeper module and extracted from the package before they are forwarded to other components of the platform, e.g., stored in catalogues. After this, FSMs/SSMs are forwarded to their specific executive plugins which instantiate them and perform several tests on them. These tests are defined and implemented inside the executive plugin and are black-box tests. They validate the behaviour of the FSMs/SSMs by interacting with their external interfaces. A placement SSM can for example be tested by sending predefined service graphs and

substrate topologies to the SSM and by validating that the placement outputs stick to certain bounds. After all tests are finalized, the test instances are then terminated and the test results are sent to the gatekeeper that might decide to start the service. If this is the case, the FSMs/SSMs are instantiated in production mode and connected to the executive plugin's broker before they register themselves to the executive. The FSM/SSM on-boarding procedure is shown in Figure 5.12.

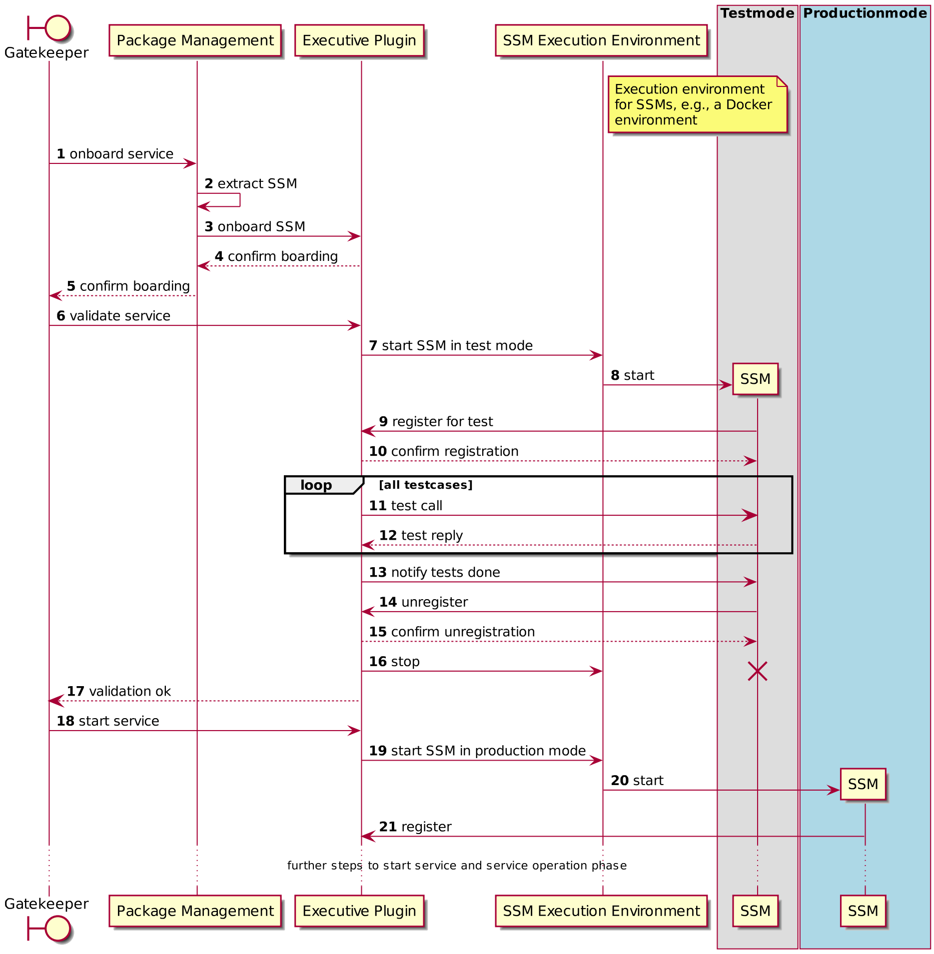

Figure 5.12: FSM/SSM onboarding, validation, and activation procedure example

Besides the initial on-boarding task, updating FMSs/SSMs during service operation is a typical use case that has to be supported. To do so, the same mechanism as MANO plugin replacement can be used. The most complicated point here is again the state transfer from the old FSM/SSM to the new one. Solutions for this are either designing stateless FSMs/SSMs or implementing an API to export and import the state. This API implementation can be supported by a FSM/SSM framework in the SDK that contains, e.g., abstract classes as basis for new FSMs/SSMs that implement state export and import functionalities.

FSM/SSM Communication Isolation

Each executive plugin in SONATA's service platform can be customized by multiple FSMs/SSMs. Each of these FSMs/SSMs belongs to exactly one platform customer and manages one or multiple functions/services of that customer. This design makes it necessary to isolate the communication between FSMs/SSMs in order to ensure that the managers of one customer can not influence or

monitor the managers of other customers. To this end, an executive plugin can either create a dedicated message broker instance for each platform customer and connect all FSMs/SSMs of this customer to it or it can use a single broker instance and use features of this broker system to achieve isolation. This can be based on a concept called *virtual hosts* that is supported by most modern broker implementations, like RabbitMQ or ActiveMQ. Using this, one isolated virtual broker will be created for each customer of the platform to separate communication of management components between customers. Additionally, topic-based permissions can be used to also ensure that each FSM/SSM is only allowed to talk to the executive plugin but not to other FSMs/SSMs of the same customer. However, the decision to use this topic-based isolation is left to the executive and can be configured by the platform operator.

5.2.3.6 Recommended MANO Plugins

SONATA's service platform architecture is based on a very flexible MANO framework consisting of a set of MANO plugins interconnected by a message broker. This design allows platform operators to customize the platform to their needs. As a result, we do not specify a fixed set of MANO plugins that have to be used within the platform. Rather, we specify a collection of typical MANO functionalities as a reference and best practice recommendation for the list of plugins. This collection contains, but is not limited to, the following MANO plugins that are also shown in Figure 5.7. The set of plugins presented here does not implement the full list of functions listed in Section 2.2 and we leave the mapping between these functions to actual plugins as a task for WP4.

Service Manifest Management

This plugin is called by the gatekeeper and unpacks service packages uploaded by service developers. The unpacked artefacts can then be returned to the gatekeeper for further validation. It is responsible to resolve references given in the service package and to download the referenced artefacts , e.g., virtual machine images. It may also encapsulate certain artefacts to be able to execute them on the underlying infrastructure. For example, the plugin may embed a network function given as a JAR file into an default virtual machine image to deploy it on an Infrastructure-as-a-Service cloud. There might by several of these plugins connected to the system, each responsible for a specific type of service package.

VNF and Service Lifecycle Decision

These decision plugins are responsible to take decisions about the lifecycle management of either a single VNF or a complete service. These decisions can, for example, be changes in the lifecycle as well as scaling and placement decisions. They decide the resources allocated to a VNF or a service. We recommend a combined approach for placement and scaling decisions because they heavily depend on each other. The exact decision behaviour of these plugins is implemented by FSMs/SSMs to allow service developers to adapt the behaviour to the needs of their VNFs and services.

To enable automated scaling of network functions and services, decision plugins need information about the key performance indicators of the functions and services. These KPIs are obtained and preprocessed by the monitoring subsystem. The decision plugins can directly access monitoring information available in the monitoring repository or they can register callbacks to the monitoring system which are triggered when certain conditions are fulfilled, e.g., the number of user sessions a service sees exceeds a certain threshold.

The network service lifecycle decision plugin always considers the entire service chain for scaling decisions rather than operating on single VNFs only. It gets a service graph with extended scaling

descriptions (e.g., max. n instances of VNF m) together with a substrate topology and a set of performance indicators as input. Based on this information, the scaling and placement is computed and a embedding of the service chain into the substrate topology is returned. The plugin can filter and validate in-/outputs of the connected SSMs and thus control which information, e.g., underlying topology or performance indicators, is available to the connected SSMs.

VNF and Service Lifecycle Execution

These plugins are responsible to *execute* lifecycle management decisions for either single VNFs or entire network services. The main behaviour of these plugins is implemented in FSMs/SSMs so that they can be customized for each VNF type or service. This is needed since different functions or services will have different requirements to their lifecycle management system. This MANO plugin comes with default FSMs/SSMs which implement a generic lifecycle manger. Since there are different lifecycles to mange, e.g., the lifecycle of a service vs. the lifecycle of an individual VNF, multiple of these plugins are connected to the system.

Conflict Resolution

Behaviours of the lifecycle decision and execution plugins can be customized for each network function or service by using FSMs/SSMs, which might result in conflicts. For example, competing placement and scaling managers will result in resource conflicts, which have to be resolved by the MANO framework. This functionality is implemented in a dedicated MANO plugin that interacts with the lifecycle decision and execution plugins. There are different alternatives how such a conflict resolution system can be implemented. Two examples for this are as follows:

- Priorities: A simple conflict resolution scheme can be based on priorities assigned to services and VNFs by the platform. The conflict resolution can then easily decide which service gets the requested resources when a conflict occurs. However, this scheme does not provide any flexibility and is most probably not suited for real-world usage.

- Auction mechanism: A better solution is to build a auction mechanism for the conflict resolution system. Such a system can communicate with the lifecycle decision plugin using a round-based interaction pattern. Based on this, a placement and scaling FSM/SSM can bet for the resources it wants to use. If a conflict occurs and the resource request can not be fulfilled, the FSMs/SSMs of the conflicting functions/services are allowed to bet again. The main challenge in such a system is guaranteeing fairness and ensuring that competing functions/services can not intentionally harm each other.

Further conflict resolution approaches may be investigated during the project. It would also be possible to add the conflict resolution functionality directly to the lifecycle decision plugins. But this has the downside of loosing modularity when, for example, the auction mechanism should be replaced.

Monitoring Processor

The monitoring management receives monitoring data and manages the monitoring sub-system. This includes monitoring data collected from the service platform and the connected infrastructure, as well as the deployed services. It implements functionalities to pre-process and filter monitoring information before writing them to the monitoring repository. A monitoring management plugin can also provide functionalities to register callbacks which are triggered when certain conditions are fulfilled. This has the advantage that only this single component has to continuously observe the

monitoring inputs and can notify other components when its needed. There can be multiple monitoring managers connected to the system, each responsible to process a certain type of monitoring data.

Slice Management

SONATA's service platform supports network slices to deploy network services in an isolated fashion. To this end, the MANO framework contains a slice management plugin which is in charge of controlling and managing slice requests. Depending on the used slicing approach, the plugin is either responsible to directly manage the slices, e.g., to create a slice, or it communicates with an external slice management system which is in charge of managing the slices. The slicing concept of SONATA's service platform is described in Section 5.6 in more detail.

Information Management

The MANO framework maintains all its information in the repositories that are part of the SONATA service platform and described in Chapter 3. Information management plugins can be used to provide a consolidated view on this data. Other plugins can query this plugin for information they need rather then accessing the repositories directly. This creates an abstraction to the repository interface and improves the modularity of the framework. Information management plugins are also used to automatically collect information from the MANO system, process them and store them. For example, an information management plugin can subscribe to all infrastructure events occurring on the main message bus and thus automatically ensure that the current status of the infrastructure is captured in the corresponding repository. There can be multiple of these plugins each responsible to process a certain type of information.

Resource Management

A resource management plugin will be in charge to request additional resources from the underlying infrastructure. It has also bookkeeping tasks and keeps track of free resources.

5.2.4 Infrastructure Abstraction

The generally accepted view of NFV deployment scenarios, regardless if multi-domain or single-domain is included, the instantiation of VNFs is considered to happen in various location within the network topology (i.e core and edges). These locations are anticipated to host datacenter infrastructure (at various scales) that are interconnected via Wide Area Network (WAN) infrastructures. These designated locations are called Network Function Virtualization Infrastructure Points of Presence (NFVI-PoP) according to ETSI NFV. Within each NFVI-PoP the resources (i.e computing, storage and network) are managed via the entity called Virtual Infrastructure Manager (VIM). The actual implementation of the VIM is directly influenced by and dependent on the technology that is used in order to provide virtualization support within the NFVI (i.e. hypervisor) and the type of resources actually being virtualized. A VIM may be specialized in handling a certain type of NFVI resource (e.g. compute-only, storage-only, networking-only), or may be capable of managing multiple types of NFVI resources (e.g. in NFVI-Nodes). Hence, VIM can be found in a variety of implementations, each exposing each own set of Northbound and Southbound interfaces and supported functionalities.

Taking the above into account SONATA aims in the definition and implementation of an Infrastructure Abstraction Layer capable of abstracting the VIM specific implementation of the VIM's exposed Southbound interfaces and APIs from the SONATA SP. In this view, the approach that will be followed within SONATA is illustrated in Figure 5.13.

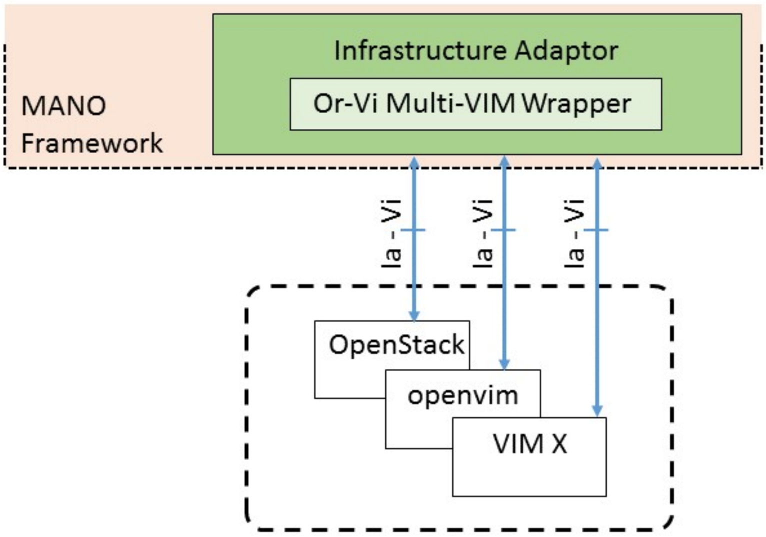

Figure 5.13: Infrastructure Abstraction Model

The approach will follow similar pattern as the one used by OpenDaylight controller in order to support the variety of Southbound interfaces that are exposed from networking devices. In this respect SONATA will implement a modular Infrastructure Abstraction model, able to support pluggable wrappers. The aforementioned wrappers will allow SONATA SP to continiously expand and support new VIM implementations or follow the evolution of the already supported ones. It is therefore planned to implement a full wrapper for Openstack [39] based VIM which is currently considered as the de-facto standard VIM implementation for NFVI. In addition to that other VIMs are considered as added value implementations, such as OpenMANO [51] or OpenBaton [12] VIM implementations.

In the case of WAN Infrastructure Manager (WIM), this functional block may be added as a specialized VIM, in particular for new virtualized deployments. Alternatively, a WIM can also be an existing component of the OSS/BSS functional block in legacy WAN deployments. In this scope the functionalities and scope greatly depend on the underlay networking technologies employed as well as on their virtualization capabilities. For SONATA, the provisioning part of the WAN networking links that interconnect the NFVI-PoPs is considered out-of-scope. Simplifying assumptions will be used in place to overcome this in order to support multi-pop use case scenarios e.g. the tunnels will be pre-provisioned and network matrix (providing resource information) will be available at the infrastructure repository.

This design choice allows to keep the same API for the Southbound interfaces towards the SP Platform and allow for VIM specific wrappers to handle the communication with each VIM. This way we avoid the burden of supporting and maintaining multi VIMs and WIMs end to end in the system, which is overly complicated task. Moreover, as each VIM and WIM still evolve release by release and APIs get updated/added/changed/removed, this usually requires changes in the data model to reflect the changes/new features etc., and in turn reflect those changes across all components and appropriate flows. The decision regarding which VIM and WIM instance to use is part of the functionality of the other components of SONATA, that interact with the infrastructure abstraction layer. The infrastructure abstraction layer should be kept relatively simple and serve

as enabler for the other more complex parts of SONATA.

5.3 Lifecycle Management

This sub-section details how SONATA's Service Platform will manage every Network Service's (NS) or Virtual Network Function's (VNF) lifecycle.

We will be using mostly ETSI's definition for this life-cycle (plus the first one), which comprises the following steps.

- **Develop:** this occurs prior to the submission of the NS/VNF to the SONATA Service Platform;
- **On-board:** validates the NS or VNF descriptors, and stores them in the Catalogue;
- **Deploy:** instantiates NSs or VNFs from the Catalogue, storing the related information in the Repositories;
- **Monitor:** collects information on the deployed instances, storing it within the platform;
- **Scale:** adds/removes resources that allow a better performance/less resource consumption;
- **Heal:** tries to recover the health of the instance;
- **Upgrade/patch:** deal with new versions of the NSs and VNFs, namely deciding when is it adequate to apply the patch or upgrade the version;
- **Terminate:** end the life of the NS or VNF.

Each one of these steps is further detailed in the sub-sections below.

5.3.1 Development of

The development of the network services and VNFs takes place outside of the SONATA Service Platform, namely using the SONATA SDK. Its descriptor is built on the SDK side and presented to the service olatform's gatekeeper.

5.3.1.1 The impact of Specific Service Managers

Specially with the introduction of the *service specific manager* concept, as outlined in Section 5.2.3.5, NS/VNF might assume a more dynamic role in the definition of the 'service'. Details on which ways exactly are left for work packages WP3 and WP4. SSMs will also simplify configuration of the network service instance, though some limits to this ability will certainly have to be imposed, in order not to jeopardize the remaining services running in the Service Platform.

5.3.2 On-board

On-boarding of a NS/VNF is the set of processes that are executed whenever a new or update versions of the NS/VNF is submitted to the SONATA Service Platform.

The main process in on-boarding a new or updated NS/VNF is the validation of its descriptor. This validation may be:

- **Syntactic:** checks that the overall document format is valid, all mandatory fields are present, etc.

- **Semantic:** checks things like tests can be executed and return a successful result, presented license(s) are valid, etc.

Every significant feature of the service should be testable on the Service Platform's side. This might include the creation of an adequate *testing environment*, the *validation of licenses* associated to the service and all its dependencies and the generation of *testing data*.

The Service Platform, specifically its Gatekeeper module, should hold enough knowledge about the needed licenses. This knowledge may be provided by the external BSS/OSS systems.

The result of a successful on-boarding process is the availability of the submitted NS/VNF in the Catalogue. More than one version of each NS/VNF may have to co-exist, since there may be deployments of previous version(s) still active. Adequate mechanisms for cleaning up the Catalogue from old and unused versions must be implemented and used.

5.3.3 Deploy

Deployment requests of existing network service may come from *OSS/BSS* (external) systems, and have to be validated in many ways, including licensing. Deploying a network service implies deploying VNFs (also from the Catalogue) on which the service depends upon.

One issue that starts appearing in commercial Cloud offerings is the possibility to *reserve resources*. This might lead to under-optimal usage of overall resources (reserved resources might never get used and can not be used by other services), although we recognize that for some resources (e.g., public IP Addresses) it makes sense to have some form of reservation. This issue is crucial to support the *slices* concept (see Section 5.6), where the orchestrator might request a dedicated set of resources from the VIM that can only be used by a particular service (slice mode 2). This kind of request should be optional: a developer can specify if dedicated resources are needed or if resource are shared like in existing cloud environments.

network service configurations may have to be run before deployment (e.g., switching on/off features of the to be deployed network service, according to the available license) and/or after deployment (e.g., ports to be connected to). After deployment configurations may be run by an external (to the network service) *configuration management tool* or by scripts that can be started automatically upon the network service start-up.

It might make sense to *suspend* a service that is deployed (i.e., *running*). We haven't considered such a state in this life-cycle, but we might have to if we want to support things like 'suspension for no payment', 'suspension for security checking', etc.

A successfully deployed network service will have all the deployment data (e.g, its IP address) stored in the corresponding Repository.

5.3.4 Monitor

One of the key issues of modern service deployment, specially when a DevOps approach is used, is the ability to *monitor* the deployed service(s), in order to know in advance how they are performing. This need to expose metrics may pose significant challenges to current monitoring processes and tools.

Within the Service Platform, the Gatekeeper should control 'who can see what'. Monitoring must be:

- very simple;
- ultra-reliable;

- catch everything that can possibly go wrong;

For these features to be implementable, monitoring should be *part of every service layer's component* and should *have an end-to-end perspective*, naturally limited to its scope when it is just one of the several components that are part of the service.

