D6.3 Final demonstration, roadmap and validation results

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	D6.3 Final demonstration, roadmap and validation results
Workpackage	WP6 Infrastructure setup, validation and pilots
Due Date	31/12/2017
Submission Date	22/02/2018
Version	0.1
Status	To be approved by EC
Editor	George Xilouris (NCSRD)
Contributors	Felipe Vicens (ATOS), Stavros Kolometsos (NCSRD), Peer Hasselmeyer (NEC), Santiago Rodríguez (Optare), José Bonnet, Alberto Rocha (ALB), Theodore Zahariadis, Panos Trakadas, Panos Karkazis (SYN), Shuaib Siddiqui (i2CAT), Dani Guija (i2CAT), Philip Eardley (BT), Thomas Soenen (imec), Dario Valocci, Francesco Tusa, Stewart Clayman (UCL), Bruno Vidalenc (THALES), Geoffroy Chollon (THALES)
Reviewer(s)	Jose Bonnet (ALB), Alex Galis (UCL)

Keywords:

Pilots, NFV, vCDN, PSA, HSP, MANO

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:

The SONATA project has followed the DevOps methodology to develop various components, which we have integrated and tested in several Pilots. This document first summarises SONATA's DevOps environments for development, integration, qualification and demonstration. It then summarises the various elements we have developed: the tools in our SDK (Service Development Kit) and the modules of our Service Platform. These provide basic functionality such as the on-boarding, deployment and instantiation of services. The main part of the document discusses our four Pilots: (1) Virtual Content Delivery Network; (2) Personalised Security; (3) Multi-orchestrator and slicing; and (4) Hierarchical Service Providers. For each Pilot we discuss the architecture, scenario, components involved (including the novel capabilities it adds to the basic SONATA functionality), validation and innovations. All pilots provide novel capabilities that go beyond the basic functionality of SONATA platform that include on-boarding, deployment, and instantiation of services. vCDN pilot innovations are mainly related to the virtual CDN service deployment optimisation and service resource management during runtime. PSA pilot innovations are mainly related to additional security-related services, specifically anonymity and safe browsing. HSP pilot innovations are mainly related to the multi MANO interworkings, SP recursiveness of the service on-boarding and abstraction layer and slice management. The Pilots thus validate and demonstrate the features and innovations of the SONATA project. The document also summarises the overall lessons learned during the project.

Contents

List of Figures

List of Tables

1 Introduction

This Deliverable is the final deliverable of the WP6 (and from SONATA) and it summarises the output and the achievements of all the tasks of the work package. WP6 was in-charge of the design, deployment, operation and maintenance of the SONATA infrastructure across all participating testbed locations (i.e Athens, Aveiro, London, Tel-Aviv and Paderborn). Over this infrastructure, which was interconnected via VPN in a single unified network, SONATA environments were created facilitating different stages of the DevOps processes. At the final stages WP6 supported the validation and evaluation campaigns as well as the demonstration of the pilots as those were formed at the last phase of the project. This document also includes information on the artefacts developed during the project, that were made available by the other technical work packages and are being evaluated in this deliverable. Furthermore the deliverable D6.3 analyses the final *selected* pilots as those where formed after an iterative process of refinement according to the capabilities available from the SONATA SP, the availability of the participating VNFs and the demonstration objectives. Finally lessons learned and key takeaways are discussed in the last part of the document.

1.1 Document structure

The structure of the document is as follows: the first section of the deliverable is devoted to the SONATA DevOps environments, the second section discusses the artefacts and presents a system level evaluation tests, the third section is devoted to the SONATA Pilot description. The following three sections discuss the validation of the main KPIs (i.e. Section 4), the lessons learnt (i.e. Section 5) and future work and gap analysis (i.e. Section 6). Finally section 7 provides the conclusions of the deliverable.

1.2 Dependencies on other deliverables of the project

This deliverable integrates and validates the implementation work carried out within other technical WPs, and as such, contains either implicit or explicit references to deliverables summarised in the following table.

Table 1.1: Deliverable documents interdependencies

Deliverable Name	Description	Reference
D6.1 - Definition of the Pilots, infrastructure setup and maintenance report	This deliverable was the first document where the original Use Cases of SONATA according to the DoW were discussed in addition to the ones that were considered during the requirements elicitation. In addition the documents provides final information on the Infrastructures used for SONATA deployment, that were further expanded into mature SONATA environments. Moreover, building on top of the Pilot selection methodology, it provides more technical information on them.	[8]
D6.2 - Configuration of pilots and pre-validation	This deliverable is being updated by the current document and also constitutes the main document for the implementation, deployment and configuration of the SONATA pilots at the demonstration environment, as it is formed by the participating and interconnected testbeds. Deviations from the described scenarios should be expected as the actual implementation was refined to fit the final deployments.	[12]

Deliverable Name	Description	Reference
D5.4 - Final release of SONATA platform	This deliverable provides the final information on the SONATA Service Platform the supported features and the deployment scenarios and configurations. According to the validations conducted in the Qualification Environment, this deliverable is validating and demonstrating the SONATA SP in the scope of the pilot scenarios. The established methodology and identified integrated components are exploited by this Deliverable in order to accommodate them in the demonstration environment.	[11]

2 DevOps environments

This section presents the final deployment of the SONATA environments. As discussed in Deliverable D6.2, the SONATA environments are:

- *Development environment:* the environment used by each developer in his own premises. Besides his preferable tools for development, the developer is required to deploy and setup the SONATA SDK in order to push developed services towards the Service Platform (regardless of the environment that those NSs are going to be operated);
- *Integration environment:* the environment deployed over a simple infrastructure required to execute integration tests for the components;
- *Qualification environment:* the environment deployed over a multi-pop infrastructure required to execute qualification tests and deployment of NS/VNFs;
- *Demonstration environment:* the environment deployed over a production (or close to production) infrastructure in order to deploy operational versions of Network Services. In SONATA the demonstration environment is being deployed over a distributed infrastructure interconnected via VPN over the internet.

The following subsections briefly present the final configuration and setup of the above environments (the development environment is omitted as is out of the scope for this deliverable - for further information on it, please read [9]).

2.1 Hosting infrastructure components

Hosting infrastructure for SONATA has been primarily deployed in Athens (NCSRD) and in AVEIRO (ALB). Both sites covered most of the integration activities for SONATA. The design of the hosting infrastructure architecture was based on the following requirements:

- The hosting infrastructure:
  - should support multiple NFVI-PoPs.
  - should support ancillary services (i.e DNS etc.);
  - should support integration with end-user devices for testing;
  - should support deployment of CI/CD toolkit and services;
  - core network may support SDN and be managed via WAN.infrastructure management;
  - should support GRE based or encrypted VPN tunnels;
- The NFV Infrastructure PoP should support SDN and service function chaining.

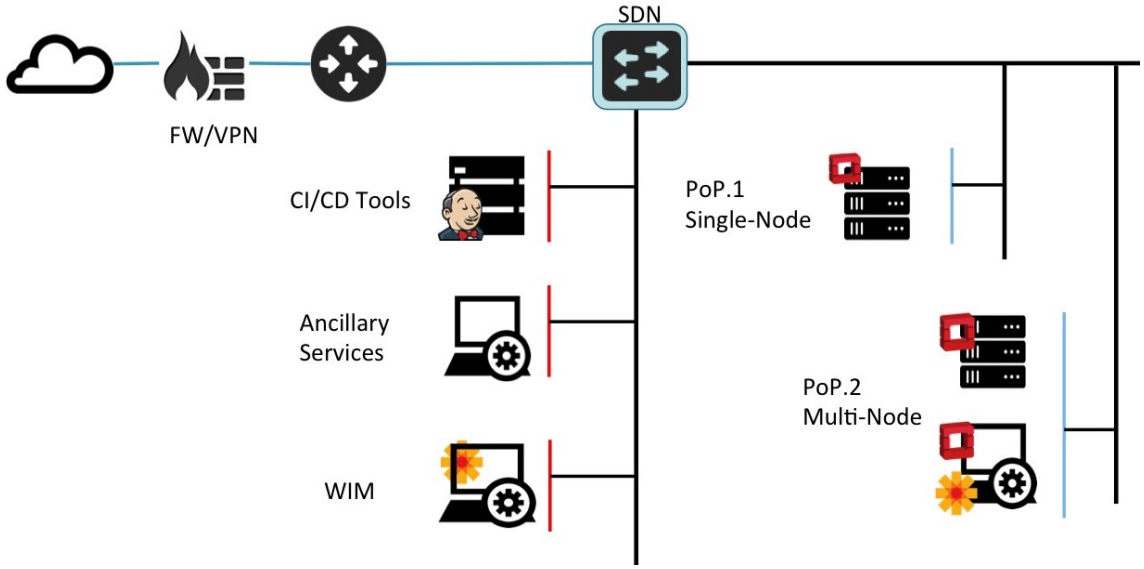

Figure 2.1: Hosting infrastructure

Figure 2.1 presents an example deployment of hosting infrastructure. Since Athens testbed was exploited from the beginning of the project to establish the core that would support the initial development, CI/CD and Ancillary components were deployed there. All the other locations implemented parts of the hosting infrastructure mainly focusing in NFVI-PoP, development versions of SONATA SP and Pilot components.

It has to be mentioned that due to infrastructure unavailability at the time, the NOKIA testbed was interconnected but also used resources in Athens (NFVI-PoP) for its 'local' testing regarding Mistral NS lifecycle management.

In order to be able to automatically deploy SONATA artifacts, the existence of an underlying available data centre infrastructure is assumed. Additionally it requires communication with VIM and WIM components in order to be able to orchestrate the available resources in the underlying physical infrastructures. This section describes the hosting infrastructure as it was laid out for the second year of the project. The hosting infrastructure is located in Athens (mainly) and Aveiro testbeds. It comprises of compute and network resources along with their management entities.

Figure 2.1 illustrates the main segments of the infrastructure currently deployed in Athens, a similar structure, apart from the CI/CD tools, that is used in the Aveiro testbed. More details on the full spectrum of testbeds used in SONATA are provided in the following sections. Components comprising the hosting infrastructure have been already mentioned in [12], with no significant changes of the hosting infrastructure to be reported.

Table 2.1 summarises each testbed's characteristics.

Table 2.1: Hosting Infrastructure deployments

Testbed	SONATA SP	CI/CD	WIM/SDN WAN	NFVI-PoP	VPN
Ancillary services	end-users	Pilot			
Athens (NCSRD)	Yes	Yes	YES (based on ODL)	Yes (2 multi/single-compute nodes)	Cisco and IP/GRE
Yes (DNS, NMS)	Yes (over two segments)	vCDN, PSA, HSP			

Testbed	SONATA SP	CI/CD	WIM/SDN WAN	NFVI-PoP	VPN
Ancillary services	end-users	Pilot			
Aveiro (ALB)	Yes	No	No	Yes (2 multi/single-compute nodes)	Cisco
No	Yes	vCDN			
Paderborn (UPB)	Yes	No	No	Yes (1 single-node)	IP/GRE
No	Yes	HSP			
London (UCL)	Yes	No	No	Yes (VLSP based)	IP/GRE
No	Yes	Multi-orch w/ slicing			
Tel-Aviv (NOKIA)	Yes	No	No	No (remote)	Cisco
No	Yes	-			

2.2 Integration environment

The integration environment is stable since D6.2 [12]. According to the monitoring data w.r.t the performance of the environment for executing SONATA tasks, the infrastructure was dimensioned correctly. Resources provided by the infrastructure were adequate not only to support unit test but also docker container creation and deployment as well as for executing integration tests.

In this context, tasks performed since D6.2 for this environment were mainly focused on the maintenance and stability of the integration platform and secondary in troubleshooting and debugging in case of problems. To this end, a set of admin tasks were designed to clean automatically the environment and deploy the integration environment more efficiently.

2.2.1 Current integration environment

Table 2.2 presents current version of Integration Environment physical server specifications.

Table 2.2: Current integration environment physical server description

Environment	**CPUs**	**RAM (GB)**	**HDD (GB)**	**Openstack Version**
Current	32xIntel Xeon(R) E5677@3.47GHz	96GB	3000	VMware ESXi 6.5

Table 2.3 provides a summary of VM characteristics.

Table 2.3: Integration Environment Setup

VM	**Function**	**SW tools**	**vCPU**	**HDD (GB)**	**Memory (GB)**
#1	CICD engine	Jenkins	8	120	8
#2	int-srv-1	SP, API, BSS, GUI	2	80	4
#3	int-srv-2	SP, API	2	80	4
#4	int-srv-3	SP	2	80	4
#5	Repos	Docker Registry	2	80	2
#6	Monitory	MONIT	4	80	8
#7	Logging	LOGs	2	80	4

2.2.2 Jenkins statistics for integration jobs

To support the DevOps cycles for SONATA development, Jenkins [15] acts as the central piece of the CI/CD pipeline. Each pull request (issued at the github repository by the developer) is checked by Jenkins. Upon completion of testing, feedback is provided that guarantees the integration of all components with the one(s) under test. This section presents information and statistics on the number of development tests and integration have been executed during the SONATA lifetime.

2.2.2.1 Development tests

The following Table 2.4 presents the number of times each test was executed for each software components. The table is sorted out by which component was tested most. It can be seen that the top 5 jobs are related to the core of SONATA, i.e., SDK-CLI, Gatekeeper, MANO, and IA. The gatekeeper is the top one since this repository is composed of twelve gatekeeper components. Also, son-mano-framework is comprised of four elements.

Table 2.4: Development jobs and tests summary table

Job Name	Times Executed	Duration
son-gkeeper	1697	8 min 11 sec
son-mano-framework	643	6 min 1 sec
son-cli	621	23 min
son-sp-infrabstract-vim	515	4 min 7 sec
son-sp-infrabstract-wim	417	5 min 32 sec
son-catalogue-repos	346	5 min 1 sec
son-emu	323	15 min
son-schema	261	1 min 7 sec
son-bss	224	10 min
son-install	208	15 sec
son-examples	176	22 min
son-analyze	142	53 sec
son-emu-pipeline-periodic	127	16 min
son-gui	117	4 min 1 sec
son-monitor	89	4 min 2 sec
son-sdk-catalogue	62	36 sec
son-monitor-probe	51	7 min 35 sec
son-cli-repo	43	36 min
son-emu-pipeline	43	40 min
son-sm	30	2 min 10 sec
son-gkeeper-pipeline	16	3 min 28 sec

Figure 2.2 visualises the data of the previous table (Table 2.4) and Figure 2.3 represents the total duration of each development job. Notably, from this graph, we can draw the conclusion that the most time-consuming tasks are related to the SDK. This is justified by the need for of SDK compilation and installation in each iteration of the testing.

2.2.2.2 Integration tests

In summary, for the integration environment, Jenkins has performed 6151 development jobs and 9594 integration tests. Table 2.5 summarizes the Integration Test jobs that run on the platform through Jenkins. Twenty-one integration tests were developed in the integration environment. Figure 2.4 shows the number of times each integration test was executed and Figure 2.5 the duration of each execution per test.

