D2.3 Updated Requirements and 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	D2.3 Updated Requirements and Architecture Design
Workpackage	WP2 Architecture Design
Due Date	October 31st, 2016
Submission Date	December 8th, 2016
Version	1.0
Status	To be approved by EC
Editor	Michael Bredel (NEC)
Contributors	all Partners
Reviewer(s)	Phillip Eardley (BT), Geoffroy Chollon (THALES), Sonia Castro (ATOS)

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	**X**
CO	Confidential, only for members of the consortium (including the Commission Services)	

Executive Summary:

This document presents the revised and updated overall architecture of the SONATA system as well as new and updated use-cases and requirements. It is based on the previous deliverables D2.1 and D2.2 and allows for the lessons learned during the implementation phases of work packages WP3, WP4, and WP5. To this end, it describes the current state of the SONATA SDK and Service Platform as well as future work items for WP3 and WP4. We highlight, however, that this deliverables strongly focuses on the updates, changes, and novelties compared to D2.1 and D2.2. Thus, we expect the reader to be familiar with these deliverables as well.

The overall contributions of D2.3 can be summarized as follows:

- Revision and zooming in the use-cases of SONATA in order to highlight the use-cases that will be most likely be implemented as pilots.
- Revision and prioritisation of requirements for the revised use-cases.
- Revision of the descriptors, packages, and catalogues based on our learnings with respect to the first prototype in order to strengthen the CI/CD and DevOps approach for NFV.
- Revision and update of the SDK architecture.
- Introduction of novel service validation and profiling tools.
- Revision of the Service Platform architecture.
- Introduction of an enhanced Service Platform management.
- Introduction of container support for NFV in the Service Platform.
- Discussion and revision of the security aspects of the SONATA platform.
- Integration of slicing in the SONATA platform.
- Elaboration on the relationship between the SONATA architecture and the ETSI reference architecture.

Contents

List of Figures

List of Tables

1 Introduction

Deliverable D2.3 is the third specification document of the SONATA project, which presents the current outcomes of the second phases of task 2.1, task 2.2, and task 2.3 as well as additional inputs for WP3 and WP4 with regard to the design and specification of the SONATA overall system architecture. Based on our learnings from the implementation of the first SONATA prototype, comprising an innovative SDK, support for DevOps and Continuous Integration and Delivery, and a comprehensive Service Platform, D2.3 presents updates of the use-cases, requirements, and the system architecture. We highlight, however, that this deliverables strongly focuses on the updates, changes, and novelties compared to D2.1 and D2.2. Thus, it presents the deltas to the previous documents and we expect the reader to be familiar with these deliverables as well.

In particular, we elaborate on new features and adaptions regarding the SONATA descriptors, packages, and catalogue system. We aim at further improving the support of NFV developers by enabling re-usage of existing artefacts, like VNF and Network Service descriptors, in an easy way. Moreover, the SDK moves on even further and implements various features to support Continuous Integration and Delivery paving the way for an holistic DevOps approach. To this end, we enhanced the test and profiling support of the Service Development Kit. Thus, we can create performance profiles for VNFs and Network Services that can be used to optimized placement decisions for example. This document describes the feature, as well as requirements and some implementation details. Likewise, we integrated a validation tool to validate various aspects of Network Service already during development time.

With respect to the SONATA Service Platform, we adapted the monitoring framework and increased its reliability and scalability. Another major improvement of the Service Platform is the support of container based VNFs. To this end, we integrate a new Virtual Infrastructure Manager, i.e. Kubernetes, into the Service Platform. Evidently, this feeds back to our SDK tools and descriptors, as they have to deals with two different kind of images, namely Virtual Machine images and container images. Moreover, we performed a detailed analysis of existing slicing approaches and added native slicing support to the Service Platform.

Finally, we performed a detailed comparison between the SONATA architecture and the ETSI reference model. Further, we describe and map the actual interfaces that built these reference points in more detail clarify how third party components may interface with the SONATA Service Platform or individual platform components.

1.1 Structure of this Document

The remainder of the document is structured as follows. First, chapter 2 revises the use cases and details some of their implications. Next, chapter 3 reveals the requirements based on the use cases. Then, chapter 4 revises the main components of the SONATA architecture. It especially highlights and motivates the changes compared to the initial architecture. Therefore, it provides the updates on the descriptors, packages, and catalogues. It focuses in detail on the SDK adaptation and introduces some new components, like the profiling tools for network services, and addresses the updates and innovations of the Service Platform, including the security checks that have been introduced. Chapter 5 describes network slicing and its impact on the SONATA architecture.

Chapter 6 outlines the relation, similarities and differences, between the SONATA architecture and the ETSI reference model. Here we provide a detailed analysis of the SONATA interfaces and the ETSI interfaces specification and show how the SONATA system could be placed in an ETSI reference implementation. Finally, chapter 7 concludes the document, covers some open issues that are addressed by different work packages, and provides an outlook on SONATA future work.

2 Use Cases and Requirements

This section attempts an update of the uses cases and related requirements considered in year one. In the frame of a technical analysis that was performed in Work Package 6, contributed in Deliverable 6.1 [17], and also the recommendations after the first project periodic report, we are presenting and updating only the UCs that are considered for the project's pilots. The pilots' fine details are subject of Work Package 6, we are providing an updated description and a high-level description of the workflow for each one. The anticipated pilots are namely: (i) Virtual Content Delivery Network (vCDN); (ii) Personal Security Application (PSA) and (iii) Service Provider to Service Provider (PS2PS).

2.1 Virtual Content Delivery Network

2.1.1 Description

As presented previously in D2.1 (initial use case discussion) and D6.1 (pilot discussion), this use case focuses on showcasing and assessing the SONATA system capabilities in order to enhance a virtual CDN service with elasticity and programmability. The business case of Content Delivery Networks is well established in the current telecommunications environment. A series of business relationships are affected by various deployment scenarios that are possible within the current setting. SONATA is building upon the aforementioned status in order to allow for an enhanced vCDN service, focused around enablers provided by the Service Platform that allow high levels of programmability and flexibility. Two scenarios are anticipated for this Use Case, namely:

- Classic vCDN mode: Content originates from a single content provider or multiple ones, distributed across the vCaches and eventually delivered to a huge number of subscribers. This scenario will be used to highlight placement and scaling functionalities of the SONATA SP.

- User Generated Content (UGC) based vCDN mode: Content also originates from the end-users (allowing various sub-cases of social networking content exchange). The SONATA SP allows the flexibility of dynamically extending the vCDN service, accommodating additional sources from alternative Content Providers. The twist of this scenario is that the UGC content is identified and cached at the edges, allowing resource optimisation at the edges. This scenario, reveals the interaction of the Service Platform with information that stems from the network (either as traffic information or content information or end-user information) in order to dynamically configure and optimise the CDN for an improved user experience.

An extension to the above functionalities can be seen by the introduction of an additional functionality for the vCDN. As non-linear editing tools normally produce non-adaptive MP4 media, which need to be transcoded and segmented to provide best user experience (QoE) for the available bandwidth, the vCDN service will optimise the QoE for the End Users of the service by introducing in the forwarding graph of the service a vTranscoder. This functionality can adapt the content per combination of elements (end-user profile/context, available bandwidth, terminal information,

etc.) by choosing the best suitable transcoding and segmentation to ensure the best QoE. Placement of vTranscoder will be decided at the service instantiation and on-demand according to the situation and customer request. During operation, based on the monitoring from the network and End-user terminal consuming the vCDN service, the vTranscoder will adapt the video transcoding and segmentation for improved QoE.

SONATA, through the SSM/FSM structure and the DevOps approach, allows developers to reuse and ingest external sources and components in addition to theirs, for the development of a functional composed Network Service. In this context the SONATA SP offers the unique capability to have a fully composed service allowing interaction between the various VNFs in order to enrich their functionalities and implement value added NS. To our knowledge, no other Orchestration Platform that assumes composition of NS with third party VNFs offers this capability. In this context, it is interesting for SONATA to implement and demonstrate this particular Use Case.

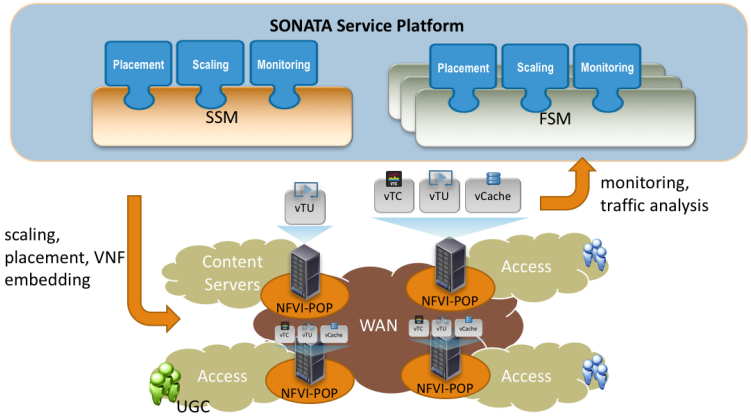

Figure 2.1: vCDN Network Service Deployment

Assessing the implementation feasibility of the discussed Use Case, most of the components that are considered will not be developed from scratch rather than readily available VNFs or modified versions of existing ones will be used. SSM and FSM plugins will be developed in order to implement the required functionalities at the management level. In this view, the implementation is realistically possible. The overall deployment of vCDN Use Case is illustrated in Figure 2.1. Groups of End Users are connected at the edges of the network (connectivity is out of the scope of SONATA), some End-Users are also able to generate content (UGC-green group). In the same figure, the upper orchestration and management layer is illustrated. For the whole service, a SSM is used in order to manage the placement of new VNFs as well as the service scaling decisions. In addition, for each VNF (i.e. instances located at various PoPs), FSM plugins deal with the placement and scaling of the VNF Components of each VNF. For example, additional instances of Virtual Traffic Classifier (VTC) DPI engine might be required to cope with the traffic load at certain PoPs. At the same time, new edge locations demand the deployment of additional vCDN VNFs (i.e. vCaches). All the interactions are driven by the monitoring and traffic analysis capabilities at the locations participating in the Network Service. The metrics are both generic (i.e. CPU, interface traffic, memory, etc.) and VNF specific (i.e. hit ratio, content classification, etc).

2.1.2 Sequence of Actions

- The SONATA framework is used for the development and instantiation of a vCDN Network Service (not part of the UC).
- The service is deployed over the Service Provider infrastructure (as requested by the customer (i.e. vCDN operator), a slice is allocated for the particular tenant.
- FSM plugins and SSM plugins are instantiated along with the respective VNFs.
- Initial placement of the VNFs is done according to the planed deployment.
- Various quality metrics (relevant to those of a vCDN) are monitored since the instantiation of the service, i.e. traffic load at the edges, content popularity, cache hit ratio, etc.
- End Users are located at various locations at the edges of the provisioned slice, consuming content.

2.1.2.1 Scenario 1 - Network Service reconfiguration

- Customer demands that additional resources at a new edge should be provisioned and should be accommodated within the same vCDN service.
- SONATA SP provisions the additional resources and updates the slice topology.
- SONATA SP deploys the required VNFs and integrates them under the same SSM instance.
- Users at the newly provisioned edge are serviced through the local cache.

NOTE: this could be the case where the edge resources are provisioned but not used until the SSM spawns the VNFs at that location based on demand increase (not scaling).

2.1.2.2 Scenario 2 - Scaling

- New load is gradually introduced at some of the edges of the provisioned slice.
- FSM for the running VNFs at those locations sends alerts for certain metrics that are used for triggering the scaling lifecycle either at VNF or NS level.
- SSM receives request for certain actions regarding to the scaling, i.e. by requesting the permission to spawn an additional VNFC for scaling out the VNF, or by instantiating a new VNF in order to load balance the traffic at certain edge locations.

2.1.2.3 Scenario 3 - User Generated Content Classification

- In an already established vCDN deployment, new functionality of User generated content caching is required.
- New SSM and FSM that will allow the management of such functionality are instantiated.
- SSM decides on the placement of VNF capable of content identification and classification.
- VTC is deployed at identified location and the service forwarding graph of the Network Service is reconfigured.

- VTC redirects identified UGC traffic to the local vCaches.
- Local content is cached and optimal use of resources is achieved as the connections of the PoP to the content servers is less utilised.

2.1.2.4 Scenario 4 - QoE enhancement

This scenario is an extension of the vCDN service including a DASH transcoding unit.

- The SONATA framework is used for the development and instantiation of a vCDN Network Service. (not part of the UC)
- The service is deployed over the Service Provider infrastructure (as requested by the customer (i.e. vCDN operator), a slice is allocated for the particular tenant.
- FSM plugins and SSM plugins are instantiated along with the respective VNFs.
- Initial placement of the VNFs is done according to the planed deployment
- Various quality metrics (relevant to those of a vCDN) are monitored since the instantiation of the service i.e. traffic load at the edges, content popularity, cache hit ratio etc
- End Users are located at various locations in the network topology

2.1.3 New Requirements

In addition to the system requirements, elicited by this Use Case during the first phase of the project and also detailed in the Deliverable D2.1 [12], the following requirements are also considered for the second phase of the project.

New Requirement Name	Scaling (updated)
Description	Developer MUST specify in the VNF Descriptor the initial number of instances and upper bounds of each VNF Component (VNFC) part of the VNF. The Service Platform will use this information for initial placement. For each VNF, the FSM will be responsible for deciding on the scaling and placement of new VNFCs in order to scale-out/in the VNF according to certain policies. In turn the SSM based on FSMs information will decide upon the Service level scaling approach i.e. instantiation of more VNFs to facilitate the additional load.
KPIs	Definition of VNFD and NSD to transfer the required information.
Category	Mandatory

New Requirement Name	SFC dynamic update
Description	The Service Platform SHOULD allow the dynamic configuration/modification of any given service function chain within a Network Service in order to insert/remove additional VNFs into the topology of the Network Service (i.e. Network Forwarding Path).
KPIs	Convergence time
Category	Mandatory

New Requirement Name	Traffic Steering among NFVI-PoPs
Description	The Service Platform MUST be able to steer traffic between the PoPs according to the Network Service definition. The implementation SHOULD be technology agnostic, using the modular implementation for the support of different WAN Infrastructure Managers (WIM).
KPIs	Level of isolation between tenants/networks, time for convergence.
Category	Mandatory

2.2 Personal Security Application

2.2.1 Description

Network Service Operators are taking a page from the Software-Defined Networking (SDN) and Networking Function Virtualisation (NFV) world to allow new services and to keep up with a fast changing consumer demands around interactive services, social networks, smart devices and Internet of Things. Network Service Operators are providing device and pervasive access protection from Internet threats by offloading execution of common security applications away from user devices with the help of Virtual Network Security functions (VNSF), like VPNs, firewalls, or parental controls services. The idea can be furthered by personalizing these VNSFs per customer's requirements, i.e. the subscribers want to remove the boundaries and limitations to create a better personalized experience. The new devices installed to deliver the connected home are all required the web-based services to much greater degree than before and, with the current route based customer premises equipment (CPE), it is very complicated to enable these innovative new services, taking into assumption that managing these services in an operator's network is already a complicated operational task. The solution is a combination of a bridge, installed in the residence (Physical CPE), and a remote hosted Virtual CPE (VCPE) service which may consist of one or more VNSFs. These VNSFs may include:

- Firewalls, Antivirus, IPS - They provide protection to the home environment.
- Parental control components - They allow control of the consumed web content by device level.
- VPN Servers - They provide remote accesses to the user LAN.

The VCPE is required to support a large number of applications and services driven by the end users' dynamics, allowing them to define their own rules and policies for security. This revised PSA use case focuses on the parental control flavor of a VCPE service. Furthermore, this use case makes the assumption that the network service operator is also the VNSF developer which allows him to gather registered end user security requirements and customizes the VCPE service accordingly. This use case also considers large network service providers with multiple SP deployments and the possibility of end users being in another SP domain than where the service is deployed.

For this UC two scenarios are envisaged. The first one is related to the prevention of DDoS attacks originating from arbitrary places of the customer network (usually from the edges). The immediate response of the PSA service will be to dynamically respond to the threat by eliminating the offending traffic, at the points of origin. The second scenario is that of Parental Control, allowing classification of family members and applying certain policies according to the end-user requirements. The next subsection elaborates on the sequence of actions for each scenario.

2.2.2 Sequence of Actions

- The SONATA framework is used for the development and instantiation of a PSA Network Service (not part of the UC).
- Differentiating from the previous UC, for each End-User a separate VNF FG is created, initially with only basic gateway functionality and a default configured firewall (vFW).
- The service is deployed over the Service Provider infrastructure (as requested by the customer, a slice is allocated for the particular tenant.
- FSM plugins and SSM plugins are instantiated along with the respective VNFs.
- Initial placement of the VNFs is done according to the planed deployment, end-user location, etc.
- Various quality metrics are monitored since the instantiation of the service, i.e. traffic load at the edges, content popularity, cache hit ratio, etc.
- End Users are located at various locations at the edges of the provisioned slice.

2.2.2.1 Scenario 1: Parental Control

- The End-User through a GUI or even off-line requests particular security functionalities is able to modify the firewall rules.
- The End-User, through a GUI or even off-line, requests the Parental Control functionality to be enabled.
- SSM for the PSA will react to the request in order to instantiate the required VNF (i.e. vPROXY) and embed it in the already established SFC.
- Upon instantiation the End-User will be able to administrate the creation of family member accounts and permissions for each one.
- Family members accessing the internet through their browsers will need to authenticate themselves in order to have access to a permitted set of websites according to their profiles.

2.2.2.2 Scenario 2: DDoS attack mitigation

- In a running PSA NS with a large number of End-Users (multiple chains), The SP wants to be able to identify DDoS attacks when they occur and blocks the traffic at the origin places.
- When requested by the SSM, IDS VNFs are instantiated in the end-users SFCs.
- If an attack is observed, the alerting system at the SSM level allows to pin point the origin and compute the policy to be applied.
- The reaction is to either instantiate vFW at certain locations protecting the end-users or passing new rules to some currently running vFW instances for each end-user.

2.2.3 New Requirements

New Requirement Name	SFC update for running network service
Description	The SONATA Service Platform SHOULD provide the means to modify SFC of any deployed Network Service to allow VNSF insertion/removal flexibility into the topology of the Network Service to adapt to the new conditions/situations.
KPIs	Time required to complete the SFC update
Category	Mandatory

New Requirement Name	Service-User spanning over Multi-SP deployments
Description	The SP platform SHOULD allow required interfaces to accommodate service and user spanning over two different SP deployment under the same network service provider.
KPIs	Service provisioning across multiple SP deployments
Category	Optional

There are no more new requirements for the updated PSA use case. The requirements of the use case are covered by requirement VNF Catalogue (4.2.1.1), Service Chaining (4.2.1.3), Placement Constraints for VNFs (4.2.1.8), Security VNF Availability (4.2.1.10), Personalized VNF (4.2.1.11), Security Simulation Tools (4.2.2.5), VNF Real-time Monitoring (4.2.4.4) and VNF Reporting to BSS/OSS and Subscriber (4.2.4.5) reported in Section 4 of D2.1 [12].

2.3 Service Provider to Service Provider

This use case considers the scenario where there are two cooperating Service Platforms (SPs) for rapid and dynamic service provisioning in a NFV environment. From a business perspective, there can be situations that justify this SP to SP cooperation:

1. One operator has segmented its network and has several service platforms that need to collaborate in order to deploy NFV end to end services across the network.

2. One operator provides NFV-Infrastructure, or even NFV Network Services, to another operator. The business relationship is a client-server one.

3. Two operators cooperate directly, for example when an instance of an NFV service function chain (SFC) spans across both operators. The business relationship is a peer-to-peer one.

SONATA does not address the business aspects of a multi-operator scenario, but the technical approaches are in scope.

2.3.1 Description

The first scenario shows a common situation in an operator. The network is segmented according to multiple criteria such as isolation of administrative domains, rationalisation of resources, etc. The situation occurs when an extended network service needs to be deployed and controlled across multiple service administrative domains, all of which are controlled by the operator. Various solutions are possible - for example, each segment could be a separate Service Platform or separate NFV Infrastructure (each running a VNF or a group of VNFs).

A trusted collaboration mechanism is needed between, for example, the service platforms. Two options can be considered as schematically depicted in the next figure:

- A first alternative, which is depicted in **??**, is that Service Platform 1 (SP1) adopts the role of 'master' and requests the second 'slave' SP (SP2) to deploy VNFs. This requires a west/east-bound API between SPs, which allows the management of services on another SONATA Service Platform. The SPs need to be able to negotiate the roles of master and slave, and then orchestrate resources.
- A second alternative, which is shown in **??**, is the establishment of a SP hierarchy where an 'umbrella' SP manages the two platforms that controls the service admin domains. This requires one level of recursivity of the SONATA service platform. The umbrella SP controls both 'underlying' SPs through the north/south-bound interface. Thus the approach is based on a hierarchical approach in MANO orchestration.

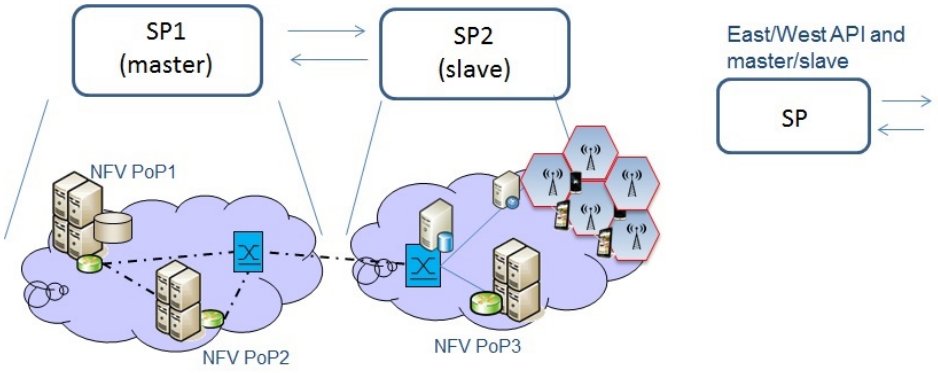

Figure 2.2: SP2SP master slave approach

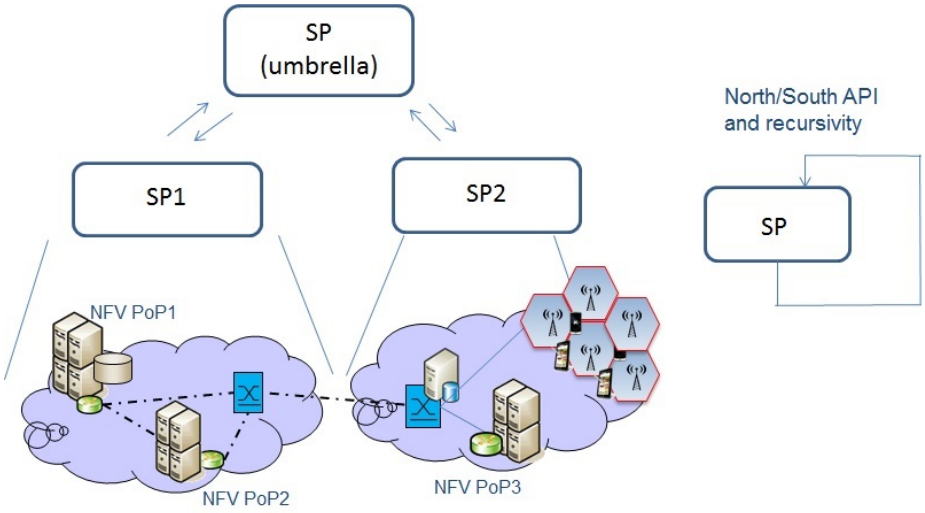

Figure 2.3: SP2SP umbrella approach

SONATA plans to pilot at least this scenario targeting the recursivity solution as a baseline.

The second scenario explores an extension of today's scenario where a "wholesale operator" offers connectivity as a 'wholesaler' to other operators, but doesn't have a business relationship with end customers. Motivations include that it allows the wholesaler to pool the compute, networking and storage resources across several 'end customer' operators; it enables an "end customer operator" to provide a better performance (latency, reliability), at lower cost and /or earlier than it could do

by its own (as a single, vertically integrated operator); and finally, some operators may prefer to keep different departments working separately, or regulation may require them to do so.