Collected monitoring data must be stored in the Service Platform for later usage. In some scenarios, more detailed and past information must be provided at request. This should be done by using a specific API, with validation by the Gatekeeper.

5.3.5 Scale

Depending on the sophistication of the NS and the VNFs that are part of it, by monitoring the performance of the service may allow automatic scaling to increase or decrease performance.

There are usually two kinds of scaling to consider:

- **Horizontal scaling:** or **scaling out/in**, on which resources are added or removed, thus allowing for better performance or less resources are consumed. This is the preferred way of scaling in cloud/virtual environments;
- **Vertical scaling:** or **scaling up/down**, on which the number of allocated resources are kept, but its characteristics (CPU, memory, storage or networking) are increased or decreased.

Vertical scaling is impossible to implement with currently available technology. **Scaling out**, needing more resources, poses some questions as to if those additional resources should be previously allocated (which leads to a waste of resources that are allocated but possibly never used) or should be requested only when needed.

Scaling a NS end-to-end and for the general case is very difficult to automate. Scaling is one of the most probable features of a SSM: only it knows how to scale a specific NS. Scaling the VNFs that constitute a NS might be delegated to the VNFM, either generic or specific.

5.3.6 Heal

Not every problem a service instance shows in the monitored data is solved by scaling it. Often radical steps such as restarting (usually into a secure or debug mode) or migrating it, after collecting log files is the only way to recover the performance of that resource. That kind of actions fall into the general kind called **healing**.

5.3.7 Upgrade/patch

Upgrading or patching existing and deployed services usually need different kinds of actions, depending both on the kind of service (or its SLA) and on the kind and urgency of the upgrade or patch. For example, security patches are usually urgent, while functional upgrades can usually wait until less loaded hours.

Depending on the technology stack used and the service's architecture, many of these upgrades or patches can be deployed without interrupting the service: a new upgraded/patched service is deployed, and the service usage is migrated (for instance, new sessions are sent to the upgraded/patched instance(s), while old sessions are left to finish in the old instance(s)) in phases out of the outdated instance(s).

There is no generic behavior for the general case. The simplest implementation is to deploy the upgraded definition (which means multiple versions must be supported) and only new 'requests'

are forwarded to the new version, while existing ones are let to finish in the old version. When no 'old' requests were active in the old version, it could be **terminated**. The remaining cases are more complex. For instance, if the upgrade is related to a **security** issue, the service should probably be interrupted and the new version deployed. Te problem with this approach is that the eventual Service Level Agreements (SLAs) that are agreed might be broken in this way. And if the service strongly depends on data (i.e., has state), this approach might lead to lost data. Strategies can be used in which the upgraded version is deployed and service users are migrated in phases from the old to the upgraded version. The only secure case would be the service to support a multi-version, high-availability architecture, in which this upgrade would be supported.

5.3.8 Terminate

Termination of a service instance depends on many factors, some of them planned (e.g., service's end-of-life) and some not (e.g., unexpected problems).

Anyway, a service should be designed to both:

- **hard-terminate:** kills all processes and then shuts down;
- **soft-terminate:** soft termination waits until the processes end (while not letting others to start) and then shuts down;

An SSM must be provided if some other behavior is expected.

Removal of existing and outdated versions of services from the Catalogue must also have to be taken into account (see above, **Upgrade/patch**).

5.4 SONATA Monitoring System Overview

The development of a monitoring system is not trivial in today's complex and heterogeneous infrastructure environments and given the applications diversity that are instantiated therein. Thus, for the purpose of developing, deploying and implementing a monitoring system for SONATA, one has to take into consideration the requirements stemming from the Use Cases, as well as generic characteristics that have to be fulfilled in order to provide a monitoring system capable of offering useful, personalized metrics to the developers, infrastructure owners and end-users.

5.4.1 Extraction of Use Case Requirements

This section starts by identifying the requirements arising from the SONATA Use Case scenarios, related to monitoring. In this perspective, requirements set out in Deliverable D2.1, have been filtered, as depicted in the following Table, acting as the enablers for the monitoring architecture design to be followed in SONATA.

Table 5.3: Use case requirements for monitoring

Req. Name	Description	KPIs
VNF specific monitoring	SP shall expose service and VNF metrics to the network application.	Availability of an API for VNFs capturing monitoring metrics.
VNF SLA Monitor	SP must provide an interface to monitor VNFs SLAs and resource usage.	Provided metrics through API.

Req. Name	Description	KPIs
VNF Status Monitor	SONATA should provide a high level state for each VNF, e.g., (i) deployment, (ii) operating, (iii) error.	Provide a quick overview through GUI displaying status data
VNF Placement	The programmability framework shall allow the customer to deploy VNFs at arbitrary points into the network.	SLA/QoS metrics related to deployment time, cost, etc.
Manual Service Function Placement	It should be possible to manually decide and configure where a service is to be placed.	An API primitive towards the orchestrator that allows manual function placement
Timely alarms for SLA violation	The monitoring system must supply alarms for SLA violations in a timely manner.	Proven performance and scalability of the selected message bus system in the service platform
VNF Real-time Monitoring	VNFs will generate in real time information useful for monitoring and response.	Monitoring frequency, time to process alerts.
Quality of service monitoring	Metrics generation and degradation detection of network traffic, should be supported and reported.	Traffic QoS, packet loss, delays.
Service Platform Scalability	The service platform must be scalable to support a very large number of devices and a high traffic load.	Support for 1000s of sensors and line-rate performance traffic processing.

In particular, the fulfilment of VNF specific monitoring and VNF SLA Monitor requirements demand the implementation of a RESTful API that will allow developers/users to monitor performance data related to their deployed Network Services. Although not specifically mentioned in the above-mentioned requirements, however taking into consideration the openness of SONATA service platform to different emerging technologies, it is required that ”’monitoring system must collect data from VNFs deployed deployed on virtual machines or containers.

Additionally, in order to facilitate the developing process, SONATA monitoring system must collect and offer through APIs, information related to the available resources of the infrastructure, as mandated by VNF Placement. For example, the developer must be informed whether special conditions required for the service deployment are satisfied in a particular infrastructure (NFVI-PoP). Thus, monitoring system must be able to collect data from the underlying infrastructures comprising the SONATA ecosystem. Given that the majority of partners rely on OpenStack cloud environment infrastructures, **the monitoring system must collect information from particular components, such as Nova, Glance, Neutron, etc.**. The collection of information from the above-mentioned components will also address the requirement of VNF Status Monitor, providing service status information (e.g. error state).

Apart from offering an API to developers for collecting and processing monitoring data related to their deployed NS/VNF, **the monitoring system must be able to accommodate VNF-specific alerting rules for real-time notifications**, as described in the Timely alarms for SLA violation and VNF Real-time Monitoring requirements. In this respect, SONATA monitoring system will offer the capability to developers to define service-specific rules, whose violation will inform them in real-time.

Finally, there is one requirement related to the Quality of Service that **demands special attention with regards to sampling period and monitoring accuracy** and another one (Service

Platform Scalability) directly related to scalability of the SONATA monitoring system with respect to the Service Platform and respective infrastructures. Hence, **the monitoring solution must comply with the scalability requirement that must be taken into consideration during the design phase**.

5.4.2 SONATA monitoring high-level objectives

From the previous analysis, in SONATA, we need to design, develop, deploy and implement a monitoring system that will allow:

- developers to monitor service (NS/VNF) or service chain status and performance through an APIs and in real-time or near real-time by defining rules tailored to their needs;
- provide infrastructure resource awareness to the Orchestrator;
- preserve the privacy of infrastructure owners' data, while providing developers with needed information;
- support different platforms, components and technologies etc. and be flexible enough to accommodate different metrics covering networking, storage and processing resources allocated for each deployed service.

5.4.3 Monitoring sources

From the analysis of the Use Case requirements related to service monitoring (D2.1), the SONATA monitoring system must collect data from:

- **Containers**

SONATA monitoring system must be able to collect data (CPU usage, disk space utilization, memory allocation, network throughput, packet loss, etc.) from VNFs running as containers.

- **Virtual Machines**

Monitoring data related to virtual machine instances may become available through a gamut of open source tools, such as Nagios, Zabbix, collectd, etc. Given the wide number of plugins supported by these tools, they are good candidates for adoption in the SONATA monitoring system.

- **SDN controller**

SONATA targets SDN technology in order to take advantage of its unique characteristics, in particular related to VNF service deployment. Thus, integrating network data from the SDN controller will be highly beneficial for the project.

- **OpenStack components**

As mentioned earlier, SONATA monitoring system can also support other SONATA plugins. In this perspective, the collection of processing, storage and networking resources available in SONATA infrastructure is of high importance for the developer in order to decide where to deploy his service.

- **Metrics related to specific VNF applications**

Last but not least, VNFs might include self-monitoring mechanisms to collect data specific to the provided service, such as an instantiation of a firewall or a vCPE. Thus, the SONATA monitoring system must be able to collect and process such information by providing interfaces that can be easily implemented and integrated by the service developer. The VNF specific metrics along with the required information for collecting and parsing the required information will be referred in the VNFD.

5.4.4 Comparison of available monitoring solutions

In accordance with the state-of-the-art analysis included in Deliverable D2.1 regarding service monitoring, this section provides a thorough look on five existing monitoring solutions that could be adopted by SONATA in the first place, presented in the next Table.It is mentioned from the very beginning that none of these solutions fully satisfy the SONATA requirements and thus proper extensions and adaptations are required, depending on each particular monitoring tool.

Table 5.4: Comparison of available monitoring solutions

	Prometheus	InfluxDB	Telemetry	Monasca	Icinga
RESTful API	Yes	Yes	Yes	Yes	Yes
Authentication	Yes	Yes	Yes	Yes	Yes
Integration with GUI tools	PromDash, Grafana	Grafana	Horizon	Horizon	IcingaWeb
Integration with sources	Collectd, HAProxy, Graphite, SNMP, StatsD, CouchDB, MySQL, RabbitMQ, PostgreSQL, cAdvisor	OpenTSDB, Graphite, collectd	OpenStack modules (Neutron, Swift, Cinder, Nova)	Ceilometer, Nagios, StatsD, MySQL, RabbitMQ	N/A
Federation/ Clustering	Yes	Yes (beta)	Yes	Yes	Yes
Client libraries	Go, Python, Java, Ruby	Go, Java, Python, JavaScript/Node.js, Lisp, Perl, PHP, Ruby	Java, Python	Java, Python	N/A
Pull/Push	Pull/Push	Pull	Pull	Pull (Kafka, Zookeeper)	Pull
Database type(s)	LevelDB (google)	InfluxDB	MongoDB	MySQL, InfluxDB	IDODB, PostgreSQL, MySQL
Alerting	Yes	No	Yes (Aodh)	Yes (Storm)	Yes

	Prometheus	InfluxDB	Telemetry	Monasca	Icinga
Resources required	Small	Small	Large	Large	Large
Installation/ Maintenance/ Configuration complexity	Small	Small	Large	Large	Large
Maturity level	Medium	Medium	High	Medium	High

The main conclusions from the comparison of these tools can be summarized as follows:

- **Prometheus**

Prometheus is an open-source service monitoring system, based on time series database that implements a highly dimensional data model, where time series are identified by a metric name and a set of key-value pairs. Moreover, Prometheus provides a flexible query language, allowing slicing and dicing of collected time series data in order to generate ad-hoc graphs, tables, and alerts, while it is integrated with visualization tools (Grafana and PromDash). Most importantly, Prometheus provides exporters that allow bridging of third-party data into Prometheus, including cAdvisor and collectd in a "pull" fashion, but also supports "push" through an already implemented gateway. Installation and maintenance is quite easy (compared to other tools) and offers alerting capabilities (although in a beta version).

- **InfluxDB**

InfluxDB is not a monitoring tool in the classic sense, but it offers many advantages, including the easy set-up and maintenance, the adoption of a database based on time-series, its integration with open source visualization tools (Grafana) and the existence of plugins to collect data from Docker containers (through cAdvisor) and VM instances (through collectd). The only disadvantage compared to Prometheus is the lack of alerting mechanisms, but it supported by several groups and has a remarkable community of developers.

- **Telemetry**

This project is the continuation of Ceilometer that provides data collected through the OpenStack components (Nova, Neutron, Swift, Cinder, etc). However, Telemetry comes with some disadvantages for the purpose of SONATA. First, it is gathering data only from OpenStack and thus it will narrow the perspective of SONATA to offer a "technology neutral" solution for the forthcoming 5G era. Second, it requires the allocation of large resources, making it difficult to implement it in small cloud environments. Third, it collects huge amount of data, but they are not all related to the SONATA needs.

- **Monasca**

This is another OpenStack project that is based on open source components, such as Kafka, Storm, etc. Just like in the case of Telemetry, Monasca collects data related to virtual machines deployed in an OpenStack environment, but lacks integration with other technologies, making the adoption of Monasca prohibitive for SONATA, given also the complexity of its installation and maintenance.

- **Icinga**

In a nutshell, Icinga is a scalable and extensible monitoring system which checks the availability of infrastructure resources, notifies users of outages and provides extensive Business Intelligence data. The main advantage of this tool is the ability to extend in clusters, offering a scalable monitoring infrastructure, in the cost of the complexity during installation and maintenance. However, another advantage is the provisioning of dynamic notifications and simple object-based configuration.

5.4.5 Justification of the selection of the monitoring tool to be adopted in SONATA

Taking into consideration the comparative analysis described in the previous section, it seems that the optimum choice for SONATA is Prometheus. However, as mentioned above, there are several issues that have to be addressed in order to fully cover the SONATA requirements. These issues will be developed during the project lifetime and will be offered to the community as open source. More specifically:

- Extending integration capabilities related to OpenFlow controller monitoring data
- Extending integration capabilities related to cloud infrastructure information
- Extending current RESTful API implementation
- Adapting and extending alerting capabilities

5.4.6 Monitoring development cycles

5.4.6.1 Monitoring system during SONATA 1st iteration

During the first iteration of SONATA development, the primary target with respect to monitoring will be to ensure that all monitoring data related to the SONATA Use Cases will be properly collected and processed. Moreover, during this iteration the proper configuration, the integration of the required components as well as the operational efficiency on local infrastructure level will be achieved, offering a RESTful API for user access. The general architecture diagram of this work is depicted in Figure 5.14.

Collecting data from several sources

First, it is of utmost importance to collect monitoring data from all interested components. As previously discussed, SONATA will take advantage of the already available Prometheus exporters for virtual machines and containers to collect data by configuring proper monitoring endpoints and pulling data from available HTTP GET commands. In particular, the collection of VM monitoring data will become available by installing the exporter related to collectd monitoring tool. Collectd gathers statistics about the system it is running on and stores or sends this information to external entities. Those statistics can then be used to find current performance bottlenecks (i.e. performance analysis) and predict future system load (i.e. capacity planning). Importantly, collectd comes with over 90 plugins which range from standard cases to very specialized and advanced topics. It provides powerful networking features and is extensible in numerous ways, while its integration with Prometheus is inherently supported.Regarding containers monitoring, SONATA will leverage Prometheus support for cAdvisor. cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters,

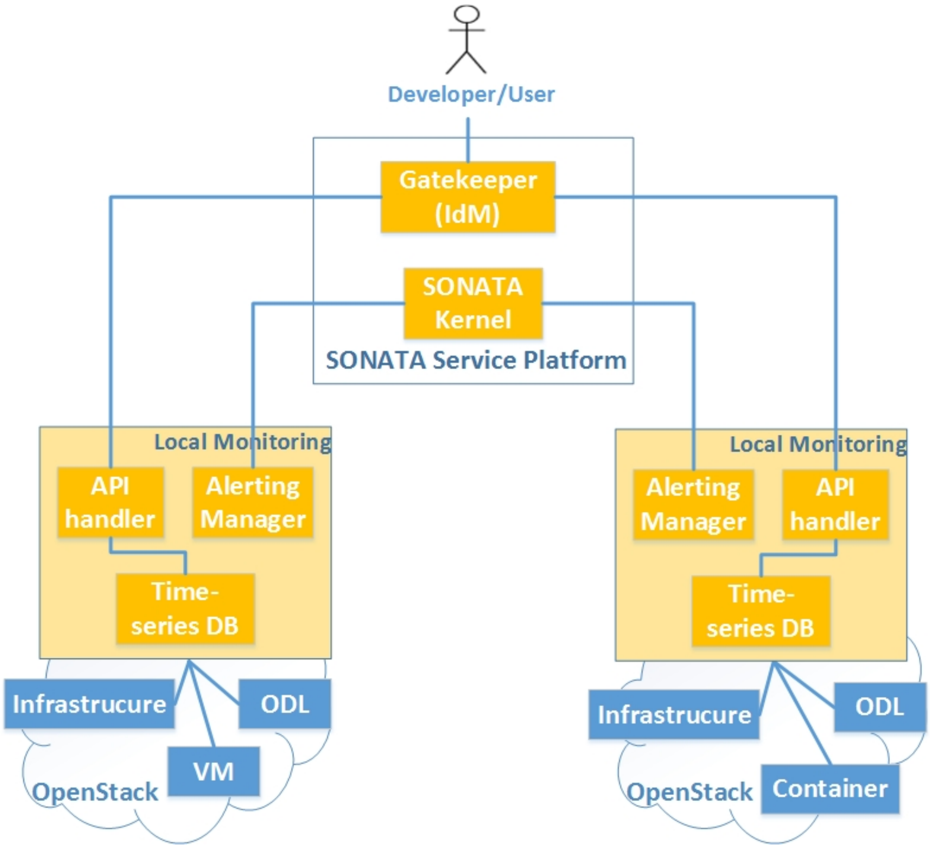

Figure 5.14: General monitoring architecture diagram

historical resource usage, histograms of complete historical resource usage and network statistics. This data is exported by container and machine-wide to Prometheus monitoring tool.Moreover, SONATA developers will be interested in network metrics related to Virtual Switches deployed within their infrastructure. SONATA will make use of OpenFlow API (OpenDayLight, ODL, as depicted in the Figure above) and define an endpoint where monitoring data will be collected. The data will be processed to fit to the Prometheus data model format and pulled through HTTP GET commands. The exporter to be developed to accommodate the integration of OpenFlow controller with Prometheus will an open source software contributed to the developers' community.Finally, in order to support other SONATA plugins, such as Placement & Scaling, monitoring system must collect data regarding the infrastructure resources. Interesting information includes: the number of instances running in the cloud environment, the list of images (VMs or containers) available for instantiation, usage of resources per tenant, available networking characteristics, etc. The collection of such information can be achieved by taking advantage of the OpenStack APIs or by listening on the OpenStack queueing mechanism. Both solutions can be integrated with Prometheus (with the proper extensions and configuration), either through the API or by running a python script to collect data from the OpenStack queueing mechanism (mongoDB). However, the difference between these solutions is that in the former, Prometheus must use the administrative password in order to reach OpenStack API, while in the latter case, a python script can be installed by the administrator himself, allowing pushing data to a Prometheus gateway, without revealing administrative privileges to external parties.The aforementioned data will be stored locally on the Prometheus time-series database and become available to the developers through the Gatekeeper, under the appropriate authentication credentials (Identity Manager, IdM). It is highlighted that in this first development iteration, monitoring system will be instantiated in each infrastructure (locally), as shown in the Figure.

Time-series database

Prometheus has a sophisticated local storage subsystem. For indexes, it uses LevelDB. For the bulk sample data, it has its own custom storage layer, which organizes sample data in chunks of constant size (1024 bytes payload). These chunks are then stored on disk in one file per time series. LevelDB is essentially dealing with data on disk and relies on the disk caches of the operating system for optimal performance. However, it maintains in-memory caches, whose size you can configure for each index via proper flags.

Querying language

Prometheus provides a functional expression language that lets the user select and aggregate time series data in real time. The result of an expression can either be shown as a graph, viewed as tabular data in Prometheus's expression browser, or consumed by external systems via the HTTP API. Arithmetic binary operators. Prometheus supports several binary arithmetic operators (addition, subtraction, multiplication, division, modulo), comparison binary operators (equal, not-equal, greater-than, less-than, greater-or-equal, less-or-equal), logical binary operators (AND, OR) and aggregation operators (sum, min, max, avg, stddev, stdvar, count).