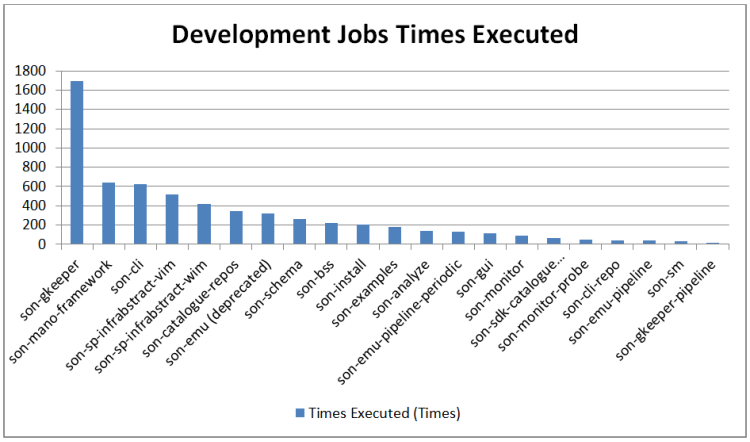

Figure 2.2: Development Jobs times executed

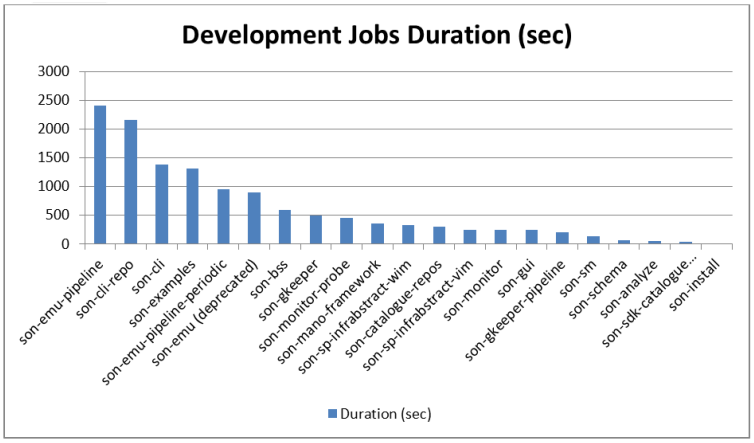

Figure 2.3: Development Jobs Duration

Table 2.5: Integration tests summary table

Job Name	Times Executed	Duration
int-9-environment-deploy	2717	3 min 12 sec
int-2-sdk-workflow	772	41 min
int-1-gtkpkg-sp-catalogue	684	46 sec
int-8-sdk-son-monitor	672	35 min
int-6-emu-push-schema	438	32 min
int-5-mano-plugin-management	413	3 min 55 sec
int-10-bss-gkeeper	407	3 min 40 sec
int-18-gtkusr-keycloak	374	2 min 28 sec
int-4-service-instantiation-E2E	357	17 min
int-15-gtkkpi-prometheus	337	9.8 sec
int-2-sdk-workflow-pipeline	325	4 min 21 sec
int-16-SSM-FSM	321	1.8 sec
int-7-slm-repositories	316	11 sec
int-14-sdk-son-analyze	264	2 min 3 sec
int-12-monitoring-SLM-broker	260	1 min 55 sec
int-17-usr-management	259	3 min 28 sec
int-13-gui-gk-mon	250	2 min 43 sec
int-11-slm-infrabstract-V2	229	2 min 25 sec
int-19-monitoring-ws	173	1 min 27 sec
int-20-sdk-gtkapi	19	5 min 46 sec
int-infrabstract-lowerSP	7	6 min 35 sec

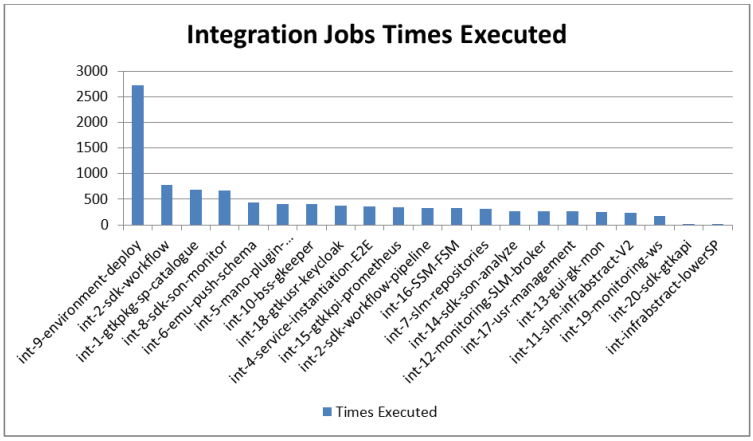

Figure 2.4: Integration Jobs times executed

It can be observed that the job with the most executions is int-9-environment-deploy. The main reason is that this job involves the actual deployment of the integration environment, hence most of the integration tests run this task before executing their own tests.

2.3 Qualification environment

SONATA CI/CD is anticipating an environment to conduct testing on integrated components that have successfully passed the integration tests in the integration environment. The qualification tests are presumed to be more complex and required additional resources and are more complex and thorough. Qualification tests are in general considered to refer to tests between integrated components that expose higher layer functionalities. The Qualification environment was described in detail in SONATA D6.2 [12]. Since the release of D6.2 additional VMs were created, improving and supporting the qualification tests.

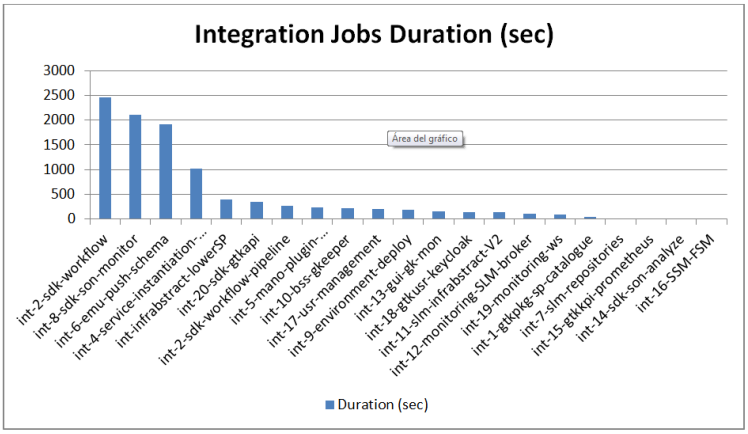

Figure 2.5: Integration Jobs Duration

In this view, Jenkins was extended with a slave worker to run the "stress tests". The reason for essentially scaling up the resources of Jenkins was to support the large number of connections against the System Under Test (SUT) that is required. In principle *stress-testing* requires to impose the system under test (SUT) to an incrementally large number of events up to the point of breaking.

During this period, the qualification environment was enhanced with the capabilities of conducting qualification for the code implementing the Hierarchical Service Platform (HSP) Pilot. Particularly for this set of qualification test, more than one Service Platform had to be considered, operating over different NFVI-PoPs. Therefore the Service Platform was installed on two Virtual Machines (VM) : (i) one placed in Athens testbed and (ii) one VM placed on Aveiro. The qualification tests are related to test of the exchange of information between those two SPs in order to deploy services. More details for this pilot are provided in the Pilot section.

Table 2.6: Qualification HSP Pilot resources

Resource Name	Location	CPU	Memory	Disk	Flavour
SONATA SP Qualification HSP	Aveiro	2	8G	40G	m1.large
SONATA SP Qualification HSP	Athens	2	8G	40G	m1.large

Table 2.6 details which resources were configured for the HSP pilot.

2.3.1 Jenkins statistics for qualification

For the qualification environment, Jenkins has performed 2045 tests. In comparison with development and integration environments, the amount of executions is lower and the time duration is higher. Statistics are depicted in Table 2.7 (duration in minutes). The average duration is 27 minutes for the qualification tests. It's important to note in the Figure 2.7 that the qualification tests taking longer are those that imply the installation of the Service Platform from scratch. The time spent is due to the deployment of a new Virtual Machine using ansible scripts and the time required for download all the Service Platform components. But 27 minutes to all this is remarkable.

Table 2.7: Qualification tests summary table

Name	Executions	Duration (min)
qual-ubuntu-install	548	54
qual-sp-alabs-ce7	360	64

Name	Executions	Duration (min)
deploy-vm2alabs-demo-u16	260	19
qual-sp-ncsrd-dem-u16	121	216
qual-stress-api	115	10
qual-prepare-sp-1PoP	75	2
qual-1VNF-1PoP	70	13
qual-2VNF-1PoP	62	13
qual-prepare-sp-1PoP-qual	56	3
qual-clean-environment	52	3
qual-deploy-and-install	41	34
qual-stress-ia	41	1
qual-prepare-sp-mockPoPs	39	3
qual-stress-mano	28	13
deploy_sp2alabs_local_vm	26	32
qual-stress-monitoring	26	2
qual-cli-centos-install	18	1
qual-push-packages	18	0
qual-check-bss-gui-certificates	17	2
qual-cli-ubuntu-install	17	2
vPSA-dockerized-alabs-u16	15	47
qual-prepare-sp-2PoPs	14	3
qual-stress-catalogue-repos	10	1
qual-prepare-sp-for-stress-sec-tests	8	8
qual-2VNF-2PoP	5	10
qual-ubuntu-install-v3-0	3	154

Figure 2.6 shows graphically the number of tests performed in qualification. The average is 78 executions per job. Some of these tests were performed automatically and others triggered by the developer. Additionally, if the integration tests related to qualification do not pass, the qualification test is not executed automatically.

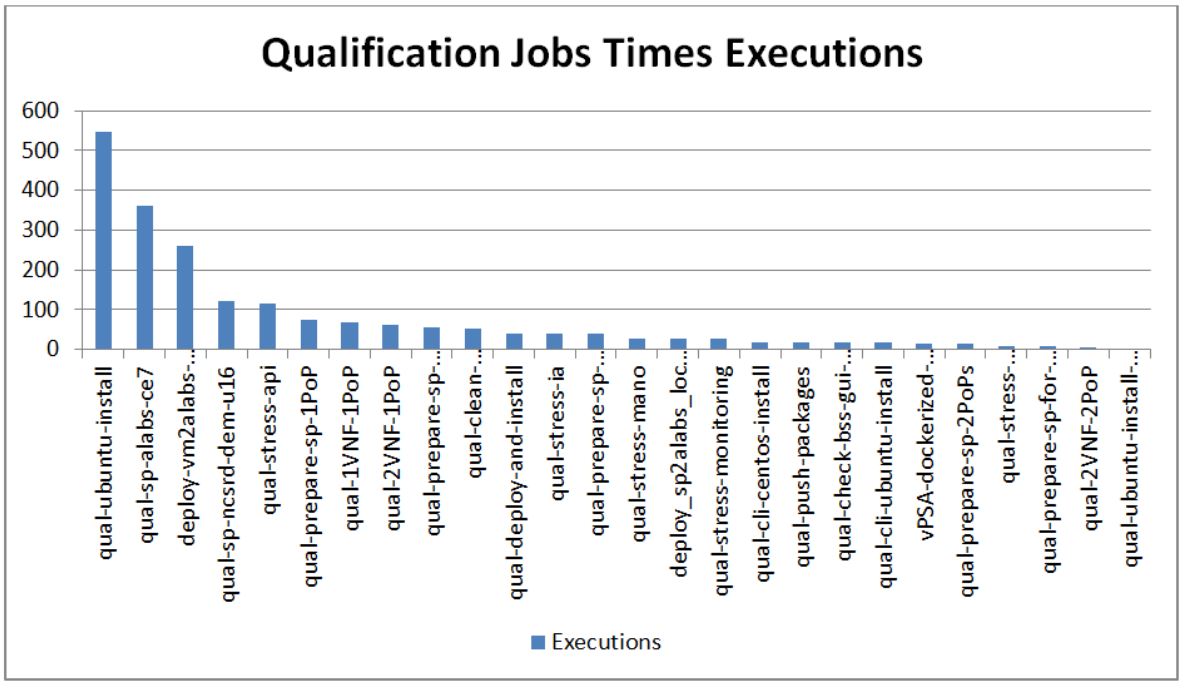

Figure 2.6: Qualification Jobs Times Execution

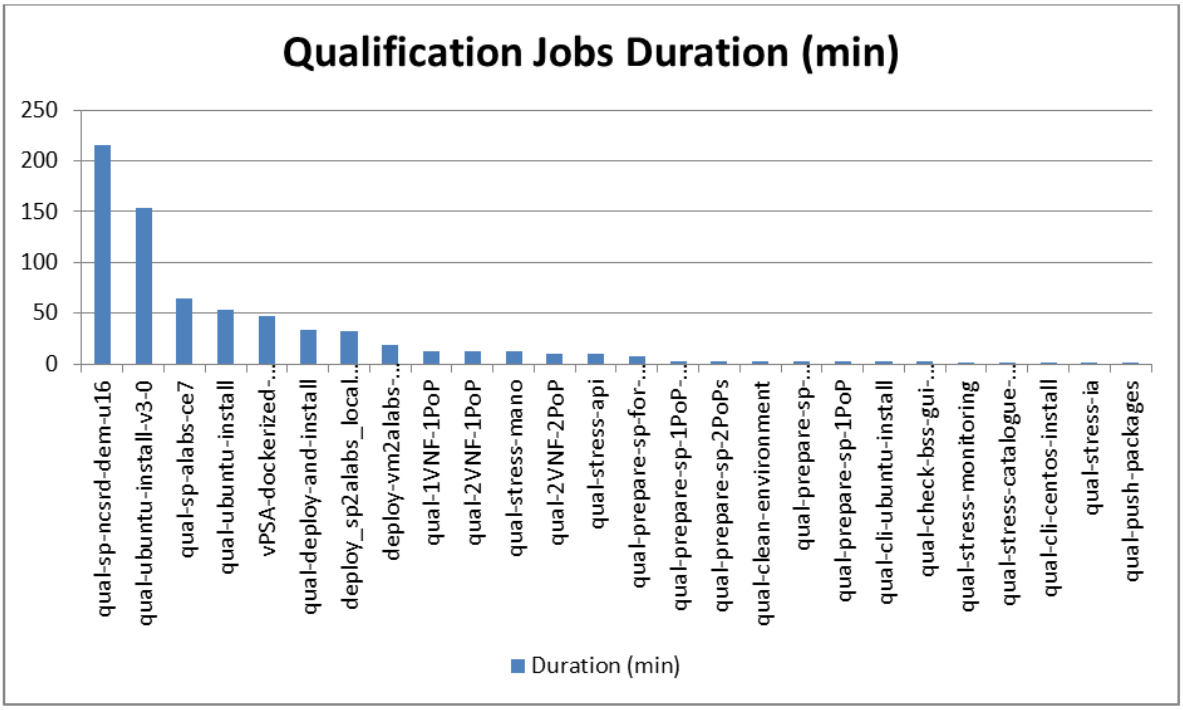

Figure 2.7: Qualification Jobs Duration

2.4 Demonstration Environment

The Demonstration environment uses the demonstration infrastructure defined in D6.2 [12] to prove the functionalities of the service platform through the Pilots. This infrastructure is detailed below.