In the NFV world, there are several possibilities for what the wholesaler provides (as described in the on-going ETSI NFV EVE work on "end to end process"):

- NFV Network Service as a Service (NSaaS) – one operator uses a NFV network service from another operator as a component in their overall network service.
- NFV Infrastructure as a Service (NFVIaaS) – one operator uses the NFVI owned and operated by another network operator; the NFVI provides virtual machines connected by virtual networks for the hosting of VNFs and providing the interconnectivity between VNFs.
- Connectivity as a Service – this is the traditional role of a wholesaler, in which one operator uses a connectivity service from another operator. It is often implicit in the previous two bullets, in order to interconnect the NFV nodes of the NFVI for instance.

As an example, the figure Figure 2.4 shows the NFVIaaS use case.

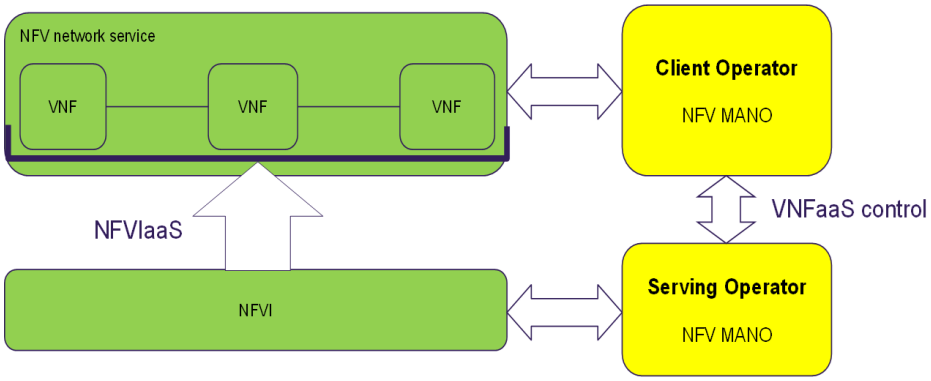

Figure 2.4: NFVIaaS provided by serving operator to client operator

SONATA's layered architecture approach is designed so that an operator uses components supplied by another operator – exactly as required by the scenarios above. The architecture is also recursive, meaning the number of layers is arbitrary, so that a service platform, as well as using components from others, can also supply extended services across service domains from others.

The SONATA project is addressing the technical, but not the business and legal aspects of the multiple operator scenario. In the Pilot we target to understand and demonstrate somehow the necessary SP data synchronization in the first bullet.

The third scenario with two SONATA operators is a peer-to-peer scenario. This is an extension to today's situation where a "global operator" provides networking to global companies, but cooperates with a "partner operator" that provides local access in countries where it doesn't have a presence. Motivations in the NFV case are similar, for example: it may be important that some VNFs are implemented close to the end customers site, to reduce latency or improve the efficiency of localised services (where the traffic is turned around); commercial sensitivity or regulation (such as data privacy rules) may require processing at a local site on country; or it may be that the "partner operator" performs some specialist function that the global operator lacks.

It would be possible to implement this scenario through direct peer-to-peer cooperation. However, another way is technically similar to the first scenario, though the commercial arrangements would be different. In the example shown in the figure Figure 2.5, the "global" (or 'client') operator offers a service to the customer, which it decides to implement with three component VNFs, two of which it runs itself and one that it gets as a VNFaaS from a "local" (or 'serving') operator. The global ("client") operator implements this as a layered architecture, where it has two serving operators, one of which it actually runs itself. Although this appears to mean it implements the three sets of capabilities (for development and definition, instance life cycle, and in-life operations) twice for two of the VNFs, note that this can be optimized (for example, calls to a common database). It also has the advantage that the global operator can readily decide to get the other VNFs from different operators, and it has a structured (hopefully standardized) way of negotiating with the local operator about in-life operations.

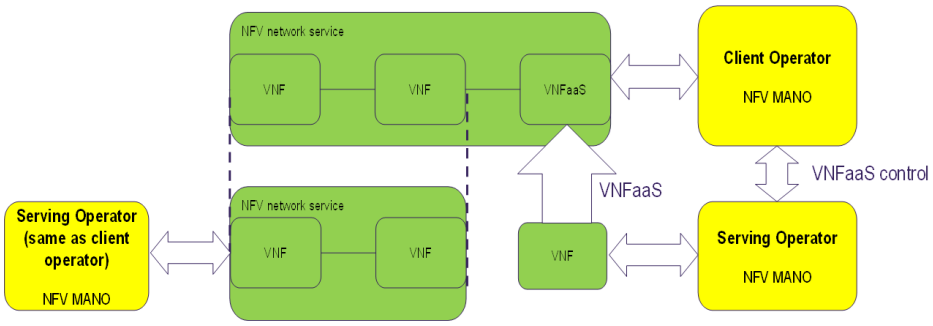

Figure 2.5: Cooperating global and local operator implemented as client and two serving operators

Again, the SONATA project is addressing the technical, but not the business and legal aspects of the peer-to-peer scenario - since we are implementing a use case where the service involves a chain of VNFs (SFC) spread across multiple PoPs of single operator, in which two SONATA service platforms collaborate as 'master-slave' (once agreed this as the peer-to-peer mechanism).

2.3.2 Sequence of Action

- Schematic Workflow

1. A user (OSS) interacts with the SONATA Service Platform to enable an extended network service (NS) that goes beyond the service domain (NFVI resources and topology) that the SP manages.

2. This originator SP requires other SONATA SPs the usage or/and deployment of VNFs that make part of the extended network service.
   Note: The type of interactions here are different depending on the adopted technical approach to address this situation.

3. If the technical solutions opts for a peer-to-peer case, the SPs will agree the master (originator) / slave roles through the east/west-bound interfaces.

4. If the solutions involves the existence of an umbrella SP, this SP instance takes the control of the request and initiates the deployment over the other SPs in a client/server mode via north/south-bound interfaces.

5. The SPs verify the existence of available resources. Initial placement strategy is used.

6. The necessary FSM / SSM plugins are initiated together with the respective VNFs and NS.

7. The deployment and interconnection of the VNF chain is executed.

8. Multiple metrics information needs to be exposed and synchronized among the SPs for overall control and status management.

9. After running instances and controlled shut-down of the extended network service is performed and collaboration links between SPs removed.

2.3.3 New Requirements

New Requirement Name	Support for Self-contained logical administration of services
Description	The logical administration of each layer SHOULD be self-contained, in that it administers full control of the services it provides. Therefore each layer must have a full set of administration functions which support the full life-cycle of the layer service: • the Development and Definition of the service, to create a template which describes the service before being uploaded to a catalogue of service/function types which can be instantiated within the layer • the Instance Life-Cycle across service domains, to instantiate an instance of the service/function and request resources from the underlying layers, and also remove instances and free the resources when no longer required • the 'In-Life' of the service, to monitor and maintain the instance across service domains. SONATA's SDK tools and Service Platform implement these three sets of administration functions. At first glance it appears that the first two items can be readily supported, whereas the in-life activity requires more work.
KPIs	Successful self-contained administration of services
Category	Mandatory

New Requirement Name	One level of SP recursivity
Description	For the umbrella approach, at least one level of SP recursiveness SHOULD be supported, i.e., • (required) a northbound service API, which offers services to the layer(s) above that can be a direct customer OSS request or another SONATA SP request • (required) a southbound API, which interacts with the service API of the layer(s) below that can be another SONATA SP or directly the NFVI
KPIs	Availability of North-bound and South-bound APIs
Category	Mandatory

New Requirement Name	East/West-bound API
Description	For the peer-to-peer approach, an east/west-bound API for the SONATA SP is required to enable communication between the two peer SPs.
KPIs	Availability of East/West-bound APIs.
Category	Mandatory

New Requirement Name	Master/Slave SP Negotiation protocol
Description	For the peer-to-peer approach, to resolve master/slave role definitions a negotiation protocol SHOULD exist. It will require a west/west-bound API to negotiate roles and establish the communication exchange means. In this case, SPs act at same level which may include synchronization of catalogues (optional), exposure of monitoring information (optional), etc but each part being fully responsible over its service admin domain.
KPIs	Successful selection of Master and Slave(s) in case of more than one SP.
Category	Mandatory

New Requirement Name	Dedicated GUI or interface for any manual intervention
Description	For both approaches, a GUI or human interface for anything which needs manual intervention (note that such actions are not done via the service API)
KPIs	Successful completion of required task via the GUI.
Category	Optional

New Requirement Name	SONATA SPs synchronization
Description	For both cases, synchronization of status information, catalogues, exposure of monitoring, etc., SHALL be provided.
KPIs	Successful synchronization/coordination of two SPs in terms of status, catalogues, and monitoring.
Category	Mandatory

2.4 Requirements Analysis and Consolidation

This subsection attempts the consolidation of the new requirements emerging from the updated SONATA use cases. The number of new requirements are comparatively less and one of the main reasons is the exhaustive requirements elicitation based on six different use cases that lead to a total number of 83 consolidated requirements (reported in D2.1).

Table 2.12: New SONATA requirements consolidation

No.	New Requirement	vCDN	PSA	SP2SP	Related Requirements from D2.1
1	Dynamic SFC update	2.2.1	2.2.1		4.2.1.3
2	Scaling (Updated)	2.2.2			4.2.2.3, 4.2.3.9
3	Traffic Steering among NFVI-PoPs	2.2.3			4.2.3.2, 4.2.3.10, 4.2.3.13
4	Support for Self-contained logical administration of services			2.2.4	
5	One level of SP recursivity		2.2.5	2.2.5	4.2.3.12
6	East/West-bound API			2.2.6	
7	Master/Slave SP Negotiation protocol			2.2.7	
8	Dedicated GUI or interface for any manual intervention			2.2.8	4.3.3
9	SONATA SPs synchronization		2.2.9	2.2.9	

3 Architecture and Design

3.1 Descriptors, Packages and Catalogues

As outlined already in deliverables D2.2 [13], D3.1 [14] and D4.1 [15], SONATA uses a variety of descriptors, catalogues and repositories to describe and store data or information regarding artefacts such as Virtual Network Function Descriptors (VNFD), virtual machine images, and Network Services Descriptors (NSD) and network service instances. The initial design and implementation, especially of the function and service descriptors, has been widely based on an early ETSI draft. However, throughout our implementation and evaluation phase, we identified several shortcomings and hence adapted our system in order to better support development workflows and our envisioned Continuous Integration/Continuous Deployment and DevOps approach. In the following sections, we describe changes and architectural improvements over the descriptors, packages and catalogues system characterized in D2.2 [13].

3.1.1 Function and Service Descriptors

In Network Functions Virtualisation (NFV), various descriptors are used to specify and describe artefacts and resources that constitute a virtual network service. To this end, you might find a Network Service Descriptor (NSD) that specifies the service characteristics, such as monitoring parameters and scaling behaviour. It also contains references to other service components, such as Virtual Network Functions (VNFs) and Physical Network Functions (PNFs). The PNFs again are specified in PNF Descriptors (PNFD) whereas the VNFs are described by VNF Descriptors (VNFDs).

Figure 3.1 depicts the various components that constitute a network service. In general, a network service contains one or multiple VNFs which again contain one or more VNF Components (VNFCs). Moreover, a VNFC contains all scaled-out instances of a Virtual Deployment Unit (VDU). To this end, a VDU-instance is a single instantiation, i.e. a running virtual machine or a container, of a VDU template that specifies the characteristics of, say, the virtual machine, e.g. in terms of CPU, RAM, and storage. In addition, the VDU template names the function of the VDU, e.g. by referring to a virtual machine image.

In SONATA the first ideas and implementation of descriptors were widely based on the preliminary ETSI work. However, during our implementation exercise, we added some room for improvement and evaluation. Thus, SONATA has adapted its initial descriptors as outlined below.

3.1.1.1 Identification of Artefacts

From the beginning, we found that the ability to identify the various artefacts in a unique way within and across descriptors is crucial for sophisticated development workflows. For instance, the re-use of artefacts, say re-using a VNF descriptor in different service descriptors, is only possible if it can be referenced uniformly. To this end, SONATA has introduced the vendor-name-version tuple to identify descriptors. Throughout our work, we further found that it is beneficial to identify more artefacts, such as network interfaces and connection points within a function descriptor, in a unique way, as it simplifies and assures consistency. Thus, we created an entity relationship

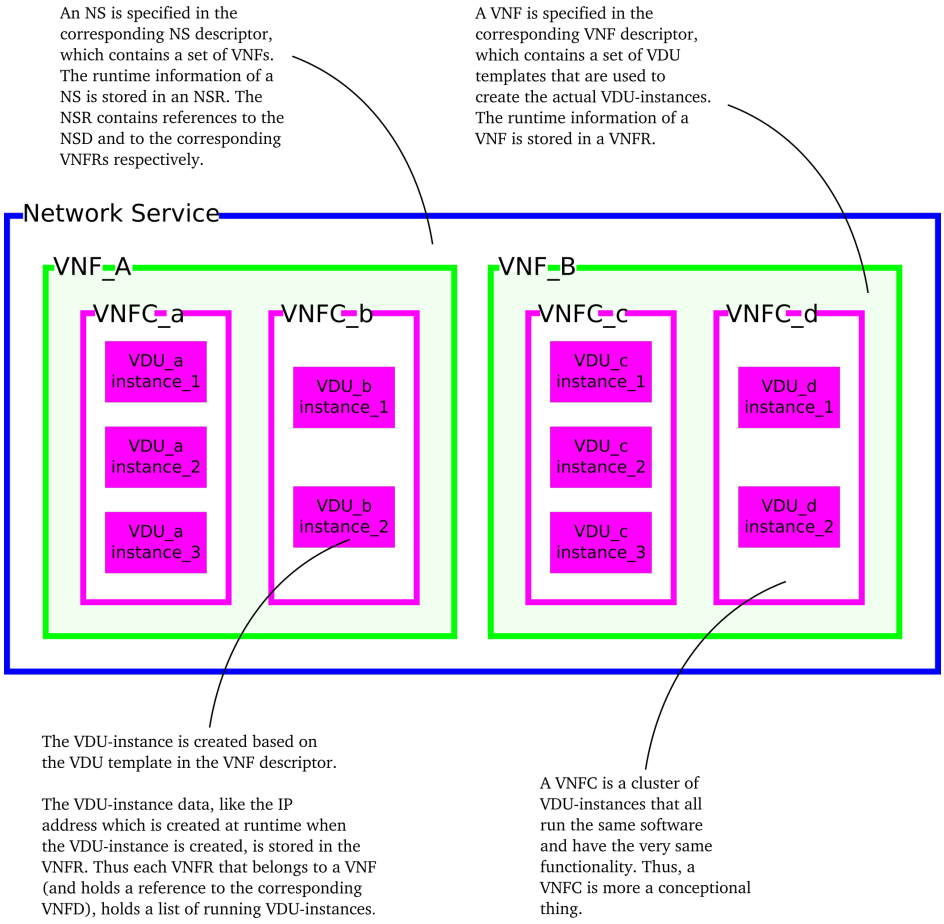

Figure 3.1: The relations between Network Service (NS), Virtual Network Functions (VNF), VNF Components (VNFC), Virtual Deployment Units (VDU) and their instances. The NS is comprised of one or multiple VNFs that again contain one or multiple VDU-instances, where VDU-instances of the same type are clustered as VNFCs.

diagram and developed an unique (hierarchical) identifier for all artefacts. This not only allows the re-use of all the artefacts across multiple systems, but also ensures consistency by using globally unique, but human-readable, identifiers.

3.1.1.2 Recursiveness

SONATA, in principle, addresses two types of recursiveness. First, the *platform recursiveness* where a SONATA Service Platform acts as a Virtualised Infrastructure Manager (VIM) to another SONATA Service Platform. And second, *service recursiveness* where a network service (description) becomes part of a new even richer network service by acting as a network function. Evidently, in the scope of descriptors we address the latter. To this end, the SONATA network service descriptor facilitates the SONATA identifiers, i.e. the vendor-name-version fields, to reference existing virtual network services and integrate them the same way we do with virtual network functions. Thus, we can create a hierarchy of network services which simplifies the development of rich network services significantly.

3.1.1.3 Network Interfaces, Connection Points and IPv6 support

Network Interface and Connection Point descriptors characterize the various inputs and outputs of a network function or service. We enhanced our initial descriptor design in a way that we now can differentiate between internal (private), external (public) and management inputs or outputs. This simplifies the integration, say of the FSMs and SSMs, which can be only connected to internal interfaces, and increases security as, say management interfaces, are not reachable from public IP addresses any more. Moreover, we enhanced the descriptors in order to support IPv6 addresses for ingress, egress and management ports.

3.1.1.4 Slicing

Slicing is an important aspect of arising NFV and 5G networks. And from a SONATA point of view, network services could run within resource slices that might be created by an operator using the SONATA Service Platform. However, slices are not created by the network service itself, which is why slicing is not considered within the descriptors.

3.1.1.5 Security and Licensing Aspects

Security and licensing of virtual network functions and services are complex tasks that need to be addressed at all the different layers of a NFV ecosystem. For example, in the SONATA descriptors family, the package descriptor contains basic security measures, like MD5 hashes for consistency checks and cryptographic signatures to check for authenticity. The next version of the SONATA descriptors now also contains reference fields for license information. Thus, we can link descriptors to external licensing systems, say a license manager, and include a variety of license systems in an easy and flexible way. The implementation and compliance assurance, however, is left to the Service Platform and might be realized by an additional SP plugin that handles the license information stored in the descriptors.

3.1.2 Updated Catalogues

Catalogues, as outlined in deliverable D2.2 [13] already, play an important role in the SONATA's DevOps approach. The various catalogues, like the SDK catalogue and the Service Platform catalogue, store complex artefacts, such as packages, descriptors, and allow to query, search, and

exchange those artefacts between developers and across platforms. Based on our findings during the first SONATA prototype, we adapted and improved the catalogues in various ways. We included comprehensives authentication, authorization, and accounting functionalities where it was necessary. Moreover, we improved performance and added functionalities to enhance the DevOps idea even further. Please find a description of the most important advances below.

3.1.2.1 Improved SONATA Catalogue System

For the second version of the SONATA system, the SDK and the Service Platform catalogues have been adapted independently.

The SDK catalogue, which resides on a developer system, say a notebook, has been quite complex in its former version. It consisted of a NoSQL database and had to be executed as a constantly running service. In order to simplify this catalogue and optimize resource utilization, we migrated the catalogue system from a daemon-based approach to a file system based approach where the SDK catalogue can be directly accessed by the SDK (CLI) software. Instead of saving the artefacts in a database, they are stored directly on disk. That is, a developer can query a 3rd party catalogues or SONATA SP Catalogue for pre-existing artefacts and store them directly on its disk.

The Service Platform catalogue has been enhanced by addition of security features in the SP, i.e. by introduction of comprehensive authentication, authorization, and accounting functionalities to address security and licensing issues. Moreover, we enhanced automated checks and tests in order to detect malformed descriptors and cyclic dependencies. Furthermore, meta-data is added for each artefact to facilitate versioning and certificate of authenticity to improve integrity of stored artefacts. Finally, we introduced the notion of a *resolver* that is used to resolve dependencies within packages and descriptors for downloading missing artefacts to make them available to the SONATA system.

3.1.2.2 Resolver

Using the SONATA identifiers, i.e. the vendor-name-version tuple, we can reference various artefacts and link them together. These references, however, have to be resolved and made available to the executing system, like the SDK or the Service Platform. In order to simplify the resolution and to further enhance the SONATA DevOps approach, we introduced a new component to the overall architecture, namely a *resolver*. The resolver component can be configured to communicate with various catalogues in order to acquire a specific artefact. Moreover, it takes SONATA identifiers as input and tries to find the related artefact in all its known catalogues. When found, the resolver downloads the artefact to the local catalogue and makes it available for the local system. The artefact may then be used by the local Service Platform. Thus, the resolver facilitates the re-use of existing artefacts as it abstracts and simplifies their retrievals.

3.2 Software Development Kit (SDK)

Deliverable D2.2 [13] documented the initial SONATA SDK architecture, consisting of editors, workspace and project functionalities, packaging tools, an emulator, a catalogue, monitoring, debugging and analysis functionalities. Each of these components follows a light-weight independent usage model, similar to the GIT-tools. This implies that each of those tools can be installed/used without necessarily requiring another SDK component. The programming model itself is heavily determined by the schema and the corresponding descriptors, which have been updated in this document's previous sections. Meanwhile, initial versions of most of these components have been implemented, and the core functionalities were described in D3.1 [14]. Since the initial release of

the SDK, its components have changed and evolved, which only modestly have an impact on the original SDK architecture.

In this document we address the following aspects:

- The impact of a deeper integration of a CI/CD DevOps process on the level of service development and deployment.
- Describing the concept and architectural impact of introducing additional SDK components such as a profiling component, a new service validation tool, improved interaction between monitoring between the SDK and the SP, and SSM/FSM development support.

3.2.1 Developing for Continuous Integration and Continuous Deployment

The support of DevOps, which is deeply integrated in SONATA, means that development and deployment of SONATA services can happen based on the level of components. Any component part of a service can be developed, deployed, updated, and moved between environments in a seamlessly and individually manner, without requiring the entire service to be updated, reducing the number of manual steps required to do so. The entire process supporting this should, however, be equally reliable as the traditional process involving a split between Development and Operations environments, which acts on silos and only considers updates of full service releases, potentially requiring a set of manual interventions.

The components and processes which are required for such an approach are the following:

- A set of independent, **parallel environments/platforms** enabling the development of services and components in an isolated and sufficiently realistic manner.
- **Testing functionality** in these environments enabling to run *developer-driven tests* as well as more generic and policy-related *Service Platform operator-driven tests*.
- **Transition processes** enabling to activate services and components in a controlled manner from one environment to another, following a well-defined order of increasing reliability and production readiness.
- Mechanisms to maintain or **migrate service/component-level state**, enabling hitless version updates of services or components in the same or between environments.

3.2.1.1 Development and test environments

The development and operations environment as described in D2.2 [13] is largely dichotomous. On one hand, it involves a developer-focused SDK environment, consisting of a set of development tools and an emulator to help the developer in the design and implementation process. On the other hand, the operations-focused Service Platform is the environment where services and functions are instantiated upon user's requests. This can be considered as a minimal viable approach for DevOps, however it largely ignores the different stages of integration, testing and staging which are required for a reliable and production-proof DevOps process. Here we refine this 2-level model into a multi-stage and a multi-environment setup.

As depicted in Figure 3.2, we propose a flexible multi-stage environment enabling multiple phases of testing, integration and staging, splits into a developer-controlled pipeline (the upper-side of the figure), an operator-controlled pipeline (the lower part of the figure), and an interaction point where the control is transferred from the developer to the operator. This approach extends the existing pipeline which includes the current minimal viable DevOps approach containing two distinct environments:

1. A SDK-empowered Developer Sandbox environment on the very left of the figure.
2. The SP-enabled Operator Production Environment on the very right.

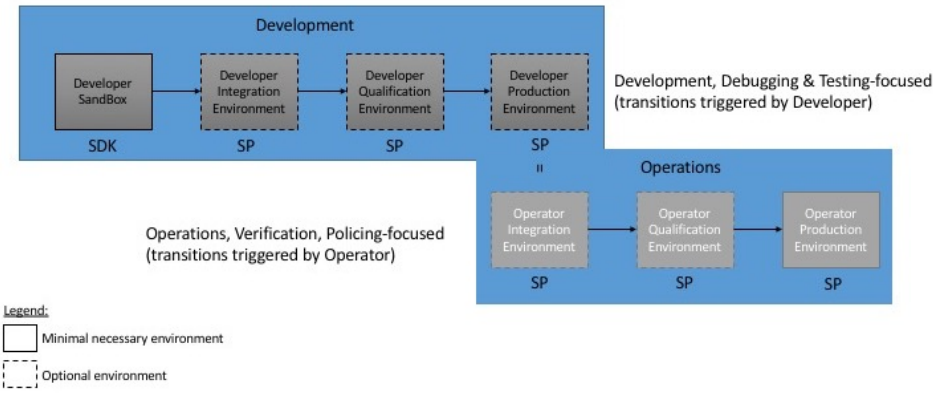

Figure 3.2: Development and test environments for CI/CD

In the new refined DevOps model, both the developer and the operator have integration, qualification and production environments.

3.2.1.2 Test support

A true DevOps process welcomes well-defined testing processes at different levels and granularities. As indicated in Table 3.1 below, tests can be executed either in an automated or on-demand way, before or after service deployment. Pre-deployment testing encompasses the checks which may be executed solely by inspecting the service and/or NF package, its descriptors and required images. Such tests may, for example, check for syntax errors or inconsistencies in the descriptors.