Alerting rules

One of the advantage of Prometheus compared to InfluxDB is the support of defining alerting rules, based on Prometheus expression language expressions and send notifications about firing alerts to an external service. Thus, in SONATA, several pre-defined alerting rules (templates) will be developed according to the Use Case needs. Moreover, a service must be developed in SONATA

that will allow the alerts to be sent to the SONATA Kernel and directly inform the user. Then, the user can utilize the SONATA API to check historical data (time-series database) to further investigate the incident (SLA violation, QoS degradation, etc).

Visualization tools

SONATA will make use of visualization tools already integrated with Prometheus monitoring tool, namely Grafana and PromDash to offer information to the users in a friendly manner. However, extensions will be required in order to cope with API extensions to cover SONATA conditions, while also integration with the Gatekeeper component.

5.4.6.2 Monitoring system during SONATA 2nd iteration

The target of the second round of SONATA monitoring system development is providing a federated solution for the SONATA ecosystem. In this perspective, some changes have to be done with respect to the previously described solution, as depicted in the Figure 5.15, based on the concept that a developer must be able to deploy a VNF service that consists of several components deployed in different SONATA infrastructures. Under this prism, the first differentiation compared to the architecture design described until now is that alerting rules and notifications must be based on monitoring data collected in different infrastructures and thus the decision must be made on a federation level. So, there is a need for installing a Master monitoring system within the SONATA Service Platform, properly configured in order to collect data from all involved infrastructures and make decisions on a federation level.

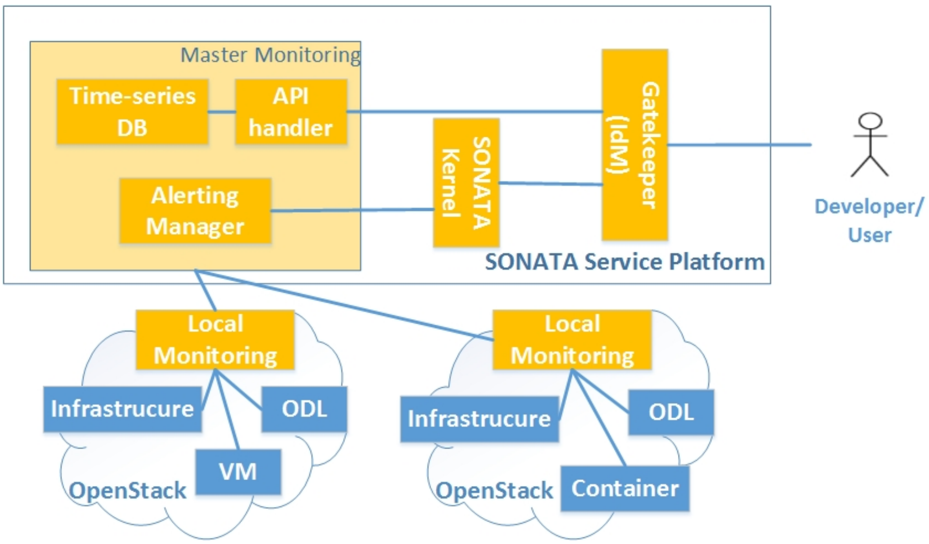

Figure 5.15: Federated monitoring architecture diagram

Hopefully, hierarchical federation allows Prometheus to scale to environments with tens of data centres and millions of nodes. In this use case, the federation topology resembles a tree, with higher-level Prometheus servers collecting aggregated time series data from a larger number of subordinated servers.For example, a setup might consist of many per-datacenter Prometheus servers that collect data in high detail (instance-level drill-down), and a set of global Prometheus servers which collect and store only aggregated data (job-level drill-down) from those local servers. This provides an aggregate global view and detailed local views. However, the proper configuration of

such a hierarchical federation monitoring system is not straightforward, taking into account the differentiations per SONATA infrastructure environment.

Additionally, during the second development iteration, enhanced functionality will be added to the monitoring system, such as automated insertion of customized (per tenant) alerting rules and discovery of new service deployments for automated monitoring.

5.5 Recursive Architectures

A recursive structure can be defined as a design, rule or procedure that can be applied repeatedly. In a network service context, this recursive structure can either be a specific part of a network service or a repeated part of the deployment platform. We will explain both approaches in the next sections. Although different challenges can be thought of (as described in Section 5.5.3), the general idea of reusing existing patterns would reduce complexity and could even add more flexible possibilities for extending the service.

Recursiveness also leads to an easier management of scalability. Monolithic software entities are prone to performance limitations from a certain workload onwards, because any scaling mechanisms were not well implemented. Scaling by delegating parts of the service to multiple instances of the same software block, is a natural way to handle more complex and larger workloads or service graphs. If this recursiveness is taken into account from the beginning of the development, the advantages of this approach will come at a minimal cost.

5.5.1 Recursive Service Definition

Basically, one could summarize the concept of recursive service definition as the ability to build a service out of existing services. A certain service or VNF could scale recursively, meaning that a certain pattern could replace part of itself. As illustrating example, a load balancing pattern can be imagined, where each VNF could be replaced by a load-balancer combining different VNFs of the same type. This leads to a more elastic service definition, where a template, describing the functionality, can be filled by a specific pattern or implementation, depending on the required performance or available infrastructure. This is illustrated in Figure 5.16 (a), showing a service graph consisting of a load-balancer (VNF1) connecting several other VNFs (VNF2,3,4). When VNF4 is getting highly loaded, it gets replaced by the same topology as the original one (VNF1a+VNF2a,3a,4a).

If a certain part of a service graph can be replaced by different patterns, this can offer some advantages:

- Each pattern might be have its own capabilities in terms of performance. Depending on the required workload, a VNF might be replaced by a pattern able to process at higher performance. Similarly, a service or VNF can be decomposed so it can be deployed on the available infrastructure.
- From an orchestrating point of view, above way of using templates in the service graph, can be beneficial for the placement algorithm used by the orchestrator. The success rate, solution quality and/or runtime of such an embedding algorithm benefits from information on both possible scaling or decomposition topologies and available infrastructure [31] [46].

The SONATA project will define in WP3, T3.1 a Service Programming Model. It is the aim to further study there how these templates or recursive service definitions can be practically modeled. As example, it can be seen in Figure 5.16 (b) that certain VNFs in the original service graph on the left, are replaced by another topology on the right. Moreover, as the service graph grows, some

parts could be placed on different infrastructure nodes. This is also shown in Figure 5.16(b) where groups of VNFs are placed in different SONATA platforms. This delegation of (partial) services leads us to the next section describing the recursive implementation of complete service platforms.

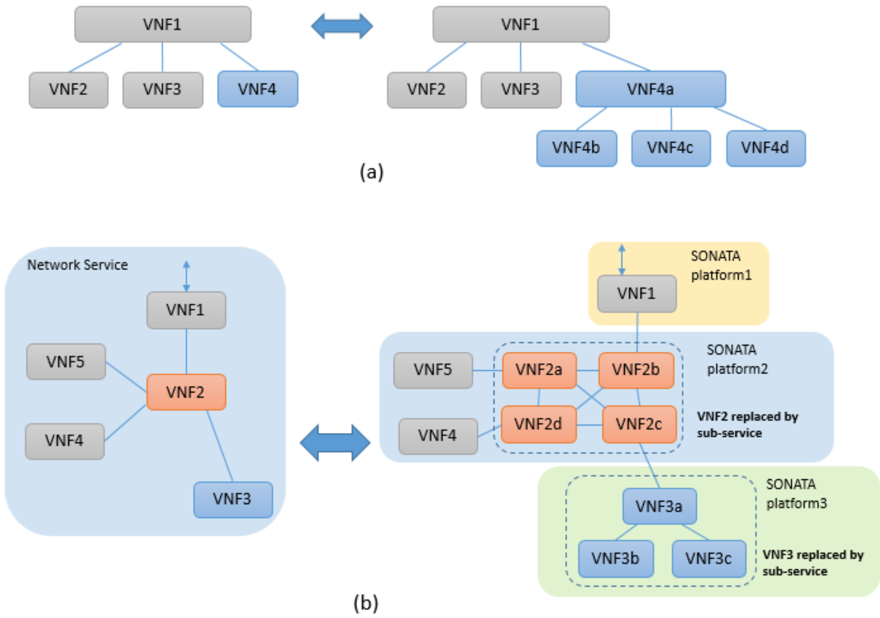

Figure 5.16: Recursive Service Definition (a) the same service graph can be recursively replaced (eg. load balancing topology) (b) some VNFs get replaced by a specific topology

ETSI GS NFV-INF 007 more formally defines recursion as a property of functional blocks: a larger functional block can be created by aggregating a number of a smaller functional block and interconnecting them with a specific topology.

5.5.2 Recursive Service Platform Deployments

A service platform, and by extension most software, can benefit from a recursive architecture. As with a recursive service definition, a recursive structure in the software architecture can be instantiated and linked repeatedly. It improves scalability, as the same instance can be deployed many times, at different places at the same time. But there are some things to be taken into account, as described further in this section.

The main ETSI NFV management and orchestration (MANO) components are the network function virtualization orchestrator (NFVO) for the lifecycle management of the services; VNF managers (VNFMs) for lifecycle management of individual VNFs and virtualized infrastructure managers (VIMs) for controlling and managing compute, storage, and network resources [29]. To make this architecture recursive, the NFVO and VNFM entities can be repeatedly deployed. It can be argued however, up to what level the orchestration functionalities need to be implemented in a recursive service platform. For example, the orchestration capabilities can be implemented as a separate orchestration layer which can be recursively instantiated (such as done in UNIFY). The comparison between the UNIFY and SONATA architecture is further detailed in Section A.1.1. To summarize we can state that the main architectural difference lies in the fact that SONATA follows more the ETSI-MANO model, with clearly defined VNFM and NFVO blocks which can get recursively deployed(see Section 5.5.2.1). UNIFY proposes a recursive architecture by repeating an

Orchestration Layer and keeping the VNFM and NFVO at a higher-level service layer, which is not repeated. The main challenges remain however the same, namely how to translate NFV services in a multi-level recursive platform down to the deployment on the infrastructure. To overcome this, SONATA will use a concept called 'slicing'. A 'slice' can be considered as a set of resources, made available by the service platform to a higher level orchestrator. This multi-slice aspect is handled more detailed in a next chapter (Section 5.6), but it is already highlighted here since it provides the first step to recursiveness in the SONATA platform. From a bottom-up perspective, the typical use-case looks like this: one operator uses its instance of SONATA to create a 'slice' which is offered to another operator which uses its own instance of SONATA to deploy Network Services (NS) on the slice. This use-case also relates to the description in an etsi-draft about SFC Use Cases on Recursive Service Function Chaining [24]. There it is described how a cloud provider could lease (part of) an existing network service to another party. Using the SONATA platform, the cloud provider is able to define in his slice which VNFs are made available to other parties. SONATA will be multi-slice capable, meaning that it can orchestrate a high-level service to different available slices, for various VIM types via specific adapters as shown in Figure 5.17.

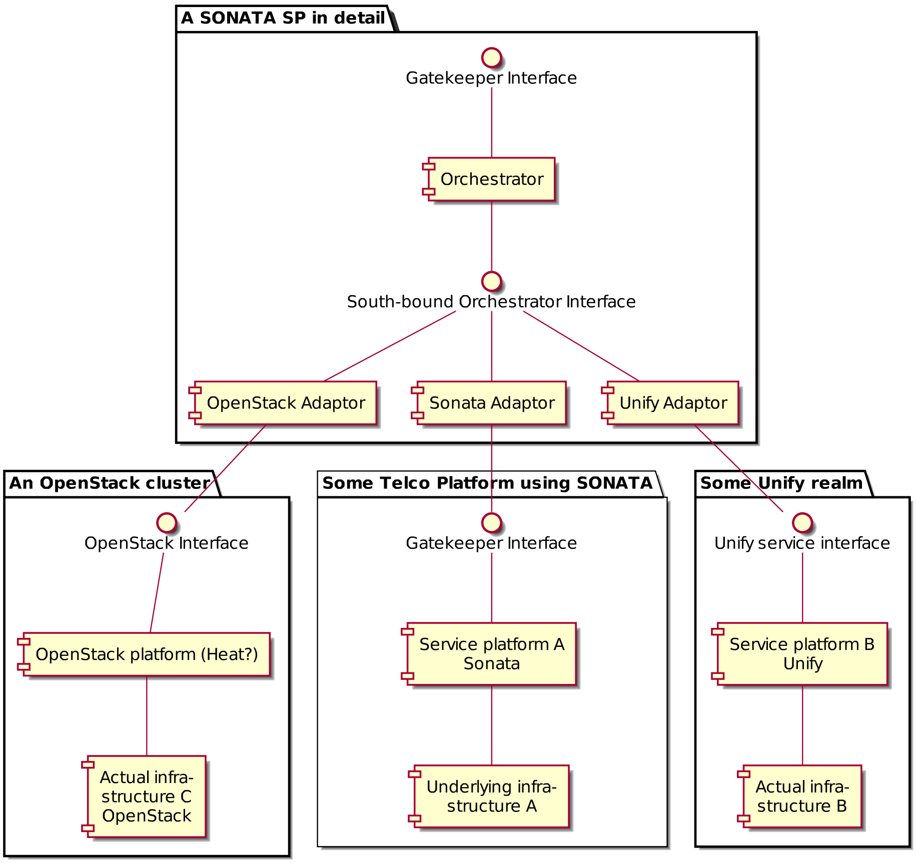

Figure 5.17: SONATA infrastructure slices

As can be seen in this same figure, there is one global orchestrator in the high-level SONATA platform, which delegates (parts of) the service to different slices. Each adapter translates from the orchestration-internal representation into the VIM-specific syntax. The level of recursiveness is now depending on the capabilities of the specific VIM:

- Expectations from a basic (or atomic) VIM interface:

- Accept an atomic network function (e.g., a virtual machine) to execute at a desired location
- Provide information about its topology (with proper annotations, like capacity, etc.)

- Expectations from a more advanced, service-capable VIM interface:
  - This is a VIM that can accept a service description containing a graph of more than one single, atomic network function
  - In addition to the Basic functionality, it accepts a service description to run internally, meaning it also has service orchestration capabilities. Note that service descriptions typically do not require / accept a location specification. This is decided by the VIM internally.

A big challenge in this context is the resource abstraction at different layers. The service will be described by different parameters, depending on which layer in the platform the service is handled. From a top-down perspective, it can be seen that the high-level SONATA platform must be able to interwork between different VIMs and existing OSSs (especially for WAN interconnectivity) This is illustrated in Figure 5.18. The different descriptors throughout the service deployment are these:

Network Service (NS) resource description

- Parameters appropriate to the network service eg: number of simultaneous sessions, packet throughput

VNF level resource description (NS decomposed into VNFs)

- Translation of NS parameters to the number of required VNFs
- Mapped to the capacity of infrastructure resource components: by resource typing, flavor definitions

Infrastructure resources

- Virtual network (VN) resources reasonably well understood (Bit rate, packet throughput, many models of traffic management)
- Virtual machine (VM) resources have large number of potentially sensitive parameters (even down to details of CPU cache, specific instructions, SR-IOV, etc)

As depicted in the figure above, SONATA will start by focusing on the highlighted, lower part of the service deployment. Starting from a basic VIM interface for the SONATA platform, a virtualized or abstracted resource view will be sent upwards, describing the specific SONATA slice, made available to a higher level SONATA orchestrator. This is also further explained in Section 5.6. Further research remains necessary to accomplish also the upper layer of Figure 5.18. The challenge is how to break down a service description into sub-services that can be deployed to different infrastructures. Once sub-services are defined, a more advanced, service-capable VIM interface could orchestrate and deploy this further.

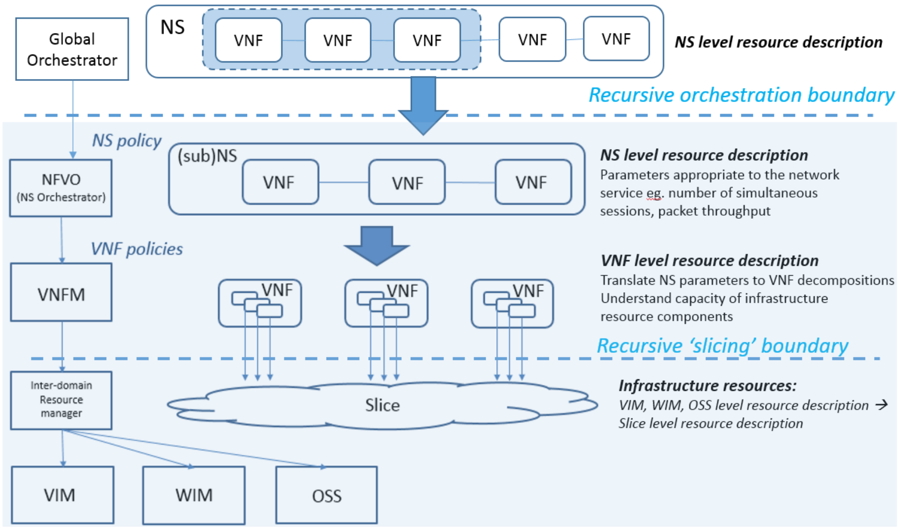

Figure 5.18: SONATA recursiveness

5.5.2.1 SONATA Recursive Architecture

The gatekeeper module is SONATA's main entry point for service packages or service requests. Along with authentication and verification functionalities, it also hands over the new service request to the dedicated placement plugin. Taking into account the slicing capabilities explained in above section, the gatekeeper can also act as an Infrastructure Endpoint, meaning it is able to send an abstract resource view of the available infrastructure in its own SONATA platform. Similarly to other infrastructure adapters, a SONATA adapter could serve as interface to a recursively deployed SONATA platform by a different infrastructure provider. This interface can be instantiated dynamically when another SONATA platform is detected. The placement plugin takes any received infrastructure abstraction into consideration in its algorithm. Through the SONATA infrastructure adapter, it can send a partial NS or VNF to the gatekeeper of another SONATA platform. Note that this could be done in the same native package format as the original service request. In a recursive SONATA hierarchy, there is thus :

- Northbound interface from SONATA through the Gatekeeper
- Southbound interface from SONATA through an infrastructure adapter plugin

This process is shown in Figure 5.19.