2.4.1 vCDN Pilot

vCDN Pilot is made of three VNFs that have been deployed in two sites. From the user point of view, vTC and vCC have to be placed in the same location near to the user, while the vTU has to be installed next to Content Server on a different site. To this end, The pilot requires the following resources from demonstration infrastructure:

- Service Platform for demonstration associated with both VIM1 and VIM2 (Athens)
- NFVI PoP OpenStack Athens - sonata.dem tenant (VIM1)
  - vTC and vCC will be deployed in Athens PoP
- NFVI PoP OpenStack Aveiro - sonata.dem tenant(VIM2)
  - vTU will be deployed in Aveiro PoP
- SDN switch to configure the service chain
- User (Physical desktop in Athens)
- Content Server (Server deployed in Aveiro)

2.4.2 PSA Pilot

PSA Pilot makes use of a set of VNFs deployed next to the user. The service can change according to the user's needs so he can add or remove VNFs while the interconnection is reconfigured to maintain the chaining. Only one PoP is needed to demonstrate the SONATA functionalities. For the presentation of this pilot the following resources are expected from the demonstration infrastructure:

- Service Platform for demonstration associated with VIM1 (Athens)
- NFVI PoP OpenStack Athens - sonata.dem tenant (VIM1)
  - vPRX, vTOR, vFW, vVPN will be deployed in Athens PoP
- SDN switch to configure the service chain
- User (Physical desktop in Athens)

2.4.3 HSP Pilot

We have chosen to use the vCDN pilot to demonstrate one unique and key feature of the SONATA Service Platform: its hability to work in a hierarchical way, with one Service Platform in Athens and the others in Aveiro and in London. The management network is shared between both Service Platforms for Management purposes. Each service platform is attached to the VIM located in the same site where it is placed. The following resources are required from demonstration infrastructure:

- Service Platform for demonstration associated with VIM1 (Athens)
- Service Platform for demonstration associated with VIM2 (Aveiro)
- NFVI PoP OpenStack Athens - sonata.dem tenant (VIM1)
  - vTC and vCC will be deployed in Athens PoP
- NFVI PoP OpenStack Aveiro - sonata.dem tenant(VIM2)
  - vTU will be deployed in Aveiro PoP
- SDN switch to configure the service chain
- User (Physical desktop in Athens)

2.4.4 MO and Slicing Pilot

As we described in D6.3 [13], this pilot is not focused on specific network functions, but on the overall inter-working between multiple MANOs. The proposed scenario was made in collaboration with 5GEx 5GPPP Project [1]. The demonstration environment uses the demonstration infrastructure located in London, where it was required to have two Sites for the proposed scenario. Resources for the demonstration infrastructure are referenced next:

London Site1:

- Sonata SP1
- Sonata SP2

- Slice controller running on hosts
- VLSP PoP1
- VLSP PoP2

London Site2:

- Host with Sonata SP Domain adapter, 5GEx RO, VLSP Domain Adapter
- Slice controller running on hosts
- VLSP PoP1

3 SONATA artefacts

This section summarises the SONATA artefacts that were implemented during SONATA development. These artefacts have released specification and implementation details in number of deliverables in the frame of WP3, WP4 and WP5. In the above context i) deliverable D3.3 [9] presents the SDK operational release; ii) deliverable D4.3 [10] presents the final release of the SONATA Service Platform architecture and iii) deliverable D5.4 [11] presents the final implementation of SONATA SP along with validation.

3.1 Service Development Kit (SDK)

The SDK consists of a number tools to assist the Service Developer in the construction, testing and profiling of the service components. Below we give a brief overview of the different functionalities and involved artefacts. More detailed descriptions can be found in the deliverables associated to the SDK ([4], [5] and [9]).

3.1.1 SDK workspace and project management

The SDK enables to organise the environment of the service developer into separated environments. The SDK provides two CLI tools for this: ‘son-workspace‘ and the related ‘son-project‘. The first enables to store re-usable environment settings including authentication tokens, or other login details to connect to multiple Service Platforms, the second enables to create a separate, isolated folder structure (skeleton) for developing a new service with associated descriptors.

3.1.2 Descriptor construction and compilation

The SONATA SDK enables the developer to construct network service descriptors (NSDs) and VNF descriptors (VNFDs) using a web-based graphical user interface (GUI) ‘son-editor‘ (consisting of a front-end and back-end component). This tool enables rapid re-use of existing descriptors including direct support for re-using descriptors of GitHub repositories, as well as a point-and-click interface for interconnecting different VNFs in constructing VNF Forwarding Graphs (VNFFGs). Individual descriptors can be modified in a rapid and user-friendly manner using GUI elements like dropdown-boxes to help the user.

3.1.3 Packaging

Packaging involves the process where the entire set of image files and descriptors can be bundled into a single file which can then be on-boarded on one or more Service Platforms. The CLI tool ‘son-package‘ provides the packaging functionality, which create such a package in a robust and efficient way. These packages can be submitted to tests to ensure that they do not contain any errors.

3.1.4 Validation

A number of bugs or issues can sneak into a range of descriptors, images or packages. To assist the service developer in catching those issues before the service is deployed, the ‘son-validate‘ tool provides a number of mechanisms to detect syntactical, as well as semantic errors including loop detection in the network forwarding graph. The GUI component of the tool allows developers to do a deep analysis of each of the service components through a click-and-zoom interface which is able to inspect details up to VDUs of VNFs.

3.1.5 On-boarding

Once a service is packaged and adequately validated using the previously described SDK tools, it can be on-boarded on one or more Service Platforms. The SONATA Service Platform provides a REST API via its Gatekeeper to on-board services (within a package). Alternate MANO platforms, such as OSM, provide alternate REST APIs. The SONATA SDK CLI tool ‘son-access‘ provides an easy interface, hiding any of the API details of the used Service Platform for on-boarding.

3.1.6 Emulation

Deploying a newly developed NFV service on a full-fledged Service Platform and associated Virtual Infrastructure Manager such as OpenStack, might be a time and resource consuming task. In many cases, when for example the developer wants to focus on testing functionality of a service or its components, a shorter development and test cycle is desired. For this purpose, the SDK provides ‘son-emu‘, an emulator tool enabling to locally on-board and deploy a developed NFV service on an efficient, low-overhead docker-based platform. ‘son-emu‘ is able to emulate multiple Points of Presence (PoP) configurations as well as their interconnection.

3.1.7 SDK profiling & monitoring

The SDK emulation environment provides extensive monitoring and profiling functionality assisting the developer to debug and assess performance of developed components. The ‘son-monitor‘ CLI tool supports timed monitoring of network metrics including packet loss or jitter and to store associated measurements in an efficient database environment which can be expected during and after test execution. The SDK tool ‘son-profile‘ enables deploying network services on SONATA’s emulation platform and to perform load testing under different resource constraints to detect scaling trends as well as to inspect performance penalties under resource constraints in a direct, on-demand manner.

3.2 Service Platform

SONATA’s Service Platform has the following components, each of which is further detailed in the next sections:

- **Gatekeeper**
- **Orchestration**
- **Infrastructure Adaptor**
- **Slice Controller**
- **Monitoring System**

3.2.1 Gatekeeper

The Gatekeeper in the SONATA Service Platform (SP) provides the interface between all the SP's components and their Northbound Interface clients. The Gatekeeper consists of a set of components that interact using a HTTP/REST-based interface. The most important components are:

- **API Gateway:** manages and orchestrates all requests that hit the SP, validating the request (e.g., does the user submitting the request has permissions to do it?), forwarding it to the appropriate component to handle it, collecting their answer(s) and constructing the response;
- **Package Manager**: handles all requests about packages, namely on-boarding (including forwarding it to the Catalogue), and package file and metadata downloading;
- **Service Manager:** handles all services' metadata download requests (including the Network Service Descriptor);
- **Function Manager:** handles all functions' metadata download requests (including the Virtual Network Function Descriptor);
- **Request Manager:** handles all services' instantiation and instance update and termination requests, passing the request to the MANO Framework and receiving the answer;
- **Record Manager:** handles all services' and functions' instantiation records download requests;
- **User Manager:** handles all users' (as well as micro-services) registration and login;
- **Licence Manager:** handles all licensing (packages, services and functions are by default 'public' -- i.e., downloadable and seen by any one, and in the case of services, instantiable -- or the developer can declare them 'private', providing a URL of a store where the SP validates the existence of a licence for that private package, service or function);
- **KPI Manager:** handles all KPIs collected during the usual operation of the SP;
- **Rate Limit Manager:** handles all requests made to the SP, validating the allowed number of requests;

Furthermore, the Gatekeeper includes some other components that are only available internally for the Gatekeeper itself, such as **User Management** component. In the case of this component, it provides a set of interfaces that have been tested under heavy load.

3.2.2 Orchestration (MANO)

The Orchestration segment of SONATA is performed by the MANO Framework. It sits between the Gatekeeper, which provides the interface with the clients of the Service Platform, and the Infrastructure Adapter, which merges the heterogeneous APIs of different VIMs and WIM into one API for the MANO Framework. The MANO Framework consists of a set of components that interact using a message bus. Its most important components are:

- **Service Lifecycle Manager**: manages and orchestrates all aspects on the level of the network service. It will invoke the placement calculations, instruct the deployment of VNFs on the correct VIM, generate records, instruct the chaining of VNFs, etc...

- **Function Lifecycle Manager**: handles all aspects on the level of the network function. It receives instructions from the Service Lifecycle Manager and translates them into the Infrastructure Adapter API.
- **Placement Plugin**: calculates how services can be embedded on top of the available infrastructure.
- **Specific Manager Registry**: manages the on-boarding, instantiation and life cycle of Service/Function Specific Managers. These Managers are pieces of code provided by the service or function developer. They are executed by the Specific Manager Registry, and they override generic MANO Framework behaviour. They allow the service/function developer to customize the behaviour of the MANO Framework in terms of their service/function.

3.2.3 Infrastructure Adaptor

The Infrastructure Abstraction module works as an abstraction layer between the MANO framework and the underlying infrastructure. The Infrastructure Abstraction allows the MANO's entities to interact with the infrastructure, regardless of the specific technology used to manage it. This allows the MANO to work independently from the Virtual Infrastructure Manager (VIM) vendor, providing extra flexibility to the whole platform. It exposes a standard interface to:

- Deploy and manage service and VNF instances
- Retrieve monitoring information about the infrastructure status
- Reserve resources for services deployment
- Configure SFC and other networking aspects
- Request, configure and interact with resource slices (see Sections Section 3.2.4 and Section 4.3 for further details)

It is composed of two main modules, the Virtual Infrastructure Manager Adaptor (VIM Adaptor) and the WAN infrastructure Manager Adaptor (WAN Adaptor). The 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 NFVI Points of Presence to the Sonata MANO framework. The WIM Adaptor allows the service platform to manage network resources connecting different NFVI-PoPs in a vendor agnostic fashion, in order to provide connectivity to the deployed services.

3.2.4 Slice Control and the Slice Controller

This section describes the Slice Controller.

A Slice is an aggregated set of resources that can be used in the context of an end-to-end networked service comprised of virtual network functions. Slices are composed of multiple resources which are isolated from other slices and allows logically isolated network partitions, with a slice being considered as the basic unit of programmability using network, computation and storage. When considering the wide variety of application domains to be supported by 5G networks, it is necessary extend the concept of slicing to cover a wider range of use cases than those targeted by the current SDN/NFV technologies.

If we have slicing everywhere, including networks and DCs, we observe the following attributes:

- there is a separation of physical resources

- there is isolation of services as no customers share physical resources
- it is secure as only specified customer can access host, no sharing or cross VM issues

In order to support service provisioning over these slices, it is necessary to have mechanisms to support the slicing of the network resources and the Data Center compute and storage resources. To manifest this slice approach, we have designed and built a DC Slice Controller which is able to allocate a slice of a DC and create a per-slice VIM in an on-demand fashion. The DC slice and the VIM are provisioned solely for use with the service. Each slice and its associated VIM are independent of the other slices and VIMs. In this way, customers will never share servers, and the worry of VMs of one customer interacting or spying on another customer will be eliminated. Also, the issue of one customer's VM consuming all the resources and starving other customer's VMs is also ameliorated to some extent.

Each of these slices will be allocated and de-allocated in an on-demand fashion. A customer can interact with a Slice Manager, and request a new slice. The resulting slice will be isolated from the other slices. Furthermore, each slice will get its own VIM or WIM, and not have management as part of shared one. The Figure 3.1 presents how the resources of a DC are isolated from each other, and how a Slice Manager is involved in such a process.

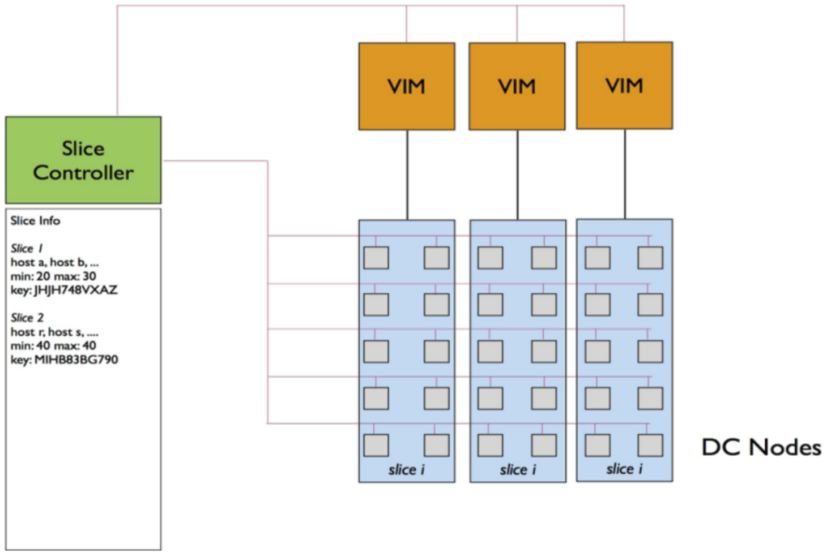

Figure 3.1: Data Center resources isolation

3.2.4.1 Slice Controller Elements

The Slice Controller has the following elements for its operation:

- a Slice Manager, which is composed of a Resource Manager manages all of the resources in the DC and keeps a track of which resources have been allocated to which slice and a User Manager that manages all of the users that can access the Slice Controller.
- a Slice Information Store, which database lists all of the slices and all of the resources in the slice, together with meta-data such as the VIM REST entry point, and the keys used for access to all the resources.

- a Slice Creator, which is responsible for handling requests for slices and interacting with the Resource Manager and the User Manager to determine if it is possible to create a new slice. If the slice creation is possible it interacts with the VIM Factory.
- the VIM Factory, which is able to allocate a VIM of a particular type, and configure it to use the resources which have been picked by the Slice Creator. Once the VIM is allocated and deployed, the REST entry point is returned to the caller.
- the VIM Placement Manager, which is responsible for determining which host should be used to execute a newly created VIM.

3.2.5 Monitoring System

SONATA monitoring framework collects and processes data from several sources, providing the developer the ability to activate metrics and thresholds in order to capture generic or service-specific behaviour. Moreover, the developer can define rules based on metrics gathered from one or more VNFs deployed in one or more NFVIs in order to receive notifications in real time. In general, the developer is able to subscribe to a message queue or he can get the alert notifications by email and/or SMS on his smartphone. Most importantly, monitoring data and alerts are also accessible through an API or directly accessing a websocket URL. The components that monitoring framework comprises of are explained below:

- **PushGateway** is a component that is utilized so that the probes/sources use HTTP PUT method to "push" monitoring data to. The advantage of this approach is that in the case of the deployment of a new service, there is no need for the Prometheus monitoring server to search for data related to the newly deployed VNF, but rather collect them from the PushGateway.
- **Monitoring server**: is based on Prometheus open-source monitoring system, based on time series database that implements a highly dimensional data model. A time series entry is identified by a metric name and a set of key-value pairs. Monitoring server has a sophisticated local storage subsystem (LevelDB), which is essentially dealing with data on disk and relies on the disk caches of the operating system for optimal performance. Monitoring server is mainly responsible for collecting the data, communicating with the time-series database for retrieving data upon request and processing the data according to predefined thresholds.
- **Monitoring Manager**: offers APIs to the users with respect to the monitoring data of their instantiated 5G services, including: 1) the relation among services, network functions, NFVIs and users, 2) the ability to modify rules and thresholds during service/function runtime, 3) the reconfiguration of Prometheus server, 4) the ability to define the notification methods in case of alert generation, 5) the definition of a new websocket to get data in real-time and many other features.
- **Alert Manager**: is responsible for sending notifications about firing alerts to the subscribed users. After this notification, the user can take advantage of the API to further investigate the fault or activate a websocket to receive real-time monitoring data.
- **Websocket server**: allows the user to collect streaming data from VNFs that have been deployed in the Service Platform. This is highly beneficial to the developers, as they would be able to monitor the performance of a new service in timely and under real conditions.