Table 3.1: Test support

	Before Deployment	After Deployment
Automated Execution	Example: automated syntax check of descriptors upon the generation of the service package	Example: execution of connectivity verification checks after deployment
On-demand	Example: check of reachability between NFs after editing the NSD based on encoded forwarding rules	Example: check configuration state of NF once deployed

Post-deployment testing might involve different types of testing. The first type could contain basic testing of the underlying Service Infrastructure, for example to ensure that NFs are up, and that the required links between them are operational. These types of tests usually do not need to be encoded in the service package or descriptor themselves, as they are agnostic to the considered service. Secondly, functional testing is specific to the considered service or component, and might involve particular tests to validate if the configuration of the component is operational. Such types of tests are usually implemented by the developer of the component and referred/included in the corresponding descriptors. Besides functional tests, testing might involve performance-related checks, for example to ensure that adequate QoS is reached by a deployed service (e.g. ensuring that a FW can handle the required number of packets per second). All levels and types of testing should be able to be triggered in an automated way throughout the DevOps process, implying that

upon each transition of a component from one environment (SDK or SP) to another (SP), pre- and post deployment checks can be possibly triggered at a functional or performance level.

Table 3.2: Test types

	Description	Service/Function Dependent	Example
Service Infrastructure Test	Test to validate the base of a service infrastructure in order to implement NFs, and if links interconnecting NFs are operational.	No	- NF liveness check - Service link check - Reachability check
Functional Test	Service or NF-specific test in order to verify if the component is functionally working.	Yes	- Check if FW is blocking appropriate traffic - Check if DHCP server is reacting on requests
Performance Test	Test which verifies if the performances of a service or a NF are correct against expectations	Yes	- Check if FW achieves required number of pps

3.2.1.3 Transition procedures

As indicated in previous deliverables, D2.2 [13], D3.1 [14] and D4.1 [15], the interface between the SDK and the SP is defined by a REST API triggered by the son-push tool from the SDK, and implemented by the Gatekeeper of the SP. This API is also specified in section . Once a service or component has passed all checks in one environment, it might be considered for a next stage environment (e.g., from Qualification environment to Production). Such a transition can be triggered by accessing the GK of both SDK or SP using the same REST API. However, it might be the case that this API will be further extended to facilitate the transition of finer-grained components (or versions of components), as well as the selection of multiple components.

3.2.1.4 State handling support

When a version of a component or service is updated in an environment, running instances using prior versions can be either upgraded or not. When upgrading the instance, runtime state (e.g., the configuration of a Firewall) can, and in most cases, should be migrated towards the instance of the component with a higher version. This process might either be in the responsibilities of the service or component itself (e.g., using SSM or FSM functionality to migrate state between NFs, or using data persistence techniques to store and retrieve data during the version swapping of components), or might rely on the MANO functionality of the SP itself. In the latter case, an API could be defined between service components and the MANO framework in order to pull and push service-related state. Such an API could for example build further on the OpenNF API [x]. Ideally, such process could happen in a hitless way, enabling zero-downtime for the service.

3.2.2 CI/CD support tool

Although each of the above considered functionalities could be achieved using existing SONATA CLI tools, DevOps shines when these steps can be visualized, persisted and automated in a common environment. Traditional software development relies on so-called CI/CD tools for managing such functionality. These tools ease the introduction of new environments and management of existing ones, the definition of development and deployment pipelines, providing means to manage and store test procedure collections, visualize the state of components in the pipeline, and provide one-click interfaces to trigger the execution of transitions and associated tests and state migrations.

Natural candidates to support such an approach are automation servers such as Jenkins [26]. In SONATA context, these will need to be customized and potentially extended in order to support the NFV-focused workflows. Both the developer and the operator might incorporate his/her own CI/CD support tool. Future releases of the SDK and/or SP might therefore include support for automation tools optimized for the SONATA workflows.

3.2.3 Profiling for NFV-based Network Services

Future 5th generation telecom networks are adopting new standards regarding the agility of network services. It envisions fast provisioning cycles, not only at the initial service deployment, but during its whole lifetime. This means that events such as configuration changes and service updates, like scaling actions, must happen with a 'zero-perceived' downtime for the service user. Nothing may interrupt the expected service quality. Furthermore, this high reliability of the service needs to go hand in hand with an optimized resource usage. It adapts the service resources elastically to the required processing power in quasi real-time, limits over-provisioning, but further stresses the assurance of the service quality. Network Function Virtualisation (NFV) offers many new degrees of flexibility to implement and deploy network functions, previously provided by dedicated hardware or middleboxes. The softwareised nature of these VNF implementations implies, however, that the performance of NFV-based services relies on a number of variables:

- The underlying software platform (e.g. programming language, compiler)and implementation quality.
- The architecture of the underlying hardware platform (e.g. ARM vs. x86, number of cores, clock speed, memory and/or available storage).
- The potential variability of the interconnecting networks between VNFs due to their flexible deployments on different locations in the network and SDN-based dynamic traffic steering.

This shows that, 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. It also illustrates that the advantages of a NFV-based network service come with a trade-off regarding its performance reliability. NFV performance profiling can remove this trade-off by linking performance metrics to underlying resource parameters. Network service developers do not only need to implement and specify their services before deployment, they also need to test them and validate their performances. Further, MANO systems do benefit from performance information of a certain service or VNF, for example, to know if a VNF performance benefits from adding additional CPU cores to it. This performance information, also called *performance profiles*, can have various practical use cases as described further.

The SONATA Software Development Kit (SDK) aims to be the starting point for both NFV-based service development and deployment. It allows to first test and validate a service in the local SDK environment, to give confidence on the correct configuration and implementation before the service is deployed in production on the Service Platform. SONATA plans to support service developers with a set of profiling solutions like already indicated in D2.2 Section 4.2.4 [13]. In this section we describe these solutions more in detail and present the SONATA approach for VNF and service profiling that is based on [33]. The ideas described here were not yet discussed in earlier deliverables and therefore this section is more elaborate than other ones in this deliverable.

3.2.3.1 Profiling as Part of the NFV DevOps Cycle

The overall goal of the DevOps methodology is to bridge the gap between development and operation of services. New service versions are directly deployed into production after they have been quickly tested in an automated fashion. In addition, information collected during service operation can be considered in the development phase to improve the service.

As a result, extensive tests on lab testbeds should be removed from the development cycle. This becomes challenging for NFV where the services are always expected to meet certain SLAs. On one hand, it becomes hard for service developers to validate that their changes result in the expected performance improvements before they put their service in production. On the other hand, MANO systems will be continuously faced with the management of new service versions, which means that their resource allocation algorithms, e.g., scaling algorithms, have to be continuously adapted. This can be tricky because historical monitoring information, available from old service versions, might not provide correct assumptions about the new version. For example, assume a developer fixes a performance bug in an intrusion prevention system (IPS) that reduces its resource requirements; a MANO system will not know about this and it will allocate too much resources to a new IPS instance.

To overcome this, a mechanism is needed that automatically gathers performance information about a service prior to its deployment without requiring dedicated testbeds or other special hardware setups. This is called *offline profiling* [33].

Another important point that motivates the need for *offline profiling* is based on the assumption that low-level metrics, like throughput, are often not sufficient to perform good resource allocation decisions. Especially for Quality of Experience (QoE) optimizations of application-level metrics, like frames/s or lag ratio of a video stream, are more interesting. However, due to encryption and privacy issues, it is not always possible to collect such metrics from operating services, e.g., when no deep packet inspection mechanisms (DPI) are available. In contrast to an offline profiling solution where a developer is allowed to collect all performance metrics he is interested in. But it is also possible, for example, to add additional measurement VNFs, called *probes*, to a profiled service chain.

Figure 3.3 shows a high-level NFV DevOps architecture. It contains artefacts and components that exist in most of today's architectures including service definitions, consisting of network service descriptors (NSD), VNF descriptors (VNFD), and VNF images as well as the MANO system that manages the service.

The figure added some components (filled boxes) to this architecture to integrate an *offline profiling* solution into the DevOps cycle. First, there is the main *profiler* component that is part of the service development toolchain and can be executed on the developer's laptop. This profiler gets the service definition and VNFs that should be profiled. Additionally, the developer specifies which resource configurations should be tested throughout the profiling runs and which performance metrics should be collected. The profiler then executes the service and its VNFs with different resource configurations and outputs profiling results for both the network service as a whole, called network service profile (NSP), and each constituting VNF, called VNF profile (VNFP). Optionally, topology information about possible target environments can be fed to the profiler. Based on this information topology-specific profiling runs can be performed in which the target topology, e.g., a multi-PoP topology with realistic inter-PoP delay, is emulated and the service is tested in this topology. The additional profiling results are called topology profiles (TP).

The profiling output of such an *offline profiler* highly depends on the host machine on which the profiling run was performed. This makes it harder to reuse them in other environments or compare them. Hence, a *normalizer* component is foreseen to be part of the profiling tool that normalizes the results with respect to the underlying machine.

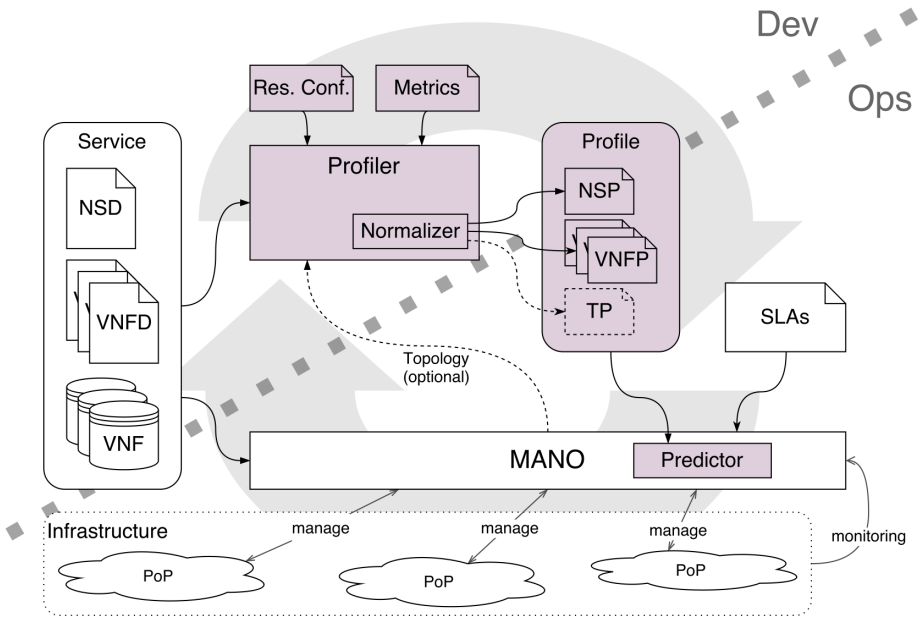

Figure 3.3: High-level DevOps architecture with integrated offline profiling solution [33]

The final, normalized profiles are then bundled with the service description and pushed to the operation side, namely the MANO system. By doing so, the MANO system has much more information about resource requirements available than in existing approaches. A *predictor* component inside the MANO system would be needed that uses these information to calculate the absolute resources required to meet the given SLAs in the target environment. Since the described performance profiles only provide *relative* profiling data and no *absolute* performance numbers for the target environment, the predictor is required to interpret the available information and predict the resource requirements of the target environment, e.g, by identifying trends in the profiles.

This approach would still be combined with monitoring-based management solutions and should not be seen as a replacement for performance monitoring functionalities. In the proposed scenario, monitoring information that becomes available after the initial deployment of a service would be used to continuously improve and refine the initial performance profiles and thus the decisions made by the management system.

3.2.3.2 Other Use-Cases for NFV-based Profiling

Next to the above described DevOps cycle, other use-cases for profiling exist:

- **VNF Benchmark-as-a-Service (VBaaS):** VNF benchmarking or profiling can be offered as a service by the SP [35]. This has the main advantage that it avoids continuous monitoring overheads. Similar to the DevOps approach, VBaaS is only provided on-demand whenever new or updated infrastructure resources are required or a service update needs to be tested. This is now done offline but on the production environment's infrastructure. This creates the possibility to estimate and compare the predictable behaviour of VNFs in different hardware environments or datacenter loads. If the VNF profile is considered reliable and its performance becomes very predictable, then this might reduce the amount of parameters to continuously monitor during the service lifetime. The QoS performance is then guaranteed as long as the profiled hardware resources are available.

- **Automated SLA translation:** Automated SLA translation to deployable VNF chains is researched in [36]. However, these efforts only take high-level policy descriptions into account and rely on a pre-defined mapping from policy expressions to deployable VNFs. It is clear that the missing link in this process is a mechanism that automatically calculates these mappings, i.e. the service or VNF performance profile. Of course the mapping becomes more complex as more parameters have to be taken into account, proportionally to the level of detail in the SLA and policy description.

- **Support for automated scaling/decomposition decisions of a MANO system:** Model-based service decomposition allows a step-wise translation from of high-level (monolithic) service functionality into more elementary or atomic VNFs, which are eventually deployed on the infrastructure. It is therefore an important part of the Service Fulfilment process. In a NFV context, the terms scalability, elasticity and decomposition are in fact related. They all depict the dynamic mechanism of deploying more or less VNF resources in function of the requested functionality, workload and hardware availability. Profiling helps to define the correct KPI threshold to trigger a scaling action for a VNF.

- **Training data for performance (prediction) models:** Profile datasets are necessary as training data for advanced resource prediction models such as neural networks, as illustrated in [27]. These machine learning techniques promise better correlation results than classic regression analysis but rely on a slow training phase which requires an earlier monitored dataset [28].

3.2.3.3 Requirements

A profiling solution that provides the functionalities described in the last section has to fulfil the following requirements:

- **R1:** *Profile production-ready VNFs.* In a DevOps approach time matters. Thus, a profiling system has to be able to execute the same VNFs that will later be executed in the production environment.

- **R2:** *Profiling could be done offline.* Network service and function developers want to quickly check the impact of their changes before a service is put to production.

- **R3:** *Support profiling of complex service chains.* Profiling a service chain as a whole will give more detailed insights about relative resource requirements of its parts.

- **R4:** *The profiling process has to be fully automated.* NFV is about automation. Thus the profiling step has to be automated as well.

- **R5:** *Profiles should contain fine-grained performance results.* Fine-grained performance profiles will make the MANO system better support prediction and decision algorithms.

3.2.3.4 SONATA Profiling Approach

There are two types of profiling approaches that can be applied to NFV use cases. The first one utilizes cloud testbeds to execute VNFs in realistic environments under different resource configurations. To do so, a VNF is executed as a VM with a pre-defined resource configuration, e.g., 2 vCPU cores and 2 GB of memory, and its performance is measured, e.g., its throughput. After this, the VM is destroyed and a new one with another resource configuration is started,

e.g, 4 vCPU core and 2 GB of memory. Based on this, performance values for different resource configurations can be measured, which creates a mapping from available resources to resulting performance. This approach provides only a limited set of possible resource configurations (CPU cores, memory, disk space) and it requires a lot of effort to configure and provision the needed VMs.

The second approach executes a single VNF and sends varying amounts of workload to it. During this, its resource consumption, like CPU and memory, is *measured* so that the results reflect a mapping from workload to resource usage. The benefit of this approach is that it comes with less configuration overhead. However, it does not generate results about the behaviour of a VNF under different resource limitations, e.g., different numbers of available CPU cores.

SONATA follows a hybrid approach and allows developers to specify both the resource configurations to be tested during a profiling run as well as the used traffic generators and their parameters. It utilizes the emulation platform with its container-based VNF execution that allows to control the resource configurations, such as CPU cores, available CPU time, memory and block I/O, in a very fine-grained way for each of the VNFs in a profiled service. The main challenge here is to come up with a profiling solution that supports network service developers and automates the profiling process as much as possible to make it applicable in a DevOps environment.

To automate the deployment and test execution as much as possible, a profile test setup is defined as a Service Function Chain (SFC), where a Test VNF is chained to the input/output of the VNF under test. The Test VNF is a dedicated VNF that can generate/analyse traffic and export the derived metrics to the monitoring framework for further analysis. In the SONATA framework, this translates to a Network Service Descriptor (NSD), which describes how the VNFs in the test setup are linked. The NSD points to Virtual Network Function Descriptors (VNFD), which describe the exact VNF images to be deployed.

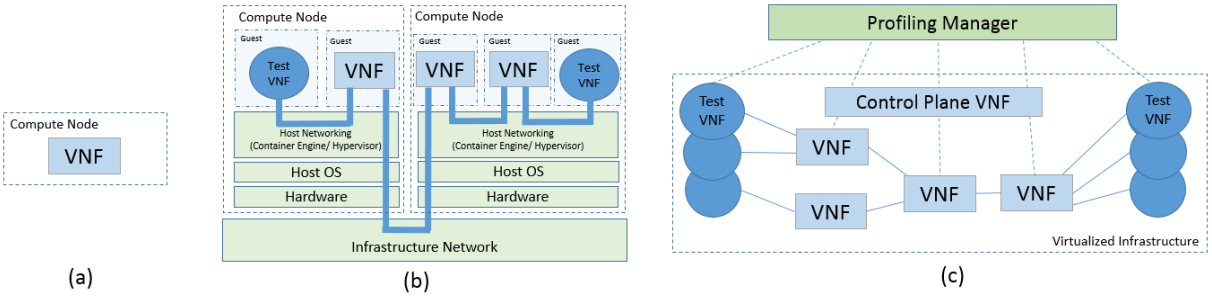

Figure 3.4: Generic profiling SFCs: (a) direct application profiling (b) installing a dedicated SFC with Test VNFs (c) larger SFC topology with multiple test-VNFs and Profiling Manager.

Figure 3.4 illustrates some generic profiling SFCs, deployed in a Virtualised Infrastructure. In a cloud computing context option Figure 3.4(a) can yield reliable results. This represents a VNF that is profiled without external chaining. So its workload is not provided via the normal networking but directly fed into the NFV application (e.g. an encryption function that processes local files). The difference with a NFV-based context can be seen in Figure 3.4(b), where the traffic flowing through a SFC is loading both the guest and the host. Ideally, this complete load and overhead should be taken into account when deriving a performance profile of a VNF. The principle of using SFCs to generate performance profiles is further exploited in Figure 3.4(c) to measure the performance of larger services, consisting of multiple VNFs in a graph-like topology including specific control plane VNFs. Similarly, multiple test VNF endpoints can be combined and controlled from a centralized profiling manager (the *profiler* in section Section 3.2.3.1). This can create an aggregation of very

large streams, or emulate a large set of different users (e.g. useful for testing vEPC functionality in a mobile network). The 2 latter options are further developed in SONATA.

Profiling Implemented in the ETSI NFV MANO Architecture

The automatic deployment and execution of profiling test runs, can be mapped on the general ETSI-MANO architecture, as shown in Figure 3.5. ETSI recommendations and methods for pre-deployment testing of the functional components of a NFV environment can be found in [25]. The modular approach of the SONATA framework can be mapped on the ETSI-MANO architecture, as discussed in earlier deliverables. The same modular implementation of profiling functionality can be followed in SONATA. This enables not only the deployment but also validation of the performance, reliability and scaling capabilities of Network Services.

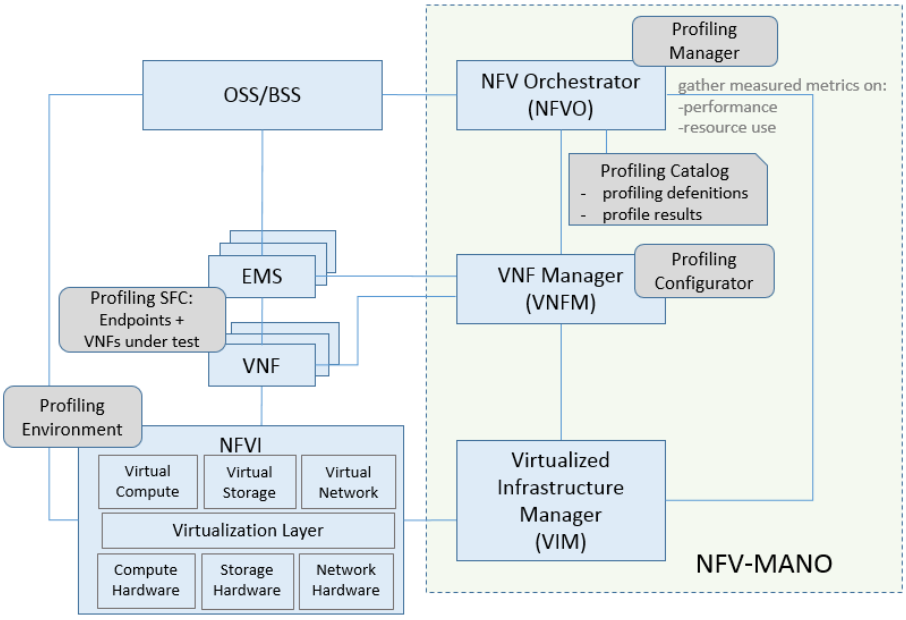

Figure 3.5: NFV Profiling mapped to the ETSI MANO architecture

- **Profiling SFC**: As described in the previous section, this is a dedicated profiling SFC with specialized test VNFs. The SFC needs to be described in the NSD format supported by the SONATA framework. VNFDs for the test VNFs and profiled VNFs need to be available.
- **Profiling Environment**: The virtual resources exposed by the Infrastructure. The VNFs in the Profiling SFC are mapped and deployed here. In SONATA, the emulator in the SDK environment will be used as infrastructure for the first profiling experiments. The emulator has been designed to be deployed on generic Linux based compute nodes so it can be used to deploy profiling setups on various hardware. We will further investigate if the emulator can also have a dedicated VIM on the SP itself.
- **VNF Manager (VNFM)**: This entity configures the VNFs in the Profiling SFC (e.g. setting the correct traffic parameters in the Test VNFs via a REST API). The VNFM can also expose performance metrics of the deployed VNFs. In the SONATA framework, this translates to a dedicated FSM plug-in. In the SONATA SDK environment, this functionality is implemented as an external layer above the emulator, adapted to be used with specific Test VNFs.

- **Profiling Catalogue**: This catalogue contains the SFCs that define different profiling runs and their configuration. Additionally, also the measured and validated datasets are stored for later use. In a first phase these will be locally stored files which describe the measured performance metrics in relation to the used resources. As described in previous sections, they can be used for further FSM/SSM development to derive the resource-optimized scaling mechanisms.

- **NFV Orchestrator (NFVO)**: This block adopts the function of general 'profiling manager'. It provides access to the Profiling Catalogue, deploys the SFC unto the infrastructure and gathers the monitored performance metrics from the VNFM and resource parameters from the VIM. In the SONATA framework, this can be implemented as a SSM plug-in module, dedicated for profiling. In a first phase we will focus on implementing the profiling functionality in the SONATA SDK environment, implemented as an external layer above the emulator. This will allow the creation of NSDs, deployment on the emulator, metric validation and analysis.

SONATA Profiling Toolchain

SONATA is developing a profiling approach that is based on the ideas of [33] and is implemented as a part of the SONATA SDK. Its main component is called *son-profile* which is a command line tool that supports network service developers by automating big parts of the profiling process. This tool interfaces with other SONATA components, like *son-emu* or *son-monitor* in order to create a complete profiling pipeline.

Figure 3.6 shows the high-level profiling concept developed by SONATA. A network service developer first creates the service package to be tested and defines a so called *Profiling Experiment Descriptor (PED)* which contains all information needed to automatically execute profiling runs with different resource setups and parameters. A developer can, for example, specify that each VNF in his service should be tested under different CPU resource allocations. Further, a PED file defines which traffic generator tools are used to measure the performance of the profiled service, e.g., iperf. To simplify these definitions, PED files allow to define parameter studies using a simple macro syntax that is inspired by Omnet++ [29] configuration files which are well known in the networking community.

The PED as well as the service package are then read by the *son-profile* command line tool (1) which then computes all possible configurations that should be tested based on the parameter studies defined in the PED file. After this, *son-profile* generates a set of new service packages (2), where each package contains exactly one service configuration that will be used for a single experiment executed during the profiling process, e.g., the package contains the service to be profiled and its resource configuration that should be tested. These service packages are then deployed on the emulation platform which also executes the defined traffic generators (3). At runtime, the performance, e.g., throughput or delay, of the tested network service is measured, e.g., using *son-monitor*, for each configuration (4). The results can then be post-processed by *son-profile* and be combined into a single performance profile that describes the behaviour of the profiled service under different resource configurations (5).