5.5.3 Challenges in Implementing Recursiveness

The idea of recursive service platforms is not new. The UNIFY project also proposes a service platform with a recursive hierarchical architecture [45]. As described in previous section, SONATA will build further upon this concept and provide recursive mechanisms in both the service graph and the service platform. When it comes to service development, different patterns to decompose a NF may be developed and dynamically used inside higher-level service descriptors which use this NF. The SONATA SDK should provide access to these catalogs. Regarding service deployment via the SONATA platform, recursive orchestration may add flexibility and new business opportunities to lease and offer network, storage and compute infrastructure. Different slices or sub-sets of the

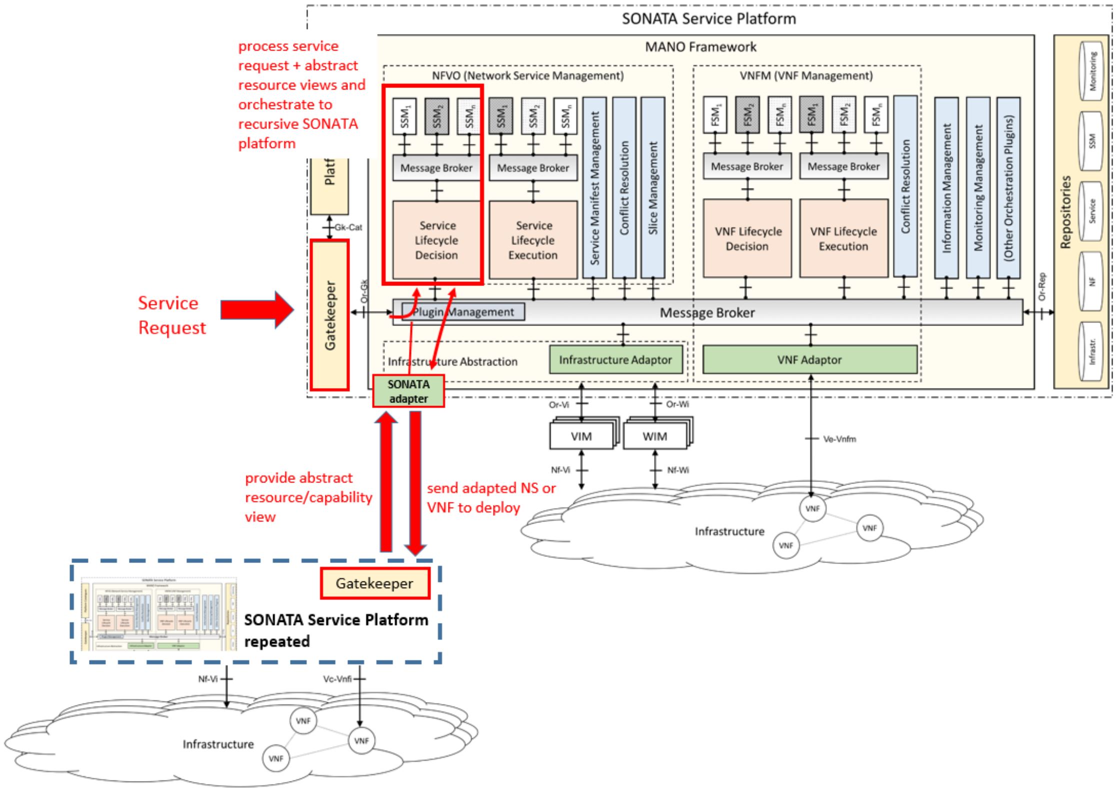

Figure 5.19: SONATA in a recursive architecture

infrastructure node's resources could be brought under the management of a separate SONATA platform instance. Each SONATA platform instance could then offer a dedicated virtualized view of its available infrastructure it wants to lease to a higher-level SONATA platform instance. Each instance is managed by a different stakeholder, and thus different configuration parameters or policies can be applied. The higher-level SONATA platform can then recursively orchestrate or delegate a network service in a distributed way to different underlying platforms. Ideally, different adapters would make it possible to delegate (parts of) the service also to other platforms such as Unify, T-Nova or OpenStack.

A standard describing a complete network service in a generic way, adoptable by all platforms is still a topic of ongoing efforts. Therefore specific development is necessary to create different adapters to connect various platforms to SONATA. Recursive orchestration between SONATA platforms would also require that the SONATA adapter can communicate an abstract or virtualized infrastructure view to the service platform above. In the SONATA architecture, it is the gatekeeper must be able to present this view and it must be generated somewhere in the platform. This will be further worked out in WP4. Ideally, the lower-level infrastructure does not even know that it works for another SONATA instance above. As generic approach, the underlying SONATA could represent its available infrastructure upwards as a single node / "big switch" abstraction. However, we should be aware that all such approaches will lose quality of solutions, as 'divide and conquer' comes for the price of not finding all possible solutions across such boundaries. (Comparable to issues in situations with multiple path computation elements in MPLS, multiple SDN controllers, etc.)

Adapters translate from the orchestration-internal representations into the VIM-specific syntax. They are mostly syntactic tools. The infrastructure abstraction needs to be consistent with an

infrastructure service offered by a hosting provider, for example as a slice. Related to resource abstraction, we suggest to use a set of VM types which the slice resources meet as a minimum and a VNF can be verified against, by using of forms of resource typing or various flavor definitions to specify important details of a VNF. The VNF developer is responsible for making a model that translates, e.g. amount of subscribers (= service-level KPIs) to resource level KPIs.

5.5.3.1 Elasticity in Recursive Platforms

A service platform should enable VNF developers to autonomously control their resource needs. Or in a broader sense, the VNF should at least be able to dynamically change its higher-level KPI requirements which the service platforms' orchestrator must translate to the needed HW resources. This means that elastic control may indeed be VNF or service-specific and may be changed with updates by the developers themselves. Adding this to the recursively hierarchical structure of the service platform architecture, resource control requests may be handled locally or closest in the hierarchy to the actual execution. SONATA can support this by supporting VNF or service-specific plugins called SSMs.

Consider the service depicted in Figure 5.20:

- service S, defined in terms of services T and U
- T contains network functions T1, T2, T3
- U contains network functions U1, U2
- S, T and U each have their own SSM, say, for placement and scaling calculations
  - SSM S only sees the high-level graph: it knows about two nodes T and U upon which it operates.

If the SONATA Platform controls two infrastructures and service T and service U are deployed unto a different infrastructure, the question arises what to do with SSM T and SSM U, as illustrated in Figure 5.21. Two options exist when SSMs, as part of the service package to be deployed, need to be placed:

1. The Infrastructures are basic, like Figure 5.21.
2. The Infrastructures are service-capable, like Figure 5.22.(Assuming that this will only be possible towards underlying SONATA service platforms.)

Apart from where the SSM is placed, it is clear that any service specific elasticity actions (related to scaling and placement) needs to be programmed inside it. The SSMs will be able to elastically change and scale a (partial) service graph or VNF. During the SONATA project this will be handled by different work groups. WP3 will provide the SDK to develop and debug such SSM functionalities related to scaling and placement. WP4 will define ways how the SSMs can be deployed, also when the platform is used in a recursive way.

5.6 Slicing Support

5G networks are expected to have to support a variety of vertical domains such as automotive, public safety, industrial networks, or telecom operators. Each of these domains comes with quite disparate requirements in terms of security, resilience, latency, flexibility, traffic load, etc. The right balance

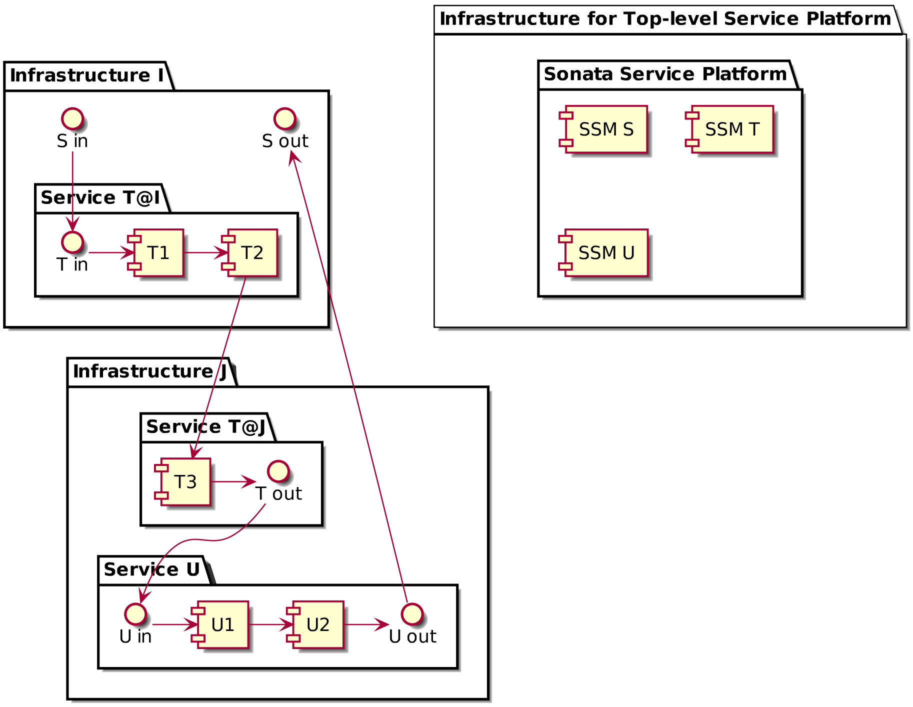

Figure 5.20: Service S, defined in terms of services T and U

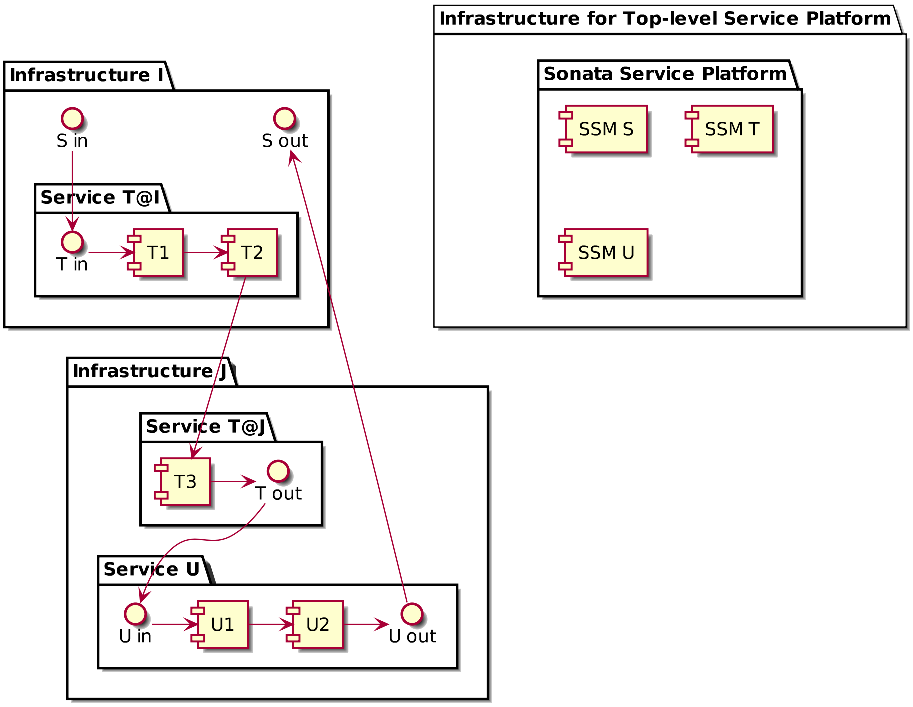

Figure 5.21: placement depending on the capabilities of the infrastructure: SSMs are kept in the high-level SONATA platform with basic, non-service capable infrastructures

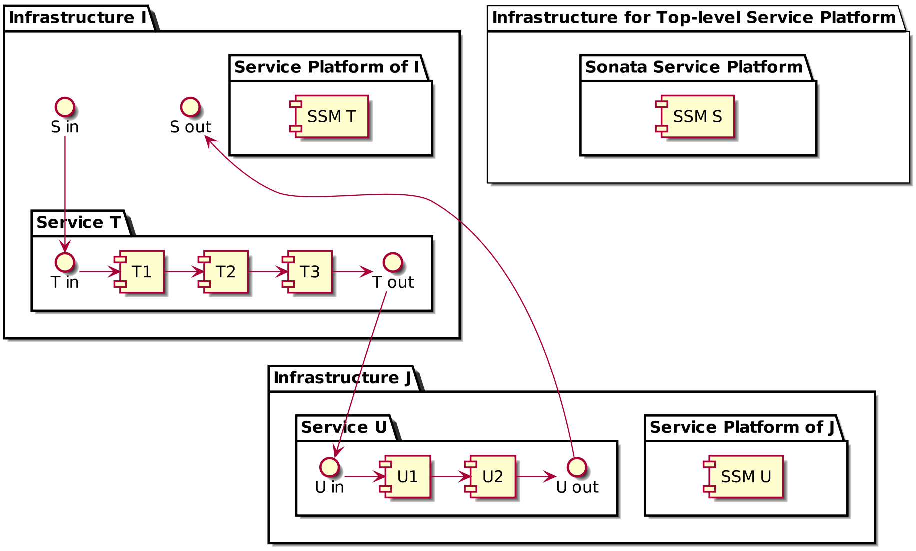

Figure 5.22: placement depending on the capabilities of the infrastructure: SSMs are delegated to the service-capable infrastructures

between a one-size-fits-all network architecture and separate architectures per vertical domain is generally agreed to be achievable with network slicing. This means that the network is partitioned into a set of logically isolated networks, each of which can host different services, be configured differently, support different performance KPIs, and so on. A slice is conceptually a subset of the available compute, storage, memory and network resources in the physical infrastructure. As such, a slice - beyond separating heterogeneous vertical domains - is also a natural concept to separate multiple tenants which jointly operate on the same physical resources. A tenant can obtain a slice from the IaaS provider and then operate his services freely within the resource boundaries of his slice.

In order to perform slice management, Sonata's architecture includes a Slice Manager. This plugin, which is connected to the Message Broker, can be accessed from outside the Service Platform via the Gatekeeper, e.g. via a self-service portal. Using that, a tenant can manually request a slice with certain properties or modify the resource boundaries of an existing slice. Practically speaking, a slice is a virtual topology graph with nodes and links between them. Since Sonata supports multi-site infrastructures, a slice's nodes can be explicitly located in different geographical sites. This is important for use cases such as Content Delivery Networks, where the whole point is to have caches in various geographic locations such as to serve users from a location that is in close proximity to them.

In terms of orchestrating multiple slices and the services within those, there are two fundamental models, both of which are supported by Sonata. The first - and simpler - model is for Sonata to only consider slices themselves (as special incarnations of network services) and leaving the orchestration of services within the slices to the slice owner. The slice owner could, for example, operate his own Sonata instance for managing his own slices, leading to recursive deployments of Sonata instances. The second - more advanced - model is for Sonata to manage both slices and their services within. This avoids duplicating the resource consumption for nested Sonata instances by the tenants. It also means that tenants can make use of the IaaS provider's (i.e. an existing) Sonata Service Platform to manage their services instead of having to deploy and operate a management system of their own. Service providers may be domain experts in terms of the service they offer (e.g. a public

safety service in a smart city), but they may not be cloud and networking experts and therefore operating their own service management and orchestration system may be a significant technical challenge and organizational overhead for them. In that sense, if the IaaS provider's Sonata Service Platform can not only offer easy slice management but also service management at the same time, this offers a much more attractive service deployment environment for 3rd party service providers.

In the following, we will briefly explain both slicing models in more detail. In view of the resulting deployment scenario of Sonata's Service Platform, we call the first model "nested" and the second model "flat". In the first model, Sonata instances are stacked on top of each other, but the managed entities are flat, namely either services or black-box slices. In the second model, the Sonata instance is "flat", i.e. without recursion, but now the managed entities are nested: slices contain services, and both need to be managed at the same time.

5.6.1 Nested Slice Management

In the nested slice management model, the Slice Manager will present the various slices to the Service Lifecycle Decision plugin, which is responsible for global service orchestration. To the Service Lifecycle Decision plugin, a slice may look like yet another network service (which also consists of resource-consuming nodes and links). The Decision plugin in this model considers slices' nodes and links to be atomic entities, it is not concerned with orchestrating the actual network services which are run inside a slice. The latter is here the responsibility of the slice owner (= tenant), who can operate his own Sonata Service Platform within his slice for this purpose, leading to a stacked or recursive deployment of Sonata Service Platforms. This case is depicted in the following diagram.

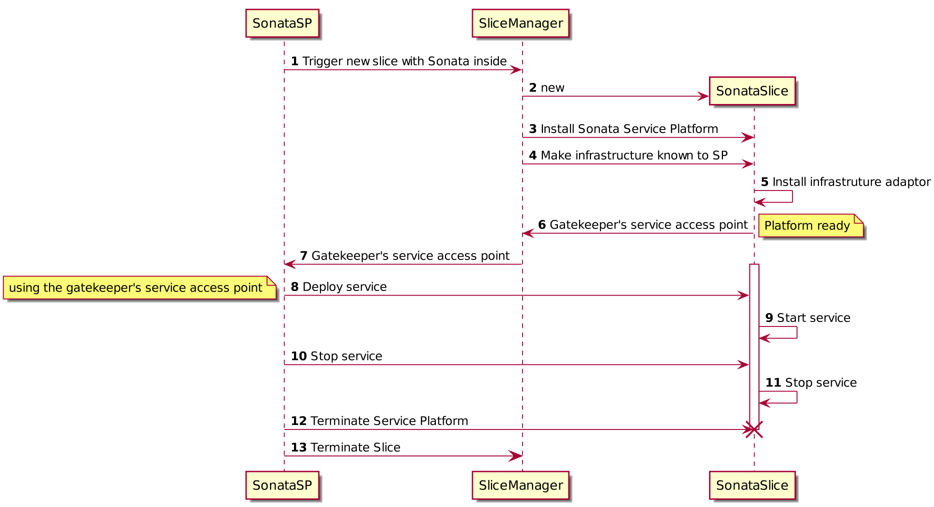

Figure 5.23: Nested slice management

In addition to manual slice management, the latter can also be triggered via policy-based automation. It is possible, for example, for the "inner", tenant-specific Sonata instance to automatically request upgrading its slice in the "outer" IaaS provider's Sonata instance whenever the tenant's services were scaled up or out. Analogously, slices can be automatically shrunk when the slice's services are scaled in or down. That way, the resource requirements of slices at the IaaS level can be made to closely follow the varying requirements of the sum of services in the tenant's slice on the tenant level, e.g. based on some safety margin. This allows a flexible and efficient implementation of the cloud world's "pay-as-you-grow" paradigm. Such automated slice management is one example

for the usefulness of customizable Service Lifecycle Execution logic. Once, for instance, the decision to scale up a particular network service has been made by the Service Lifecycle Decision plugin (potentially customized by a tenant's SSM), the execution of this decision is delegated to the Service Lifecycle Execution plugin. The same tenant's SSM for the Service Lifecycle Execution plugin could then trigger the Slice Manager of it's Sonata instance to automatically request modification of the tenant's slice via the Gatekeeper of the underlying IaaS provider's Sonata instance.

5.6.2 Flat Slice Management

In the flat slice management model, a single Sonata instance manages both slices and the services within the slices. The following diagram illustrates how slices and services are set up in this model. Basically, if a User A requests a new slice, he will do so via Sonata's Gatekeeper, which will in turn forward the request to the Slice Manager. The new slice will appear as a new virtual infrastructure. Service A of User A will then be deployed into User A's slice. The same is true for User B. Subsequent services (e.g. service B2 in step 15) will then be deployed into the same existing slices of the respective tenants.

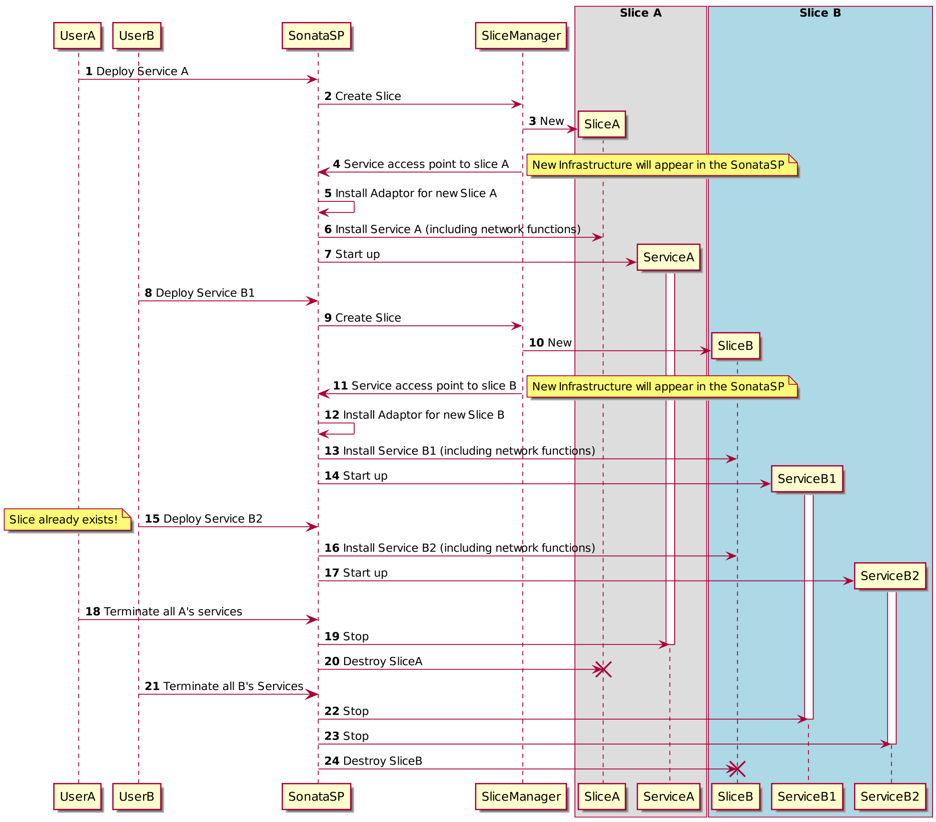

Figure 5.24: Flat slice management

5.6.3 Performance Isolation

Slices are an intuitive way of separating services belonging to one tenant from those belonging to another tenant. An important question is how much performance isolation slices are expected to guarantee to their tenants. Looking into the wider industry, there seem to be two dominant schools of thought in this regard. The first one is the traditional cloud community. In the cloud world,

a key paradigm is the illusion of unlimited resources. According to this school, cloud users (i.e. service providers on top of Sonata) should neither know nor care about the resource situation on the infrastructure level. Resources are considered to be elastic, which means that whenever users require more resources, the cloud platform will grant them. On the physical infrastructure level, this could lead to resource congestion, if the granted virtual resources are overbooking the physically available resources too much. The second school of thought argues that such elastic "illusionary" resource management can never allow for predictable service quality. Especially carriers, who want to run their communication network (e.g. an LTE virtual Evolved Packet Core) on top of a cloud platform, belong to this school. In their view, resources need to be allocated in a dedicated way, leading to more predictable performance guarantees. A way to implement such dedicated resource allocation would be via CPU pinning, for example. Obviously, this approach loses statistical multiplexing opportunities, as unused resources cannot be reused by other services. However, for certain service providers, this seems to be the only viable way of running high-value services on top of a generic cloud infrastructure.