4 SONATA's pilots

The selected pilots considered for the validation and evaluation are: (i) Virtual Content Delivery Network (vCDN), (ii) the Personal Security Application (PSA), (iii) Hierarchical Service Provider (HSP) and Multi-orchestrator and slicing. All pilots provide capabilities that go beyond the basic functionality of SONATA platform that include on-boarding, deployment, and instantiation of services. vCDN pilot innovations are mainly related to the virtual CDN service deployment optimisation and service resource management during runtime. PSA pilot innovations are mainly related to additional security-related services, specifically anonymity and safe browsing. HSP pilot innovations are mainly related to the multi MANO interworkings, SP recursiveness of the service on-boarding and abstraction layer and slice management.

SONATA's pilots were elaborated after an analysis conducted in deliverable [12]. We have produced Table 4.1, which maps main requirements of SONATA to pilots as means of validation. The table presents functional and non-functional means for verification for each requirement. In order to present the status of the validation conducted through the pilots an additional column is used (**Validation result**) with a short summary of the result.

Table 4.1: Requirements mapping

Requirement Name	Pilot	Functional Verification	Non-functional Verification	Validation result
VNF Catalogue	1, 2	On-boarding of functions from the VNF Catalogue	Time to on-board (measure time(sec) required to on-board a VNF to be available by the platform)	The functional requirement was validated through the successful deployment of all pilots. The non-functional was also measured and is presented in Section 5.
VNF Placement	1, 2, 3	Placement of the various NS components to specific locations in the underlying infrastructure	Time (sec) to calculate the placement decision (depends on the complexity of the infrastructure and of the NS)	Functional: two algorithms were developed for the placement of VNFs, Non-Functional: the time required for placement is provided in Section 5.
Service Function Chaining	1, 2, 3	Chaining of VNFs components of the same NS in a single end-to-end chain (inside the PoP)	Time (sec) to setup the chain in the PoP; Latency (sec) caused by the chained path; Number of rules applied for the SFC to apply	Functional: successful operation of the pilots validates this requirement, Non-Functional: time to setup is measured, other metrics were not useful due to small number of nodes in the SFC
VNF Specific Monitoring	1, 2	Monitoring of VNF specific metrics via SONATA monitoring framework	Overall metrics transmission overhead (bps); Latency for the transmission of alert (sec)	Functional: validated, Non-Functional: overhead has been measured in Section 5
NS/VNF Scaling	1	Scaling a VNF or a Network Service based on monitoring and alert thresholds	Time (sec) to scale	Functional: validated for vCDN and PSA

Requirement Name	Pilot	Functional Verification	Non-functional Verification	Validation result
NS reconfiguration	1, 2	Reconfiguration of NS (i.e. update of VNFFG or extension)	Time (sec) to reconfigure	Functional: validated for PSA, Non-functional not measured
SP Recursiveness	3	SP to SP communication for deployment of NS on separate administrative domains	Time (sec) to deploy; Security features	Functional: validated, Non-Functional: not measured

4.1 Virtual Content Delivery Network pilot

4.1.1 VCDN's architecture and components

As presented previously in deliverable D2.1 **??** (initial use case discussion) and D6.2 (pilot discussion), vCDN focuses on showcasing the SONATA system capabilities in order to succeed in the development, deployment and enhancement of a virtual CDN service with features leverage the elasticity and programmability that the SONATA framework provides. In this context the service orchestration part of a vCDN that manages all the lifecycle of the service coordinating the service deployment, placement and management during runtime is handled through the Service Specific Management plugins. At the same the monolithic Virtual Network Function Manager (VNFM) as envisaged by ETSI NFV is implemented in SONATA via the Function Specific Management plugins. The anticipated benefits is flexibility in the service orchestration, ease of updates or service upgrades with new functionalities or algorithms and service responsiveness to network conditions or events. The deployment and operation of CDNs is in our days well exploited, we argue that in our implementation, we are not aiming into direct comparison with commercial solutions. Even performance wise, our implementation is merely a prototype or a proof of concept rather than a full blow, rich featured implementation. The main SONATA scoping for the vCDN service we are presenting is to highlight and demonstrate SONATA's capabilities adhering to the use case selection methodology as laid out in deliverable D6.2 **??**. Originally two scenarios are anticipated for this pilot, 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)*: Identify and redirect user generated content through the vCaches in order to ingest user created content in the vCDN. The SONATA SP allows the flexibility of dynamically extending the vCDN service, accommodating additional sources from alternative Content Providers.
- *QoE adaptive mode*: an extension to the above functionalities can be seen by the introduction of an additional transcoding 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 service chain vTranscoder.

As mentioned previously, the components that are considered for the implementation of vCDN, will not be developed from scratch rather than use readily available VNFs or modified versions of existing ones. however, SSM and FSM plugins need to be be developed as the SONATA paradigm

affects the way a Virtual Network Function Manager (VNFM) or a Service Orchestrator will be implemented. The overall deployment of vCDN Use Case is illustrated in Figure 4.1. 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, an SSM plugin is used in order to manage the placement of new VNFs as well as any service scaling or re-configuration decisions. In addition, for each VNF (i.e. instances located at various PoPs), FSM plugins deal with the placement, reconfiguration or scaling of the VNF Components (VNFC) of each VNF. For example, additional instances of Virtual Traffic Classifier (vTC) DPI engine might be required, in order to handle the traffic load at certain PoPs. At the same time, new edge locations may require deployment of additional vCDN VNFs (i.e. vCaches) based on deployment policies.

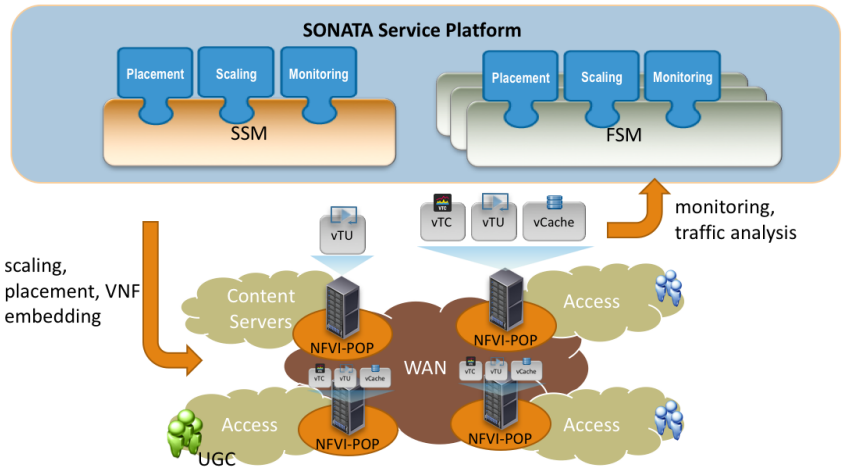

Figure 4.1: vCDN network service deployment

All the interactions are driven by the monitoring and traffic analysis capabilities at the locations part of 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.).

4.1.2 Deployment scenarios

In the following subsections the various deployment scenarios for vCDN are discussed.

4.1.2.1 Scenario 1 - Network Service reconfiguration

- The NS is instantiated by the SP on top of the already provisioned network slice. (Assumption 2).
- SSM and the FSMs are instantiated.
- SSM placement plugin is deciding on the proper placement of the VNFs to the available NFVI-PoPs, taking into account the explicit placement and also resource availability.

- The VNFs are instantiated and signaling to the FSMs is established.
- The SFC is established and traffic from the content servers is now passing through the deployed VNFs.
- Content is now received on end users' terminals.
- monitoring information is collected by the VNFs and the infrastructure elements.
- alerts are issued by SONATA monitoring framework and collected by the ssm-mon plugin (SSM Monitoring plugin).
- In another end point of the network service, end users start to consume content however there is no vCache for them.

Case 1 : Manual triggering of the new placements

- Customer decided that a new vCache is needed so he reconfigures his service.
- SSM Placement plugin receives a request (external) that a new vCache is to be instantiated at a prefered (explicit) location to accommodate the traffic at the edge.
- The ssm-placement (SSM Placement) commands the instantiation of the new VNF and modifies the established chain.
- The new SFC might have two branches originating from the same Content Servers segment and directed to two edges PoP2 and PoP3.
- The service at the edge where the new users reside is a) better quality or b) just available depending on the objectives set by this scenario.

Case 2 : Automatic triggering of new placement

- Alerts are received by ssm-mon (related to usage of the service on all branches towards the edge locations).
- ssm-placement detects that at a particular edge new users are connected, thus increase of the aggregate traffic towards that particular edge is detected.
- ssm-placement checks if that edge is served locally by a vCache.
- When alerts surpass the configured threshold, automatic placement of vCache is requested.
- The PoP in proximity to the edge location is identified (ssm-placement).
- the SSM coordinated the required lifecycle operations in order to deploy the vCache and update the NS.

4.1.2.2 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.

4.1.2.3 Scenario 3 - QoE enhancement

This scenario is an extension of the vCDN service including a DASH transcoding unit. The transcoding functionality can produce new content per combination of elements (available bandwidth, terminal information, etc.). By choosing the best suitable transcoding and segmentation, it ensures the best Quality of Experience (QoE). The vTranscoder will be exploited on-demand according to the situation and customer needs.

- Upon user request of a content format or quality that is not available
- vTC forwards request to the vTU
- vTU manages the transcoding of the content based on the user request
- As soon as the initial segments are transcoded, they are made almost immediately available to the content server
- Upon new request for the new content format the content server streams the content to the user.
- vTC is monitoring the whole process.

4.1.3 vCDN SSM and FSM components

The vCDN pilot is able to take advantage of the enhanced management capabilities of the SONATA platform (SP) provided through Service Specific Managers (SSM) and Function Specific Managers (FSM). Through these managers, the developer is able to replace default orchestration operations for both services and functions by customized and more dynamic orchestration patterns. For the vCDN pilot, two SSMs (placement SSM and configure-monitoring SSM) and three FSMs (vTC FSM, vCC FSM and vTU FSM) were developed and are actively customizing the lifecycles of the service and each function.

4.1.3.1 Placement SSM

The vCDN placement SSM overwrites the generic placement logic of the SONATA SP. The generic SP placement doesn't take proximity between VNFs on one side and the source and destination of the service on the other side into account when performing placement calculations. Since the QoS of the vCDN will improve if the vCC and the vTC are located close to the user of the service and the vTU close to the content server, we designed a placement SSM to establish just that. By comparing the IP addresses of the users and the content server to those of the available PoPs, this placement SSM locates those PoPs closest to the users/content server, and nominates them as hosts of the correct VNFs.

4.1.3.2 Configure-monitoring SSM

The configure-monitoring SSM takes care of customizing orchestration behavior when it comes to configuring the service and in terms of dynamically reacting to monitoring triggers. The default NS deployment behavior of the SONATA SP doesn't include an additional configuration step for the VNFs after they are started (which already includes a configuration step). Since its only input is the VNFR, the start event of a VNF sometimes has not enough information to completely configure it, e.g. when it needs IP addresses of the data interfaces of its adjacent VNFs or when it needs the IP address of the source/destination of the service. Since this is the case for both the vTC and

vTU, the configure-monitoring SSM will trigger the SP to execute an additional configuration step for these VNFs while also defining the input for these events.

Since this SSM is also defined as a monitoring SSM, the SP will forward it all received monitoring information. This allows the SSM to determine whether the NS needs a configuration change to adapt to a new context (e.g. when the amount of requests of the user increases). Once such a decision is reached, the configure-monitoring SSM can instruct the SP to execute this configuration change.

4.1.3.3 vTC FSM

There are a number of functions and configurations inside the vTC FSM component. Initially, the `start_event` of the FSM start event plugin is responsible to start the vTC in order to have the traffic classified and monitored. It is also used as a configuration tool for the monitoring services running inside the VNF. Another FSM plugin is the `configuration_event`, which configures the VNF to be able to identify the Transcoding requests (by its destination port) and able to redirect them towards the Transcoding Unit (by changing the destination IP). Finally, the FSM is also able to stop the vTC's functions upon request.

4.1.3.4 vCC FSM

The FSM attributed to the vCC VNF only has a few default functions. It is responsible and able to start or stop the VNF services upon request and proceed to the initial configuration and initiation of the monitoring tools used by the network service.

4.1.3.5 vTU FSM

The FSM in the vTU is responsible for the initial configuration of the VNF. This includes mounting the NFS folders from the Content Server, done dynamically with the IP discovery, and publishing the Job_id (different every time the VNF boots) to the Content Server, required for the transcoding service. It is also responsible for the configuration of the monitoring tools. Another function of the FSM includes the source NATing of the responding transcoding requests, originating from the user, so that the request is successfully done.

4.1.4 vCDN VNF validation

Functionality and performance of the individual VNFs on the vCDN pilot have been evaluated in the following ways:

4.1.4.1 Virtual Traffic Classifier

The vTC has been tested and evaluated by confirming that network traffic is not only able to pass through the VNF but also identified. Through the VNFs web admin page it is possible to visually observe the traffic passing through, as well as observe the different types of traffic due to the various charts available. The validation process includes two different hosts trying to connect to each other, configured though to pass through the vTC. In order to confirm the requested functionalities, the vTC core component, pfbridge, can be started and stopped at will, which allows to observe the successful or not flow of the traffic.

4.1.4.2 Virtual Cache-Content

In order to validate the vCC, a new vCC VNF is initiated which, by default, has an empty cache. Through the appropriate network configurations and using two external nodes, the user and a content server, it is able to access the server's content from the user using the cache. As it is normal, the user receives the data requested at the permitted network speed. On subsequent tries however, since the requested data was successfully cached, the content is received in a much faster way.

4.1.4.3 Virtual Transcoding Unit

The vTU is validated by its ability to perform the incoming transcoding requests. First step is to confirm that the vTU has successfully mounted the required files from the Content Server and that the docker containers with the transcoding services have been initiated. The process continues with transcoding requests directed to the VNF. At that stage in order to validate the service, a file inside the vTU, containing the available video qualities, must have been updated, which will subsequently trigger the update on the content server as well, due to their mounting connection.

4.1.5 vCDN NS validation

4.1.5.1 SONATA SDK-based validation

Before the vCDN pilot is actually deployed on SONATA evaluation environments, it is already statically validated using the validation capabilities of the SONATA SDK. To do this, the NS developer has to install the `son-cli` tools on her development machine and clone the pilot repository available on GitHub. In this repository, all artifacts for the pilot can be found, for example, the NSD, the VNFDs, or the FSM and SSM implementations.

To validate the descriptors, the `son-validate` tool can be either used on the command line, like shown in Figure 4.2, or its GUI can be used as shown in Figure 4.3. As described in SONATA's SDK software release deliverable D3.3 [9], the validation tool is able to verify the following aspects of a give set of descriptors:

- *Syntax:* Checks the given descriptors against the corresponding schema definitions.
- *Integrity:* Performs additional semantics checks for the descriptors, e.g., does the NS contain at least one network function? Are the management interfaces well-defined?
- *Topology:* Checks the topology of the defined network service and its VNFs. Can, for example, detect loops in forwarding paths and throw a warning.

The verification considers four kinds of artifacts of an SONATA SDK NS project:

- *Project:* The folder and file structure and configuration of a SONATA SDK project.
- *Package:* The meta description and structure of generated SONATA packages.
- *Service:* The network service descriptors (NSDs) including the defined forwarding paths.
- *Function:* The virtualized network function descriptors (VNFDs) of the project.