The benefit of our design, with its intermediate service package generation for each configuration to be tested, is that it can conceptually be integrated with other platforms apart from *son-emu*. It can for example be connected with a real service platform as long as the corresponding platform offers a way to monitor the performance of the tested service.

Another approach is to use the *son-profile* functionality to enhance SDK-based service validation, as illustrated in Figure 3.7. Here, only a single profile test setup is generated that links the needed

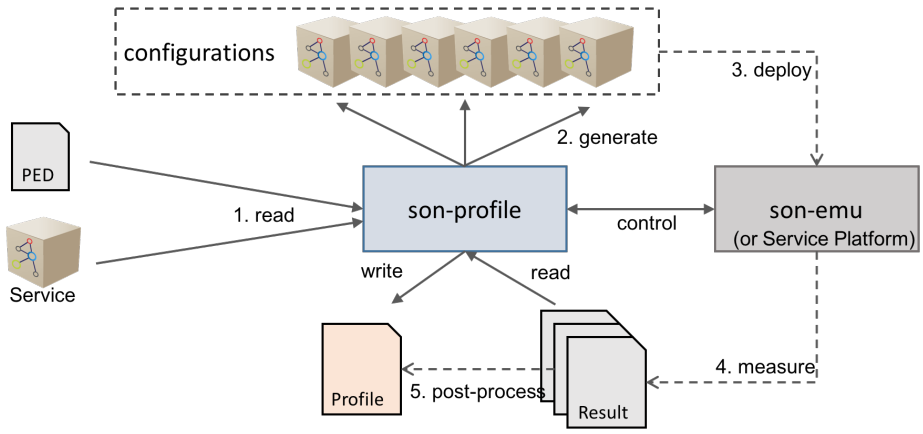

Figure 3.6: SONATA profiling concept based on the SDK tool son-profile that integrates with son-emu or (optionally) the service platform

Test-VNFs to the VNF or Service under test. Once this test setup is deployed, both its resource and functional configuration can be altered during runtime by *son-profile*. This can include actions such as altering CPU/memory allocation, modifying traffic generation in the Test VNFs or changing the functional configuration of the VNFs under test. The *son-profile* tool can read in a PED file as described above to define the parameters that can be configured. Additionally, a *Monitoring Service Descriptor (MSD)* describes all metrics that need to be monitored and exported. By analysing the monitored metrics, the configurations' parameters can be steered, for example to identify bottlenecks or debug anomalous behaviour.

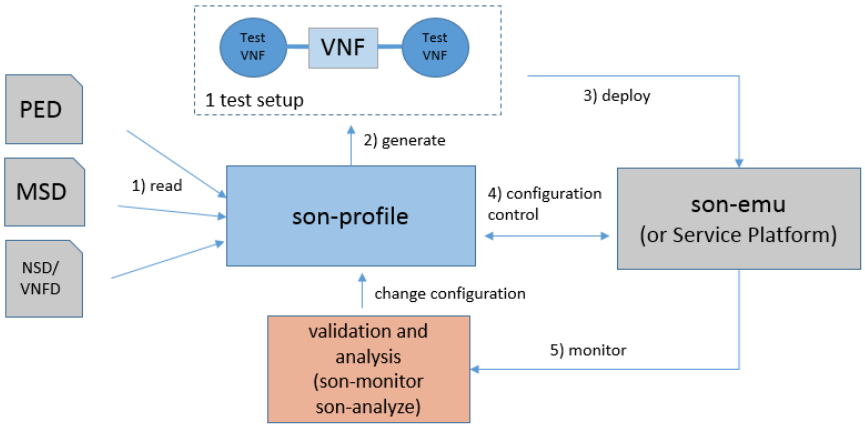

Figure 3.7: SONATA profiling concept used for service validation

More details about *son-profile* and its design will be presented in D3.2. The detailed implementation is involving these WP3 repositories (also further described in D3.2):

- **son-emu** : The emulator that deploys SONATA service packages consisting of Docker-based VNFs on a user-definable infrastructure topology.
- **son-cli** : A toolset (including son-monitor) that enables SFC creation and customizable monitoring of the deployed service, as illustrated in [38].

- **son-analyze** : An environment where monitored sets of metrics can be further analysed (e.g. statistical or regression analysis)

3.2.4 Service Validation

The service validation tool, namely `son-validate`, aims at supporting the development of services by providing examination algorithms. `son-validate` addresses the following validation scopes:

- **Syntax**
- **Integrity**
- **Network Topology**

3.2.4.1 Syntax

The service descriptor and corresponding function descriptors are syntactically validated against the schema templates, available at the son-schema repository.

3.2.4.2 Integrity

Service descriptors contain references to the function descriptor files that compose the service itself. This referencing relies in the combination of the vendor, name and version of the VNF. Thus, a referencing validation is performed to assert the existence of the referenced VNF descriptors. The inter-connection between the service and functions is carried out using connection points and the links associated with them. Connection points of the service are defined in the service descriptor itself, whereas VNF connection points are defined in the VNF descriptor and referenced in the service descriptor. Therefore, a verification of the correct relation of defined and referenced connection points must take place.

3.2.4.3 Network topology

The `son-validate` provides a set of mechanisms to validate and aid the development of the network connectivity logic. Typically, a service contains several inter-connected VNFs and each VNF may also contain several inter-connected VDUs. The connection topology between VNFs and VDUs (within VNFs) must be analysed to ensure a correct connectivity topology. The `son-validate` tool comprises the following validation mechanisms. Figure 3.8 shows a service example used to better illustrate validation issues.

- **unlinked VNFs, VDUs and connection points** - unconnected VNFs, VDUs and unreferenced connection points will trigger alerts to inform the developer of an incomplete service definition. For instance, *VNF#5* would trigger a message to inform that it is not being used.
- **network loops/cycles** - the existence of cycles in the network graph of the service may not be intentional, particularly in the case of self loops. For instance, *VNF#1* contains a self linking loop, which was probably not intended. Another example is the connection between *vdu#1* and *vdu#3* which may not be deliberate. The `son-validate` tool analyses the network graph and returns a list of existing cycles to help the developer in the topology design. In this example, `son-validate` would return the cycles:
  - [*VNF#1*, *VNF#1*]

– [*vdu#1*, *vdu#2*, *vdu#3*, *vdu#1*]

- **node bottlenecks** - warnings about possible network congestions, associated with nodes, are provided. Taking into account the bandwidth specified for the interfaces, weights are assigned to the edges of the network graph in order to assess possible bottlenecks in the path. As specified in the example, the inter-connection between *vdu#2* and *vdu#3* represents a significant bandwidth loss when compared with the remaining links along the path.

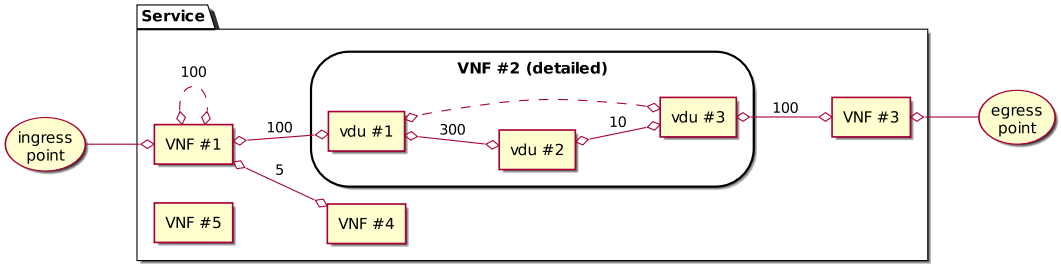

Figure 3.8: Example of a Service Network Topology

3.2.5 Monitor Data Transfer from the SP to the SDK

The monitoring framework in SONATA is based on the Prometheus open-source software, as described in D2.2 section 5.4 [13] and D4.1 section 5.2.4 [15]. In them, it was already explained that a developer/BSS can access the monitoring framework on the SP via the Gatekeeper and the Monitoring Manager. After Service deployment on the SP, it is possible in the SDK environment to:

- Receive alerts from the SP, as defined in the monitoring part of the service descriptor via the message broker and given that the developer has subscribed to specific message broker topics.
- Retrieve monitoring data from the deployed service via a REST API using the Monitoring Manager to relates each metric in the Prometheus DB with SONATA's monitored entities like NS/VNFs.
- Modify the monitoring configuration via the same REST API, the developer can add a new metric, change the monitoring frequency of an existing metric or even include a new alerting rule.

In addition to this, a new way of streaming monitor data from the SP to the SDK is introduced, to further close the DevOps cycle between service development and operation. This further enhances the SONATA monitoring framework allowing to inspect monitored data of a deployed network function or service in different ways:

- Past monitored values are stored in the Prometheus Database at SDK-side when a service is deployed via the SDK's service emulator.
- Past monitored values are stored in the Prometheus Database at SP-side when a service is deployed via the SP.
- Upon request, the SDK can request the SP to start streaming monitored data to the SDK (filtering the data first then specifying service/VNF and time-period).

A service developer should be able to process all monitored data of his/her services with these SDK tools, whether a service is deployed locally in the SDK's emulator or in production on the SP. The SDK has advanced debug and analysis tools that are not available in the SP, using the functionalities implemented in the SDK's: the *son-analyze* tool for example. This way the developer can write customized functions to process several time series metrics and do things like anomaly detection or load prediction. By implementing an automatic data transfer mechanism, we avoid that the SDK must continuously poll the SP to check if new data is available.

3.2.5.1 Finding the correct data transfer method

Since we have a Prometheus Push Gateway and Database both at the SDK side (for the emulator monitoring) and at the SP side, we can connect both sides together using these tools. We explore different data transfer mechanisms before settling with a push model, where the SP pushes data to the SDK, through dedicated websockets per user and service.

Table 3.3: Monitoring data transfer methods

Data transfer method	Explanation	Pro	Con
Pull from SDK via REST API	Use the GK's API to query data from the SP's Prometheus DB	Already implemented	SDK is polling, SP is loaded with multiple REST GET requests
Push from SP via web hook	data transfer using POST requests from the SP to the SDK	SP can push and stream data when it is available	SDK needs to be reachable via a public IP address
Push from SP via web socket	SP acts as a server where the SDK can connect to	SDK can connect to the Monitoring Manager public IP address	SP needs to open a dedicated websocket for each SDK/user that wants to receive data

The last method where the SP pushes metrics into a web socket seems to be the most preferred one. It satisfies the need for an automatic data transfer, without needing the SDK to continuously poll the SP for new data. We must ensure that the SDK can connect to the web socket at the SP side and read from it. Data received at the SDK side should be placed in the Prometheus DB for later use, or it could be streamed directly to the input of a monitor data analysis tool.

Push from SP based on websockets

In this push-based mechanism, we need the SP to actively send the metric values out of the SP. The disadvantage of this model is that the SDK side must be reachable for the SP. This can cause extra overhead e.g. requiring a public IP or a tunnel setup. But, as discussed in the table above, this can be avoided by using web sockets. This mechanism can be explained as the SDK being a client that automatically receives data from a server (the SP). The SDK is like a client browser and the GK is like a webserver that pushes data to the client. The GK can make the websocket available at its public IP. The normal GK authentication and authorization mechanism can be used to securely connect the SDK.

This is illustrated below in the Figure 3.9. Upon a request from the SDK, a web socket is opened on the SP (via the GK) where metric values can be pushed into. The SP is in control of the data transfer and it can choose to push monitored data as soon as it is available. The SDK can connect to this web socket to automatically receive the (filtered) monitored data.

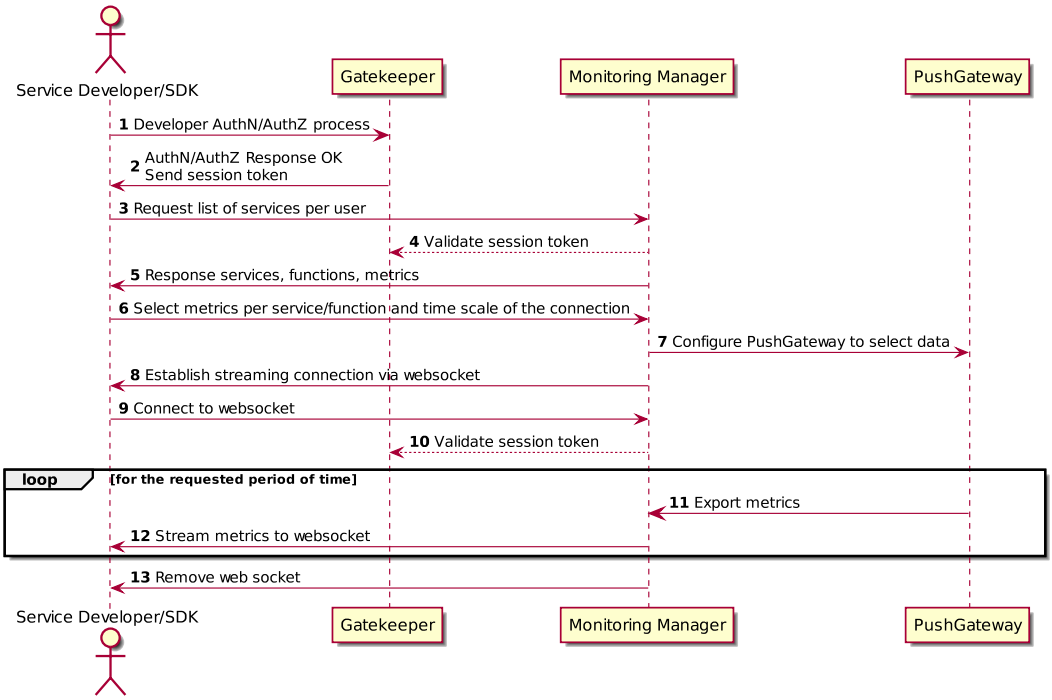

Figure 3.9: Retrieval of Monitoring Data

3.2.5.2 Secure the monitoring data transfer

The Gatekeeper in the SONATA SP will be the first point of access to reach the SP from the SDK. A service developer can use his/her identity from the SDK to access monitor data from the SP. This user authentication and authorization step is described in Section 3.3.4. After successful AuthN/AuthZ phase, the developer is granted a session token to access a dedicated web socket on the Monitoring Manager. Multiple developers will use different web sockets to receive their own set of allowed monitor data.

It is preferred that the Gatekeeper would be transparent in the websocket data transfer (acting like a proxy, apart from the AuthN/AuthZ phase). It will be further explored in D4.2 how this can be implemented. Either the session token will be validated in the Gatekeeper before the request is transferred to the Monitoring Manager, or the Monitoring Manager should query the Gatekeeper to check the session token itself first.

Optionally, the Gatekeeper could filter non-numerical monitored data (e.g. obfuscate domain names, ip/mac addresses) transferred in logfiles or packet streams. However, the data export described here only addresses numerical data gathered via the Prometheus Framework (containing compute, network and storage metrics). The export of this numeric monitor data is filtered in another way: per SDK/developer there is only a limited set of metrics that is allowed to be exported; these are only the metrics specified in the NSD/VNFD. So metrics of another user, other services not started by the developer or the SP itself can never be queried by the SDK. Also, next to this, other limits can be installed:

- Limited number of metrics at once.
- Limited quantity of data.
- Limited time frame during which metrics are exported.

3.2.6 SSM-FSM Development Support

This section describes tools that SONATA provides to support developers in the context of Service Specific Managers (SSM) and Function Specific Managers (FSM).

3.2.6.1 SSM-FSM general template

SONATA's SDK provides a template that can be used by service developers to create a new SSM/FSM. The template is a Python class containing functions that are needed to register a FSM/SSM into a Specific Manager Registry (SMR). A SMR is a MANO framework component, responsible for managing SSMs/FSMs lifecycle including SSM/FSM on-boarding, instantiation, updating, and termination. The SSM/FSM instantiation starts with registration which deals with storing a corresponding record in the SSM/FSM repository; that is done by the SMR.

Figure 3.10 illustrates the process of FSM/SSM registration which is implemented in the FSM/SSM template. It starts with the initialization function, responsible for receiving the SSM/FSM record. The record consists of the SSM/FSM name, type (either SSM or FSM), id number, version, and description. The next step, validation, checks whether the received record is valid or not, e.g., checks if the chosen SSM name can be used as a container name or not. Now that the received data is validated, the registration function can be triggered. The registration function forwards the record to the SMR through the message broker and waits for the response. Eventually, the SSM/FSM will be started if the SMR replies by a message containing "Registration OK".

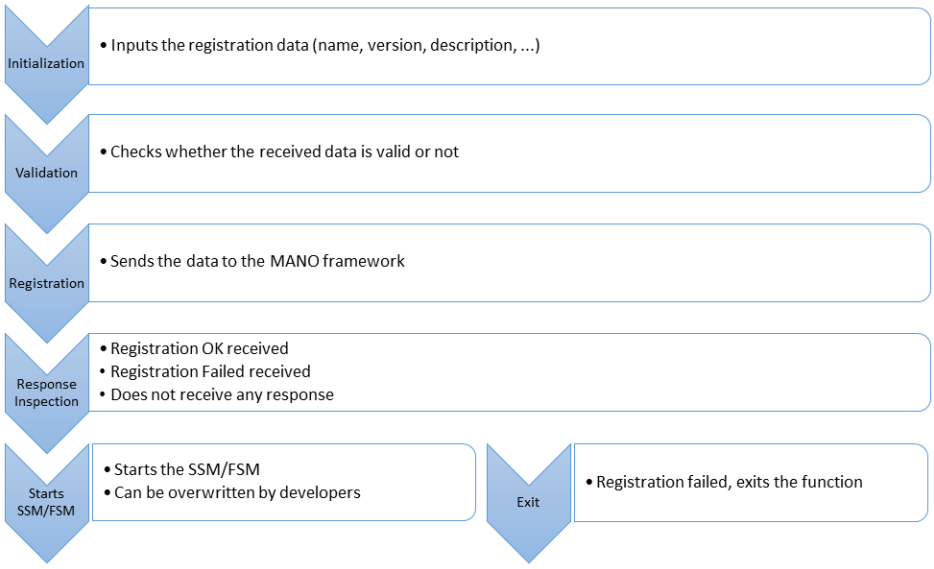

Figure 3.10: SSM-FSM registration process

Since all SSM/FSMs are required to have the registration functionality, the template can help developers to accelerate the SMM-FSM development.

3.2.6.2 Placement and Scaling SSM-FSM template

Placement and scaling SSM/FSM templates are designed to provide not only the SSM/FSM registration but basic placement and scaling algorithms as well. These templates can be used by developers to develop placement/scaling SSM/FSMs based on their own policy, e.g., developing scaling FSMs based on different CPU usage thresholds.

The input data that a SSM-FSM needs to process, is received via the SP's message bus. To further enhance the debug possibilities during SSM-FSM development, two approaches will be further investigated:

1. The SDK can have its own isolated message bus where a SSM/FSM can connect to and receive alerts.
2. The SP as a whole can be run locally on the developer's machine, so the same message bus, SLM and other SP functional blocks are available at the developer's side.

In both cases, the developer has access to the message bus and can publish messages on it. The SDK will provide a sequence/set of messages that tests the behaviour of the placement/scaling algorithm when published. Since the developer has access, he/she can create additional messages to test another functionality that is not covered by the messages provided by the SDK. The messages provided by the SDK can contain varying topologies to test the performance of the placement algorithm, monitoring alarms that trigger the scaling, etc.

SDK support for Scaling SSM-FSM

Optimized, dynamic resource allocation is a main functionality for a service scaling algorithm. The resource usage must at all times be adapted to the required traffic load, limiting over-provisioning and meeting QoS requirements. As explained in Section 3.2.3, performance profiling of NFV-based services gives a mapping between the required resources and the expected performance of a certain VNF or service. This can serve as input for a scaling algorithm. The format of the profiles generated by the tools described in Section 3.2.3 will be devised so that a parsing library for these profile results can be easily implemented in the scaling SSM/FSM. The datasets derived during profile runs initiated from the SDK, will therefore support the implementation of service specific scaling algorithms.

SDK support for Placement SSM-FSM

The SDK emulator supports the emulation of custom, virtualised infrastructure topologies. Different datacenters can be placed in a network, where additional bandwidth and delay constraints can be applied to the infrastructure links. The network and datacenters are implemented by virtual switches. On service deployment, the different VNFs in the service need to be connected to those datacenters. The placement algorithm that maps each VNF to a certain virtual switch, can be easily customized. A default round-robin placement algorithm is installed as example, but a service developer can implement and test any of his/her mapping algorithms on user-defined infrastructure topologies.

3.3 Service Platform Architecture

This section describes the Service Platform's architecture, focused on its modularity. We present both the existing architecture, as well as the expected changes for its second release.

We start by describing the main Service Platform's components and (briefly) their interfaces (these will be covered in detail in another deliverable [16]). Then, we highlight changes to be made in the Monitoring and to the Infrastructure Abstraction components of the Service Platform, in order to accommodate a different kind of VIM, based on containers. Next, we explain the security aspects we are including in this second version and detail how the entire Service Platform can programmatically be installed and uninstalled on our infrastructure.

3.3.1 Component Interfaces

This section describes the interfaces between the components in the SONATA Service Platform, and between the Service Platform's components and external entities: the Software Development Kit (on the left hand side of Figure 3.11) and the Network Function Virtualisation Infrastructure (on the right hand side of Figure 3.11).

These interfaces are listed per component duo that interacts with each other. The interfaces can be divided into two main categories:

1. RESTful interfaces (which are listed first)
2. Message based publish/subscribe interfaces.

The message based publish/subscribe interfaces are realised by a RabbitMQ message broker. Figure 3.11 shows the different components that are listed in this section, and the interfaces between them.

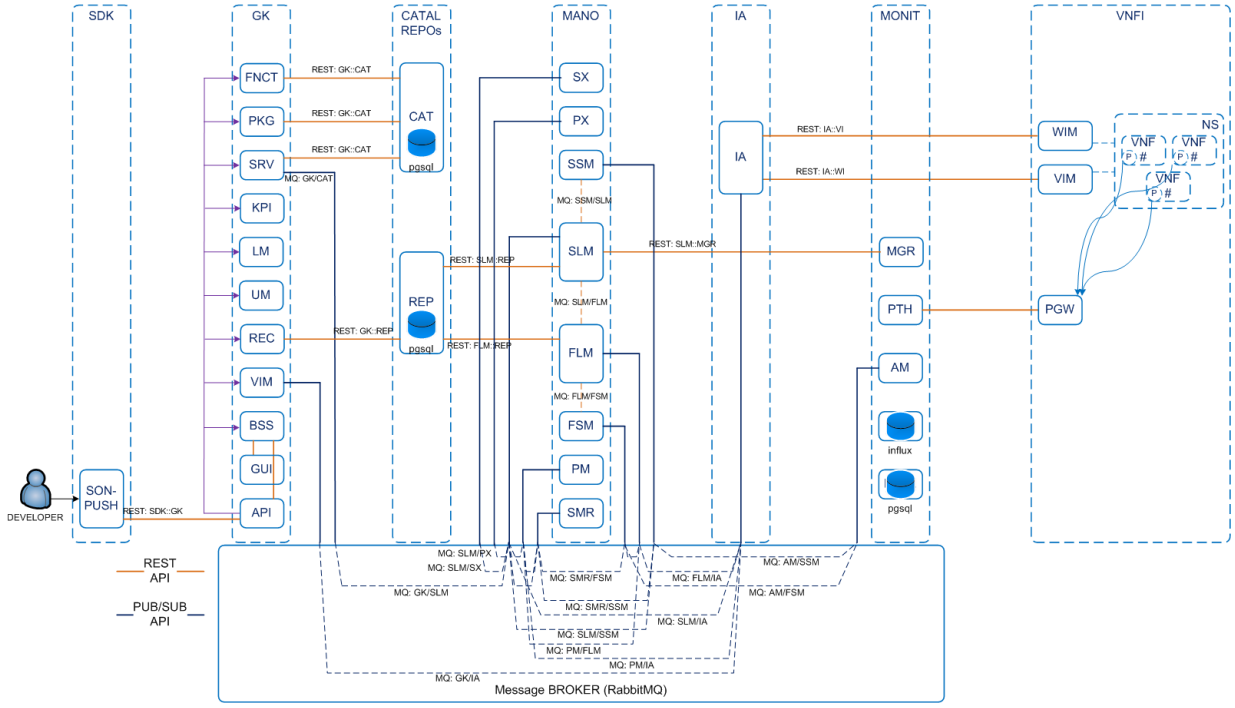

Figure 3.11: SP Component Interfaces

Interfaces to the outside components of the Service Platform, already described in [15], are implemented with a **REST** [20] API over HTTP. These components are the **SDK** and the NFV Infrastructure (**NFVI**) (which contains the **VIM** for the computational and storage resources, and the **WIM** for the wide-area network resources).