In Sonata, we will support both approaches: the "cloud mode" with its illusion of unlimited and elastic resources, as well as the "exclusive mode" with per-service resource guarantees. In addition, we foresee a third mode of operation, which is somewhere between the two previous extremes. In this mode, the Sonata Service Platform guarantees that only a single slice (i.e. a single tenant) is hosted on an individual physical node. This prevents slices from multiple tenants to unpredictably interfere with one another. On the other hand, it allows a tenant to manage services within his slice himself with some level of statistical multiplexing being possible. This way, the tenant can individually decide how to protect his premium services from becoming resource-congested, prioritizing some services over others. Supporting all three modes of operation gives potential service operators on top of the Sonata platform large degrees of freedom and flexibility and should support all possible requirements in terms of performance isolation between tenants and services. The Sonata platform operator still has some flexibility to perform resource optimization to maximize asset utilization and minimize additional investment. Obviously, the different isolation modes can be coupled with different charging models, giving Sonata platform operators additional revenue opportunities for demanding premium services of their users.

5.7 DevOps

DevOps, which is a clipped compound of **Dev**elopment and **Op**ration**s**, emphasizes the collaboration and communication of both software developers and the operators. The DevOps movement started around 2007 ([21]) with a project that needed close cooperation between developers and operations in a complex migration task to succeed ([32]). At the same time, the idea of continuously integrating the code that was being produced started to become popular, though it roughly left the *developers* side, towards the *operations*.

The big difference of DevOps when compared to a more traditional is a strong *emphasis on people and culture*, improving collaboration between operations and development teams. The key to this is a dynamic and programmable infrastructure, on which automation and testing tools, before seen only on the developer's side, can be used, dramatically shortening times to deploy newly developed features.

In the remaining of this section, we describe the multiple technologies that have evolved in parallel and how they have converged into making DevOps possible.

5.7.1 Infrastructure Virtualization

If only one fact could be chosen as the DevOps trigger, it would probably be the *virtualization of the infrastructure*.

Just seeing the once leading times for buying the supporting hardware for a new project going down from ninety days or similar to mere minutes or even seconds, just pushed the remaining pillars of the whole DevOps practice. Suddenly, the cost of just having the needed infrastructure there was so low that tests could be executed in machines that could just be created and destroyed efficiently.

The NFV and SDN movement has clearly helped this virtualization trend.

Of course that there are still some areas for which virtualization does not help much. These are the specific hardware areas, either because its revolutionary performance needed. But even in these areas, as that specific starts to become more and more common, there will be more and more data centres having that hardware virtualized and able to support those functions.

But many hurdles still stand in this kind of processes. In many organizations, with several layers of approval, this process can still take much longer than the time the actual infrastructure is provisioned and made available to the requesters.

5.7.2 Infrastructure as Code

When treated as code, infrastructure can benefit from the tools and techniques that development also benefits from, like *version control*, *code analysis*, *tests*, etc.

Infrastructure as Code principles are ([33]):

- **Reproducibility:** any element of an infrastructure should be built without spending a significant effort (there are no significant decisions to make, like which software and versions to install on the server, how to choose a hostname, etc., about how to rebuild the thing) and reliably (decisions such as the ones mentioned before should be simply capture in scripts that can be run the needed number of times, to obtain the required infrastructure);

- **Consistency:** servers playing similar roles (e.g., web servers) should be as identical as possible;

- **Repeatability:** being 'reproducible' allows for the endless repeatability of the server definition;

- **Disposability:** as the 'cattle' principle mentioned before, one should assume that any infrastructure element can be destroyed at any time, without notice, whether because the supporting hardware have failed, because something have failed miserably, or for any other reason;

- **Small changes:** one aspect that is often ignored when using DevOps is that everybody's work is eased when the whole work is split into smaller changes. This implies a careful split of the whole into significant chunks, each one delivering an acceptable value to the 'customer'/product owner;

- **Version everything:** being code, infrastructure specification and configuration should be easily versioned, leading to traceability between every change in that infrastructure and the code running on top of it;

- **Service continuity:** for any service hosted on the infrastructure, it is expected that lite infrastructure failures, e.g., one infrastructure fails completely, does not lead to service interruption. This can only be achieved with adequate application architectures, ones that are not monolithic;
- **Self-testing systems:** when automated testing is the core of the development process, it easily spreads also to the infrastructure specification and execution, as code. The flexibility provided by Infrastructure as Code further supports the adoption of automated tests techniques, for which different environments can be easily setup and teared down;
- **Self-documenting systems:** relevant documentation must be generated and maintained, which is often a costly and hard to execute task. The consequences of this is that documentation quickly drifts from the existing and deployed systems. Documentation should be an as automated as possible task, using cooperation tools such as wikis, for the final and human touch (by using page templates, for example);

This approach to infrastructure management is not easily accepted in many fields. For example, *monitoring* is usually connected to a specific resource, and its id: if that resource is teared down and replaced by another executing exactly the same function, it is usually not easy for monitoring tools to connect both resources and show a unified portrait.

5.7.3 Configuration Management

Configuration management, also described in deliverable D5.1 is another DevOps pillar, designed to eliminate error and inconsistency by managing system changes made during development, production, and subsequent modification. Provision environments, deployment of applications, infrastructure maintenance are critical and complex tasks that traditionally were done by hand. In the virtualization and NFV era the above tools play serious role as a facilitator of the automation within the DevOps cycle.

5.7.4 Automated Tests

Since tests do delivered code have to be executed to guarantee its quality and we want to deliver frequently, executing tests manually is simply not feasible. The solution to this is to write code that tests code and executing that test code as frequently as possible. This is what is called *automated tests* (or *continuous tests*, when they are automatically triggered by each change in code).

Writing and maintaining these automated tests obviously have an extra cost, that is hopefully much less than the gains from being able to repeat the execution of those tests *at anytime* (i.e., *unattended*), any number of times. This is specially important in environments where automation is a must, and quality of the whole code base, both of the changes that are introduced and of the existing base, is to be assured. In a DevOps environment, it is only when the costs of writing and maintaining automated tests are higher than executing them manually, e.g., when those tests are rarely executed or involve very specific and costly components, that manually executed tests are considered

There are several kinds of automated tests, from the lowest level ones (*Unit Tests*) to the more broader in scope (e.g., *Functional Tests*), executed in different times of the whole life-cycle of the code base.

Automated Tests imply also some kind of management of *Test Environments*, which might need to be recreated for each test run, specially for the *Integration* and *Functional Tests* (due to its broader scope). Managing *Test Data*, i.e., maintaining or generating data for the tests to be executed, is

also a crucial activity in *Automated Tests*. Many solutions exist that address these two problems, but they depend on the technology stack used.

5.7.5 Continuous Build, Integration and Delivery/Deployment

Successfully tested code must be (automatically) *built*, depending on the technology stack used (programming languages, frameworks, etc.), as soon as it becomes available. Depending on each specific case, building might imply linking with libraries, accessing services, etc., for which more or less complex validations have to be made. Build may also be done only after *integration* of the new piece of software into the existing code base. Again, this depends on the specific case and specific programming languages and frameworks used.

Newly and successfully integrated code must then the *delivered* (or *deployed*, depending on the specific case). Delivering/deploying new features usually make sense if done *without service interruption*, unless the service is not in itself continuous (e.g., when a download has to be manually triggered by the user). This *continuous service* also helps when something goes terribly wrong with the delivery or deployment of the most recent version: *rollback* to the previous version(s) is usually very simple, given the mechanisms of continuously deliver new versions. Another possible approach might also be a new version that is made available to only a *subset of the current users* (or only to a limited number of those who start using the service from a predefined moment on). This subset is then extended as the confidence in the new version grows, until all users are affected. When no service interruption is desired, the usual strategy to be used is to let current users finish their sessions (eventually imposing a hard time limit for that to happen) and let new sessions be created using the new version of the service. This is almost never simple in real cases, specially when data is being generated, updated or deleted.

6 Conclusions and Future Work

In this deliverable, we presented a first approach on a high-level architecture for the SONATA system. To this end, we presented the main components, such as the service platform, the software development kit, the catalogues and repositories, and the main interfaces and reference points. This architecture comprises contributions of all SONATA partners and, therefore, reflects consensus among the consortium members on its initial vision. Moreover, it provides the main building blocks for all the related work packages (WP3 and WP4). At the same time, the architecture is flexible enough to be adapted to later leanings based on implementations.

The Sonata architecture has two main parts, namely the service platform and the software development kit. Both parts integrate state of the art results and best practices proposed by industry, R&D projects and standardisation bodies.

The service platform reflects the ETSI NFV reference design and, at the same time, adds extensions to it. Moreover, its design is structured around a flexible message bus system that allows for loosely coupled components. We specified the main components, such as service life cycle, VNF live cycle, and other management components, as well as the related interfaces. In addition, the SONATA platform facilitates multi-tenancy support. That is, SONATA allows slicing of resources that than can be mapped to users exclusively. These slices allow for dedicated and guaranteed resources used by a tenant and therefore support the high quality of service expectations that are expressed especially by telco operators. Similarly, SONATA supports recursive installations where a SONATA instance can run on top of a SONATA instance. Again this addresses typical telco use-cases and simplifies the management and operation of SONATA installations as such.

In addition to the service platform, the software development kit supports SONATA users to develop, manage, and operate virtual network service that run inside the SONATA platform. Thus, the SDK integrates tightly with the service platform using a well-defined API.

In short, the main contributions of this document are:

- Full SONATA functionality presented in an implementation free approach.
- Initial positioning of the SONATA functionality in a wider 5G Networking context.
- A high-level structure of the SONATA architecture incorporating a software development kit and a service platform.
- A pluggable architecture for the service platform based on a message bus system that allows for easy extensibility using loosely coupled plugins and inter-components communications and operability.
- A design for slicing and multi-tenancy support for the service platform that allow for dedicated and guaranteed resources per tenant.
- A design to support recursive installations that allows running a SONATA platform on top of a SONATA platform.
- A design for pluggable service and function specific managers that can be provided by a SONATA user as part of a service package.

- A layout for repositories and catalogues to store SONATA artefacts, such as service descriptors and virtual network function images.
- An architecture of a software development kit that integrates with the service platform and supports development and operations of complex VNFs.
- Interfaces for advanced monitoring and debugging functionalities that integrate with the SONATA service platform and the SONATA software development kit.

Although this document already covers the main aspects of the SONATA architecture, there are some open issues that need to be addressed in the future. Some of the architectural decisions mainly related to software architecture, hosting the SONATA architectural systems have been left open intentionally to allow flexibility. We, for instance, specified the functions of software components and how they have to communicate. Their internal architecture, however, is up to the developers. In this document, non-functional considerations, such as usability, security, and performance considerations, were taken into account only to a limited extend. That is, we incorporated well-known best practices of architectural design with respect to performance, but it is quite challenging to address all relevant aspects at this early point in time. However, since we believe that performance will be a crucial factor for the final SONATA system, we will evaluate and monitor the SONATA performance during the development phase on a regular basis and revise the architecture if we discover any issues here. In general, following the agile development approach, we will refine and amend the early overall architecture based on leanings and feedback from the detailed subsystems design and implementation. In addition, we may want to integrate more existing external components and products, like external network management systems, into the service platform. SONATA's current design is flexible due to using the message bus and additional plugins, but it may require some more work with respect to interfaces.

In short, future work might include:

- Prioritising the SONATA functions.
- Appraising the setting of the SONATA functionality in a wider 5G Networking context.
- Adaptations of the early architecture by following an agile development process.
- Detailed (software-) design of the SONATA components taking into account high availability considerations, performance, DevOps operations, and the hosting environment
- Usability, reliability, performance, supportability, security, safety, resilience, compliance, extensibility, inter-operability, privacy and scalability evaluations of the software platform as a whole.
- Interfaces and adapters to integrate external, possibly complex, systems as SONATA components.

A State of the Art and Related Work

This chapter presents a brief summary and update to the description of State of the Art provided in Deliverable D2.1, now being focused in architectural aspects, and its relation to SONATA architecture when applicable.

It is split in firstly EU-funded collaborative projects, opensource solutions and finally a brief overview of commercial solutions is provided.

A.1 EU-funded Collaborative Projects

A.1.1 UNIFY

UNIFY aims at an architecture to reduce operational costs by removing the need for costly on-site hardware upgrades, taking advantage of Software Defined Networking (SDN) and networking virtualization technologies (NFV). The project envisions an automated, dynamic service creation platform, leveraging a fine-granular service chaining architecture. For this, a service abstraction model will be created and a proper service creation language to enable dynamic and automatic placement of networking, computing and storage components across the infrastructure. Its global orchestrator includes optimization algorithms to ensure optimal placement of elementary service components across the infrastructure.

Both in [45] (section 6.3) and [50] it is argued that ETSI-MANO architecture [29] is not taking into account any infrastructure abstraction which enables automated, recursive resource orchestration. The main arguments are explained below, for further details the reader is referred to the respective documents.

The Open Networking Foundation works on the definition of an SDN architecture [35]. They focus on three layers: data, control, and application plane layers, but also include the traditional management plane and end user systems into their architecture. SDN applications are defined as control plane functions operating over the forwarding abstraction offered by the SDN controller. Since the architecture allows other SDN controllers (clients) to connect to the north of an SDN controller, the architecture is recursive. Therefore, automated network orchestration can be executed in multi-level virtualization environments, as long as resource virtualization and client policies related to resource use can be set. Such recursive automation enables clear separation of roles, responsibilities, information hiding, and scalability. It also provides efficient operations in multi-technology, multi-vendor, multi-domain, or multi-operator environments. There is, however, at least one major difference between the designs of the NFV and SDN architectures: SDN relies on a basic forwarding abstraction, which can be reused recursively for virtualization of topology and forwarding elements, while the NFV framework offers significantly different services on the top (service-level policies) compared to what it consumes at the bottom (HW resources: storage, compute and network resources). Therefore, it is not straightforward, how to offer NFV services in a multi-level hierarchy with the MANO framework. Additionally, a central and generic management entity such as the VNFM or NFVO in the ETSI-MANO architecture, does not allow flexible elastic control that is VNF-specific and may be changed with updates by the developers themselves.

In the UNIFY project, it is believed that with combined abstraction of compute and network resources, all the resource orchestration related functionalities existing distributively in the MANO

framework can be logically centralized. Such architecture can enable automated recursive resource orchestration and domain virtualization similar to the ONF architecture for NFV services. This can be seen in Figure A.1, taken from [45], where the UNIFY architecture is compared with ETSI-MANO and ONF-SDN architectures.

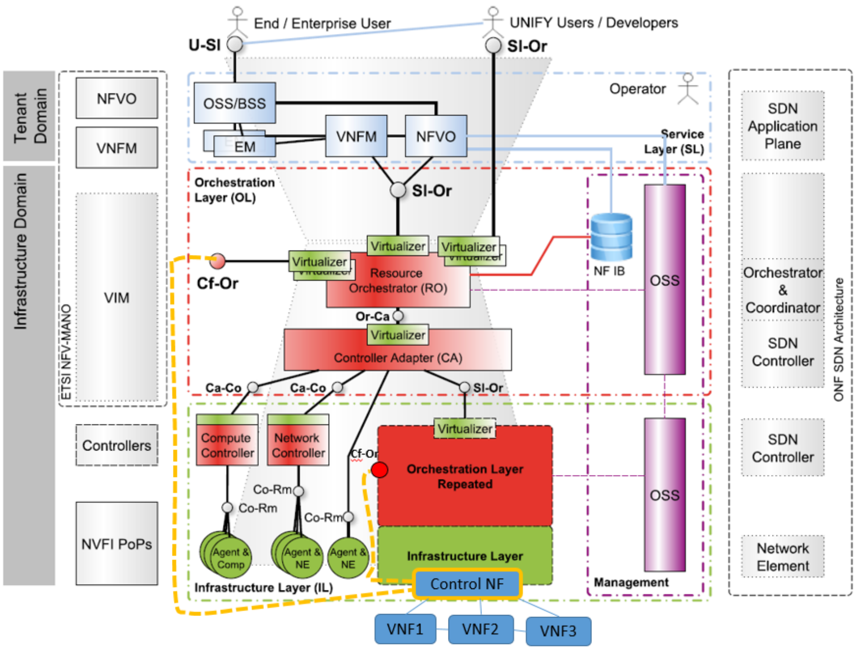

Figure A.1: ETSI NFV, ONF SDN and recursive UNIFY architectures side by side and illustration of the elastic control loop via the Control NF and Cf-Or interface[45]

A.1.1.1 SDK-DevOps

The UNIFY project announced a first version of the SP-DevOps Toolkit at the Network Function Virtualization Research Group (NFVRG) meeting held at IETF94 in Yokohama. The SP-DevOps Toolkit is an assembly of tools and support functions developed in WP4 of UNIFY to support developers and operators to efficiently observe, verify and troubleshoot software-defined infrastructure. Its main purpose is to allow the community at large to experiment with solutions developed in the project to address challenges outlined in [10].

The SP-DevOps toolkit has the following components and are available from the UNIFY website [7]:

- DoubleDecker, a support function that provides scalable communication services for virtual monitoring functions
- EPOXIDE, a troubleshooting framework that enables integrating different tools into a troubleshooting graph
- RateMon, a scalable distributed congestion detector for software-defined networks
- AutoTPG, a verification tool that actively probes the FlowMatch part of OpenFlow rules and is particularly useful in case of wildcarded flow descriptors

A.1.1.2 Monitoring

Furthermore, a monitoring mechanism is implemented called MEASURE. It specifies which measurement functions should be activated, what and where they should measure, how they should be configured, how the measurement results should be aggregated, and what the reactions to the measurements should be. MEASURE is a generic way to define a monitoring intent (or abstract request) from the service definition down to the infrastructure. A monitoring annotation will be defined along with the service model. During instantiation of a service, the abstract service model is then decomposed through the UNIFY architecture, and as the monitoring intents described in the MEASURE annotation are associated to this specific service, it will follow similar decomposition and scoping rules. The work on MEASURE was inspired by the work done in OpenStack Telemetry with Ceilometer. Ceilometer is a centralized system targeted to 1) collecting measurement data from distributed nodes, 2) persistently store the data, and 3) analyse the data and trigger actions based on defined criteria's. MEASURE on the other hand relies on distributed aggregation functionality and does not define how to collect and store the data, rather it 1) describes the configuration of the measurement functions, together with where they should be placed and where they should send results, 2) describes how results should be aggregated, and 3) triggers actions based on defined criteria's. When combined with DoubleDecker, measurement results and triggers can be sent to aggregation points for aggregation and analysis, to databases for persistent storage, and/or to higher layer components such as Resource Orchestrators.