This process only takes a couple of seconds and helps the NS developer to already detect a lot of potential bugs in the NSDs and VNFDs, for example, missing connection points or typos in

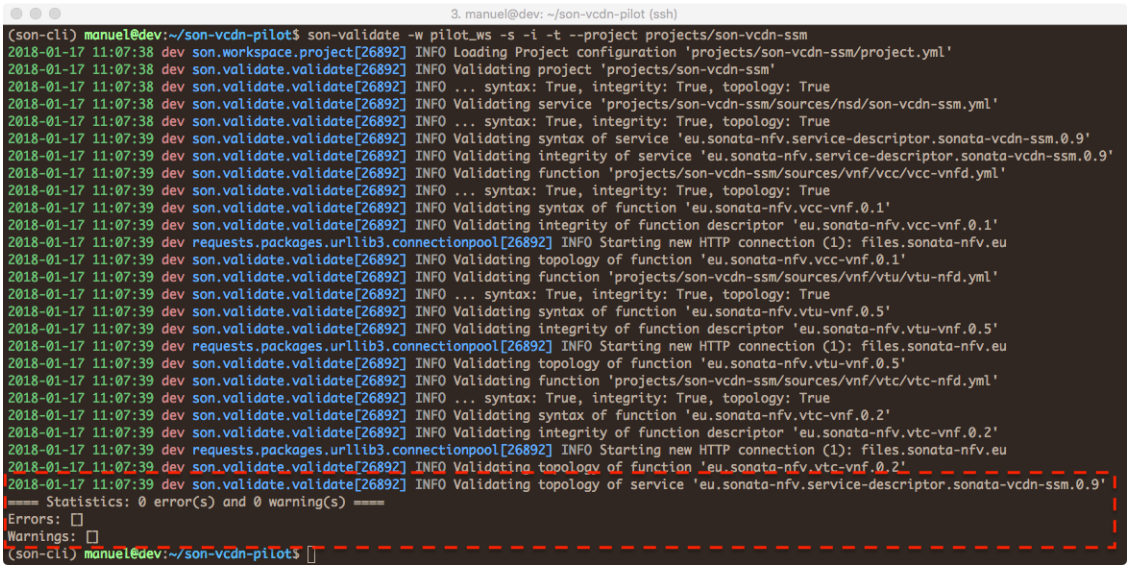

Figure 4.2: Descriptor validation of vCDN service using the SONATA son-cli tools

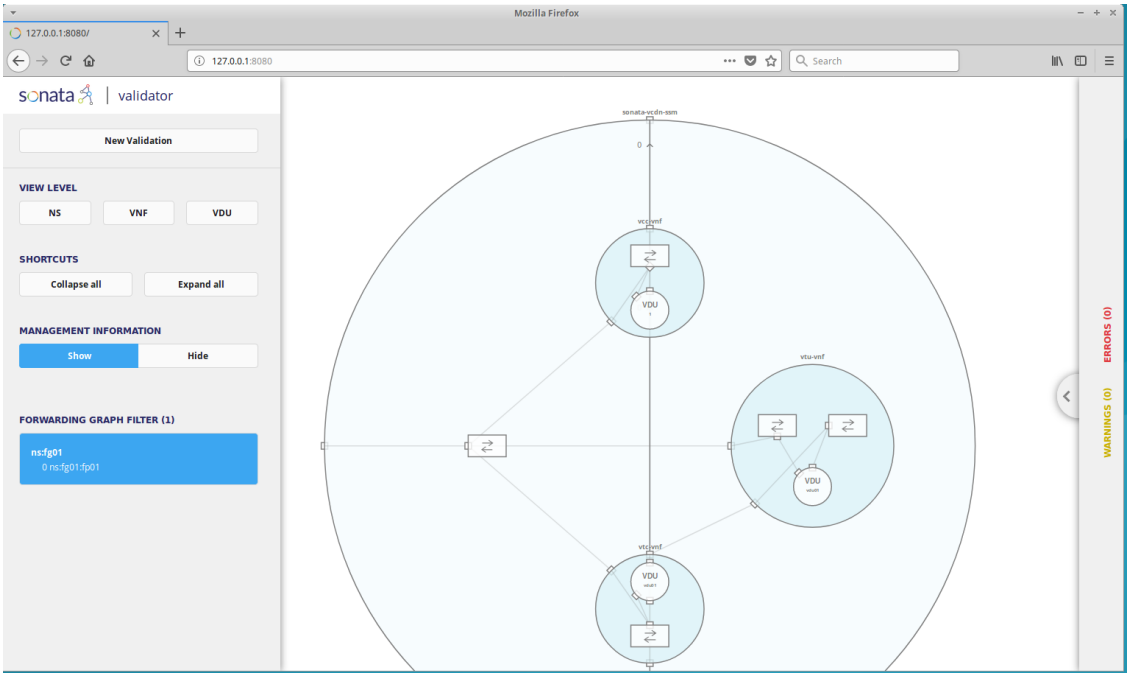

Figure 4.3: Descriptor validation of vCDN service using the SONATA son-validator GUI

references. Since these bugs can be detected before the NS is deployed on a service platform, the development times for NSs is drastically reduced.

After the developer has validated the NS, she needs to package it in order to on-board it to a service platform for instantiation. This tasks is also done by the SONATA SDK using the `son-package` tool. This tool requires a single CLI command to trigger the packaging process of a complete NS project, including its NSDs and all VNFDs as shown in Figure 4.4. After the packaging process has been completed, the package is available on the developer's hard disk as a single file for further use, e.g., `eu.sonata-nfv.son-vcdn-ssm.0.1.son`. During the packaging process, the syntax validations are always performed as default.

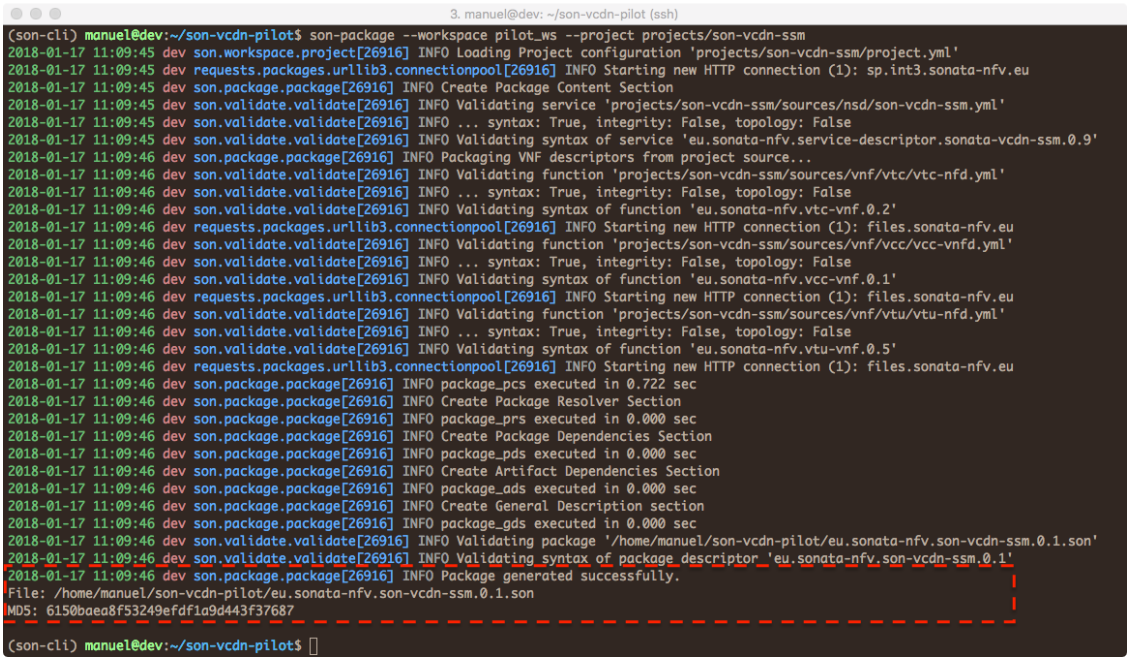

Figure 4.4: Packaging of the vCDN service using the SONATA son-cli tools

4.1.5.2 vCDN Pilot deployment and validation

In order to be able to validate the Network Service that is used in the vCDN pilot, a deployment of the NS is needed and a verification that all its components have deployed successfully. Subsequently there is the need to manually confirm that the traffic flows through the service as it was defined on the NS originally. In order to achieve this steps, it is needed for the Network Service package to be uploaded on the SONATA platform, so that it can be ready for deployment. In Figure 4.5 the marked service is the one that is needed for deployment of the pilot.

After the successful Network Service deployment of the vCDN pilot from the SONATA BSS, the next step is to check if the VNFs were deployed correctly. In Figure 4.6, there are two stacks visible, each one in a different PoP. Due to the placement rules from the SSM two of the VNFs, vTC and vCC, are located in the PoP that is closer to the end user side in order to provide the best QoS, and the last one, vTU, is located in the PoP closer to the Content Server.

When the deployment and validation of the VNFs is completed, then the network traffic flowing and the integrated functionality of the VNFs in the Network Service can be validated. At first, the Client/User host that is going to consume the video from the Content Server (CS) has to be accessed. There, using the installed DASH player, the Client can access the video located in

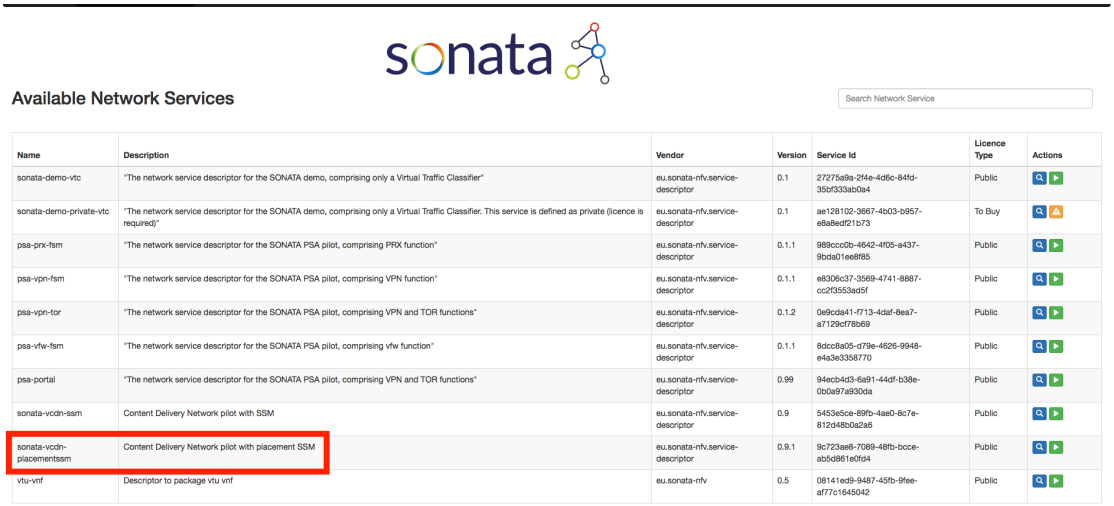

Figure 4.5: NS deployment request for the vCDN pilot

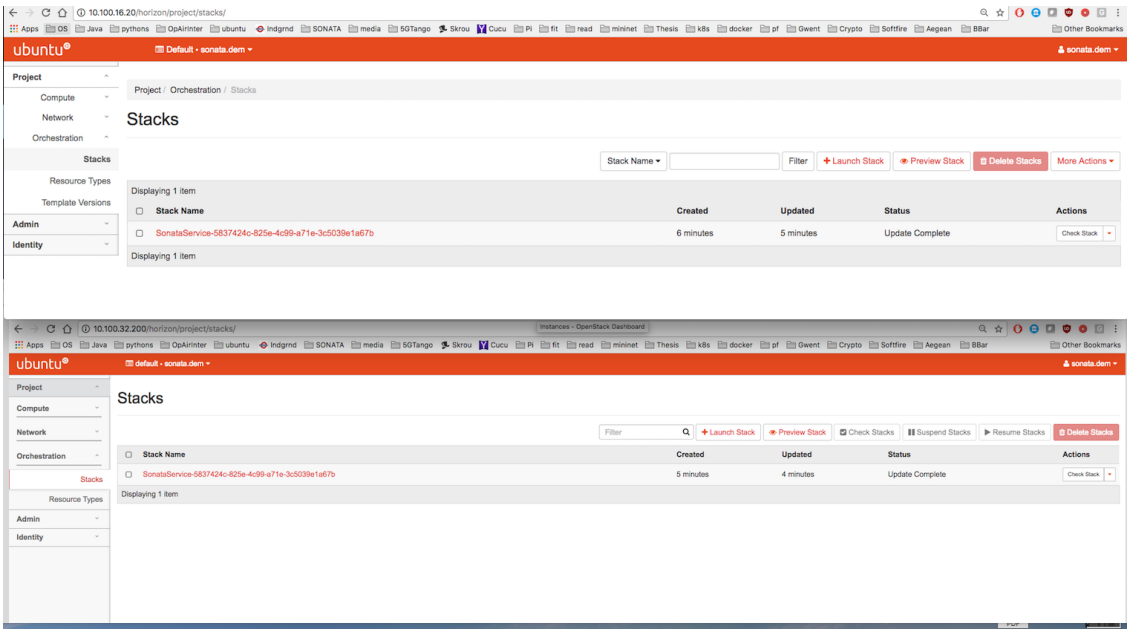

Figure 4.6: NS deployment request for the vCDN pilot

the Content Server, though the Service and the VNFs, vTC and vCC, in between. This can be validated with the following steps:

- The Client is able to actually see the video streamed
- Through the vTC dashboard, the traffic produced from the video streamed can be observed
- When the Client requests content from the CS multiple times, then the time required for the process is significantly reduced after the initial effort due to the vCC caching the content
- Finally in order to confirm that the Transcoding Request has been completed, the file responsible in the Content Server that provides the different resolutions has to be altered. Since the file is mounted on the vTU the file must change there as well. In Figure 4.7, the file can be seen before and after the Transcoding process, where the added content is highlighted.

A visual representation of the above steps is available to be observed in the Figure 4.7

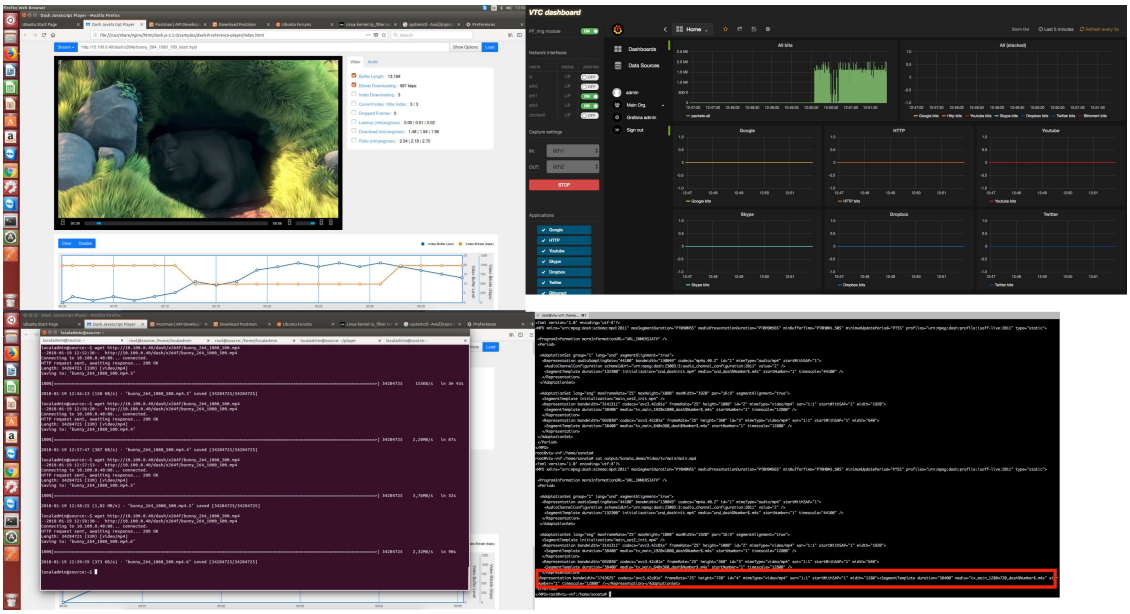

Figure 4.7: `Streaming video through vTC and vCC caching`

4.1.5.3 vCDN monitoring

The final step to the overall Network Service validation is to confirm that the Monitoring probes located inside the VNFs are installed and configured successfully. This can be done by checking the Service Platform devoted tab on monitoring. Each VNF should be represented there graphically, transmitting the data from the VMs, to the Service Platform. In Figure 4.8 a couple of screenshots have been produced. One from the typical view of the graphical representation of the monitoring probes in the VNFs, and another one with a more detailed view of the VNF monitoring metrics.