The `son-push` module of the SDK accesses the Service Platform through the Gatekeeper's **API**, which then delegates to other micro-services. These micro-services are then responsible for connecting with the other components of the Service Platform.

The Gatekeeper is built by several micro-services:

- **API**, playing the role of an API-Gateway;
- **Package Manager** (PKG), which receives, validates and stores packages;

- **Service Manager** (SRV), which manages service instantiation requests and queries on service descriptors;
- **Function Manager** (FNCT), which manages queries on function descriptors;
- **Record Manager** (REC), responsible for gathering service and function instance records and providing them to the developer;
- **VIM Manager** (VIM), which manages the set-up of VIMs through the GUI.

There are two different applications under the Gatekeeper's scope that access its resources through the API:

- **GUI**, a (Web) Graphical User Interface that allows the configuration and generic management of the Service Platform;
- **BSS**, representing the Business Support System, facing the **End-User** (who might be a person or a company).

In the second year version the following modules will be added to the Gatekeeper:

- **User Manager**, concentrating all the authentication and authorisation mechanisms the Service Platform will support;
- **Licence Manager**, with a first implementation of a licence mechanism that allows the monetisation of services and functions;
- **KPIs**, where the Key Performance Indicators will be stored/calculated.

The **Catalogues and Repositories** are components that were available from the first version, and they are where descriptors and records are stored.
The **MANO Framework** is the core of the Service Platform. Here, all components talk to each other through the Message Broker. These components are the following:

- **Plugin Manager** (PM): accepts, validates and controls all plugins of the MANO Framework;
- **Service Lifecycle Manager** (SLM): one of the main components of the Framework, accepts requests related to the service lifecycle and passes them to the FLM (for the function-related lifecycle) and the IA (the WIM part, see below);
- **Function Lifecycle Manager** (FLM): accepts requests related to the service lifecycle and passes them to the IA (the VIM part, see below);
- **Service Specific Manager(s)** (SSM): one of the key innovations of SONATA, together with FSMs (see below), these components are meant to adapt the behaviour of the Service Platform in certain and controlled ways, specific to the service it is related to. SSMs are usually specific to one feature, like scaling, placement, etc.;
- **Function Specific Manager(s)** (FSM): similar to the SSMs, they deal with function specific aspects, like scaling, placement, etc.;
- **Specific Manager Registry** (SMR): is responsible for managing the deployment and lifecycle of the F/SSM containers as such;

- **Placement Executive** (PX): responsible for interfacing with the placement SSMs/FSMs that are uploaded to the platform;
- **Scaling Executive** (SX): responsible for interfacing with the scaling SSMs/FSMs that are uploaded to the platform.

Besides the above shown MANO components, there are a few additional components responsible to manage the platform as such that aren't shown for the sake of simplicity.

The **Infrastructure Abstraction** (IA) allows for the transparent usage of different kinds of VIM/WIM infrastructure, like OpenStack, Kubernetes, etc.

The **Monitoring** components were already available from the first year, although specific extensions are planned for the second year, as described in the relevant sections of this deliverable. In particular, the extensions can be categorized to:

- those related to the scalability and reliability of the monitoring framework,
- the extensions to monitoring metrics (with an emphasis on the support of custom metrics defined by the developers),
- those related to the integration with SONATA Service Platform as a whole.

The fine details of these interfaces are left for **D4.2** [16].

3.3.2 Service Platform Monitoring Framework enhancements

This section describes and specifies the additional functionalities and enhanced operational characteristics of the SONATA monitoring framework to be developed on top of the characteristics achieved during the first year of the project.

In particular, during the first year, and in accordance with the description presented in the respective section in Deliverable 4.2 [16], the monitoring framework has been designed, developed and implemented as an integral part of the SONATA Service Platform. The components that the Monitoring Framework consists of have been developed, tested (both as separate units as well as in conformance with other SONATA components), validated and integrated.

However, since new technologies as well as functional and non-functional requirements are added to the SONATA Service Platform (also adopted by other components of the Service Platform and SDK), there is a clear need for adaptations and additions on the architecture of the Service Platform to allow for supporting the new functionalities from the monitoring system, in parallel with the developments already planned for the second year of the project (see the relevant section concerning the monitoring framework roadmap in Deliverable 4.2 [16]).

In the following sections, all planned activities related to the extension of the SONATA monitoring framework during the second year of the project are analysed.

3.3.2.1 Monitoring framework scalability & reliability

One of the major objectives regarding the development planned for the second year, is related to the extension of the SONATA Service Platform to more than 2 PoPs and VIMs. From the monitoring framework viewpoint, this means modifications and adaptations of the current implementation in order to address issues related to scalability and reliability.

API enhancements

The first version of the API calls provided by the Monitoring Manager has been already available during the first year of the project. However, specific needs have arisen, especially with regard to the ability to modify the configuration of the monitoring parameters and metrics in the VNFs (either deployed within containers or VMs) deployed in the NFVIs. Moreover, due to the decision to support streaming data from the Service Platform to the SDK, some new API calls must also be designed and implemented, providing management capabilities to the developers and thus closing the DevOps cycle that is followed in SONATA.

Implementing filtering mechanisms in monitoring probes

During the testing of the monitoring framework, a large flow of data from the monitoring probes to the Monitoring Server and its respective database has been identified that might affect the Service Platform performance in extreme cases, without adding knowledge to the system. In this respect, an architectural decision to address this scalability issue was to support a distributed architecture regarding the monitoring server and its database, working in a cascaded fashion along with proper modifications on component level. In particular, the functionality of the monitoring probe will change so that it will not send data to the monitoring server in cases where the value difference is less than a threshold defined by the developer. The same will be the case in the communication between the monitoring server within a NFVI and the monitoring server in the Service Platform.

Homogenize databases and schemas

Last but not least, an important decision towards achieving better scalability and reliability response, databases and schemas will be modified accordingly. This will also simplify maintenance effort of the SONATA platform as a whole.

3.3.2.2 Monitoring metrics extensions

One of the issues that are of paramount importance for the second year is the support of the SONATA's Use Cases to be demonstrated as pilots during the project lifetime. From the monitoring point of view, addressing this issue comes with a plethora of requirements for supporting custom metrics, as discussed below.

Development of the monitoring mechanism to collect data from OpenFlow switches (ODL server)

It is a priority of the project to finalize the development and implementation of a mechanism that will not only collect data from OVS switches (through the SDN controller) but most importantly handle the data and trigger the actions from the developer (or SSM/FSM) side.

Support metrics for VMs/Containers and custom developer metrics

Moreover, in order to support the SONATA's Use Cases, there is a clear need to extend the current list of supported metrics and rules. This extended list of metrics and rules does not only include metrics tailored to specific deployment technologies (such as Virtual Machines, LXC, etc.) but also the development and support of any stringent requirement for integration of user-specific metrics (either related to business or technical characteristics). This case also includes the definition of a clear strategy on the way that a custom metric software will be automatically installed along with the VNF, without the need for any human intervention, just as it is the case until now.

Rules based on multiple metrics (running on different VNFs/POPs)

Another capability that will be supported on the new version of the monitoring framework release, as described in D4.2 [16], will be the ability for a developer to define metrics and thresholds on each VNF (even if deployed in different PoPs) and further define alerting rules for the monitoring framework at the Service Platform level. Thus, it would be possible to define a rule such as: "send an email when the packet loss on VNF #1 deployed on PoP #3 exceeds 30% AND when the latency in VNF #2 deployed on PoP #2 exceeds 200msecs".

3.3.2.3 Integration with other SONATA components and adopted mechanisms

The monitoring framework is part of the SONATA solution as a whole, and in this sense, it has to comply with rules implied by other components. As it became evident in previous deliverables that, the monitoring framework communicates with several components (e.g. SLM, IA, GK, GUI, SSM/FSM, etc).

For the second year, and given the functionality enhancements and priorities described in the rest of the SONATA components, the monitoring framework will comply with the mechanisms presented in the following components.

Integrate AuthN/AuthZ mechanism with monitoring framework

The existing solution will be enhanced, providing much more secure communication not only between end-users and the SONATA Service Platform but also between the SP's components themselves.

In this respect, monitoring data must be part of this concept, requiring to follow specific architectural decisions, as described in the respective section of the deliverable.

Align with SONATA user management policies

Moreover, one of the main concerns of the potential customers of SONATA with regard to its adoption, is the user management policies enforced (or foreseen) in this context. Thus, the monitoring framework will take all the appropriate development actions to support user management at the Service Platform level.

Extend unit and integration tests

As part of the activities related to the seamless integration of the monitoring framework with the rest of the SONATA components, it is foreseen that the list of unit and integration tests will be enhanced, targeting the successful deployment of a Network Service without compatibility issues in the Operational environment of the SONATA ecosystem.

3.3.3 Additional Infrastructure Abstractions

During the second iteration of requirements elicitation and use cases definition, the addition of new features at the infrastructure abstraction layer is considered. The new additions are:

1. IPv6 Support
2. Support for alternative VIMs.

For the latter topic, SONATA targets a complete implementation for a suggested VIM deployment (after analysis of the candidate technologies). Whereas for the first topic an assessment approach is anticipated, based on the fact that IPv6 support is considered mainstream for most of the technologies used at the Infrastructure substrate, although levels of support may vary.

3.3.3.1 IP Protocol Version 6 support

The project will consider support for IPv6 both for the deployment of the Service Platform itself, but also for the instantiation of Network Services (NS) on top of IPv6 NFVI-PoPs. The activity and implementations related to this topic will initially include assessments of the current level of support of all involved components of the SP. The process will allow fixes in communication between the components following the DevOps approach used for the IPv4 version of the SP. In cases where dual stack approaches are operational, implying existence of both protocols, no modification on the current implemented SP components will be made.

In relation to the support for the deployment and instantiation of NS, the support of IPv6 at the PoP level is required. SONATA involvement in this layer is only for integration and deployment of particular features required for the integration, qualification and demonstration environments. In this context, the current implemented NFVI-PoPs are based on OpenStack. The used version is Juno, which natively supports IPv6. More specifically, OpenStack allows the following features:

- Dual-stack (IPv4 and IPv6 enabled) instances.
- Allocation of instance with an IPv6 address.
- Communicate across a router to other subnets or the internet.
- Interaction of instances with other OpenStack services.

Since Kilo, OpenStack supports for ensuring the tenant network can handle dual stack IPv6 and IPv4 transport across a variety of configurations. This same level of scrutiny has not been applied to running the OpenStack control network in a dual stack configuration. Similarly, little scrutiny has gone into ensuring that the OpenStack API endpoints can be accessed via an IPv6 network. At this time, Open vSwitch (OVS) tunnel types (STT, VXLAN, GRE) only support IPv4 endpoints, not IPv6, so a full IPv6-only deployment is not possible with that technology.

In addition to the above, it has to be noted that the deployment of VNFs on top of OpenStack requires steps and implementations from the developers to support IPv6 (single or dual stack too). The anticipation is that in this assessment, IPv6 ready VNFs will be used in order to assess the level of SONATA support for IPv6 considering the whole provisioning chain, from the development to the deployment and instantiation of an IPv6 enabled NS.

3.3.3.2 Alternate VIMs support

An extension of the set of VIMs supported by the SONATA service platform requires an analysis on the implications brought by different kind of virtualisation technologies available. An initial analysis was done as part of deliverable 6.1 [17]. This section contains a comparison between the technologies that have been taken into consideration, focusing on the functionalities they offer and on the implication of their usage on the whole SONATA architecture. The section is divided in two parts: the first part analyses the VIMs based on hypervisors, offering services in the form of virtual machines similarly to OpenStack (the first VIM managed by SONATA). The second section analyses VIMs based on Linux Container.

Virtual Machine based VIM

The available VM-based VIMs can be further divided in two sets: open source VIMs that are more targeted to the research community, and enterprise solutions, widely accepted and used both in operators and cloud companies. We took into consideration a total of three VIMs, as detailed

below.

OpenVIM [21] - is an open source VIM available under the Apache 2.0 license. It is a VIM aiming to be optimized for VNF high and predictable performance. Although it is comparable to other VIMs, like OpenStack, it includes control over SDN with plugins (floodlight, OpenDaylight) aiming for high performance data plane connectivity. It offers a CLI tool and a northbound API. The orchestration component (OpenMANO) uses this northbound API to allocate resources from the underlying infrastructure; which includes the creation, deletion and management of images, flavours, instances and networks. OpenVIM provides a lightweight design that does not require additional agents to be installed on the managed compute nodes.

VMWare vCloud Suite [39] - VMWare vCloud Suite is an integrated solution that blends together VMware's vSphere hypervisor and VMWare vRealize Suite cloud management platform. This combined solution allows the infrastructure owner to offer to the SONATA platform APIs to deploy VMs on the flight.

Amazon Web Services (AWS) [5] - AWS is a well-known platform for cloud computing offered by Amazon. The API offered by the system to deploy virtual machines is extensive and used as reference in many other cloud managers. Anyway, since the service is operated as a black box, it offers limited programmability and flexibility to the service platform, in terms of placement or advanced networking features.

In general these VIMs offer a good start for supporting SONATA adoption in an industrial context. As different VIM backends expose an heterogeneous set of APIs, it increases the effort needed to design the proper infrastructure adaptor in the relevant architectural layer. But they may natively offer an extended support for security, resource isolation and advanced features for networking (support for tenant networks , facilities for function chaining). These native features ease the development of the SONATA IA. As a final note, these solutions come with license constraints and limited terms of use, due to their corporate nature. Moreover they offer limited support for trial.

Container Based VIM

Software containers are being more popular these days. Even if VNF development had mainly used VM, software development heads more and more towards the use of containers. It is easy, fast to deploy and the application have a much lighter overhead than with VMs. Moreover using VNF VM image with docker is possible by converting it to a Docker image [7]. Various add-ons are available and various software such as Kubernetes [8], Swarm [11] and CoreOS [6] provide infrastructure for container clustered deployments.

Kubernetes - Among these tools, Kubernetes is the most complete. It is quite mature and used by other projects such as MESOS, Mesosphere and DC/OS. Kubernetes is a container (Docker) manager that provides a platform for automating deployment, scaling, and operations of application containers across a cluster of hosts. It provides, for example, an automatic DNS for service discovery or Virtual IPs for services load balancing. Kubernetes could be considered as a VIM

abstraction by itself. If so, SONATA would be compatible with any IaaS, as long as you can create a Kubernetes cluster inside it. The implications of using this model would be many, in terms of resource allocation and isolation. In particular, the control over resource allocation might become more loose. Currently, the Infrastructure Adaptor takes care of steering traffic around the WAN, inside the NFVi-PoP, within the OpenStack (VIM) tenant network and, later, inside the Kubernetes space. By using a higher level of abstraction, this architecture provides more granularity to control and manage the network.

Unfortunately Kubernetes doesn't manage the network, it delegates the responsibility to an underlying backend. So there isn't any network resource to manipulate in Kubernetes apart for the network policies [9]. The network policies are "firewall" rules for isolating the flow across containers. But there are several projects that work on the networking aspects of Docker based orchestration.

- **Calico** networking relies on BGP routing tables to steer traffic corresponding to a FG or a SFC. Calico creates a L3 network fabric using the components displayed in the Figure 3.12. Moreover Calico supports OpenStack and is able to talk to Neutron. The choice of BGP is to mimic the internet architecture and to provide a high level of scalability. But relying on BGP is old fashion while OpenFlow allows much more flexibility and control for implementing SFC.

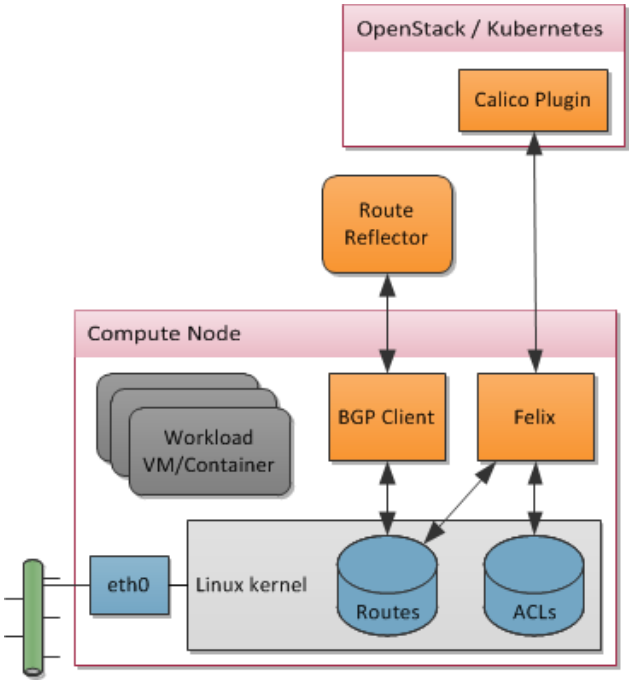

Figure 3.12: Calico network architecture *(source: [30])*

- Another alternative is to create an overlay network to connect containers running on different hosts. It can be achieved using projects such as **Weave** [40] or **Flannel** [18]. Among these tools, Flannel is the most complete and compatible with Kubernetes. The Figure 3.13 shows the internal setup of a Flannel overlay network. It manages IP addresses and reserves a subnet to each host with containers. Then the packets are boxed into a VXLAN frame. The Flannel daemon customizes the OS routing table to relay the packet to the correct host onto the underlying private network. With its current form, a SFC is not available without customizing the Flannel daemon or injecting new rules.

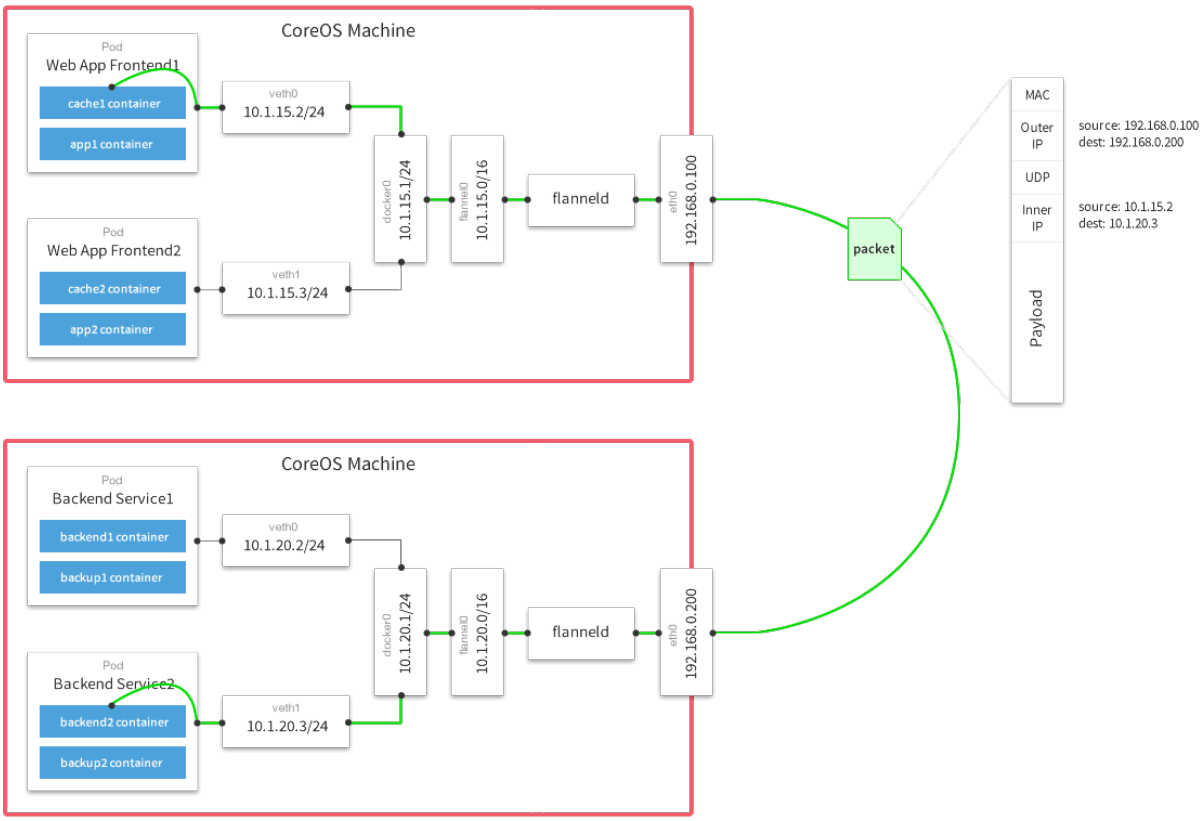

Figure 3.13: Flannel network architecture *(source: [18])*

- The next option is to create a GRE mesh and to configure Kubernetes on top of **OVS**. This option is more difficult but takes full advantage of SDN's features by having a more flexible network and Service Function Chaining through OpenFlow rules. The challenge will be to manage a controller and push the adequate rules. The Figure 3.14 shows a simplified setup based on OVS.

- **Romana** is a L3 fabric that try to avoid the overhead of encapsulation. It uses a complex IP Address Management (IPAM) to split the IP range into buckets. Each bucket level has its own semantic: host, tenant or segment. The Figure 3.15 shows how Romana controls the kernel routes to forward packets to the corresponding destination using the information stored into the IP.

3.3.3.3 Implications on SONATA's Architecture

In order to support two different virtualisation technologies, the service platform should offer a furthered level of validation to service on-board in the catalogue. In fact specific Service Platform instances could be configured to use just NFVi-PoP where a VM-based VIM is deployed, or vice versa. In this case, developers can on-board, into the Service Platform, just services and VNFs based on the technology that matches the one available in the platform. To this aim, the Gatekeeper must validate each on-boarded function against the list of available NFVi-PoP and the relevant deployed VIMs.

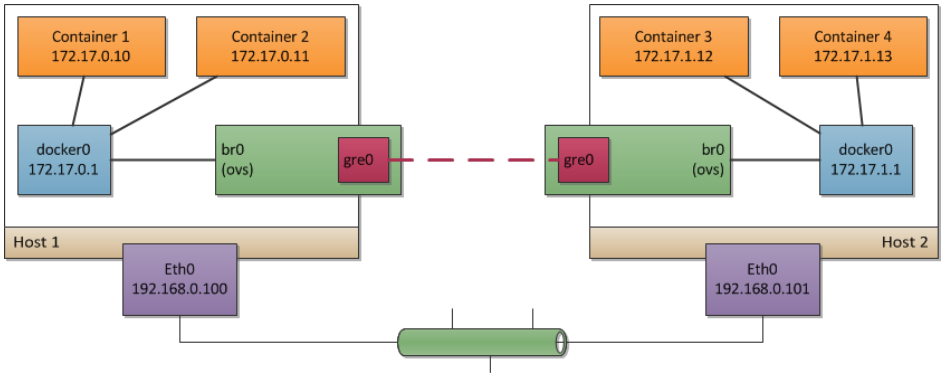

Figure 3.14: OVS network architecture *(source: [31])*

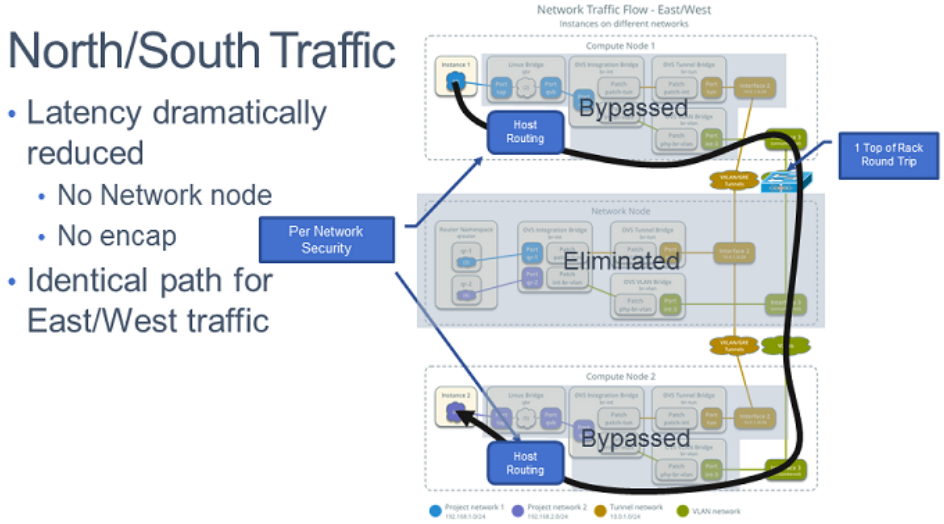

Figure 3.15: Romana network architecture *(source: [34])*

In the most general case, a Service Platform instance offers both NFVi-PoPs based on hypervisors and NFVi-PoPs supporting container virtualisation. Therefore during the instantiation of a network service, the default placement algorithm, or the placement-SSM shipped with the service, are responsible of selecting, for each VNF deployed, a PoP(VIM) which matches the VNF virtualisation technology. To this aim the Infrastructure Abstraction layer offers the functionality to retrieve the list of PoPs and the relevant VIMs deployed in them.