For the detailed description of these tools and how they are used in the UNIFY architecture, the reader is referred to the respective deliverable [43]. Further updates to will be made in a soon to be published document [48].

A.1.1.3 Comparison to SONATA

Although the main idea of a recursive service platform is common between UNIFY and SONATA, the implementation is different. UNIFY places a recursive orchestration functional block in the infrastructure domain. This Orchestration Layer is repeated and implemented in each infrastructure domain. A Cf-Or interface point is made available on each orchestration layer, to which a VNF can connect to trigger any service-specific actions, related to eg. placement or scaling. It is clear that this Cf-Or interface bypasses the VNFM and NFVO, which reside at a higher level in the hierarchy. Basically, from a UNIFY perspective, the service-specific logic is placed inside a VNF which is deployed as part of the service graph on the infrastructure. By connecting via this Cf-Or interface directly to the Orchestrator, an elastic control loop is formed which bypasses the higher level generic VNFM and NFVO. (For further details the reader is referred to the UNIFY deliverables).

This differs from the SONATA architecture showed in Figure 2.4. The SONATA architecture is clearly structured into VNFM and NFVO alike blocks. Also the service-specific logic is placed inside these entities as SSM or FSM modules. The VNFM and NFVO have a generic part, which works as a default for every service, but can be extended with service-specific modules. To make the architecture recursive, the VNFM and NFVO blocks are repeated as a whole and connected via a dedicated adapter, as explained later on in this chapter. This is a different but maybe more obvious approach to a recursive service platform and will be further worked out in WP4.

To summarize, we can point out these differences between the UNIFY and SONATA architecture regarding recursiveness and service-specific logic:

- UNIFY Figure A.1:
  - Recursiveness is implemented as a repeatable Orchestration Layer, which is instantiated in each infrastructure providing entity. In that sense it compares to the 'multi-slice'

concept in SONATA. The functionality of the Orchestrator Layer is however different as it contains no service specific logic. The VNFM and NFVO are considered as high-level generic entities, not service-specific, and reside in the upper Service Layer.

  - Service specific functionality added by developer inside a Control NF, as a dedicated part of the Service Graph, running in the infrastructure. A dedicated interface can feedback from the running Control NF in the service to the service platform. The Control NF is running as close as possible to the other deployed instances that are part of the service, this keeps the monitoring cycle and elastic control loop short. Via an interface (Cf-Or) it can trigger the orchestrator to handle elastic actions like placement or scaling requests. This is done by eg. sending an updated service model via this interface to be deployed by the orchestrator. The updated service model is generated by this Control NF, eg. by analyzing monitoring data or specific logic programmed by the developer.

- SONATA Figure 2.4:
  - Recursiveness is implemented as the repeated deployment of a complete SONATA platform, similar to repeating the VNFM and NFVO functional blocks in the ETSI architecture. Through the gatekeeper, infrastructure abstractions and updated service graphs are communicated.
  - SSMs contain service specific logic, as plugin in the service platform. This functionality is thus running in the platform and not necessarily on the same infrastructure where the service VNFs is running. This also implies that monitoring data needs to propagate to the platform in order to reach the SSMs, where this data is analyzed.

The recursive aspects of the SONATA architecture are further explained in Section 5.5.

A.1.2 T-NOVA

T-NOVA project, **Network Functions as-a-Service over Virtualised Infrastructures**, aims at promoting the Network Functions Virtualisation (NFV), enabling the migration of certain network functionalities, traditionally performed by hardware elements, to virtualized IT infrastructures, where they are deployed as software components. This is done by a novel framework, allowing operators not only to deploy virtualized Network Functions (NFs) for their own needs, but also to offer them to their customers, as value-added services. Virtual network appliances (gateways, proxies, firewalls, transcoders, analyzers etc.) can be provided on-demand as-a-Service, eliminating the need to acquire, install and maintain specialized hardware at customers' premises. For these purposes, T-NOVA designed and implemented a management/orchestration platform for the automated provision, configuration, monitoring and optimization of Network Functions-as-a-Service (NFaaS) over virtualised Network/IT infrastructures. This platform leverages and enhances cloud management architectures for the elastic provision and (re-) allocation of IT resources assigned to the hosting of Network Functions. It also exploits and extends Software Defined Networking platforms for efficient management of the network infrastructure.Furthermore, in order to facilitate the involvement of diverse actors in the NFV scene and attract new market entrants, T-NOVA establishes a "NFV Marketplace", in which network services and Functions by several developers can be published and brokered/traded. Via the Marketplace, customers can browse and select the services and virtual appliances which best match their needs, as well as negotiate the associated SLAs and be charged under various billing models. A novel business case for NFV is thus introduced and promoted.

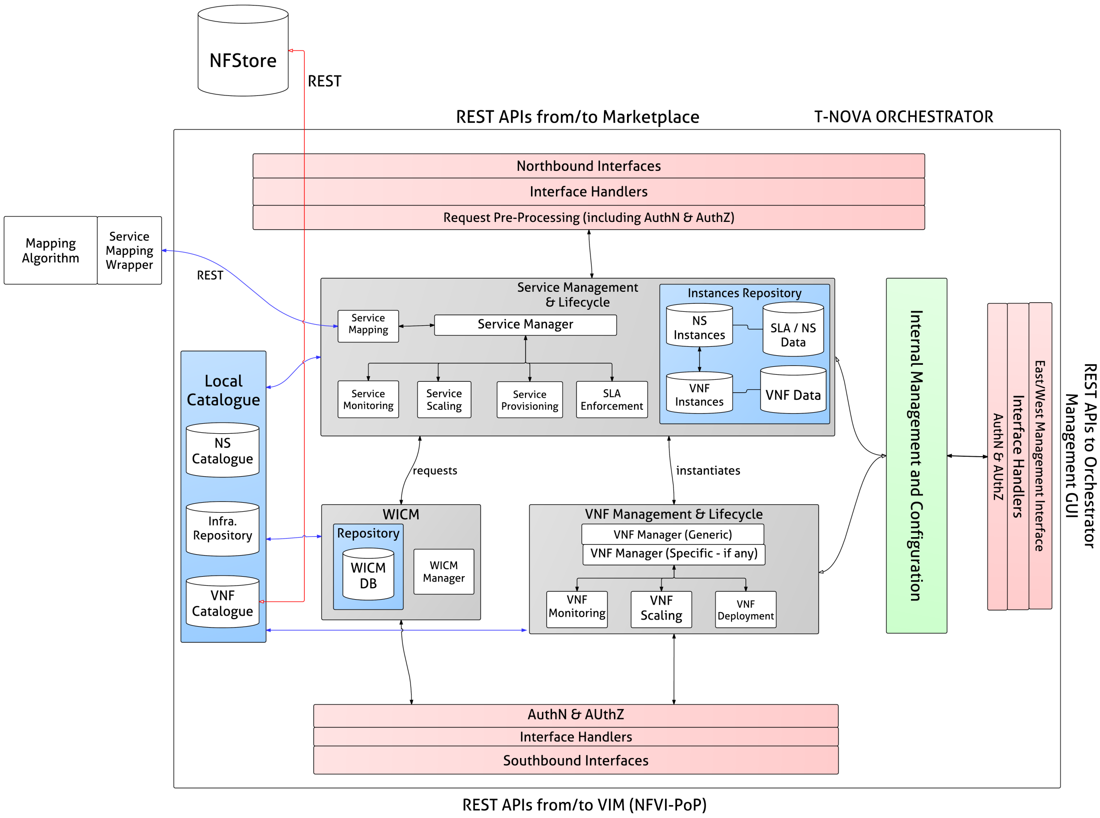

Figure A.2: TeNOR's, T-NOVA's Orceshtrator, architecture (updated from [4])

T-NOVA Orchestrator has been named as TeNOR, which is ETSI-NFV compliant VNF deployment and operation, and whose high level architecture can be seen in Figure A.2 (see [4]).

Highlights of TeNOR architecture are:

1. Features have been split into three main blocks:
   - **Service Management and Life-cycle**, which concentrates all the features at the Network Service level;
   - **VNF Management and Life-cycle**, which concentrates all the features at the Virtual Network Function level;
   - **WIM**, from **Wide-area network Interconnection Management**, which abstracts away all the interactions with the WAN (e.g., communication between VNFs that may leave in different Datacenters, connection to a specific customer's network, etc.);
2. both external and internal interfaces are **REST/JSON**, in a micro-service oriented architecture;
3. this kind of architecture allows TeNOR to provide a **'generic**' VNF Manager, while allowing **'specific**' VNF Managers to be provided together with a (set of) VNFs (with an authentication/authorization scheme to control which functions should or shouldn't be delegated to those 'specific' VNF Managers;
4. VNFs enter TeNOR coming from the **NFStore**, while NSs do it from the *Marketplace*';

A.1.2.1 Service composition and relation to SDK

T-NOVA does not include a proper toolkit or SDK for service programming. In T-NOVA service description is facilitated to the Service Provider by means of the GUI via the Marketplace, were existing network functions offered by several developers can be published and brokered/traded. The Service Provider will compose services upon inheriting the description of the VNFs that will be part of the service. Therefore DevOps is not supported in T-NOVA orchestration platform but services are "on-boarded" once and in a static manner. T-NOVA offers the Service Provider a graphical interface to select (purchase) available VNFs and to combine them in order to build the forwarding graph (VNFFG) to be included in the descriptor (NSD).

A.1.2.2 Comparison to SONATA

Comparing to SONATA (see Figure A.1):

1. SONATA's architecture is much more **fine grained** than T-NOVA's: e.g., **Service Specific Managers** (**SSMs**) are really **Specific VNF Managers**, as described above, with their authentication/authorization scheme concentrated on the Gatekeepr;
2. in SONATA, **placement**, if very specific, is left for the service's SSM. **Conflict resolution** plays a very important role in guaranteeing that every conflict gets solved;
3. both architectures share a VIM/WIM abstraction layer.
4. service programmability as well as DevOps approach are not provided by T-NOVA, in contrast with the SDK that is part of SONATA. T-NOVA interface with Service Providers by means of T-NOVA marketplace is a static GUI that helps in the process of describing a service but neither programmability nor DevOps approach is supported.

A.1.3 NetIDE

The NetIDE project [20] which hast started early 2014 and is ongoing until the end of 2016 tries to create an integrated development environment for portable network applications. The main goal of the project is enabling vendor-independent SDN application execution and support for the end-to-end development lifecycle of SDN based network applications. To do so, three major components are developed. The first one is called *network engine* and is responsible to execute controller independent network applications. The second component is an application repository which provides predefined network applications which can be reused by developers. The last component is an software development kit (SDK) including an Eclipse based integrated development environment (IDE) that interacts with the network engine for full DevOps support [11] [8]. They plan to provide extended tool support for SDN application development, including garbage collector, resource management, model checking and profiling tools. The current release integrates Mininet as lightweight test environment. The general setup of these three components is very similar to SONATA's approach having an *service platform*, *catalogues*, and *SDK*.

A.1.3.1 Development Toolkit

The NetIDE development toolkit consists of a single IDE that integrates all editors and tools needed to develop a NetIDE compatible SDN application. To do so, a SDN application developer has to specify two things. First, a topology model has to be specified that describes the available SDN switches and their interconnections. This is done with a graph model. Second, the developer has to implement the actual SDN application behavior which can be done in a programming language of his choice depending on the target SDN controller platform, e.g., Python or Java. The IDE then automatically maps the application behavior to the topology and generates an output format that can either be executed in the Mininet based emulation environment or on top of NetIDE's network engine platform [36].

The IDE supports all development steps with customized editors, e.g., with a graphical editor to edit topology graphs. Furthermore, it integrates seamlessly with the emulation environment and provides direct control of the underlying Mininet instance through the Eclipse console. This Mininet instance is executed in a VM that runs on the developer's machine and is automatically provisioned by the IDE [37].

A.1.3.2 Comparison to SONATA

The Network Engine developed by NetIDE targets the composition of SDN applications running on SDN controllers. Since it does not contemplate VNFs, it is outside the scope of SONATA and therefore, state-of-the-art components provided by OpenStack should be preferred. Additionally, given the scope of NetIDE Applications, the application repository they plan to release is out-of-scope for SONATA.

The development toolkit provided by NetIDE is an interesting reference for SONATA even though it does not consider virtualized network function and service chain development. Hence, it is not a direct competitor to SONATA but rather a possible starting point to build on. In principle the SONATA SDK could directly be integrated with the NetIDE IDE. In such a scenario the NetIDE tools would support the development of SDN based network applications and the SONATA tools would be used to compose services of virtualized network functions.

A.2 Open Source Initiatives

A.2.1 OpenMANO

OpenMANO is an open source project initiated by Telefonica that aims to provide a practical implementation of the reference architecture for NFV management and orchestration proposed by ETSI NFV ISG, and being enhanced to address wider service orchestration functions. The project is available under the Apache 2.0 license, it was first released in early 2015 and it is currently under active development. The OpenMANO framework is essentially focused on resource orchestration for NFV and consists of three major components: *openvim*, *openmano*, and *openmano-gui*.

The first component is essentially focused on the resource infrastructure orchestration, implementing express EPA (Enhanced Platform Awareness) requirements to provide the functionality of a Virtual Infrastructure Manager (VIM) optimized for virtual network functions and high and predictable performance. Although *openvim* is comparable to other VIMs, like OpenStack, it provides:

- Direct control over SDN controllers by means of specific plugins (currently available for Floodlight and OpenDaylight), aiming at high performance dataplane connectivity.
- A northbound API available to the functional resource orchestration component *openmano* to allocate resources from the underlying infrastructure, by direct requests for the creation, deletion and management of images, flavours, instances and networks.
- A lightweight design that does not require additional agents to be installed on the managed infrastructural nodes.

The functional resource orchestration component itself is controlled by a northbound API, which are currently suitable to be used directly by network administrators via a web-based interface (*openmano-gui*) or by a command line interface (CLI) that eases integration in heterogeneous infrastructures and with legacy network management systems. The functional resource orchestrator is able to manage entire function chains that are called *network scenarios* and that correspond to what ETSI NFV calls network services. These network scenarios consist of several interconnected VNFs and are specified by the function/service developer by means of easy-to-manage YAML/JSON descriptors. It currently supports a basic life-cycle for VNF or scenarios (supporting the following events: define/start/stop/undefine). The OpenMANO framework includes catalogues for both predefined VNFs and entire network scenarios, and infrastructure descriptions carrying EPA information.

As described above, OpenMANO is in the position of being extended to incorporate the higher-layer service orchestration facilities considered in SONATA, and provides support for advance functional resource orchestration able to support high and predictable performance.

A.2.2 OpenBaton

OpenBaton [12] is an open source project by Fraunhofer FOKUS that provides an implementation of the ETSI Management and Orchestration specification. Its main components are a Network Function Virtualisation Orchestrator (NFVO), a generic Virtual Network Function Manager (VNFM) that manages VNF life cycles based on he VNF description, and an SDK comprising a set of libraries that could be used for building a specific VNFM.

The NFVO, which is the main component of OpenBaton, is written in Java using the spring.io framework. To interconnect the NFVO to different VNFMs, OpenBaton relies on the Java Messaging System (JMS). To this end, it uses ActiveMQ [16] as a message broker.

The NFVO is currently using OpenStack as first integrated NFV PoP VIM, supporting dynamic registration of NFV PoPs and deploys in parallel multiple slices one for each tenant, consisting of one or multiple VNFs. Through this functionality the orchestrator provides a multi-tenant environment distributed on top of multiple cloud instances.

A.2.3 OpenStack

OpenStack [39] is an open source project, mainly written in Python, that provides an Infrastructure-as-a-Service solution through a variety of loosely coupled services. Each service offers an API that facilitates the integration. Due to its variety of components, OpenStack today not only provides a pure VIM implementation but spans various parts of the ETSI-NFV architecture [27]. *OpenStack Keystone* [38], for instance, offers authentication and authorization not only to the VIM part, but can be integrated to other services as well. *OpenStack Ceilometer* [42] provides a plugable monitoring infrastructure that consolidates various monitoring information from various sources and makes the available to OpenStack users and other services. *OpenStack Tacker* [40] aims at the management and orchestration functionality described by ETSI-NFV. Thus, we take a closer look at the OpenStack architecture and its design choices.

The overall architecture relies on messages buses to interconnect the various OpenStack components. To this end, OpenStack uses AMQP [34] as messaging technology and an AMQP broker, namely either RabbitMQ [49] or Qpid [14], sits between any two components and allows them to communicate in a loosely coupled fashion. More precisely, OpenStack components use Remote Procedure Calls (RPCs) to communicate to one another; however such a paradigm is built atop the publish/subscribe paradigm. OpenStack implements RPC over AMQP by providing adapter classes which take care of marshaling and unmarshaling messages into function calls. This decouples a service client from the server and also allows for an easy integration of new services or service instantiations in a plugable manner.

Openstack messaging has two modes, i.e. **rpc.cast** that does not wait for any response, and **rpc.call** that waits for results, given there is something to return.

The OpenStack architecture as described above has been proven to be scalable and flexible. Therefore, it could act as a blueprint for the SONATA architecture. In the following, we take a closer look at specific OpenStack Components that potentially could be leveraged or extended to suite some of the SONATA needs.

A.2.3.1 OpenStack Tacker

Tacker is a new OpenStack project that aims at building an open NFV orchestrator with a general purpose VNF manager to deploy and operate virtual network functions on an NFV Platform. It is based on ETSI MANO Architectural Framework and aims at providing full functional stack to orchestrate VNFs end-to-end. Today, Tacker offers features like a *VNF Catalog*, a basic *VNFM Life Cycle Management*, *VNF Configuration Management Framework*, and a *VNF KPI Health Monitoring Framework*. The VNF Catalog makes use of TOSCA for VNF meta-data definition and OpenStack Glance to store an manage the VNF images. The Tacker VNFM Life Cycle Management takes care of instantiation and termination of VMs, self-healing and auto-scaling, and VNF image updates. It also takes care of interfaces to vendor specific element management systems. Like the VNF Catalog, the basic VNFM Life Cycle Management relies on existing OpenStack services and uses OpenStack Heat to start and stop VMs that contain the VNF. Thus, the TOSCA templates are automatically translated to OpenStack Heat templates.

OpenStack Tacker is under heavy development. As of today, several crucial features, such as service function chaining and VNF decomposition, are still missing and under discussion.

A.2.3.2 Heat

Heat is the main project in the OpenStack Orchestration program. It implements an orchestration engine to launch multiple composite cloud applications based on templates in the form of text files that can be treated like code. I works in the following way:

- A Heat template describes the infrastructure for a cloud application in a text file that is readable and writable by humans, and can be checked into version control, diffed, etc'.
- Infrastructure resources that can be described include: servers, floating ips, volumes, security groups, users, etc.
- Heat also provides an autoscaling service that integrates with Ceilometer, so you can include a scaling group as a resource in a template.
- Templates can also specify the relationships between resources (e.g. this volume is connected to this server). This enables Heat to call out to the OpenStack APIs to create all of your infrastructure in the correct order to completely launch your application.
- Heat manages the whole lifecycle of the application - when you need to change your infrastructure, simply modify the template and use it to update your existing stack. Heat knows how to make the necessary changes. It will delete all of the resources when you are finished with the application, too.
- Heat primarily manages infrastructure, but the templates integrate well with software configuration management tools such as Puppet and Chef. The Heat team is working on providing even better integration between infrastructure and software.

A.2.3.3 OpenStack Ceilometer

In recent years, the Ceilometer project became the infrastructure to collect measurements within OpenStack. Its primary targets are monitoring and metering in order to provide data and mechanisms for functionalities such as billing, alarming or capacity planning. However, the framework can be extended to collect data for other needs as well. The OpenStack design is gathered around samples. Every time Ceilometer measures something a sample is generated by one of the Ceilometer components, such as an agent and a pollster, and forwarded to a sample collector via a message bus. The collector is responsible for storing the samples into a database. Moreover, Ceilometer exposes a REST API that allows to execute various reading requests on this data store and to retrieve the measurement data

In order to achieve the required performance, data storage has to be optimized. Thus, Ceilometer uses two separate databases to store events and time series data. Events are stored in an SQL data base and exposed via the Events REST API. Resource meters, basically a list of (timestamp, value) for a given entity, are stored in a Time Series Database as a Service called Gnochi [CITE]. Like in many other OpenStack components, the storage driver is abstracted so it is possible to use whatever technology is available.