4.1.6 vCDN innovations

The vCDN pilot is a proof of concept for the deployment of the components comprising a Content Delivery Network using an NFV framework. Similar efforts have been proposed by other projects

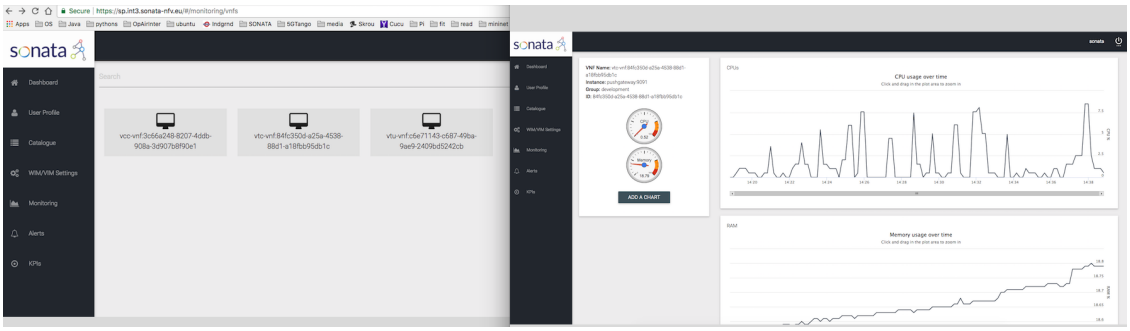

Figure 4.8: Service Platform monitoring Network Service

(e.g., 5GEx as well as others) that focus mostly on the end-to-end inter-domain deployment of the vCDN. SONATA's version attempts to focus more in the SONATA Service Platform programmability features highlighting the flexible service orchestration and NS placement capabilities. The demonstrated vCDN innovations are mainly referring to the service deployment optimisation and service resource management during runtime. Moreover the introduction of a transcoding unit allows additional optimisations to occur during runtime, depending on the status of the service, the number of sessions, location based event triggering etc. Summarising the innovations from this pilot are:

- Optimised placement, taking into account the functionality of each VNF and the context of each end-point (i.e. Access, Application Servers etc). This is achieved thanks to the FSM/SSM plugin structure at the SP level that enable the introduction of alternative placement algorithms
- Dynamic adaptation of service: This is achieved in the case that the service orchestration and management detects

4.2 Personal Security Application pilot

The personal security application (PSA) scenario is loosely based on an access provider wanting to sell value-added services, specifically anonymity and safe browsing. The technical focus of the demonstration scenario is the on-demand reconfiguration of the service chain for a specific end-user. Selection happens via a self-service portal at which the user can choose the service chain he would like to use. Changing of the service chain is on-demand and takes effect nearly immediately.

4.2.1 PSA architecture and components

The PSA pilot consists of three large subsystems: the self-service portal, the MANO plug-ins, and the virtual network functions (VNFs). These components and their interactions are shown in Figure 4.9. The self-service portal is the component which interacts with the end-user and which translates user-requests to service requests to the MANO system. The MANO plug-ins realize the specific management functions which are needed to provide on-demand adaptable service chains to individual users. The MANO plug-ins extend SONATA's service platform's orchestration system with the capabilities to reconfigure the VNFs in a way that realizes the specific service chain an end-user selects.

A more detailed view of the architecture is given in Figure 4.10. The self-service portal is a stand-alone component, running in its own virtual machine. It is implemented as a small set of

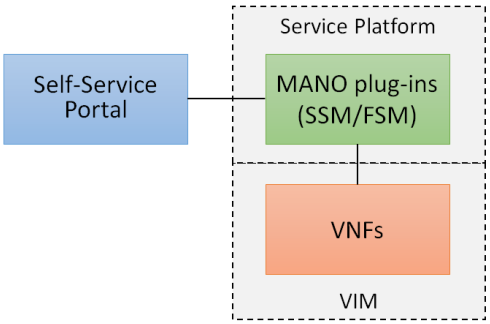

Figure 4.9: PSA Overview

communicating micro-services. A front-end micro-service is interacting with the end-user's Web browser. It serves a Web page which is visualizing the available service chains and their states to the end-user. The front-end is accepting service chain selection requests from the user and forwards them to the back-end. The back-end keeps information about the selected service chain and persists it to a data base. The back-end propagates service chain selection requests to the SSM of the service. To that end, the back-end maintains contact with the SSM. Upon start-up, the SSM connects to the back-end and a Websocket connection is kept alive for the lifetime of the service (and, therefore, the SSM).

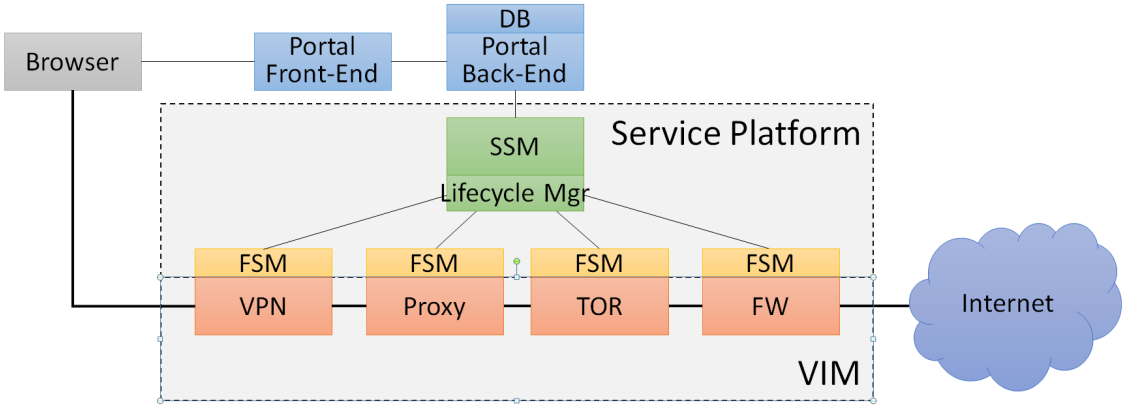

Figure 4.10: PSA Components

The SSM keeps track of all the VNFs that are part of the network service. Upon request of the self-service portal, the SSM calculates the set and order of reconfigurations necessary and communicates them to the FSMs of the affected VNFs. The FSMs accept requests for reconfiguration from the SSM and pass them on to their associated VNFs. As the VNFs have different capabilities and mechanisms for reconfiguration, the FSMs translate the requests from the SSM to instructions appropriate for "their" VNFs.

4.2.1.1 PSA's VNFs description

The PSA pilot makes use of the following three VNFs: VPN server, Web proxy, TOR client. These are described in the following sections.

VPN

The VPN VNF is the access point of the PSA service for the user's data. It contains a VPN server provided by OpenVPN [19]. The end-user starts a VPN client on his host to create a VPN tunnel to the PSA service. This client will re-route all the host traffic into the PSA service. The VPN encryption will protect the forwarded packets.

Proxy

The proxy VNF caches HTTP requests to accelerate their delivery and lowers the bandwidth by sharing the same remote content to a number of end-users. The current implementation is based on Squid. Moreover, it also filters the access to some resources by blacklisting some URLs. This feature is used as a remote ad-blocking system.

TOR

The anonymity VNF is based on TOR [22] and is always used in conjunction with the VPN service. The TOR VNF sends all traffic to the TOR network and makes sure the user cannot be traced back. The anonymity service forwards Internet traffic through a free, worldwide volunteer network consisting of more than seven thousand relays to conceal a user's location and usage from anyone conducting network surveillance or traffic analysis. As part of the Personal Security Application (PSA), this makes it more difficult for Internet activity to be traced back to the end user. This includes visits to Web sites, online posts, instant messages, and other communication forms. Thus, it offers an additional security layer for the user.

Firewall

The firewall service is implemented by the PFSense commercial software based on Freebsd. Additionally to the possibility to block or allow specific traffic, PFSense provide a user friendly graphical interface to easily configure the security policy, and monitor the malicious activities.

4.2.2 PSA demonstration scenarios

As described in deliverable D2.3 [3], the personal security application (PSA) scenario is loosely based on an access provider wanting to sell value-added services. In addition to simply providing Internet connectivity, the access provider wants to sell services which enhance the user experience. In the PSA scenario, these additional services are security-related, specifically anonymity and safe browsing.

The technical focus of the demonstration scenario is the on-demand reconfiguration of the service chain for an end-user. There are multiple service chains a user can select. Selection happens via a self-service portal at which the user can choose the service chain he would like to use. Changing of the service chain is on-demand and takes effect nearly immediately.

Although the system supports arbitrary service chains, three service chains are used in the demonstration scenario. These are shown in Figure 4.11. The first service chain, the so-called "default" service chain consists of only a VPN server and a Firewall with the latter being connected to the Internet. The end-user can connect to that VPN server and browse the Internet. The firewall protects the user from hiker attacks and log them in a graphical interface for future deeper analysis.

The first value-added service is the "basic" service chain. It consists of the VPN server, the firewall and a TOR node for anonymity. This service chain offers anonymized browsing services for the end-user.

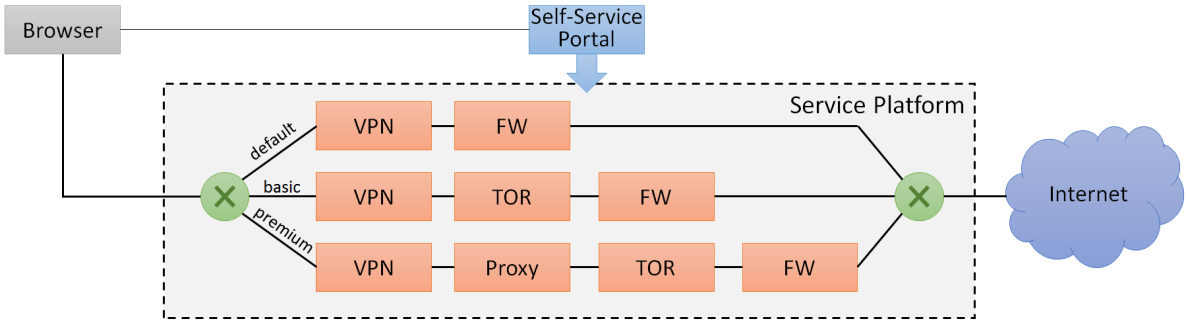

Figure 4.11: PSA Demonstration Scenario

The second value-added service is the "premium" service chain. In addition to the VPN server, firewall and TOR components, it contains a URL-filtering proxy. The proxy filters and denies access to sites which are deemed advertising sites. By using this chain, the end-user is not only offered anonymized browsing capabilities, but in addition is protected from accessing advertisement sites which are often employing user tracking mechanisms and protected from outside malicious attacks.

The self-service portal offers the user the choice of these two value-added services, allowing him to start or stop the desired service. The self-service portal connects to the MANO components of the service platform (specifically, the PSA SSM, see below) and instructs them to reconfigure the service chain as selected by the user. The reconfiguration is effected immediately, so that the end-user only experiences a short outage (or even none at all).

The PSA demonstration scenario contains four VNFs, the VPN, the Proxy, the TOR server and the firewall. Although the different service chains could contain an arbitrary number of VNFs, the main goal of the PSA pilots is to demonstrate the network chaining and its dynamic reconfiguration. To focus on this aspect, the actual pilot has been simplified by reducing the number of VNFs down to the four mentioned ones.

In the following, the concrete scenario steps are listed.

4.2.2.1 Deployment

- The Self-Service Portal (SSP) is deployed in a virtual machine on the virtualized infrastructure.
  - A VM is created and an image containing a Linux operation system is booted.
  - The self-service portal code is cloned from its github repository (https://github.com/sonata-nfv/son-selfservice).
  - The self-service portal is started.
- The PSA-NS is instantiated by the SP on the same virtualized infrastructure as the SSP.
- SSM and the FSMs are instantiated.
- The SSM establishes a connection to the self-service portal.
- The SSM placement plugin is deciding on the placement of the VNFs and instantiates them.
- The VNFs are started and signaling to the FSMs is established.

- The SSM instructs the FSMs to set up the default service chain, which consists of the VPN and Firewall VNFs only.
- The user can now connect to the VPN and browse the Internet through the VPN and the Firewall.

4.2.2.2 Using the "basic" service chain

- The user connects to the self-service portal with a Web browser.
- A page with the service chain selection is displayed.
- The user selects the "basic" service chain and triggers its creation by clicking on the "start" button.
- The portal forwards the chain selection action to the SSM.
- The SSM sends configuration commands for the "basic" service chain (i.e., VPN, TOR, FW) to the FSMs.
- The FSMs trigger reconfiguration of their associated VNFs.
- Routing inside the VPN and TOR VNFs is changed, so that traffic flows from the VPN to the TOR to the Firewall to the Internet.
- The user can now benefit from the TOR anonymization capabilities when surfing the Internet through the chain.
- Usage of TOR can be checked by looking at the IP address the user is browsing with.

4.2.2.3 Using the "premium" service chain

- The user now selects the "premium" service chain from the self-service portal and triggers its creation by clicking on the "start" button.
- The portal forwards the chain selection action to the SSM.
- The SSM sends configuration commands for the "premium" service chain (i.e., VPN, TOR, FW and proxy) to the FSMs.
- The FSMs trigger reconfiguration of their associated VNFs.
- Routing inside the VPN, TOR, Firewall and Proxy VNFs is changed, so that traffic flows from the VPN to the proxy to the TOR to the firewall, to the Internet.
- The user can now benefit from protection of the firewall, TOR anonymization and Proxy URL filtering when surfing the Internet through the VPN.
- Usage of the proxy can be checked by trying to access a black-listed URL.

4.2.3 PSA SSM and FSM components

The PSA pilot makes heavy use of the enhanced management capabilities of the SONATA platform. It includes a service-specific manager (SSM) as well as function-specific managers (FSMs). The SSM has control of the overall service provided to the end-user. It keeps track of the VNFs involved in the PSA service and orchestrates updates of the service chain. It connects to the self-service portal and receives requests for changing the active service chain. Based on these requests, the SSM instructs the FSMs of the affected service chain VNFs to reconfigure appropriately.

The PSA SSM is triggered once upon instantiation of the PSA service. At this point, it passes on the initial configuration (VPN/FW-only service chain) to the VNFs. The SSM is then triggered by the portal for each request to change the service chain (e.g., to the "basic" or "premium" service). It calculates the set and order of reconfigurations necessary and communicates them to the FSMs of the affected VNFs.

The FSMs accept requests for reconfiguration from the SSM and pass them on to their associated VNFs. As the VNFs have different capabilities and mechanisms for reconfiguration, the FSMs translate the requests from the SSM to instructions appropriate for "their" VNFs.

4.2.3.1 FSM objective

The FSM has the purpose to initiate and configure a specific VNF on behalf of the SONATA orchestrator. To reach this objective, it relies on the VNF containing a predefined OS image having the desired network functions (based on some open software product). This image is instantiated on the SONATA system and the FSM is relied upon to perform fine grained configurations inside the VNF image to adapt it to the specific network service the VNF belongs to.

4.2.3.2 FSM operation

An FSM requires two main steps to perform its functions, the registration in the SONATA platform and the processing of events that will trigger the VNF configuration.