Container Based Options with VM isolation

For VNF development, we have mainly two approaches. Use the most conservative, but also secure approach by embedding the network function into VM. In the other side, developers can also choose the container approach. This option has many advantages, the VNF is much lighter, faster but its isolation is less efficient and secure compare to a VM. In contrast with VMs that rely on hypervisors and hardware instructions for isolation, all containers, running on the same host, share the same kernel and rely on kernel tools such as cgroups and namespace for isolation. As a consequence, the container based option is considered less secure than the VM based solution. Each approach has pros and cons. But if a developer wants to provide his VNF in the two worlds, it requires two separate developments tree. The container based option with VM isolation not only allows a developer to develop his VNFs for containers but also to run them with VM isolation. The container may be executed through a hypervisor to bring the extra level of security that a VM can assure.

This setup is possible by using Kubernetes with Rkt. Kubelet must be configured to use Rkt as the container runtime. Then, Rkt brings an extra level of flexibility by being able to launch Docker image with basic namespace/cgroups isolation (default) but it is also able to run Docker image under the KVM hypervisor [10].

3.3.4 Service Platform (SP) Security

This section describes different security aspects taken into account by the SONATA SP in terms of its operation. The main security approach of SONATA SP is to verify “Who” can access (Authentication) and “What” can be accessed (Authorization). As SONATA is pure software project, the security of the SONATA code plays a vital role in improving the security of SONATA SP as a platform. However the security related to the SONATA source code will be dealt in detail in D5.3, as it is related to the CI/CD process of SONATA development. This section covers:

- User Management.
- Micro-Services Security.
- SONATA Catalogues Artefacts Security.

3.3.4.1 User Management

This section treats the user management of the SONATA Service platform, under the scope of the users and their respective roles.

The planned features that will be included in the SONATA architecture are:

- Single Sign-On (SSO) with a centralized user authentication and authorization approach.
- Social login: a single sign-on procedure using existing information from a social networking service such as GitHub, Twitter or Google+ (Git accounts in case of SONATA), to sign with a third party site instead of creating a new login account.

- The use of HTTPS protocol and digital certificates.

Users and Roles

There are two types of users in SONATA: the SONATA SP users and the End-Users of the deployed services on the SONATA SP. This document only contemplates the Service Platform users and assumes that the service End Users shall be managed by the business logic of each service itself.

Figure 3.16 shows the roles in which the SP users can be classified: platform admin, customer or service manager and developer.

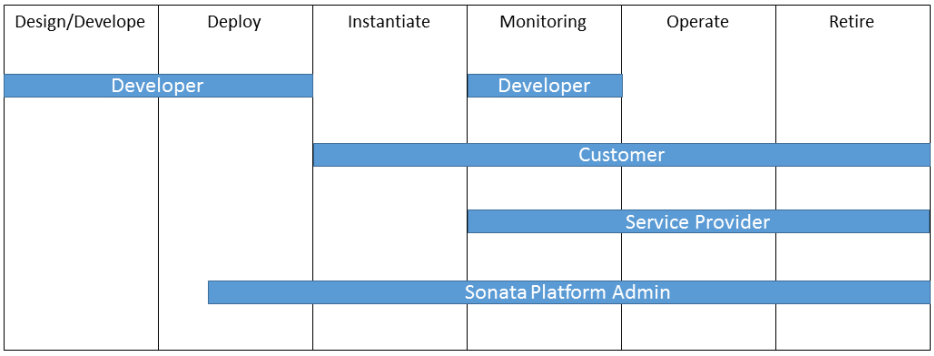

Figure 3.16: SONATA Platform Users and Roles

Attending to the roles, the different SONATA platform users can be:

- Developer: responsible to provide and support VNFs and NSs as products to some service customers or service providers. Responsibilities:
  - Create/Develop NS/NFVs for the platform.
  - Deploy NS/NFVs into the platform for later instantiation.
  - Monitor instances to understand the requirements to build/modify a NS/VNF (issues, bugs, etc.).
- Customer: responsible for the operation of network services, for service users to consume them. Responsibilities:
  - Request new instances of the NSs.
  - Request management operations after a service is deployed: pause, resume, retire, update/upgrade.
  - Monitor instances.
- Service Provider: offers the SP's infrastructure, management and orchestration services to the service customers, in order to host instances that support service customers' users. Responsibilities:
  - Request management operations after a service is deployed: pause, resume, retire, update/upgrade.
  - Monitor instances.
  - Manage customer permissions through the User Management Module.
- Platform admin

- Manage the resources, the NFV specs, the NFV instances, the NS specs and the NS instances, allowing their publications and instantiations.
- Operational management based on security, SLAs (scale up), resources (scale down), malfunctions, etc.

User Authentication and Authorization

Figure 3.17 describes the user registration, authentication and authorization process.

The current focus on the User Management component is to use OpenID Connect (OIDC), OAuth 2.0 and Json Web Tokens (JWT). OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an AuthC/AuthZ Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

In this context, the user management architecture can be split into two main components:

- the son-gtkuser adapter: the sonata component who connects the user clients (GUI/BSS) with the authorization and authentication system.
- the identity and access manager open source tool.

3.3.4.2 Micro-Services Security

As SONATA's architecture is based on micro-services, it considers the micro-services security in terms of authentication and authorization. That is, all the micro-services in the SONATA SP can be authenticated and regulated based on defined policies. Every micro-service can register itself with the AuthZ/AuthZ module in the Gatekeeper and receive a token. This token refers to both the identity and the policy/role of the service and hence it can be used for authorized communication with other micro-services. The AuthC/AuthZ module keeps tracks of all the valid and revoked tokens and any micro-service can query this module for validating a token. Figure 3.18 represents the centralized architecture that will be used for micro service authentication/authorization:

Figure 3.19 shows the message sequence chart that describes the Micro-Service Registration and Authorization process.

The Gatekeeper serves as the first point of contact for the rest of the world and, with its AuthC/AuthZ module, it ensures strict user authentication and authorization, hence reducing the chances of external attacks. On the other hand, SONATA SP plans to implement all its micro-services' API using HTTPS to improve the overall security. In that light, implementation of micro-service authorization is not necessary. However, if required, it could be achieved as described above. Even after all this, a rogue developer may try to execute malicious code by exploiting the SSM/FSM feature of SONATA architecture. This potential vulnerability is addressed in the subsequent subsection.

FSM/SSM security considerations

Function-/Service-Specific Managers (FSMs/SSMs) are executed in the service developer's domain, in complete isolation and with controlled interfaces towards the Service Platform. As shown in Figure 3.20, each FSM/SSM can only communicate with the corresponding executive plugin through an isolated message broker. The executive plugin controls and, if necessary, filters the information that the FSM/SSM requires (e.g., information about underlying network resources). Similarly, the results produced by the FSM/SSM are checked by the executive plugin, to make sure no unauthorized action is taken by the FSM/SSM.

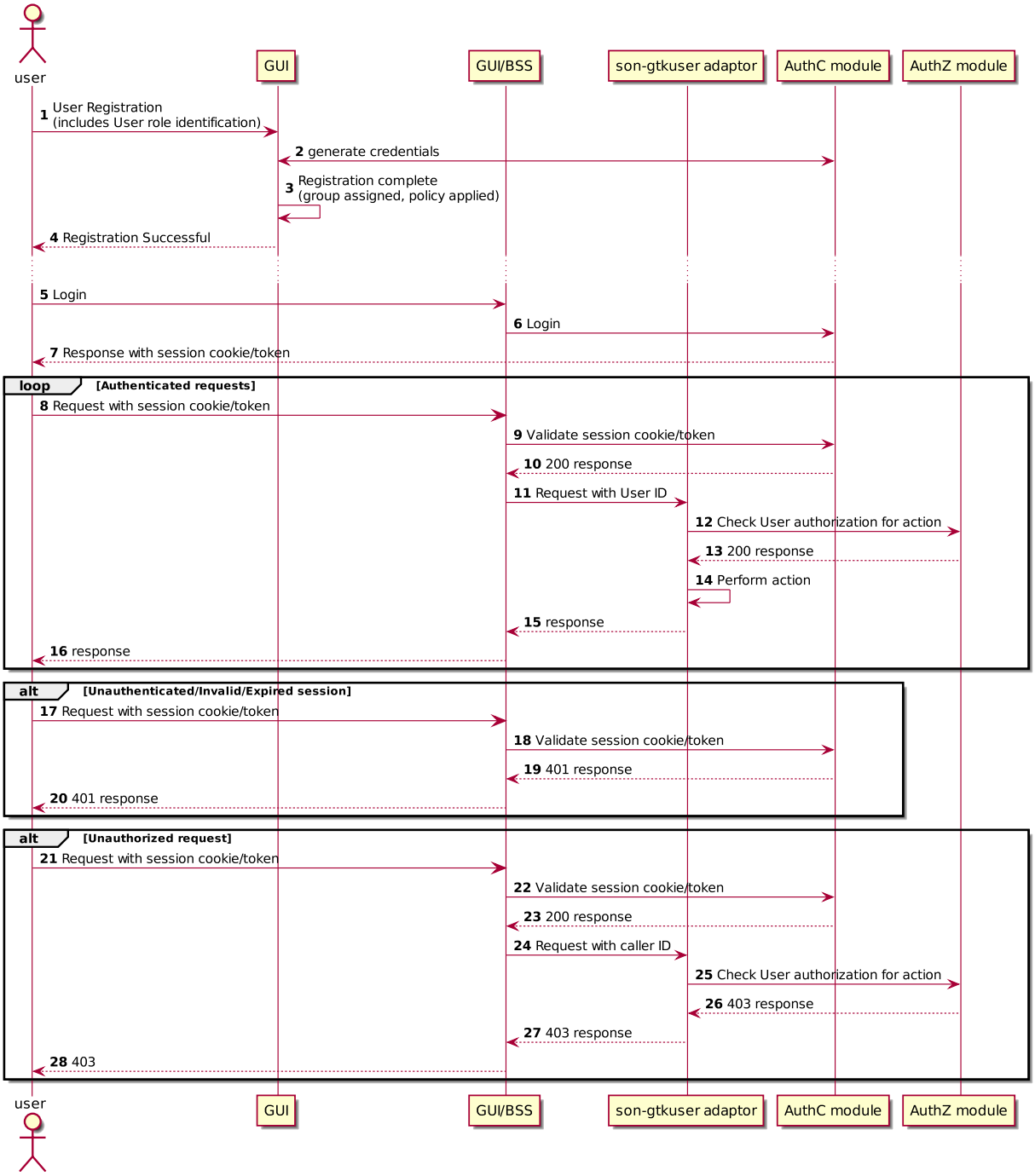

Figure 3.17: User Registration, Authentication and Authorization

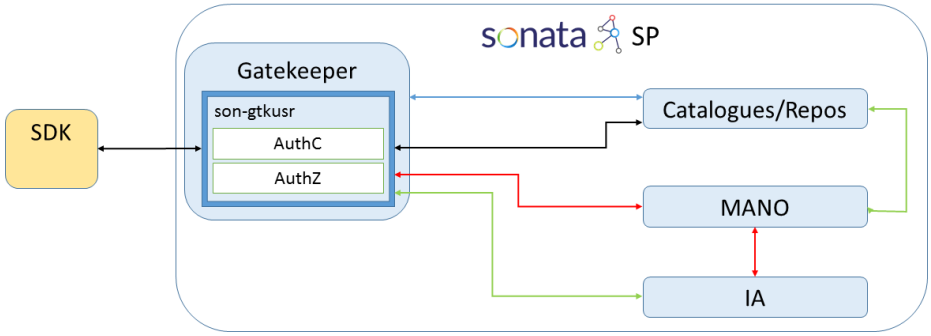

Figure 3.18: SP Architecture: centralized approach

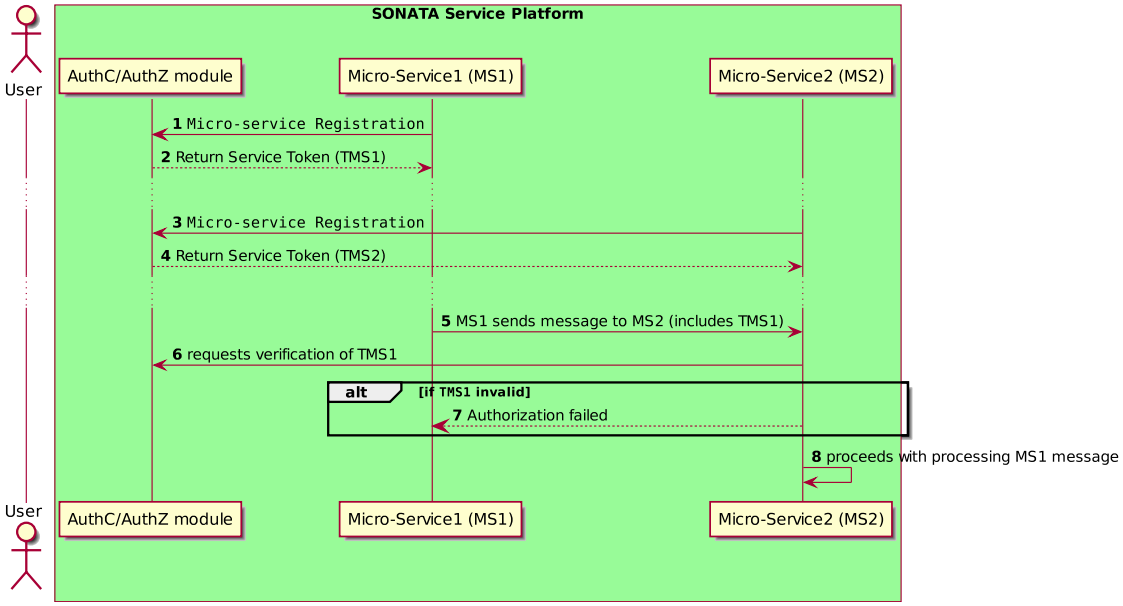

Figure 3.19: Micro-Service Registration and Authorization

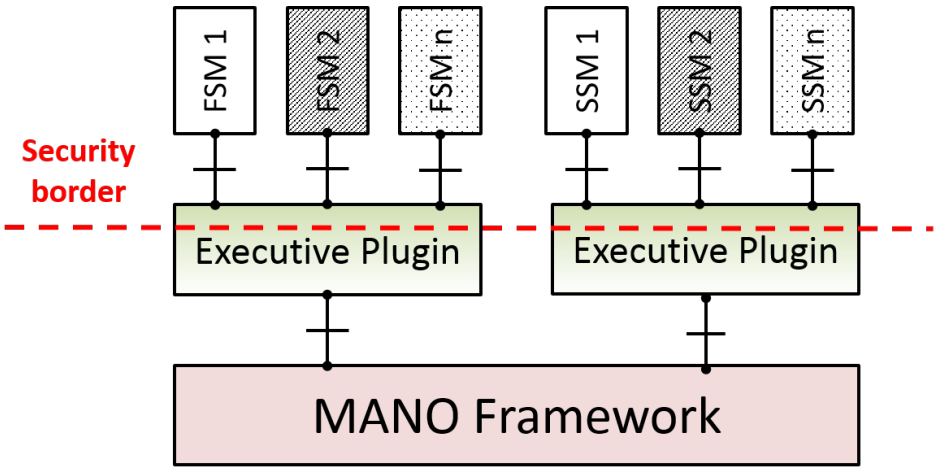

Figure 3.20: Executive plugins acting as a security border between FSMs/SSMs and the MANO framework

3.3.4.3 SONATA Catalogue Artefacts' Security

As mentioned before, the SP Catalogues store different artefacts including Package Descriptors (PDs), NSDs, and VNFDs. It is of utmost importance to be able to verify the authenticity and integrity of these artefacts. That is, to ensure

- the identity of the developer who is submitting the artefact to the SP via the Gatekeeper
- the integrity of the artefact itself

The above two elements are important to avoid malicious users submitting bogus or pernicious packages to the SP Catalogues and also to mitigate Man-in-the-Middle attack where a user session hijacking is followed up by changing the package contents, thus compromising the integrity of the package. The User Management module, in the Gatekeeper, handles the former one to only allow access to registered developers (over HTTPS). The latter one is addressed with aid of accompanying digital signatures and message authentication codes (MAC) using certificates for the artefacts. The certificates of each artefact are also stored in the SP's Catalogues along with them as meta-data.

3.3.4.4 User Management Module design

This section defines the User Management module architecture which is responsible of users and services authentication and authorization.

Module architecture

The User Management module architecture defines two main components:

- Adapter (son-gtkusr)
- Identity and Access Management open-source tool (Keycloak)

These two components communicate through secured RESTful interfaces as they follow the micro-service architecture pattern inside the module.

The Figure 3.21 shows the initial design for the User Management module architecture where the Adapter component enables an Access REST API and interacts with the Access Management and Identity Provider tool.

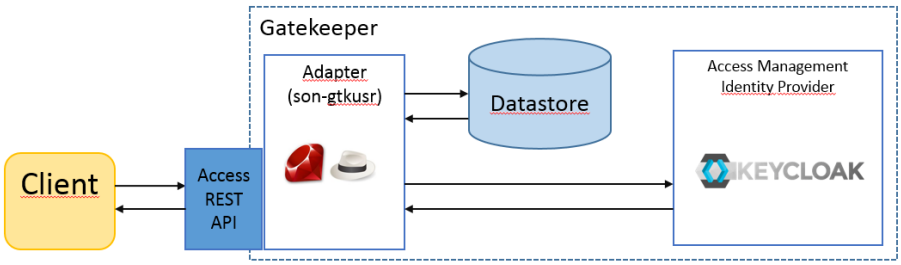

Figure 3.21: User Management sub-module architecture

The Adapter (son-gtkusr) will be implemented from scratch using Ruby Programming language and Sinatra Framework (following the Gatekeeper philosophy). It will communicate with the User management - Identity Provider securely for the authentication and authorization processes.

The open source tool Keycloak has been chosen to act as Identity / Authority Provider, while other candidates were Hydra or Anvil Connect. However, we foresee that some features may require implementation that can be integrated into the son-gtkusr as a Ruby plus Sinatra microservice with the support of a Database.

Keycloak is an open source identity and access management solution that enables authentication to applications and secure services. It supports features like Social Login (Github accounts), Standard Protocols such OpenID Connect and OAuth 2.0, and it is lightweight, fast and scalable.

The Adapter acts as a Resource Provider (RP) and uses the Keycloak to provide access to external interfaces. Keycloak exposes the endpoints that the Adapter component functionalities require. Currently 3 main functionalities have been identified for the Adapter:

1. Set/Get configuration: The Adapter must be configured to obtain the endpoints available from the Access/Identity Provider and act as a client with admin credentials. The Adapter can request a special access token or use certificates to communicate securely with the Access/Identity Provider.

2. Get access: The Adapter will be a microservice using the OAuth 2.0 specification using admin Client Credentials as authorization grant, which is obtained using its Client ID/Secret pair or Private/Public key pair. Microservices inside/outside the Platform will be using Client Credentials grants (Services accounts), while users will be using Resource Owner Password Credentials grants (User accounts).

3. Requests/Responses: The Adapter will enable a REST API to accept operations from users/microservices. However this API will expose public interfaces (e.g. register to the Platform) and secure interfaces (e.g. authenticated/authorized processes). The secure interfaces of the API will work with JWT tokens and OpenID Connect protocol (on top of OAuth 2.0).

Identity and permissions management

The Access/Identity Provider needs to feature SSO and Social-Login (Git accounts) from external Identity Providers. OpenID Connect allows to generate Access/Authentication Tokens to support SSO, where a user is granted with an access token for sending requests to the Platform until the access token expires.

- Authorization will be based on roles/scopes encoded on granted tokens.

- Roles/Policies can be matched to Groups for different types of users (e.g. End-user).

Identity: OpenID Connect

OpenID Connect (OIDC) is an interoperable authentication protocol based on the OAuth 2.0 family of specifications. It is the new emerging standard for Single Sign-on (SSO) and Identity Provision on the internet. Its formula is based on simple JSON-based Identity Tokens (JWT) and uses straightforward REST/JSON message flows allowing clients of all types, including Web-based, to request and receive information about authenticated sessions and end-users.

While OAuth 2.0 is only a framework for building authorization protocols, OIDC is a full-fledged authentication and authorization protocol. JWT standards and OIDC define an identity token JSON format and ways to digitally sign and encrypt that data in a compact and web-friendly way.

There is really two types of use cases when using OIDC:
An application client asks the Keycloak server to authenticate a user. After a successful login, the application receives an Identity Token and an Access Token. The Identity Token contains information about the user and other profile information. The Access Token is digitally signed by the Authorization Server and contains access information (like user role mappings) that determines what resources the user is allowed to access.

- An application client wants to gain access to remote services. In this case, the client asks Keycloak to obtain an Access Token it can use to invoke on other remote services on behalf of the user. Keycloak authenticates the user and the client receives the Access Token. This Access Token is digitally signed by the Authorization Server. The client can make REST invocations on remote services using this access token. The REST service extracts the access token, verifies the signature of the token, then decides based on access information within the token whether or not to process the request.

OpenID Connect is used for authentication and authorization following the "Authentication using the Authorization Code Flow". When using the Authorization Code Flow, all tokens are returned from the Token Endpoint, from Keycloak through the Adapter. The Authorization Code Flow returns an Authorization Code to the end-user client, which can then exchange it for an ID Token and an Access Token directly. The Authorization Server can also authenticate the client before exchanging the Authorization Code for an Access Token. The Authorization Code flow is suitable for clients that can securely maintain a client Secret between themselves and the Authorization Server. The Adapter forwards an end-user or service client authentication and authorization request to the Authorization Server, Keycloak. It directly authenticates and authorizes the client with an Access Token.

The User Management module will consider two types of accounts regarding end-users or services:

- User Account
- Service Account

User Account

Each user has a User Account and will belong to a particular type of User as defined above. This account uses the User Credentials grant type (a.k.a. Resource Owner Password Credentials) when the user has a trusted relationship with the client, and so can supply credentials directly.

This is covered in the OAuth 2.0 specification under Resource Owner Password Credentials Grant [23].

Use Cases:

- When the user Client wishes to display a login form
- For applications owned and operated by the Resource Server
- For applications migrating away from using direct authentication and stored credentials

Service Account

This feature allows to authenticate the client application with an authentication and authorization server and retrieve the access token dedicated to this application. No interaction with users is needed. This is great for tasks executed on behalf of a service instead of individual user. Any registered micro-service will be assigned a Service Account.

To support service accounts, Client Credential grant type is used to retrieve access token granted to the client. The Client Credentials grant type is used when the client is requesting access to protected resources under its control (i.e. there is no third party).

This is covered in the OAuth 2.0 specification under Client Credentials Grant [24].

Use Cases:

- Service calls
- Calls on behalf of the user who created the micro-service client

Permissions

From the Authorization Server (Keycloak), a permission associates the object being protected and the policies that must be evaluated to decide whether access should be granted. To protect resources of the Service Platform and authorize actions, policies and permissions have to be created. Role-Based policies define conditions for permissions where a set of one or more roles is permitted to access an object or perform an action. Roles assigned to a policy will grant access if the user requesting access has been granted any of these roles. Permissions provide more granularity to define allowed access to resources and the actions that can be performed on them. Permissions can be created based on two main types of objects:

- Resources, e.g. descriptors in the Service Platform Catalogue
- Scopes, e.g. what actions an end-user is allowed to perform

3.4 Service Platform Management and Setup

The SONATA Service Platform (SP) can be deployed and managed by the "**son-cmud.yml**"[37] Ansible playbook.