A.2.3.4 Murano

The Murano Project introduces an **application catalog** to OpenStack, enabling application developers and cloud administrators to publish various cloud-ready applications in a browsable categorized catalog. Cloud users can then use the catalog to compose reliable application environments with the push of a button. The key goal of the Murano project is to provide UI and API which

allows to compose and deploy composite environments on the Application abstraction level and then manage their lifecycle. The Service should be able to orchestrate complex circular dependent cases in order to setup complete environments with many dependent applications and services.

A.2.3.5 Mistral

Mistral is a **workflow service**. One can describe any process as a set of tasks and task relations and upload such description to Mistral so that it takes care of state management, correct execution order, parallelism, synchronization and high availability. Mistral also provides flexible task scheduling so that we can run a process according to a specified schedule (i.e. every Sunday at 4.00pm) instead of running it immediately.

A.2.3.6 Lessons Learned for SONATA

- OpenStack is practically a compatibility standard for the private cloud market. It is the most conmanly IaaS used by both enterprises and telcos.
- OpenStack's APIs are defector a standard for IaaS APIs and are (almost) aligned with ETSI NFV VIM apis
- All the OpenStack components are loosely coupled using an AMQP message bus to establish communication between the different entities. Thus, OpenStack is highly scalable, very flexible, and allows for an easy, plugable integration of new services. The architecture using messages buses could act as a blueprint for SONATA.

A.2.4 Terraform

Terraform [23] and its ecosystem promote an Infrastucture as Code (IAC) approach. A developer can use this toolset to deploy and manage an immutable infrastructure. Terraform supports Software Defined Networking (SDN) architectures even if it doesn't have any notion of Network Functions (NF). Virtual Network appliances can be provisioned as long as they are shipped as images or containers. SONATA has a deep knowledge about the Virtual Network Functions (VNF) it manipulates. Terraform, in contrary, views them as an another piece of software. It ships with an orchestrator doing automatic dependencies inference. In this context, SONATA has more information to optimise VNFs and manages orchestration too.

For the creation of appliances, Terraform got in its ecosystem a tool promoting Configuration as Code: Packer. Using Packer, a developer can package code, binaries and configuration into deployable artefacts. Freezing a maximum of moving parts at development time makes the infrastructure immutable and decreases the risk of dynamic failures at run-time. The SONATA SDK supports this approach as it incorporates the creation of VNFs in its DevOps loop. To add dynamism and to lower the risk of a tooling lock, Terraform uses Consul. Consul is a distributed and decentralized key/value database. It offers service discovery, configuration and life-cycle orchestration. With it, the developer can add events, triggers and health-checking to manage automatically its appliances at runtime. The SONATA SP will provide equivalent mechanisms but add a layer of provider's abstraction missing in Terraform.

In the end, these three tools make a DevOps loop. SONATA supports such an approach with the combination of its SDK and SP. The main difference resides in SONATA being a complete multi-tenant system with a deep understanding of NFs. Terraform is mainly a command line tool targeting generic appliances. Its architecture is much simpler and extend-able with plugins. Every

resource type is handled by a specific plugin. Each plugin is a separate executable. A collection of several binaries makes the whole Terraform platform.

The Figure A.3 shows how Terraform orchestrates every plugins.

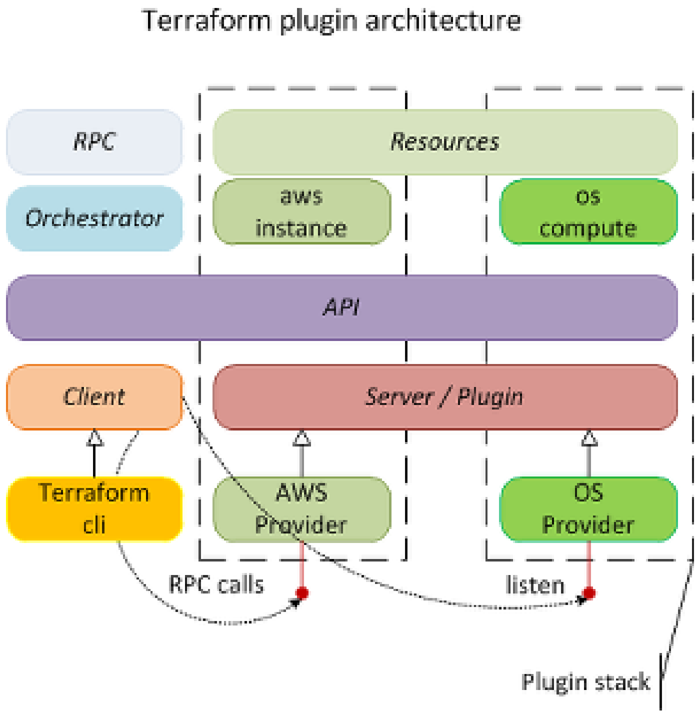

Figure A.3: Terraform plugin architecture

A plugin is a server listening on a random port and managed by the root Terraform executable. Terraform distributes resources to each plugins depending on their types. To ensure the rpc calls's correctness, all the plugins are built using the same *official* library. In fact, the bare skeleton of a plugin is a simple one-liner. The plugin developer must only focus on implementing a simple *CRUD* (Create, Read, Update, Delete) API over the resources it manages.

A.3 Commercial Solutions

Commercial vendors have begun to market solutions for the orchestration layer to accompany their growing NFV portfolios. This first generation of NFVO is based off ETSI MANO specifications, although originate from various paths to their NFV context. Several, for example, are orchestration solutions developed by established network vendors to further expand a larger NFV ecosystem. Others have been extended from existing cloud service orchestration solutions that started before the market for software networks took flight. Regardless of their origin, they offer similar functionality and characteristics when they are confined to the NFVO definition, and larger differentiation seems more apparent when comparing their larger, integrated suite offering beyond the scope of orchestration.

For the scope of D2.2's SotA Annex, a variety of public material has been used to provide a baseline look at the commonalities of a first generation of commercial NFVOs. It should be noted that the material used differs between technical white papers and marketing, varying the level of detail available, and provides only a biased and limited look at their offerings (from the vendor). This is to be expected from proprietary solutions in a competitive market. The sample of

commercial solutions are listed below, in decreasing order of detail available for the initial study, ranging from technical white papers to marketing fact-sheets. The figures referenced in the list can be found at the end of the section.

- HP NFV Director (Figure A.4)
- Amdoc Network Cloud Orchestrator (Figure A.5)
- Oracle Network Service Orchestrator (Figure A.6)
- Ciena BluePlanet NFV Orchestrator
- IBM SmartCloud Orchestrator (Figure A.7)
- NetCracker Orchestration
- Cisco NFV Management and Orchestration (Figure A.8)
- Luxsoft SuperCloud Orchestrator
- Overture Ensemble Service Orchestrtor

All characteristics and diagrams below can be found on the vendor's public website, either through online material or hosted documentation.)

There is a common understanding between commercial solutions that the NFV orchestration concept applies to the complete VNF and network service lifecycles, including onboarding, test and validation, scaling, assurance and maintenance. Vendor marketing material and white papers showcase their upcoming products as holistic solutions for both service and network orchestration, compatible with current ETSI MANO specifications. These solutions cover the orchestration and management of Virtual Network Functions (VNFs), VNF Forwarding Graphs (VNF-FG) and network services (NS).

Service and Network Orchestration: A common distinction is made between service and network orchestration. Service orchestration components are responsible for end-to-end service lifecycle management between virtualized or hybrid (virtual/physical) networks. Support is also included for automated service chaining, policy-based service fulfillment and configuration of VNFs via various protocols. Their network orchestration components are responsible for onboarding multi-vendor VNFs and optimizing them based on performance, latency or security requirements. This includes end-to-end network connectivity for the created and deployed NS.

Interaction with VNFM: Almost universally found in this first generation of NFVO solutions is a generic VNFM by the same vendor. However, material is often quick to present compatibility with a third-party VNFM. Although the separation of NFVO and VNFM components and functionality are a distinction made in ESTI MANO specification, commercial solutions are often integrated together, and not transparent if this is an extension of core NFVO functionality, or in fact two distinct, interacting solutions. An important conclusion, however, is that there has not been a marketed NFVO solution without a generic VNFM included in the offering, although recognition of third-party VNFM support is an important feature.

Interaction with VIM: Solutions tend to advertise multi-VIM support, but such compatibility is limited. The common compromise is OpenStack compatibility plus an additional proprietary VIM (the latter via a suite-based solution). Wider VIM support is sometimes hinted at, but assumed to be for built-to-spec integration for the customer.

Legacy Support: Commercial orchestration solutions, particularly from established vendors, are often citing legacy support for hybrid environments, referring to physical (legacy), software/virtualized

(SDN/NFV) and hybrid network environments. This refers to both VNFs and existing physical network functions.

VNF Catalogs and Third-Party Support: There is a heavy emphasis on an ecosystem approach, with accompanying VNF catalog, and in the majority of cases, support for VNFs from third-party vendors (the most common definition of their marketing "multi-vendor"). The initial VNFs included are often in the form of integrated network services. For example, Amdoc's NFV Services Partner Program also pre-integrates VNFs (also with a deployed NFVI) for packaged services, including vCPE, vEPC and Virtual IMS Core solutions. NEC/Netcracker similarly provides a core VNF portfolio, including a commercially developed vEPC solution. Such packaging is common and extends to other large vendors' ecosystems, including Oracle, Cisco and HP.

Monitoring: Solutions such as HP Director include integrated monitoring, covering both VNF and NS; and for the latter correlating across all VNFs and physical network functions that are part of the NS. Provisioning and monitoring are then coordinated through rules that define manual or autonomous scaling and placement actions, based on KPIs.

Service Modeling: Solutions like NEC/NetCracker's orchestrator supports a DevOps-like approach to service modeling using design environment with automated import of YANG service/resource models and OASIS TOSCA deployment templates, provided by VNF vendors. Cisco's Management and Orchestration Architecture and Ciena's BluetPlant NFV Orchestrations provide similar support.

Integrated NFV Architectures, Platforms, Suites: Beyond the overlap between NFVO/VNFM functionality and an emphasis on vendor supplied VNF and NS ecosystems, these orchestration solutions are commonly part of a fully integrated NFV management platform, including NFVO, VNFM, NFVI and extended services such as enhanced monitoring and analytics. For example, IBM's SmartCloud Orchestrator can be integrated with its counterpart solutions, SmartCloud Monitoring and IBM Netcool Network Management System, providing an end-to-end offering. A multi-partner ecosystem is common in the architectures, as well, complimenting core expertise. Such partnerships include IBM's NFV platforms coupling with Juniper's SDN controller; NEC and NetCracker; RedHat and a variety of partnerships, etc. This of course relates as much to business strategy as it does architecture, and will be covered in deliverable D7.5 Market Feasibility Study.

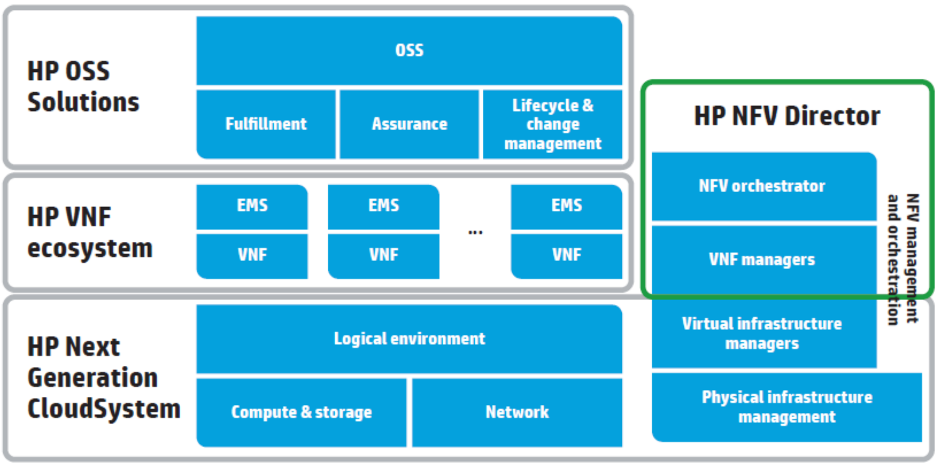

Figure A.4: HP NFV Director and ecosystem

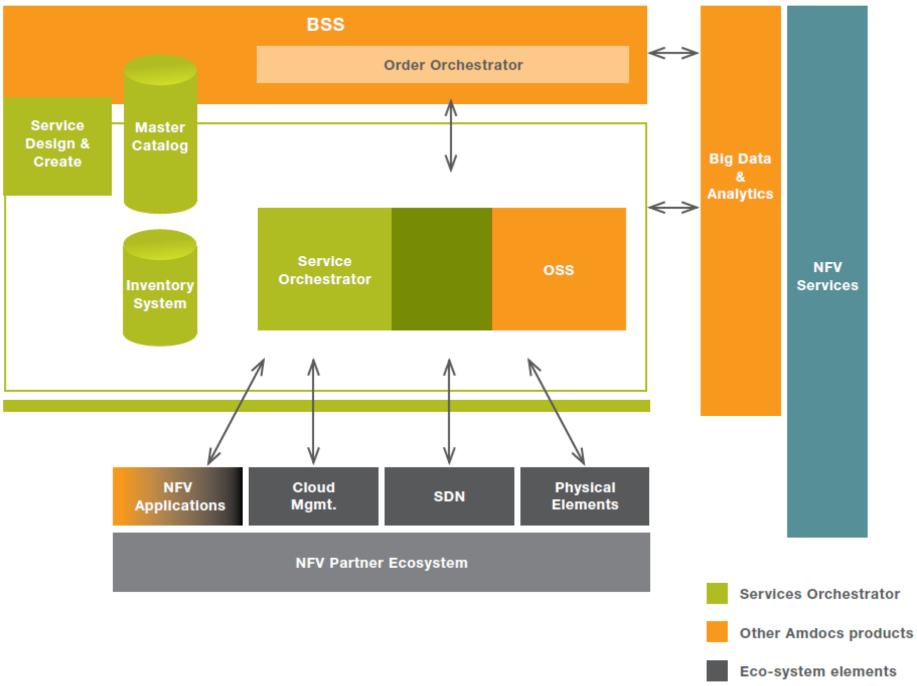

Figure A.5: Amdocs Network Cloud Orchestrator and ecosystem

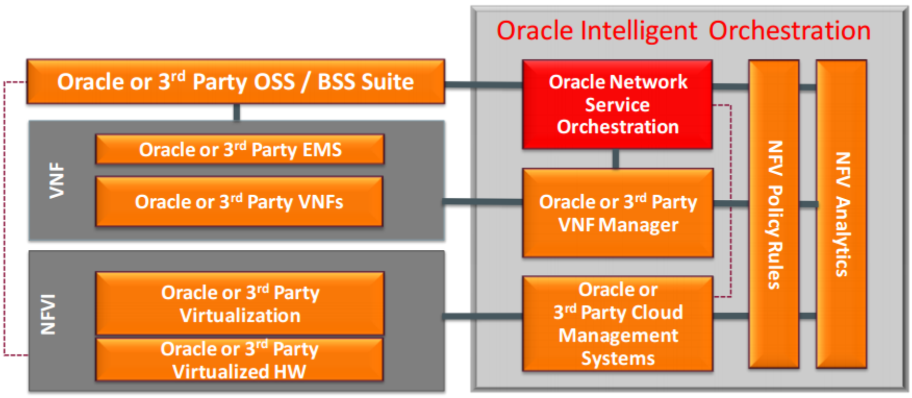

Figure A.6: Oracle Network Service Orchestrator and ecosystem

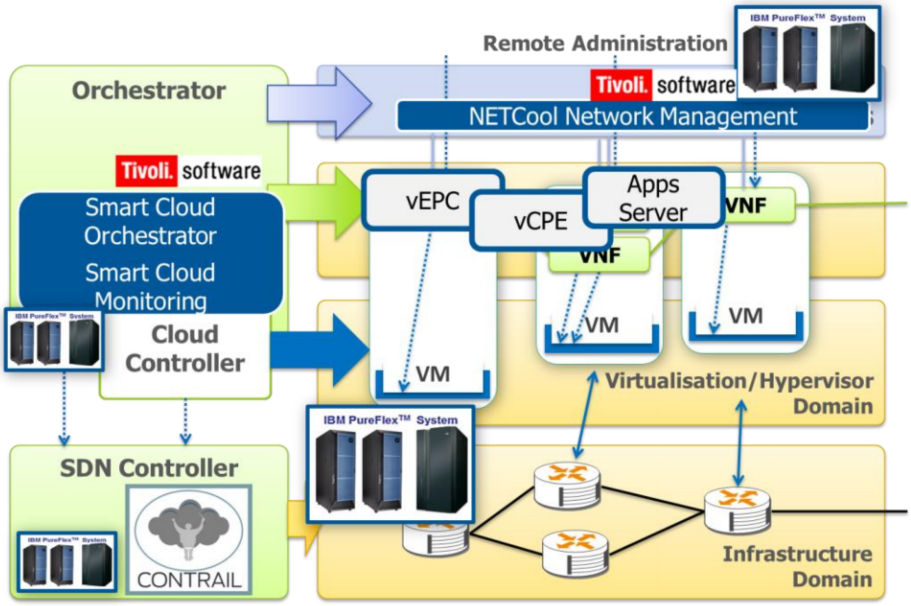

Figure A.7: IBM SmartCloud Orchestrator with Juniper SDN controller

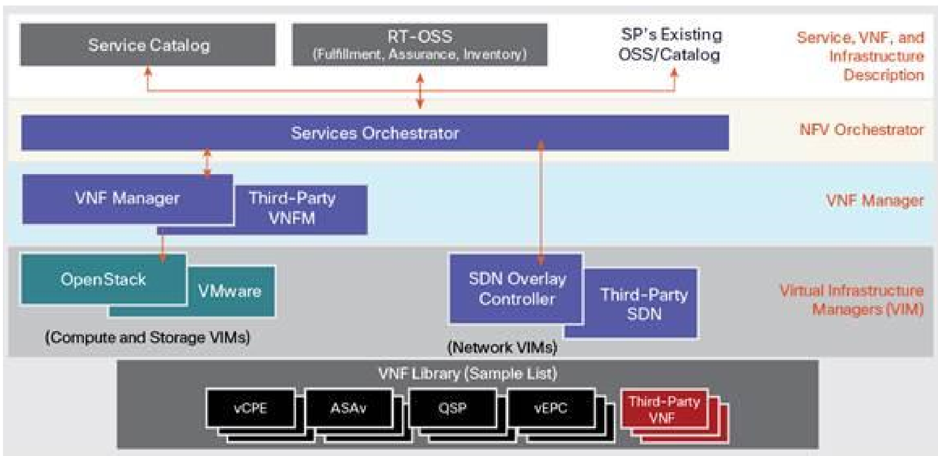

Figure A.8: Cisco NFV Management and Orchestration Architecture

B Event Hierarchy

B.1 Event Hierarchy for Message Bus

As described in Section 5.2.3.2, the message broker system to be implemented in SONATA Service Platform will consist of topic-based exchanges supporting publish/subscribe communication paradigm. Due to their flexible design, topic-based exchanges can cover stringent requirements especially in the cases where multiple users must selectively choose which type of messages they want to receive. This is achieved by using special characters ("*" and "#") as bindings between exchanges and queues. More specifically, * (star) can substitute for exactly one word, while # (hash) can substitute for zero or more words. The use of these special characters can make a topic-based exchange to act like a fanout or even a direct exchange, making topic exchange a powerful choice for SONATA message bus realization.

Given that the SONATA architecture consists of two main blocks, related to the management of the network services (NS) and virtual network functions (VNF), supported by the infrastructure abstraction layer, we consider the use of three top-level types of artefacts in the event hierarchy model of SONATA message broker, namely *service*, *function* and *infrastructure*, as depicted in the following Table.