Registration

A new FSM must perform a registration inside the SONATA system in order to be able to configure its target VNF. Only when the registration is achieved with success does SONATA permit the VNF to be created. The FSMs are coded in Python and the registration concretely is attained on the class constructor that implements the FSM - in fact the trick is done on an inherited class provided by the SDK framework.

The registration consists of a message sent on a message bus channel using all the specific class parameters referring to the service. The SONATA system will accept this new registration and inform the FSM of the acceptance.

To complete the three way handshake the FSM replies to the system acknowledging the registration and will finally accept events from SONATA relating with its associated VNF. Only after this handshake will the FSM be "open for business".

Event processing

The FSM will only process 4 types of events from the SONATA system: start, stop, configure and scale. The other events are considered invalid.

All these events processing by default use an SSH connection through the management network interface on the VNF OS for the actions involved. Obviously the start and stop events are related with actions concerned with the beginning and the end of the VNF operations. On the start event

it is necessary to open the data network interfaces and start the local network service and the monitoring service present on SONATA. The stop event obviously performs the closing operations of service and monitoring on the VNF.

The configure event will perform modifications on the configuration of the VNF, permitting the dynamic modification of the VNF behavior and also network behavior modification related with network chaining. The scale event processing will involve actions related with the scaling up or down of resources used on the VNF operation and is currently not used in the PSA pilot.

4.2.4 PSA VNF validation

Functionality and performance of the individual VNFs have been evaluated in the following ways.

4.2.4.1 Self-Service Portal

The portal has been validated manually by interacting with its Web page. The reaction to clicks on different service chains was observed and compared to the intended behavior. Portal deployment has been performed on multiple test systems, including local ones (on the developer side), and on the SONATA PoPs. The variety of systems used for testing provides confidence that the micro-services are robust to handle different hardware types and performance levels and different start-up sequences.

4.2.4.2 VPN, TOR, Firewall, Proxy Server

The VNFs that participate in the service chain have been tested individually. For each one, a front-end instance has been created. Its next hop (gateway) has been configured to point to the tested VNF. For the VPN, the front-end instance was loaded with a VPN client. With this setup, the packet flow was confirmed while the core functionality of the VNF was verified (for example, that the Proxy VNF was filtering specific URLs).

4.2.4.3 VPN server

Individual VPN tests were performed to ensure the functionality of the VNF service. The VPN FSM is prepared to perform the necessary configuration for each possible scenario of the PSA, including the VPN standalone. Concerning the standalone scenario, the VPN configuration, such as routes and authentication rules, was tested to ensure the correct client access and authentication to the internal network.

4.2.4.4 TOR server

The TOR service was implemented to act as a transparent proxy to enter the TOR network. Similarly to the remaining services, its FSM is prepared to handle any necessary configurations for the different PSA scenarios. Standalone tests to this VNF were performed having only a service with the TOR VNF to ensure correct forwarding of the traffic to clients that use the service.

4.2.4.5 Firewall server

The PfSense firewall as been configured with NAT. Standalone tests have been made into the service platform to test that allowed traffic can pass through the firewall with IP address translation and that traffic from the outside is completely blocked. Then, the same tests have been executed with the VPN, and with a chain containing the VPN and the TOR services.

4.2.4.6 Proxy server

To perform a correct validation of this specific VNF, we must refer to its supposed functionality relatively to the PSA service - a proxy HTTP service. As such, when an adequate image of an approved OS with the required function - in this case implemented by a recent version of squid - is obtained, it is supplied to the SONATA system to be instantiated on demand. The instantiation of this image is done by a snippet of Python code in the FSM. The validation of the VNF consists primarily of testing the standalone VNF inside a service that would solely consist of this VNF. If its supposed functionality is achieved during an HTTP request, this VNF should be capable of joining a chain of VNFs in a more complex service.

4.2.5 PSA NS validation

The PSA network service (NS) consists of the three VNFs VPN, TOR, and Proxy. The NS contains an SSM, which takes care of function configuration, reconfiguration, and monitoring. The individual VNFs have associated FSMs which are taking configuration and reconfiguration requests from the SSM and enacting them on their associated VNFs.

The whole service has two external connection points, one towards users, the other one towards the Internet. The one towards users directly links to the VPN function, as this is the first function in all the possible service chains. The other functions are dynamically linked by configuration requests via the SSM and FSMs. The network service is deployed on a single PoP. As described in the vCDN pilot section, the SONATA SDK tools are used to perform initial validations and the packaging of the NSDs and VNFDs prior to the on-boarding process.

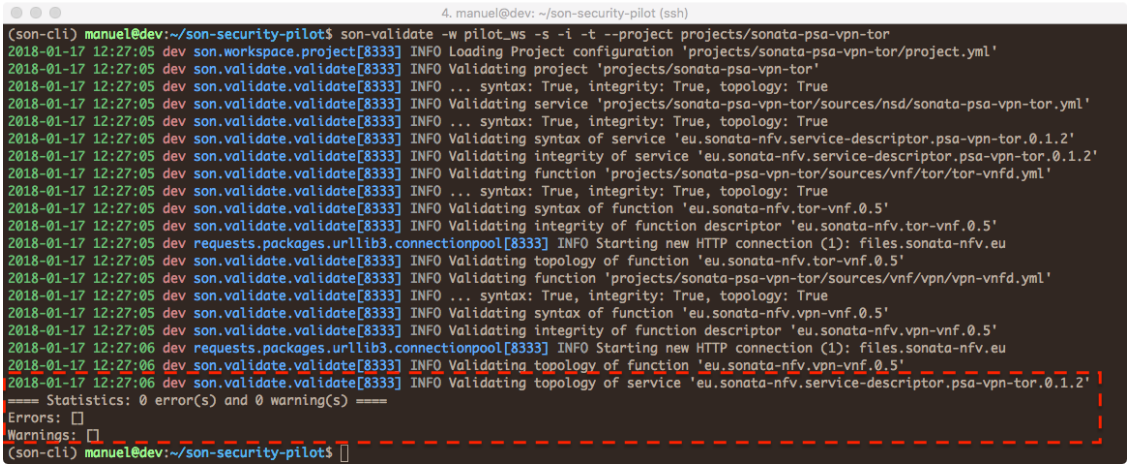

Figure 4.12: Descriptor validation of PSA service using the SONATA son-cli tools

Figure 4.12 shows how a developer can use SONATA's SDK validator to validate the network service descriptors and VNF descriptors of the PSA. It shows that the PSA project neither generates any errors nor warnings.

Figure 4.13 shows the graphical interface of the SONATA validator. One can see the PSA service showing the VPN VNF at the bottom and a zoomed-in view of the TOR VNF. A developer can easily see the interconnections and connection points defined in the descriptors and how they are chained together. The small box with the arrows in the TOR VNF circle represents the management network which can interconnect multiple VDUs of a single VNF and connect them to the outside.

Finally, Figure 4.14 shows how the PSA service is packaged using SONATA's SDK packaging tool on the command line. The SONATA SDK tools are then used to upload the network service

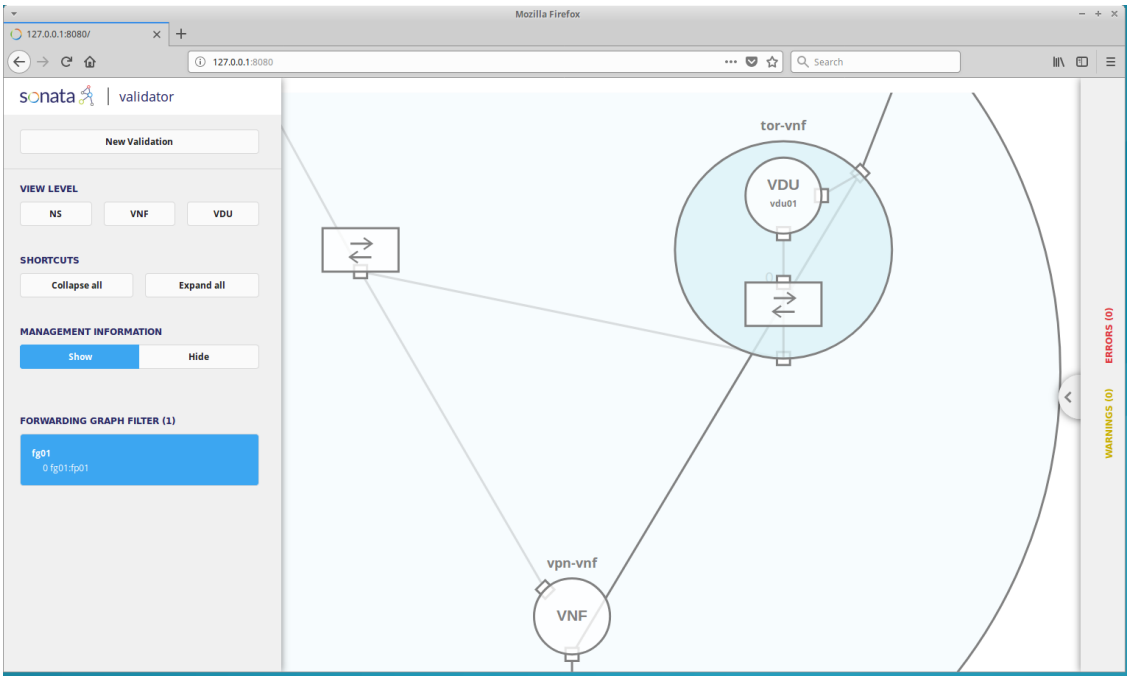

Figure 4.13: Descriptor validation of PSA service using the SONATA son-validate GUI

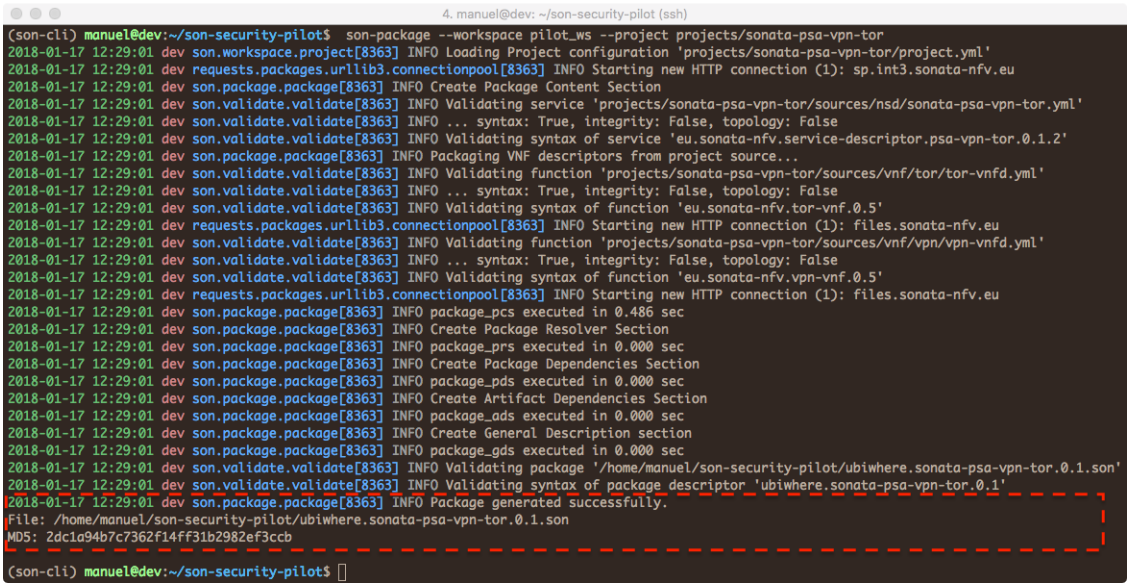

Figure 4.14: Packaging of the PSA service using the SONATA son-cli tools

and to instantiate it.

The next UML message sequence chart illustrates the user stories for PSA pilot.

4.2.6 Actions from the DevOps perspective

Figure 4.15: PSA onboard process

4.2.7 Actions from the SON-ADMIN perspective

4.2.8 Actions from the EndUser perspective

Validation of deployment success of the PSA service is done via the BSS GUI of the PoP, validating that the VNFs are instantiated and running properly. An example screenshot is shown in Figure 4.18.

Looking to the NFVI (Openstack) Dashboard will also show the deployed resources (namely Instance, Network Topology and Stack) on that PoP. An example is shown in Figure 4.19.

The IA component has correctly generated and deployed a Heat template for the PSA service, as shown in the Figure 4.20.

The self-service portal (SSP) is a separate component which is manually installed in its own virtual machine on the same PoP. Installation success of the portal is verified by accessing the portal's Web GUI. The initial GUI of the SSP is depicted in Figure 4.21 showing the selection of