- Create a new SP from the scratch
- Manage the SP or just individual services (eg., start, stop, restart)
- Upgrade the SP (not implemented yet, but in the roadmap)
- Destroy the SP or just individual services (ie., remove/uninstall from the system)

In the SP version 1, all services run inside Docker containers behaving as isolated micro-services with interfaces and dependencies among them. The following sub-chapters present the correct sequence that is applied due to the precedence on individual services.

The execution of the playbooks in the next sub-chapters will act over the SP with the following options:

- Target: is the VM where the SP will run
- Operation: is the operation to execute, namely: INSTALL, MANAGE, UPGRADE or DESTROY
- Service: is the SP service, namely: ALL or a specific service
- Env: is one of the environments defined, namely: INTEGRATION, QUALIFICATION or DEMONSTRATION

- Distro: is one of the Linux distros supported to run the SP, namely: Ubuntu 14.04 (trusty), Ubuntu 16.04 (xenial) or CentOS 7

The "son-cmud.yml" playbook looks like:

```
---
- name: SONATA-NVF Service Platform CMUD
  hosts: "{ { target } }"
  become: true

- include: "./ { { operation } } / { { service } }.yml"
```

3.4.1 Service Platform Installation

The Service Platform is installed by executing an `ansible-playbook`, like in

```
$ ansible-playbook son-cmud.yml -e "target=localhost operation=install \
    service=all env=''ENV''"
```

Figure 3.22 depicts the sequence for the installation of the Service Platform.

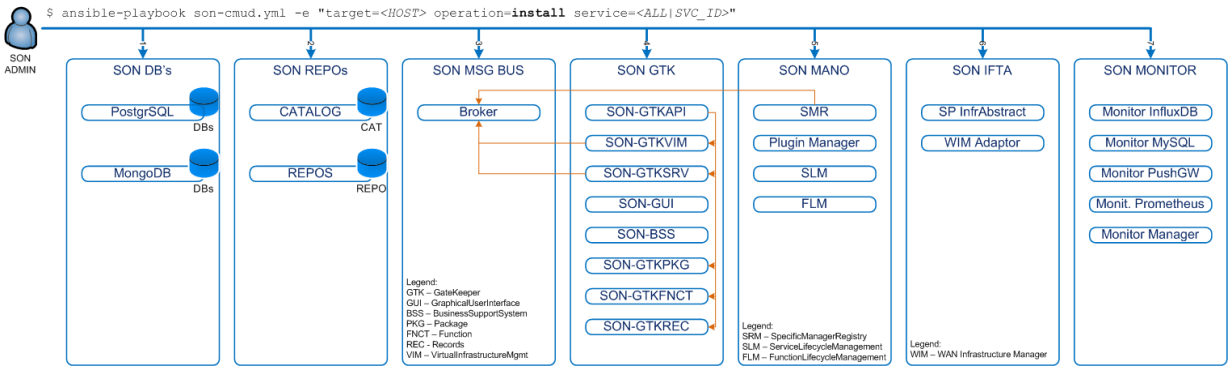

Figure 3.22: Service Platform Installation

The Ansible playbook for the "son-cmud.yml" INSTALL operation looks like:

```
---
- include: common.yml
- include: docker.yml
- include: pgsql.yml
- include: mongo.yml
- include: repos.yml
- include: broker.yml
- include: gtkall.yml
- include: mano.yml
- include: ifta.yml
- include: monit.yml
```

In this case, the `playbook` just includes other `playbooks`, where more detailed instructions are defined.

3.4.2 Service Platform Removal

The execution of the following playbook will deploy a full SP to a specific environment and/or distro:

```
$ ansible-playbook son-cmud.yml -e "target=localhost operation=destroy \
    service=all env=''ENV''"
```

Figure 3.23 shows the sequence for the removal of the installed Service Platform.

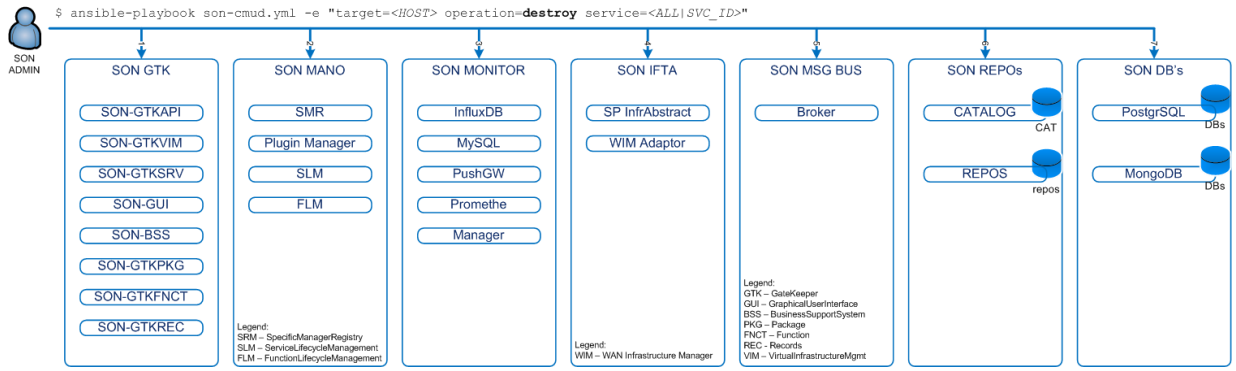

Figure 3.23: Service Platform Uninstallation

The Ansible playbook for the "son-cmud.yml" DESTROY operation looks like:

```
---
- include: monit.yml
  when: service == "all" or service == "monit"
- include: ifta.yml
  when: service == "all" or service == "ifta"
- include: mano.yml
  when: service == "all" or service == "mano"
- include: gtkall.yml
  when: service == "all" or service == "gtk"
- include: broker.yml
  when: service == "all" or service == "broker"
- include: repos.yml
  when: service == "all" or service == "repos"
- include: mongo.yml
  when: service == "all" or service == "mongo"
- include: pgsql.yml
  when: service == "all" or service == "pgsql"
```

As before, this `playbook` is just a collection of includes of other, more specific, `playbooks`, but now this inclusion is restricted to a given condition (in this case, the value of the `service` parameter).

4 Integration of Network Slicing in SONATA Platform

This chapter describes the technical requirements and the related reference model for the integrating network slicing in the SONATA platform, allowing for functional and protocol specifications to be developed in an approach consistent with the SONATA architecture.

Network Slicing is an end-to-end concept covering the radio and non-radio networks inclusive of access, core and edge / enterprise networks. It enables the concurrent deployment of multiple logical, self-contained and independent shared or partitioned networks on a common infrastructure platform.

From a business point of view, a slice includes combination of all relevant network resources / functions / assets required to fulfil a specific business case or service, including OSS, BSS and DevOps processes.

From the network infrastructure point of view, slicing instances require the partitioning and assignment of a set of resources that can be used in an isolated, disjunctive or non- disjunctive manner.

Examples of physical or virtual resources to be shared or partitioned would include: bandwidth on a network link, forwarding tables in a network element (switch, router), processing capacity of servers, processing capacity of network or network clouds elements. As such slice instances would contain:

1. A combination/group of the above resources which can act as a network,
2. Appropriate resource abstractions,
3. Exposure of abstract resources towards service and management clients that are needed for the operation of slices.

The establishment of slices is both business-driven (i.e. slices are in support for different types and service characteristics and business cases) and technology-driven as a slice is a grouping of physical (or virtual) resources (network, compute, storage) which can act as a sub network and/or a cloud. A slice can accommodate service components and network functions (physical or virtual) in all network segments: access, core and edge / enterprise networks.

A complete slice is composed of not only various network functions which are based on virtual machines at C-RAN and C-Core, but it also transports network resources which can be assigned to the slice at the radio access/transport network level. Different future businesses require different throughput, delay and mobility.

4.1 High Level Requirements for Slice Networking

- **Slice creation**: the management plane creates virtual or physical network functions and connects them as appropriate and instantiates them in the slice.

- The **instance of slice management** then takes over the management and operates all the (virtualised) network functions and network programmability functions assigned to the slice, and (re-)configures them as appropriate to provide the end-to-end service.

- **A complete slice is composed of** not only various network functions which are based on virtual machines at C-RAN and C-Core level, but also transport network resources which can be assigned to the slice at the radio access/transport network level. Different future businesses require different throughput, delay and mobility, and some businesses need very high throughput or/and low delay. Transport network shall provide QoS isolation, flexible network operation and management, and improve network utilization among different business.

- **QoS Isolation**: Although traditional VPN technology can provide physical network resource isolation across multiple network segments, it is deemed far less capable of supporting QoS hard isolation, which means that QoS isolation on forwarding plane requires better coordination with management plane.

- **Independent Management Plane**: Like above, network isolation is not sufficient, a flexible and, more importantly, a management plane per instance is required to operate on a slice independently and autonomously within the constraints of resources allocated to the slice.

- Another **flexibility requirement** is that an operator can deploy their new business application or a service in network slices with low cost and high speed, and ensure that it does not affect existing business applications adversely.

- **Programmability**: Operator not only can slice a common physical infrastructure into different logical networks to meet all kinds of new business requirements, but also can use SDN based technology to improve the overall network utilization. By providing a flexible programmable interface; a third party can develop and deploy new network business rapidly. Further, if a network slicing can run with its own slice controller, this network slicing will get more granular control capabilities [4] to retrieve slice status, and issuing slicing flow table, statistics fetch, etc.

- **Life cycle self-management**: It includes creation, operations, re-configuration, composition, decomposition, deletion of slices. It would be performed automatically, without human intervention and based on a governance configurable model of the operators. As such protocols for slice set-up / operations / (de)composition / deletion must also work completely automatically. Self-management (i.e. self-configuration, self-composition, self-monitoring, self-optimisation, self-elasticity) is carried as part of the slice protocol characterization.

- **Extensibility**: Since the Autonomic Slice Networking Infrastructure is a relatively new concept, it is likely that changes in the way of operation will happen over time. As such new networking functions will be introduced later, which allow changes to the way the slices operate.

- **Transport network shall provide QoS isolation**, flexible network operation and management, and improve network utilization among different business. The flexibility behind the slice concept needs to address QoS guarantees on the transport network and enable network openness.

4.2 Network Slices - Key Terms and Characteristics

A number of slice definitions were used in the last 10 years in distributed and federated testbed research [1], future internet research [19] and more recently in the context of 5G research [22] [32] [2] [3]. A unified Slice definition usable in the context of 5G Networking consist of 4 components as proposed at IETF [4]:

- **Service Instance** component
- **Network Slice Instance** component
- **Resources** component
- **Slice Capability exposure/Manager** component

The **Service Instance component** represents the end-user services or business services which are to be supported. It is an instance of an end-user service or a business service that is realized within or by a Network Slice. Each service is represented by a Service Instance. Services and service instances would be provided by the network operator or by third parties.

A **Network Slice Instance component** is represented by a set of network functions, and resources to run these network functions, forming a complete instantiated logical network to meet certain network characteristics required by the Service Instance(s). It provides the network characteristics which are required by a Service Instance. A Network Slice Instance may also be shared across multiple Service Instances provided by the network operator. The Network Slice Instance may be composed by none, one or more Sub-network Instances, which may be shared by another Network Slice Instance.

Slice Capability Exposure/Manager component is allowing 3rd parties to access / use, via APIs, information regarding services provided by the slice (e.g. connectivity information, QoS, mobility, autonomy, etc.) and to dynamically customize the network characteristics for different diverse use cases (e.g. ultra-low latency, ultra-reliability, value-added services for enterprises, etc.) within the limits set of functions by the operator. It includes a description of the structure (and contained components) and configuration of the slice instance.

Logical resource is an independently manageable partition of a physical resource, which inherits the same characteristics as the physical resource and whose capability is bound to the capability of the physical resource. It is dedicated to a Network Function or shared between a set of Network Functions.

Virtual resource is an abstraction of a physical or logical resource, which may have different characteristics from that resource, and whose capability may not be bound to the capability of that resource.

Network Function refers to processing functions in a network. This includes but is not limited to telecom nodes functionality, as well as switching functions e.g. switching function, IP routing functions.

Virtual Network Function is one or more virtual machines running different software and processes on top of 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.).

Figure 4.1 shows the high level models of a Slice Networking.

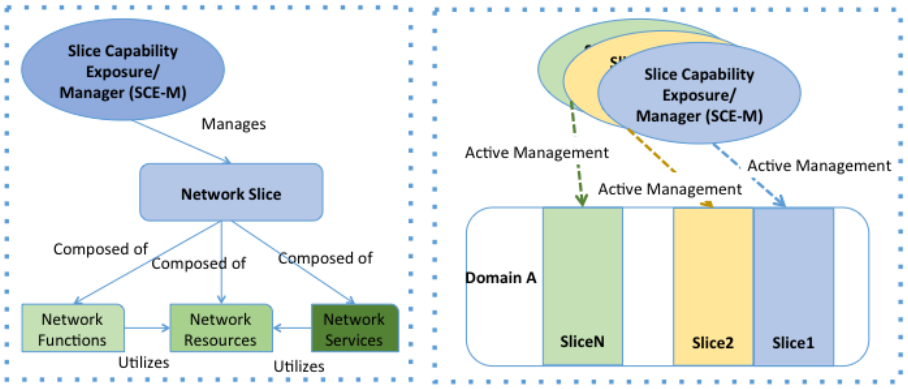

Figure 4.1: Network Slicing Models

4.2.1 Managing a Network Slice

Slice network management is driven by the Slice Manager which is performing four categories of management operations:

- Creating a network slice: Receive a network slice resource description request and, upon successful negotiation with infrastructure, allocate the resources for it.
- Shrink/Expand slice network: Dynamically alter resource requirements for a running slice network according to a service load.
- (Re-) configure slice network: The slice management user deploys a user level service into the slice. The slice control takes over the management of all the virtualised network functions and network programmability functions assigned to the slice, and (re-)configure them as appropriate to provide the end-to-end service.
- Destroy slice network: Recycle all resources from the infrastructure.

As such the **following control APIs** are needed for slicing:

- Create a slice network on user request. The request includes resource descriptions. A unique ID identifies a slice network, which groups all the resources, network functions and service elements.
- Destroy a slice network identified by its id.
- Query a slice network slicing state by its id.
- Modify a slice network.

A number of **key characteristics makes the Network Slicing concept, with added value and serviceability** thanks to the DevOps approach adopted by the SONATA project:

- Concurrent deployment of multiple logical, self-contained and independent, shared or partitioned networks on a common infrastructure platform.

- Supports dynamic multi-service support, multi-tenancy and the integration means for vertical market players.
- Separation of functions simplifies (1) the provisioning of services, (2) manageability of networks and (3) integration and operational challenges especially for supporting communication services.
- Network operators can exploit network slicing for (1) reducing significantly operations expenditures, (2) allowing also programmability and innovation, necessary to enrich the offered services, (3) creating tailored services and (4) supporting network programmability to OTT providers and other market players without changing the physical infrastructure.
- Considerably transform the networking perspective by abstracting, isolating, orchestrating and separating logical network behaviours from the underlying physical network resources.

4.3 Integration of Network Slicing in the SONATA platform

Figure 4.2 depicts the Network Slicing integration in the SONATA platform.

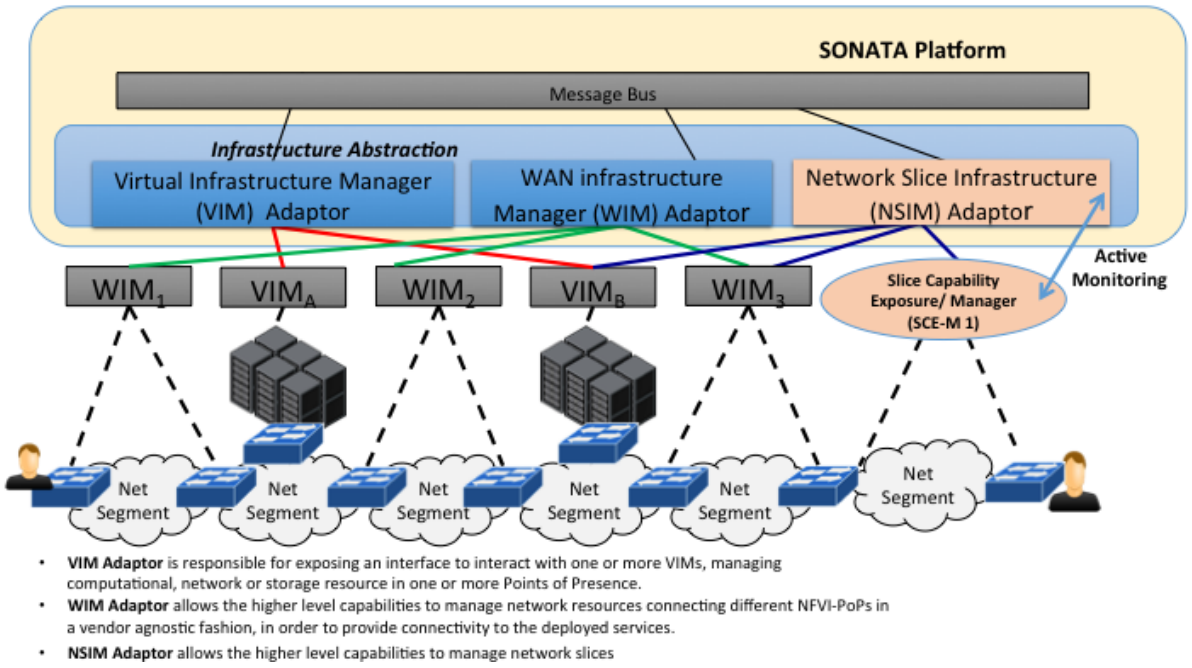

Figure 4.2: Integration of Network Slicing in the SONATA Platform

- VIM Adaptor is responsible for exposing an interface to interact with one or more VIMs, managing computational, network or storage resource in one or more Points of Presence.
- WIM Adaptor allows the higher level capabilities to manage network resources connecting different NFVI-PoPs in a vendor agnostic fashion, in order to provide connectivity to the deployed services.
- NSIM Adaptor allows the higher level capabilities to manage network slices.

New designs for Service chaining based on slicing, network slice life cycle management, Network Slice Infrastructure (NSIM) Adaptor, active slice monitoring and management are planed for the next period of work.

5 Relationship between SONATA and the ETSI Architecture

5.1 Update on the ETSI architecture

ETSI released its first NFV documents several years ago. It now has an on-going activity to revisit and update several aspects of its original work. SONATA partners are contributing with their learning from their experience as NFVs operators and with their conclusions from fruitful discussions during the project to date. Network operators face a number of practical aspects when deploying NFVs. For example, network operators need complete solutions covering all aspects of service, but not all of them have been fully detailed within ETSI NFV's scope. Then most network operators cannot deploy NFVs in isolation from their existing networks and services and these operators need to interwork NFV systems with their existing systems. Moreover, some aspects such as in-life functions (service assurance) have not yet been considered.

5.1.1 Requirements for MANO's functionalities

In the following, we describe the requirements for the MANO's functionalities.

MANO's three groups of capabilities

First we consider what operational processes a NFV system needs to support, in order for the network operator to offer services successfully. These collectively constitute the Management and Orchestration (MANO). We identify three groups of capabilities, which are used at different times:

1. During the development of the service, the creation of a template which describes the service and is uploaded to a catalogue. The SDK tools developed by SONATA are used for this purpose.

2. During the activation and deployment of the service, the instantiation of a service instance and the request of resources to the underlying layer(s). In the SONATA's service platform, the resource orchestration and repositories support this phase.

3. During the runtime of the service, its monitoring and its maintenance. The latter is often called 'service assurance'. SONATA's work on monitoring and reliability goes towards this topic.

We now describe each of the phases at a higher level. We plan to contribute the next level of details into ETSI, building on SONATA's detailed work for enabling its various capabilities.

The development and definition phase is outlined in Figure 5.1. The phase creates templates, each of which describes a service/function offered by the layer. It includes the processes by which the templates are constructed and tested/validated. SONATA'S development environment has a development toolkit, libraries of draft templates and resource types, and an isolated test and validation 'sandbox' execution environment. The templates are uploaded to a catalogue which can then be accessed by the Instance Life-Cycle phase for the creation of service/function instances.

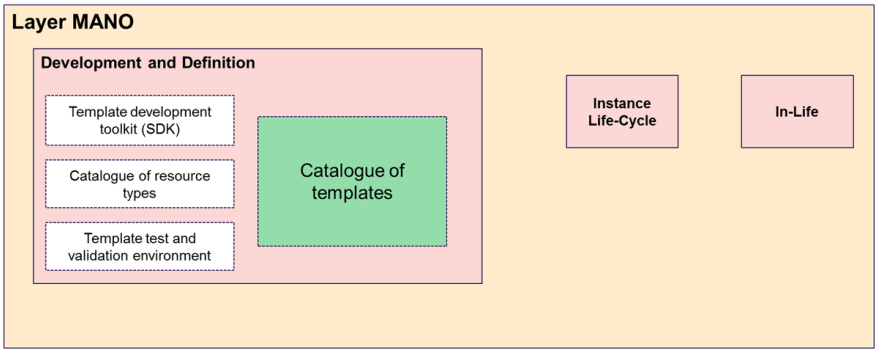

Figure 5.1: High level view of the development and definition phase

The instance life-cycle phase is outlined in Figure 5.2. It results in instances of services/functions. This phase is triggered by a request from a client, which is first validated and authorized. It then selects an appropriate template from the template catalogue, calculates the resources required to instantiate the service/function, and then triggers the actions required to achieve the instantiation. The phase can also be triggered by a request from the in-life phase. For management purposes, each instance is registered in an inventory database together with information about the resources it uses. The phase maintains a full inventory of instances, together with the mapping to the resources supporting each instance.

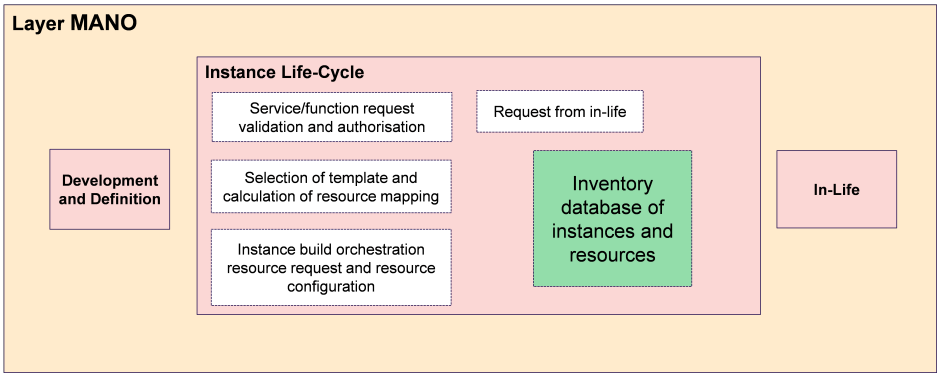

Figure 5.2: High level view of the instance life-cycle view

The in-life phase is outlined in Figure 5.3. Its objective is to maintain the health and performance of instances. The phase monitors the instances, and when it detects that one is faulty or underperforming, it calculates what action to take to solve the problem. This would typically be a restoration or scaling activity, and triggers a request to the Instance Life-Cycle phase. The in-life cycle phase can also report the status of services/functions to clients.

MANO's operation in a wider context

An NFV system does not operate in isolation. In order to achieve end to end operations, the MANO interacts with the wider context of existing network services, functionality and associated

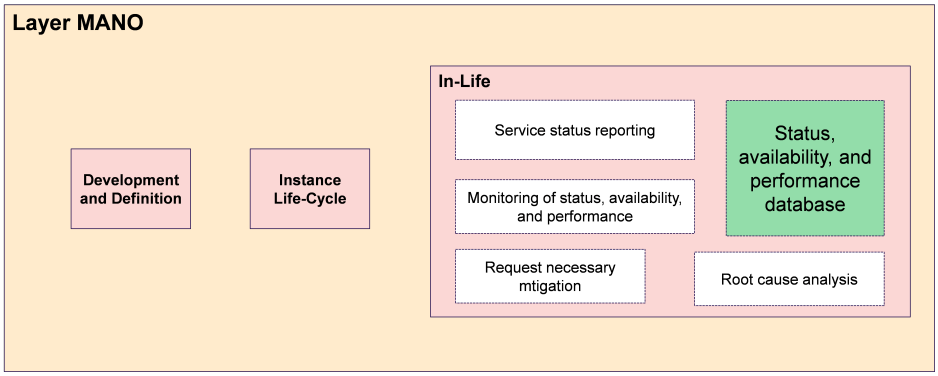

Figure 5.3: High level view of the in-life phase

management and orchestration systems. Firstly, in the context of a single network operator, a NFV system needs to:

1. Interwork with existing systems
2. Support future system evolution
3. Support heterogeneous operator systems

In essence, the network operator will have hybrid services – a mix of the old and new, and systems with overlapping functionality (for instance from different vendors). Information models are likely to help with the integration and the mapping between systems. Secondly, if several network operators are involved, then operators will offer services to each other in order to build end to end services. For example, a network operator needs to be able to:

1. Support VNF as a Service
2. Support NFVI as a Service
3. Support Connectivity as a Service
4. Support NFV network service as a Service

In essence, an operator uses components supplied by another operator (and in turn it supplies components to others). Perhaps the prototypical example is where one operator uses connectivity from another network operator to interconnect NFV nodes of their NFVI. Another example could be the provisioning of a slice.