Table B.1: Top level hierarchy of topics

Topic	Description
service.*	Includes messages related to the management and orchestration activities of a NS
function.*	Includes messages related to the management and monitoring activities of a VNF
infrastructure.*	Includes messages related to the management and monitoring activities of an infrastructure

B.1.1 Global topic messages

At functional level, we consider three main global activities for the previous artifacts: their **specification and instantiation**, their **management** covering their whole lifecycle and finally their **monitoring** aspects:

- The specification of the artefacts, scoping the typification and characterization of different artefacts used as well as their relationships, will be included in a Catalogue, while the derived instances will be stored in a Repository. For the purposes of the Event Hierarchy topic-based definition, both are considered **inventory** activities, where CRUD (Create, Read, Update, Delete) operations must be stored, in order to support **management** and **monitoring** activities of specific NS and VNF.
- The **management** activities support the control of the lifecycle of the artefacts, performing the appropriate tasks needed, such as instantiation of a VNF or NS, connecting a set of

functions or allocating resources on an infrastructure.

- Finally **monitoring** activities are considered as a separate topic, related to the performance of services.

Based on these assumptions, we define the following global topic messages, as the first level of the event hierarchy scheme:

Table B.2: Global topics

Topic	Description
service.management.*	Messages for the operations needed for managing a NS
service.inventory.catalog.*	Messages for the operations needed for managing the specifications of a NS
service.inventory.instances.*	Messages for the operations needed for managing the instances of a NS
service.monitoring.performance.*	Messages for the performance management of a NS
function.management.*	Messages for the operations needed for managing a VNF
function.inventory.catalog.*	Messages for the operations needed for managing the specifications of a VNF
function.inventory.instances.*	Messages for the operations needed for managing the instances of a VNF
function.monitoring.performance.*	Messages for the performance management of a VNF
infrastructure.management.*	Messages for the operations needed for interacting with the infrastructure
infrastructure.inventory.catalog.*	Messages for the operations needed for interacting with the definitions of the infrastructure types
infrastructure.inventory.instances.*	Messages for the operations needed for interacting and managing the instances of the infrastructure
infrastructure.monitoring.performance.*	Messages for the performance management of the infrastructure

B.1.2 Extended topic messages

In order to get a better control of the different activities, the granularity of the operations is further refined. The following Tables include the definition of extended topics for the top-level hierarchy artefacts, followed by their description.

Table B.3: Extended service specific topics

Topic	Description
service.management.lifecycle.onboard.*	Messages for the operations needed for managing the onboard of a NS
service.management.lifecycle.start.*	Messages for the operations needed for managing the start of a NS

Topic	Description
service.management.lifecycle.stop.*	Messages for the operations needed for managing the stop of a NS
service.management.lifecycle.destroy.*	Messages for the operations needed for managing the destroy of a NS
service.management.scale.*	Messages for the operations needed for managing the scaling of a NS
service.inventory.instances.create.*	Messages for the operations needed for creating instances of a NS
service.inventory.instances.update.*	Messages for the operations needed for updating instances of a NS
service.inventory.instances.delete.*	Messages for the operations needed for deleting instances of a NS
service.inventory.instances.recover.*	Messages for the operations needed for recover instances of a NS
service.monitoring.performance.sla.*	Messages for the control on performance management for a NS associated with an SLA
service.monitoring.performance.status.*	Messages for the control on performance management for a NS associated with an status

The same approach used by the service scenario will be used for the infrastructure topics:

Table B.4: Extended infrastructure specific topics

infrastructure.management.image.*	Messages for the operations needed for managing image infrastructure
infrastructure.management.compute.*	Messages for the operations needed for managing compute infrastructure
infrastructure.management.network.*	Messages for the operations needed for managing net infrastructure
infrastructure.monitoring.storage.*	Messages for monitoring the storage infrastructure

The definition of extended topics will have a more precise name when the message that the topic will support is going to be approved. Also some relevant information needed for the SONATA project could have a dedicated name in the hierarchy, e.g. for the activities between the plugins and the Specific Service Managers (SSMs) for the placement and scaling purposes or lifecycle management. In particular, for the network function scenario the messages included in the next Table are defined.

Table B.5: Extended function specific topics

function.management.scale.*	Messages for the operations needed for managing the scaling of a VNF
function.management.placement.*	Messages for the operations needed for managing placement of a VNF
function.management.lifecycle.*	Messages for the operations needed for managing the lifecycle of a VNF

B.1.3 Plugin-specific topic messages

There are several specific plugins that have a functional description but not as closed interfaces (such as service manifest management, conflict resolution, slice management, etc) and thus it is supposed that the operations that they will offer as interfaces could differ from the generic aspects described above, so special topics must be defined. In these cases, we define plugin-specific messages, explicitly including *plugin* in the definition of its scheme, as shown in the next Tables for plugin-specific topic messages related to services and functions.

Table B.6: Service and Function specific topics

Topic	Description
service.management.plugin.mnfmgm.*	Messages for the operations needed for the specific plugin ServiceManifestManagement for a NS
service.management.plugin.confres.*	Messages for the operations needed for the specific plugin ConflictResolution for a NS
service.management.plugin.slimgm.*	Messages for the operations needed for the specific plugin SliceManagement for a NS
service.management.plugin.register	Message for the registration of a new SSM
service.management.plugin.start	Message for starting a new SSM
service.management.plugin.stop	Message for stopping a new SSM
service.management.plugin.update	Message for updating an SSM
function.management.plugin.mnfmgm.*	Messages for the operations needed for the specific plugin ConflictResolution for a VNF
function.management.plugin.register	Message for the registration of a new FSM
function.management.plugin.start	Message for starting a new FSM
function.management.plugin.stop	Message for stopping a new FSM
function.management.plugin.update	Message for updating an FSM
service.management.plugin.specific.[id].*	Messages for the operations needed for the specific plugin ServiceSpecificMananger[id] for a NS
function.management.plugin.specific.[id].*	Messages for the operations needed for the specific plugin FunctionSpecificMananger[id] for a VNF

B.1.4 Platform topic messages

The SONATA platform itself has several components that need to be managed/monitor from a high level perspective. Some of these components are included as plugins in previous version but other could need a specific topic for its purposes. For these elements we consider to use the **platform** name as a complementary high level hierarchy. In the next Table is included some detail about possible candidate definitions:

Topic	Description

Table B.7: Platform specific topics

Topic	Description
platform.management.catalog.start.*	Message for include a new catalog in the platform
platform.management.catalog.stop.*	Message for disable a catalog in the platform
platform.monitor.login.error.*	Messages for monitoring error on login

C Abbreviations

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

CM Configuration Management

CRUD Create, Read, Update, Delete

DSL Domain-Specific Language

ETSI European Telecommunications Standards Institute

FSM Function-Specific Manager

GUI Graphical User Interface

IaaS Infrastructure as a Service

IDE Integrated Development Environment

IoT Internet of Things

JMS Java Messaging System

KPI Key Performance Indicator

MANO Management and Orchestration

NF Network Function

NFV Network Function Virtualization

NFVI-PoP Network Function Virtualisation Points of Presence

NFVO Network Function Virtualization Orchestrator

NFVRG Network Function Virtualization Research Group

NS Network Service

NSD Network Service Descriptor

OASIS Organization for the Advancement of Structured Information Standards

OSS Operations Support System

PSA Personal Security Applications

REST Representational State Transfer

RPC Remote Procedure Call

SDK Software Development Kit

SDN Software-Defined Networking or Software-Defined Network

SLA Service Level Agreement

SNMP Simple Network Management Protocol

SP Service Platform

SSM Service-Specific Manager

TOSCA Topology and Orchestration Specification for Cloud Applications

UD User Device

UE User Equipment

vCDN Virtual Content Distribution Network

VDU Virtual Deployment Unit

vEPC Virtualized Evolved Packet Core

VIM Virtual Infrastructure Manager

VLD Virtual Link Descriptor

VM Virtual Machine

VN Virtual Network

VNF Virtual Network Function

VNFD Virtual Network Function Descriptor

VNFFGD VNF Forwarding Graph Descriptor

VNFM Virtual Network Function Manager

WAN Wide Area Network

WIM Wide area network Infrastructure Manager

XMPP Extensible Messaging and Presence Protocol

D Glossary

DevOps A term popularized since a series of conferences emphasizing a higher degree of communication between **Dev**elopers and **Op**eration**s**, those who deploy the developed applications.

Function-Specific Manager A function-specific manager is a small management program implemented by a service developer with the help of SONATA's SDK. It is executed by the SONATA service platform to manage individual network functions based on inputs, say monitoring data, specific to the network function it belongs to.

Gatekeeper In general, gatekeeping is the process through which information is filtered for dissemination, whether for publication, broadcasting, the Internet, or some other mode of communication. In SONATA, the gatekeeper is the central point of authentication and authorization of users and (external) Services.

Management and Orchestration (MANO) In the ETSI NFV framework ETSI-NFV-MANO, MANO is the global entity responsible for management and orchestration of NFV lifecycle.

Message Broker A message broker, or message bus, is an intermediary program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. Message brokers are elements in telecommunication networks where software applications communicate by exchanging formally-defined messages. Message brokers are a building block of Message oriented middleware.

Network Function The atomic entity of execution anything in the context of a service. Cannot be further subdivided. Runs as a single executing entity, such as a single process and a single virtual machine. Treated as atomic from the point of view of the orchestration framework.

Network Function Virtualization (NFV) The principle of separating network functions from the hardware they run on by using virtual hardware abstraction.

Network Function Virtualization Infrastructure Point of Presence (NFVI PoP) Any combination of virtualized compute, storage and network resources.

Network Function Virtualization Infrastructure (NFVI) Collection of NFVI PoPs under one orchestrator.

Network Service A network service is a composition of network functions.

Network Service Descriptor A manifest file that describes a network service. Usually, it consists of the description of the network functions in the server, the links between the functions, a service graph, and service specifications, like SLAs.

Resource Orchestrator (RO) Entity responsible for domain wide global orchestration of network services and software resource reservations in terms of network functions over the physical or virtual resources the RO owns. The domain an RO oversees may consist of slices of other domains.

Service-Specific Manager (SSM) A service-specific manager is a small management program implemented by a service developer with the help of SONATA's SDK. It is executed by the SONATA service platform to manage individual services based on inputs, say monitoring data, specific to the service it belongs to.

Service Level Agreement (SLA) A service-level agreement is a part of a standardized service contract where a service is formally defined.

Service Platform One of the key contributions of SONATA. Realizes management functionality to deploy, provision, manage, scale, and place service on the infrastructure. a service developer/operator can use SONATA's SDK to deploy a service on a selected service platform.

Slice A provider-created subset of virtual networking and compute resources, created from physical or virtual resources available to the (slice) provider.

Software Development Kit (SDK) A set of tools and utilities which help developers to create, monitor, manage, optimize network services. A key component of the SONATA system.

Virtualised Infrastructure Manager (VIM) provides computing and networking capabilities and deploys virtual machines.

Virtual Network Function (VNF) One or more virtual machines running different software and processes on top of industry-standard high-volume servers, switches and storage, or cloud computing infrastructure, and capable of implementing network functions traditionally implemented via custom hardware appliances and middleboxes (e.g. router, NAT, firewall, load balancer, etc.).

Virtualized Network Function Forwarding Graph (VNF FG) An ordered list of VNFs creating a service chain.

E Bibliography

[1] Oren Ben-Kiki and Clark Evans. Yaml - yaml ain't markup language. Website, October 2009. Online at http://yaml.org.

[2] SONATA Consortium. Sonata deliverable 2.1.

[3] SONATA Consortium. Sonata description of work.

[4] T-NOVA consortium. D3.01: Interim report on orchestrator platform implementation. Website, December 2014. Online at http://www.t-nova.eu/wp-content/uploads/2015/06/Deliverable_3-01_Interim_Report_on_Orchestrator_Platform_Implementation.pdf.

[5] T-NOVA consortium. Specification of the infrastructure virtualisation, management and orchestration. Website, September 2014. Online at http://www.t-nova.eu/wp-content/uploads/2014/12/TNOVA_D2.31_Spec_of_IVM_and_Orchestrator_I.pdf.

[6] UNIFY consortium. Programmability framework. Website, November 2014. Online at https://www.fp7-unify.eu/files/fp7-unify-eu-docs/Results/Deliverables/UNIFY_D3.1%20Programmability%20framework.pdf.

[7] UNIFY consortium. Unify open source software and devops tools. Website, November 2015. Online at https://www.fp7-unify.eu/index.php/results.html#OpenSource.

[8] R Doriguzzi-Corin, E Salvadori, Aranda Gutierrez, C Stritzke, A Leckey, K Phemius, E Rojas, and C Guerrero. NetIde: Removing vendor lock-in in Sdn. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–2. IEEE, 2015.

[9] David Durman. Jsonmate. Website, 2012. Online at http://jsonmate.com.

[10] C. Meirosu et al. Devops for software-defined telecom infrastructures. Website, October 2015. Online at https://datatracker.ietf.org/doc/draft-unify-nfvrg-devops.

[11] Federico M Facca, Elio Salvadori, Holger Karl, Diego R López, Pedro Andrés ArandGutiérrez, Dragan Kostic, and Roberto Riggio. NetIde: First steps towards an integrated development environment for portable network apps. In *Software Defined Networks (EWSDN), 2013 Second European Workshop on*, pages 105–110. IEEE, 2013.

[12] Fraunhofer FOKUS. Openbaton. Website, October 2015. Online at http://openbaton.github.io/.

[13] Internet Engineering Task Force. Json - the javascript object notation data interchange format. Website, March 2014. Online at http://json.org.

[14] Apache Software Foundation. Qpid. Website, 2015. Online at https://qpid.apache.org/.

[15] Open Networking Foundation. Sdn architecture. Website, June 2014. Online at https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf.

[16] The Apache Software Foundation. Apache activemq. Website, January 2004. Online at https://activemq.apache.org/.

[17] The Apache Software Foundation. ActiveMq. Website, 2015. Online at http://activemq.apache.org.

[18] The Apache Software Foundation. Apache Kafka - A high-throughput distributed messaging system. Website, 2015. Online at http://kafka.apache.org.

[19] The Apache Software Foundation. Apollo - ActiveMq's next generation of messaging. Website, 2015. Online at https://activemq.apache.org/apollo/.

[20] FP-7. NetIde. Website, October 2015. Online at http://www.netide.eu/.

[21] Gartner. Devops. Website, 2014. Online at http://www.gartner.com/it-glossary/devops/.

[22] Thomas Spatzier (IBM) Gerd Breiter, Frank Leymann. Topology and orchestration specification for cloud applications (tosca): Cloud service archive (csar). Website, May 2012. Online at https://www.oasis-open.org/committees/download.php/46057/CSAR%20V0-1.docx.

[23] HashiCorp. Terraform: Infrastructure as code, July 2014, howpublished=Website. Online at https://terraform.io/.

[24] C. Huang, Jiafeng Zhu, and Peng He. Sfc use cases on recursive service function chaining. Website, July 2014. Online at https://tools.ietf.org/html/draft-huang-sfc-use-case-recursive-service-00.

[25] iMatrix Corporation. OpenAmq - Enterprise AmqP messaging. Website, 2015. Online at http://www.openamq.org.

[26] iMatrix Corporation. ZeroMq - Distributed Messaging. Website, 2015. Online at http://zeromq.org.

[27] ETSI European Telecommunications Standards Institute. NFv: Network Function Virtualization. Website, November 2012. Online at http://www.etsi.org/technologies-clusters/technologies/nfv/.

[28] ETSI European Telecommunications Standards Institute. Network functions virtualisation (nfv); management and orchestration v1.1.1. Website, December 2014. Online at http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf.

[29] ETSI European Telecommunications Standards Institute. Network Functions Virtualisation (Nfv);Management and Orchestration v1.1.1. Website, December 2014. Online at http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf.

[30] ITU-T. Imt2020 5g draft recommendations. Website, 2015. Online at http://www.itu.int/md/T13-SG13-151130-TD-PLEN-0208/en.

[31] Matthias Keller, Christoph Robbert, and Holger Karl. Template Embedding: Using Application Architecture to Allocate Resources in Distributed Clouds. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 387–395. IEEE Computer Society, 2014.

[32] MidVision. Devops: The past, the present and the future part one, devopsthe birth of devops. Website, 2014. Online at http://www.midvision.com/resources-blog/bid/275507/DevOps-The-Past-The-Present-and-the-Future-Part-One.

[33] Kief Morris. *Infrastructure as Code*. OReilly, October 2015.

[34] OASIS. Advanced messaging queuing protocol. Website. Online at https://www.amqp.org/.

[35] Open Networking Forum (ONF). Sdn architecture, June 2014. Online at https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf.

[36] NetIDE Project. D3.1 Developer Toolkit Specification. http://www.netide.eu/sites/www.netide.eu/files/documents/NetIDE_D3_1-FINAL.pdf, 2014.

[37] NetIDE Project. D3.2 Developer Toolkit v1. http://www.netide.eu/sites/www.netide.eu/files/documents/NetIDE_D3_2-revised.pdf, 2015.

[38] The OpenStack Project. Openstack keystone developer. Website, February 2012. Online at http://docs.openstack.org/developer/keystone.

[39] The OpenStack Project. OpenStack: The Open Source Cloud Operating System. Website, July 2012. Online at http://www.openstack.org/.

[40] The OpenStack Project. Openstack tacker: An open nfv orchestrator on top of openstack. Website, June 2014. Online at https://wiki.openstack.org/wiki/Tacker.

[41] The Openstack Project. Murano/documentation/how to create application package. Website, November 2015. Online at https://wiki.openstack.org/wiki/Murano.

[42] The OpenStack Project. Openstack ceilometer developer. Website, October 2015. Online at http://docs.openstack.org/developer/ceilometer.

[43] Wolfgang John et al. Rebecca Steinert. Deliverable 4.2: Proposal for Sp-DevOps network capabilities and tools. https://www.fp7-unify.eu/files/fp7-unify-eu-docs/UNIFY-WP4-D4.2%20Proposal%20for%20SP-DevOps%20network%20capabilities%20and%20tools.pdf, sep 2015.

[44] RedHat. HornetQ - Putting the buzz in messaging. Website, 2015. Online at http://hornetq.jboss.org.

[45] Mario Kind et al. Robert Szabo, Balazs Sonkoly. Deliverable 2.2: Final Architecture. https://www.fp7-unify.eu/files/fp7-unify-eu-docs/Results/Deliverables/UNIFY%20Deliverable%202.2%20Final%20Architecture.pdf.

[46] Sahel Sahhaf, Wouter Tavernier, Didier Colle, and Mario Pickavet. Network service chaining with efficient network function mapping based on service decompositions. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–5. IEEE, 2015.

[47] Jon Skinner. Sublime text - a sophisticated text editor for code, markup and prose. Website, January 2008. Online at http://www.sublimetext.com.

[48] Pontus Sköldström, Felician Nemeth, Bertrand Pechenot, Catalin Meirosu, Sachin Sharma, Ioanna Papafili, Guido Marchetto, Riccardo Sisto, Rebecca Steinert, Antonio Manzalini, Juhoon Kim, Wolfgang John, Xuejun Cai, Kostas Pentikousis, Serena Spinoso, Matteo Virgilio, and Apoorv Shukla. M4.3 Update on Sp DevOps with focus on automated tools., 2015.

[49] Pivotal Software. RabbitMq - Messaging. Website, 2015. Online at `https://www.rabbitmq.com`.

[50] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, and A. Csaszar. Elastic network functions: opportunities and challenges. *Network, IEEE*, 29(3):15–21, May 2015.

[51] Telefonica. OpenManO. Website, October 2015. Online at `http://www.tid.es/long-term-innovation/network-innovation/telefonica-nfv-reference-lab/openmano`.

[52] Oystein Torget. Yedit - a yaml editor for eclipse. Website, June 2009. Online at `https://marketplace.eclipse.org/content/yedit`.

[53] Canonical Ltd. Ubuntu. Juju: Model, build and scale your environments on any cloud. Website, 2015. Online at `https://jujucharms.com/`.

[54] ETSI NFV WG. Network functions virtualisation (nfv); infrastructure overview. Website, January 2015. Online at `http://www.etsi.org/deliver/etsi_gs/NFV-INF/001_099/001/01.01.01_60/gs_nfv-inf001v010101p.pdf`.