Layering of MANO functionality

The high level architecture consists of a number of layers. Layering, and its inherent modularity, helps us to meet the requirements outlined above. The details of each layer are hidden from the other layers, and so can be implemented in different ways or changed as long as the service presented at the interfaces is maintained unchanged. In this manner, the requirement mentioned earlier for the ability to include existing services (like WAN connectivity) can be achieved. On the other hand, the same software code could be used to achieve a capability at more than one layer, for instance through a separate call by each layer. The management of each layer is also self-contained, in that it manages the complete control of the services it provides. Therefore every layer needs to include

the three groups of capabilities (or phases) discussed above (development and definition, instance life-cycle, in-life). At an abstract level, as illustrated in Figure 5.4, each layer should have the same interfaces:

1. an integrated service API, which offers services, resources to the layer(s) above
2. a southbound API, which interacts with the service API of the layer(s) below, and as part of this interaction defines what information it wants to hear about – this enables for example the OSS to avoid being flooded with lower layer messages
3. a GUI or human interface for items that need manual intervention (note that such actions are not done via the service API)

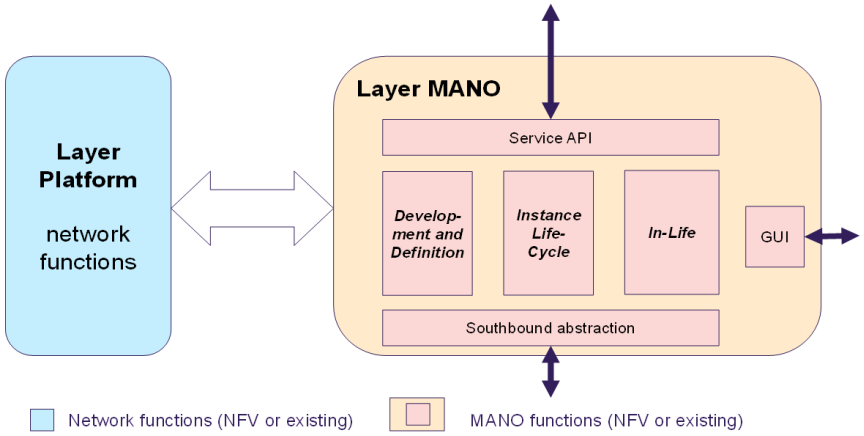

Figure 5.4: Basic entities of a layer MANO

The principle here is that a layer, in order to achieve its operation, will utilise services provided by underlying layers, as they provide it with resources. In a complementary fashion, a layer, in order to be useful, will offer services to overlying layers, as it provides them with resources. This consistent interaction between layers facilitates operation in a wider context: interoperability, operation in a heterogeneous environment, evolvability, XaaS and so on. SONATA provides an example of how to instantiate this. The service platform and the associated MANO framework is a layer. The infrastructure adaptor is the southbound interface and provides resources such as VIM access, whereas the gatekeeper provides the northbound interface (the GUI is implicit). D2.1 [12] describes how the architecture is recursive, so achieving multiple layers.

5.2 Mapping between SONATA and ETSI Interfaces

ETSI describes an architectural framework that acts as a reference for many service platforms and MANO solutions. Thus, it is important to provide compatibility with this framework to support interoperability with other systems and architectures that are also ETSI aligned. This section shows how the SONATA architecture, and especially its reference points, maps to the ETSI architecture. Further, we describe and map the actual interfaces that built these reference points in more detail to clarify how third party components may interface with the SONATA service platform or single platform components.

5.2.1 General mapping between ETSI and SONATA reference points

ETSI defines a set of reference points that define interaction points between different components of the architectural framework. The bottom part of Figure 5.5 shows these reference points in the scope of the ETSI NFV-MANO architecture and maps them to the SONATA architecture. This mapping shows that some reference points of both architectures are more or less the same, e.g., the reference points to the catalogues and repositories. This indicates that these SONATA components can easily be replaced or combined with other ETSI compatible service platforms. For example, if someone wants replace the SONATA catalogue by a third party catalogue component only the reference point between gatekeeper and catalogue needs to be modified. However, technically this means that intermediate wrapper layers might be needed that translate between the actual implemented interfaces.

The figure also shows that the main ETSI reference points can be completely mapped to the SONATA architecture. But it also shows that there are many more reference points in SONATA. This is, at one hand, based on the fact that ETSI is an ongoing initiative which will provide more and more details over time and, on the other hand, that the SONATA architecture already includes advanced customizability features, like FSMs and SSMs as well as a monitoring solution which are not explicitly covered by the rather abstract ETSI reference framework.

Reference point specifications, like shown in Figure 5.5, still define component interactions on a very abstract level. A single reference point may appear with different characteristics in an actual system, for example, the *Os-Ma-Nfvo* reference point appears between BSS and Gatekeeper as well as between SDK and Gatekeeper in the SONATA service platform. To shed more light on this, the next section provides a more detailed view on the actual interfaces SONATA uses and their counterparts in the ETSI architecture.

5.2.2 Specific mapping between ETSI and SONATA interfaces

Figure 5.6 shows a more concrete view on an example SONATA service platform setup, its components and the interfaces between them. It explicitly defines the interface names for all used interfaces, for example, the two interfaces that are mapped to the *Os-Ma-Nfvo* reference point of the ETSI architecture, namely the *Bss-Gk* and *SDK-Gk* interface. It is important to note that this figure does only represent one example configuration of the platform and that it may change if other MANO plugins are connected to the system, e.g., a placement plugin and a scaling plugin might require an additional interface to talk to each other as well as an interface to talk to the service lifecycle manager. However, these changes affect only the MANO framework internal interfaces and the interfaces to external components, like gatekeeper or catalogues are stable.

The following table describes all interfaces shown in Figure 5.6 in more detail and provides a concrete mapping to ETSI reference points:

Table 5.1: SONATA reference points and their mapping to ETSI

ETSI Reference Point(s)	SONATA Reference Point	SONATA Interface	Interface Description
Os-Ma-Nfvo	Son-Gk	BSS-Gk	BSS uses this interface to list, instantiate and update services.
Os-Ma-Nfvo	Son-Gk	SDK-Gk	SDK uses this interface to on-board, instantiate, update, list services.
n.a.	Gk-Mano	Gk-F/Slm	Gatekeeper to MANO framework (F/SLM) interface. Manage service orchestration, e.g., request instantiation.
Nfvo-Nscat, Nfvo-Vnfcat	Gk-Cat	Gk-Cat	Gatekeeper to catalogue interface. Used to store and retrieve artefacts from the service platform catalogue.
Nfvo-Nfvins, Nfvo-Nfvires	Gk-Rep	Gk-Rep	Gatekeeper to repository interface. Access NSRs/VNFRs from the Gatekeeper.

ETSI Reference Point(s)	SONATA Reference Point	SONATA Interface	Interface Description
Vnfm-Rcmi	Mano-Rep	F/Slm-Rep	F/SLM to repository interface. Access NSRs/VNFRs from the F/SLM.
n.a.	Mano-Mon	F/Slm-Mon	F/SLM to monitoring manager interface. Configure monitoring for a service.
n.a.	Mano-Mano	F/Slm-Ia	F/SLM to infrastructure abstraction interface. Actual interaction between orchestration and underlying infrastructure.
n.a.	Mano-Mano	Pm-P	Plugin management interface. Used to register plugins, deregister plugins, heartbeat, and broadcast active plugin status.
n.a.	Mano-Mano	F/Slm-Smr	F/SLM to Specific manager registry interface. F/SLM can request to start, stop, update SSMs.
n.a.	Mano-FSSM	Smr-F/Ssm	Interface to register and control F/SSMs.
n.a.	FSM-Vnf	Fsm-Vnf	Connection to interact, e.g., configure VNFs from a FSM. Actual interface specification depends on VNF and its assigned FSM.
n.a.	Mano-Rep	Ia-Rep	Interface towards the infrastructure repository to hold the state of the connected infrastructure.
Vi-Vnfm	Mano-Vi	Ia-Vi	Interface towards the VIM(s).
Vi-Vnfm	Mano-Wi	Ia-Wi	Supports communication of Infrastructure Adaptor with the WIM(s), e.g., to setup inter-PoP connections.

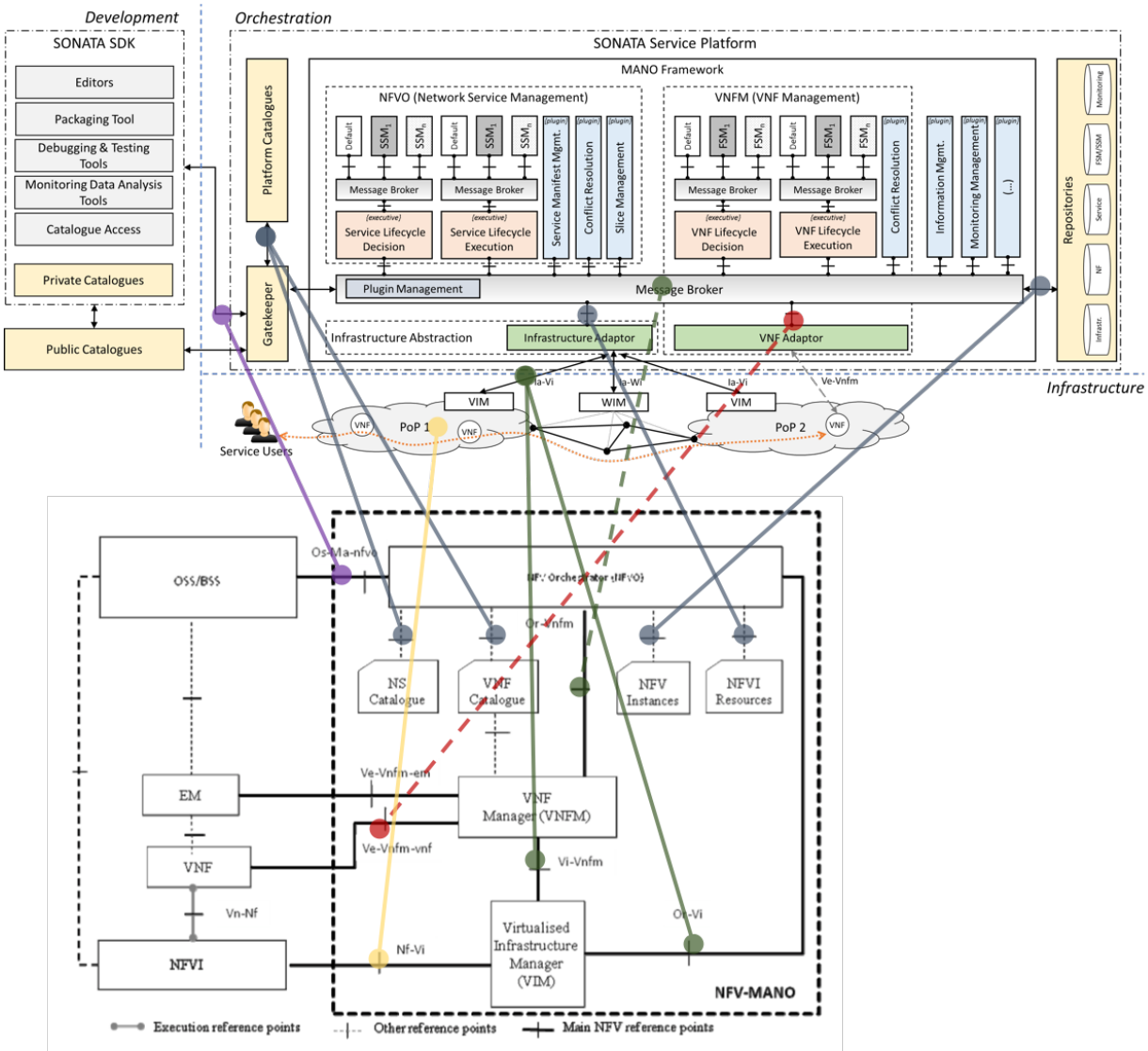

Figure 5.5: Mapping of reference points between SONATA (top) and ETSI NFV-MANO (bottom) reference architectures

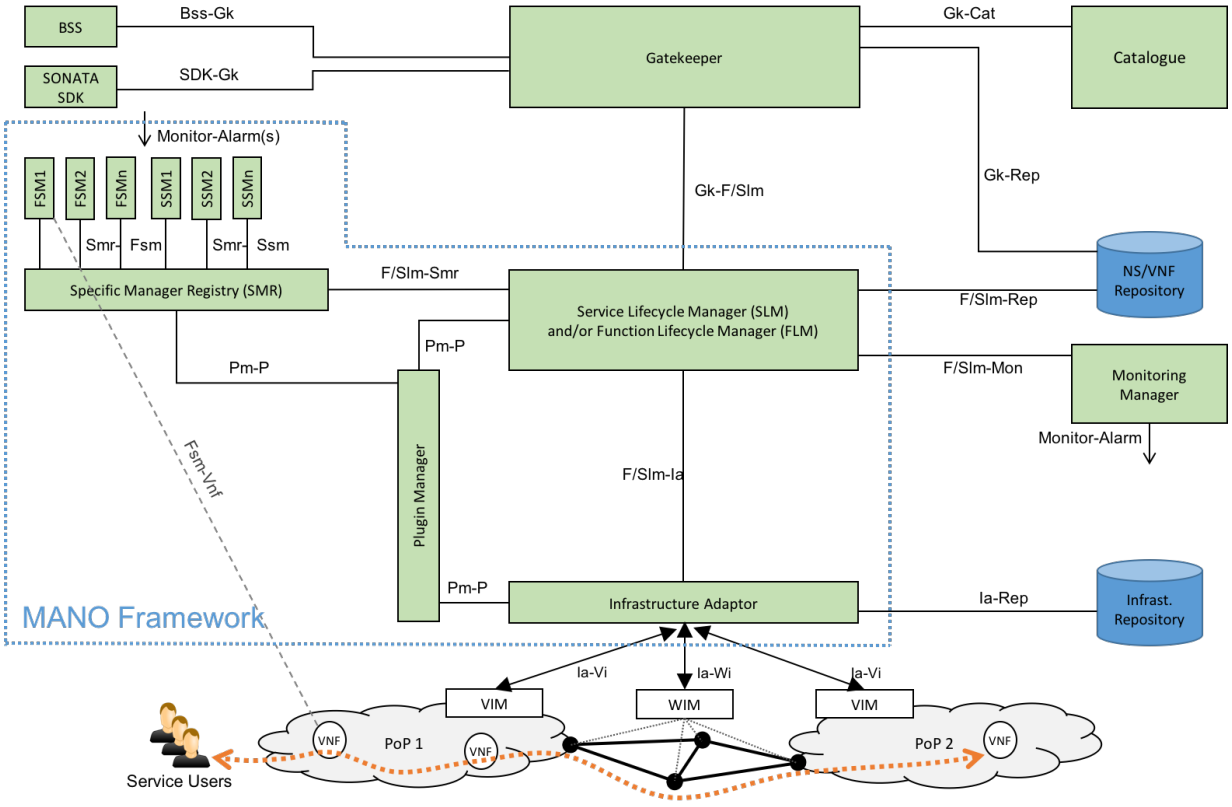

Figure 5.6: Logical view on the SONATA service platform architecture and its interfaces

6 Conclusion

This deliverable presented a revised architecture for the SONATA system. To this end, we presented the changes and updates of 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 ongoing and related work packages (WP3 and WP4).

In short, the main contributions of this document are

- Updated and revised use cases and requirements derived from these use cases.
- Updated and revised SONATA functionality presented in an implementation free approach.
- An updated structure of the SONATA architecture incorporating a Software Development Kit and a Service Platform.
- A new profiling system for Network Services integrated in the SDK and tightly coupled with the Service Platform.
- A system to perform automated tests in order to validate and verify Network Services.
- A security analysis of the Service Platform and suggestions to harden the Service Platform such that is avoid threats and failures.
- A detailed analysis of the similarities between the SONATA architecture and reference points against the ETSI reference model.

Compared to the former deliverables D2.1 and D2.2, this document covers, updates, sharpens all main aspects of the SONATA architecture. Future work, however, is needed in improving the software design, implementing the new functionalities, and (automated) security and performance evaluation. For the fine details, we refer to the (upcoming) deliverables D3.2, D4.2, and D5.3.

A Bibliography

[1] GEni Key Concepts. *Global Environment for Network Innovations (GENI)*. Online at `http://groups.geni.net/geni/wiki/GENIConcepts`.

[2] Report on Gap Analysis. *ITU-T IMT2020 document*, December 2015. Online at `http://www.itu.int/en/ITU-T/focusgroups/imt-2020/Pages/default.aspx`.

[3] Study on Architecture for Next Generation System. September 2016. Latest version v1.0.2, online at `http://www.3gpp.org/ftp/tsg_sa/WG2_Arch/Latest_SA2_Specs/Latest_draft_S2_Specs`.

[4] D. Yu A. Galis, K. Makhijani. Autonomic Slice Networking-Requirements and Reference Model. *IETF*, November 2016. Work in progress, Online at `https://tools.ietf.org/html/draft-galis-anima-autonomic-slice-networking-01`.

[5] Amazon. Amazon web services. Website, 2016. Online at `https://aws.amazon.com/`.

[6] CoreOS Community. Coreos. Website, November 2016. Online at `https://coreos.com`.

[7] Docker Community. Docker. Website, June 2016. Online at `https://github.com/docker/docker/pull/11860`.

[8] Kubernetes Community. Kubernetes. Website, November 2016. Online at `http://kubernetes.io/`.

[9] Kubernetes Community. Kubernetes network policies. Website, November 2016. Online at `http://kubernetes.io/docs/user-guide/networkpolicies/`.

[10] Rocket Community. Rkt. kubelet. Website, November 2016. Online at `https://rocket.readthedocs.io/en/latest/Documentation/running-kvm-stage1/`.

[11] Swarm Community. Swarm. Website, November 2016. Online at `https://docs.docker.com/swarm/`.

[12] SONATA consortium. D2.1: Use cases and requirements. Website, October 2015. Online at `http://www.sonata-nfv.eu/content/d21-use-cases-and-requirements`.

[13] SONATA consortium. D2.2 architecture design. Website, December 2015. Online at `http://www.sonata-nfv.eu/content/d22-architecture-design-0`.

[14] SONATA consortium. D3.1: Basic sdk prototype. Website, May 2016. Online at `http://www.sonata-nfv.eu/content/d31-basic-sdk-prototype`.

[15] SONATA consortium. D4.1: Orchestrator prototype. Website, May 2016. Online at `http://www.sonata-nfv.eu/content/d41-orchestrator-prototype`.

[16] SONATA consortium. D4.2: Service platform operational release and documentation. Website, December 2016.

[17] SONATA consortium. D6.1: Definition of the pilots, infrastructure setup and maintenance report. Website, June 2016. Online at http://www.sonata-nfv.eu/content/d61-definition-pilots-infrastructure-setup-and-maintenance-report.

[18] CoreOS. Flannel. Website, November 2016. Online at https://github.com/coreos/flannel.

[19] A. Galis et all. Management and Service-aware Networking Architectures (ManA) for Future Internet. *Invited paper IEEE 2009 Fourth International Conference on Communications and Networking in China*, August 2009. Online at http://www.chinacom.org/2009/index.html.

[20] Roy Fielding. Architectural styles and the design of network-based software architectures. Website, 2000. Online at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[21] Open Source MANO (OSM Group). Openvim. Website, 2016. Online at https://osm.etsi.org/wikipub/index.php/OpenVIM_installation_%28Release_One%29.

[22] Mschner K. et all Hedmar, P. Description of Network Slicing Concept. *NGMN Alliance document*, January 2016. Online at https://www.ngmn.org/uploads/media/160113_Network_Slicing_v1_0.pdf.

[23] Internet Engineering Task Force (IETF). The oauth 2.0 authorization framework. Website, October 2012. Online at https://tools.ietf.org/html/rfc6749#section-4.3.

[24] Internet Engineering Task Force (IETF). The oauth 2.0 authorization framework. Website, October 2012. Online at https://tools.ietf.org/html/rfc6749#section-4.4.

[25] ETSI European Telecommunications Standards Institute. Network Functions Virtualisation (Nfv);Pre-deployment Testing;Report on Validation of Nfv Environments and Services v1.1.1. Website, April 2016. Online at http://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/001/01.01.01_60/gs_NFV-TST001v010101p.pdf.

[26] Jenkins. Jenkins documentation. Website, June 2016. Online at https://jenkins.io/doc/.

[27] George Kousiouris, Andreas Menychtas, Dimosthenis Kyriazis, Spyridon Gogouvitis, and Theodora Varvarigou. Dynamic, behavioral-based estimation of resource provisioning based on high-level application terms in Cloud platforms. *Future Generation Computer Systems*, 32:27–40, 2014.

[28] Sunil Kumar, Manish Kumar Pandey, Abhigyan Nath, Karthikeyan Subbiah, and Manoj Kumar Singh. Comparative study on machine learning techniques in predicting the Qos-values for web-services recommendations. In *Computing, Communication & Automation (ICCCA), 2015 International Conference on*, pages 161–167. IEEE, 2015.

[29] OpenSim Ltd. Omnet++ Network Simulator. Website, 2016. Online at https://omnetpp.org.

[30] Packet. Intro to Project Calico: a pure layer 3 approach to scale-out networking. Website, November 2016. Online at http://www.slideshare.net/packethost/intro-to-project-calico-a-pure-layer-3-approach-to-scaleout-networking.

[31] Packet. Packet Project Calico Keynote Presentation. Website, November 2016. Online at http://www.slideshare.net/packethost/packet-calico-keynote-47122317.

[32] Schallen S. Betts M. Hood D. Shirazipor M. Lopes D. Kaippallimalit J. Paul, M. Applying Sdn Architecture to 5G slicing. *Open Network Fundation document*, April 2016. Online at `https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/Applying_SDN_Architecture_to_5G_Slicing_TR-526.pdf`.

[33] Manuel Peuster and Holger Karl. Understand Your Chains: Towards Performance Profile-based Network Service Management. In *5th European Workshop on Software Defined Networks (EWSDN'16)*. IEEE, 2016.

[34] Romana. Romana Performance. Website, November 2016. Online at `http://romana.io/how/performance/`.

[35] Raphael Vicente Rosa, Christian Esteve Rothenberg, and Robert Szabo. VBaaS: Vnf benchmark-as-a-service. In *2015 Fourth European Workshop on Software Defined Networks*, pages 79–84. IEEE, 2015.

[36] Eder J Scheid, Cristian C Machado, Ricardo L dos Santos, Alberto E Schaeffer-Filho, and Lisandro Z Granville. Policy-based dynamic service chaining in Network Functions Virtualization. In *Computers and Communication (ISCC), 2016 IEEE Symposium on*, pages 340–345. IEEE, 2016.

[37] SONATA. son-cmud.yml. Website, 2016. Online at `https://github.com/sonata-nfv/son-install`.

[38] Steven Van Rossem, Wouter Tavernier, Manuel Peuster, Didier Colle, Mario Pickavet, and Piet Demeester. Monitoring and debugging using an Sdk for Nfv-powered telecom applications. In *IEEE NFV-SDN (NFVSDN2016)*. IEEE, 2016.

[39] VMWare. Vmware. Website, 2016. Online at `https://www.vmware.com/products/vcloud-suite.html`.

[40] WeaveWorks. Weave. Website, November 2016. Online at `https://github.com/weaveworks/weave/`.