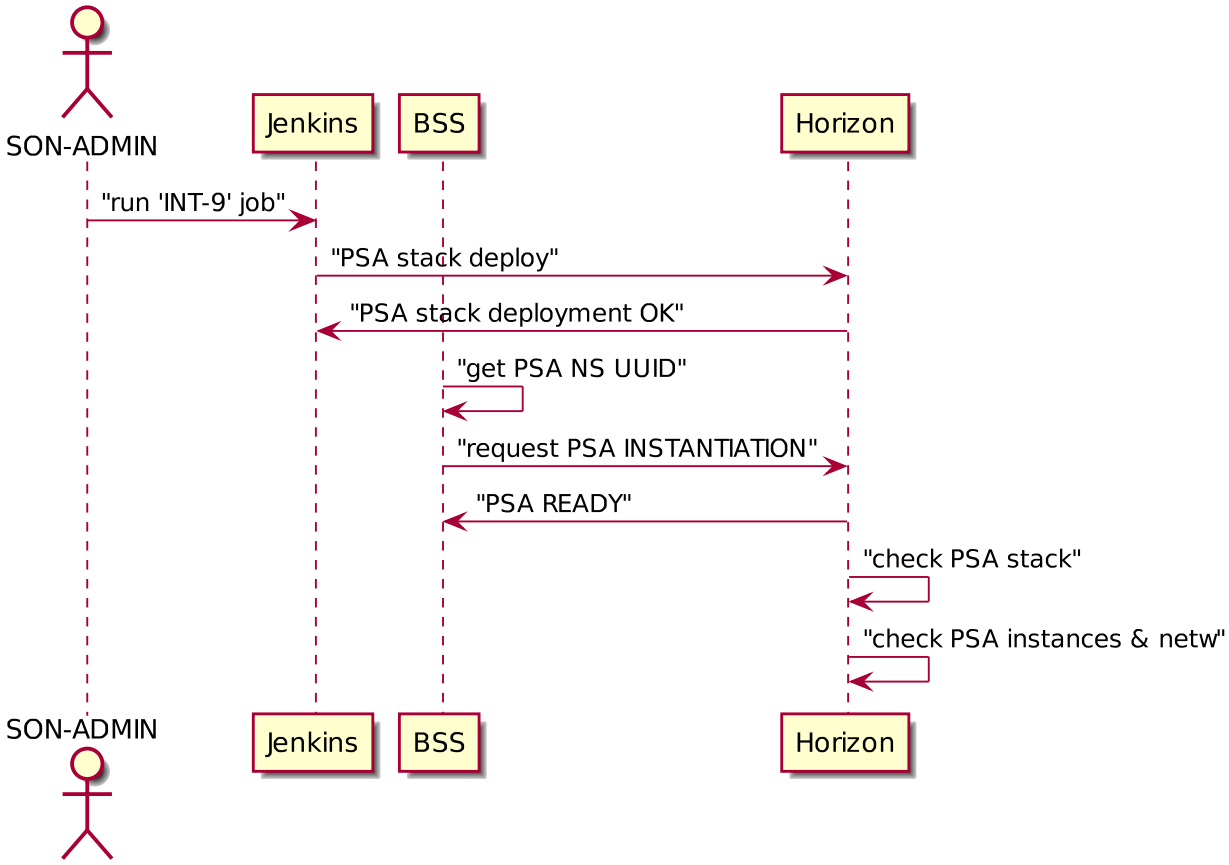

Figure 4.16: PSA NS INSTANTIATION process

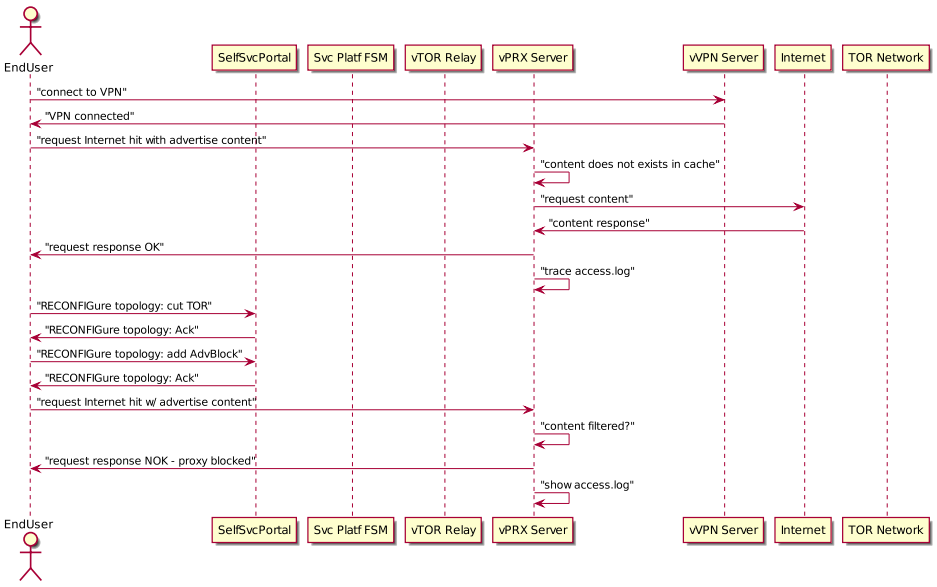

Figure 4.17: PSA features usage

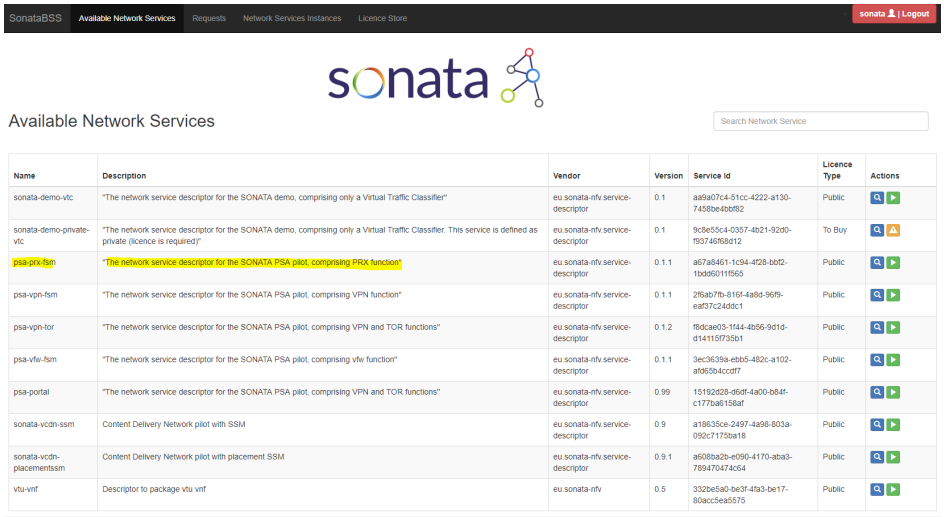

Figure 4.18: Deployment of the PSA service in SONATA's BSS GUI

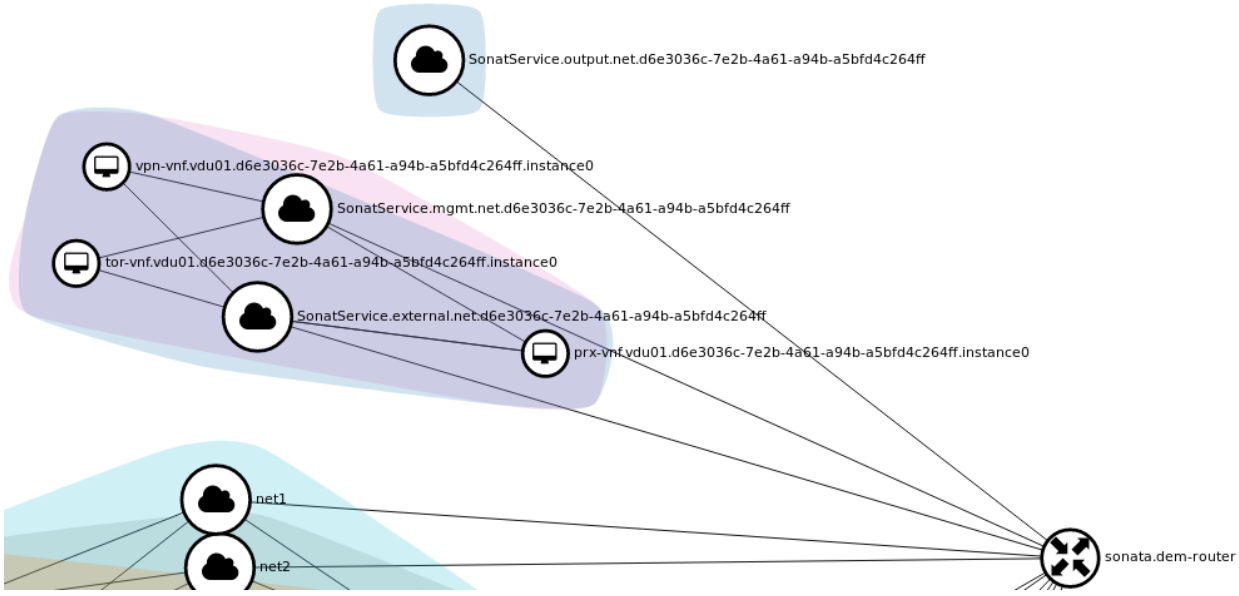

Figure 4.19: PSA network topology in Horizon

Topology | Overview | Resources | Events | Template

Stack Resource	Resource	Stack Resource Type
floating.vpn-vnf.vdu01.inout.48a3f264-22da-45b9-9312-ba0b51f128f9	a5477b50-657d-47cc-9333-e0cc2bb95038	OS::Neutron::FloatingIP
SonataService.output.net.48a3f264-22da-45b9-9312-ba0b51f128f9	3dec82fd-4b4b-4370-850d-a3e7ad1464dc	OS::Neutron::Net
SonataService.input.internal.48a3f264-22da-45b9-9312-ba0b51f128f9	e8cdd5c7-191f-4215-83f3-53ee1113db86:subnet_id=e91a7f79-1975-40ba-ba2e-58723d561a37	OS::Neutron::RouterInterface
vpn-vnf.vdu01.inout.48a3f264-22da-45b9-9312-ba0b51f128f9	6842bec6-2938-430d-a42b-7230a0d93fa2	OS::Neutron::Port
tor-vnf_48a3f264-22da-45b9-9312-ba0b51f128f9_spAddressCloudConfig	ed82ee22-348d-47f1-bcd2-fbe44dd4edaa	OS::Heat::CloudConfig
vpn-vnf.vdu01.eth0.48a3f264-22da-45b9-9312-ba0b51f128f9	0703f794-11ad-48e9-9429-e4915a549212	OS::Neutron::Port
SonataService.mgmt.internal.48a3f264-22da-45b9-9312-ba0b51f128f9	e8cdd5c7-191f-4215-83f3-53ee1113db86:subnet_id=57e24cb0-5c25-4bc4-bc82-faba22ed81d1	OS::Neutron::RouterInterface
SonataService.input.subnet.48a3f264-22da-45b9-9312-ba0b51f128f9	e91a7f79-1975-40ba-ba2e-58723d561a37	OS::Neutron::Subnet
SonataService.internal.subnet.48a3f264-22da-45b9-9312-ba0b51f128f9	986fa383-0d20-42d8-83b4-3d40c20a3b45	OS::Neutron::Subnet

Figure 4.20: PSA Heat stack

the two service offerings: "basic" and "premium". Figure 4.22 shows the self-service portal's GUI after selection and activation of the "premium" service chain.

Checking the correct functioning of the network service is done manually via a browser connected to the service. Following that connection to the service happens via the VPN server function, a VPN client is installed on the user's machine and connected to the VPN function. Initially, the only function in the service is this VPN server, allowing access to the Internet. Validation of this default service chain happens via using a Web browser to access a number of Web sites and verifying the result (connection succeeded or not). This step is shown in the Figure 4.23, where the VPN client applies locally the rules sent by the PSA VPN server.

Once this has been verified, chain reconfiguration is tested. In order to do that, the self-service portal's GUI is accessed and another service chain is selected and started. First, the basic service chain is selected. Reconfiguration includes the addition of the TOR service function. Proper inclusion of that function involves verifying that access to the Internet is still available. Then, the anonymization feature of the TOR function is tested by accessing a Web site which echoes back the IP address from which it is accessed (e.g., `http://www.whatismyip.com`). With a working TOR function, the IP address changes with every request.

The functions chaining is done at layer 3. So the chaining can be verified by inspecting each VNF's routing table. In Figure 4.24, the VPN VNF has a default route pointing to the TOR's IP as gateway for its next hop (`172.16.0.195`).

The TOR's IP can be cross-checked by inspecting the Horizon interface (Figure 4.25) where the `tor-vnf.vdu01.48a3f264...` has this IP attached to an external network.

As for the TOR VNF itself, the route is checked by verifying that the default route inserted by its FSM (`172.16.0.193`) is the gateway of the corresponding Neutron internal network. The route is shown in Figure 4.26.

Then the TOR function is verified by doing a curl to `http://www.whatismyip.com/`. The result is displayed in the Figure 4.27: the end-user has the IP `171.25.193.78`.

This IP `171.25.193.78` is listed as a TOR exit point as shown in Figure 4.28.

Next is the validation of the Proxy function. Again, the service chain is changed by selecting

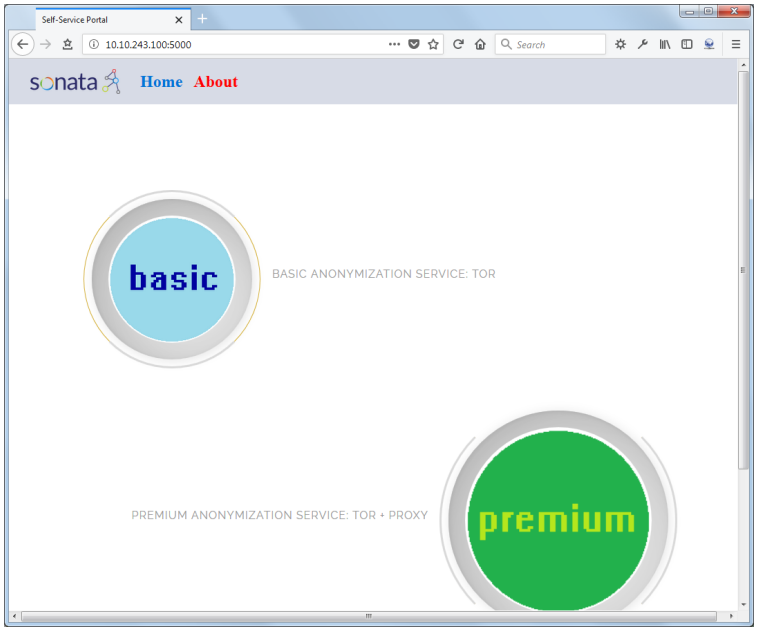

Figure 4.21: Self-Service GUI

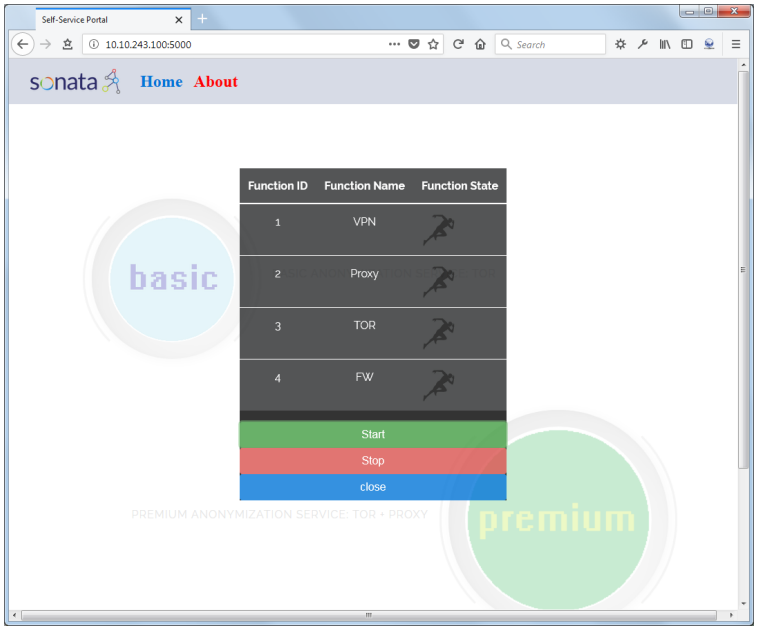

Figure 4.22: Self-Service GUI showing started premium service

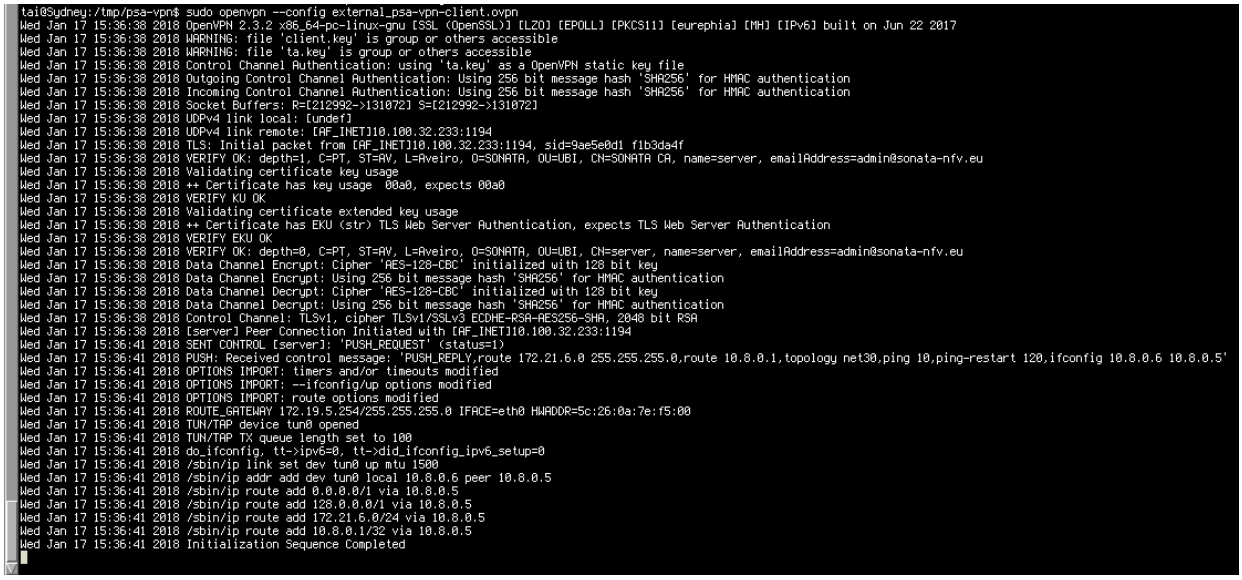

Figure 4.23: The end-user VPN client

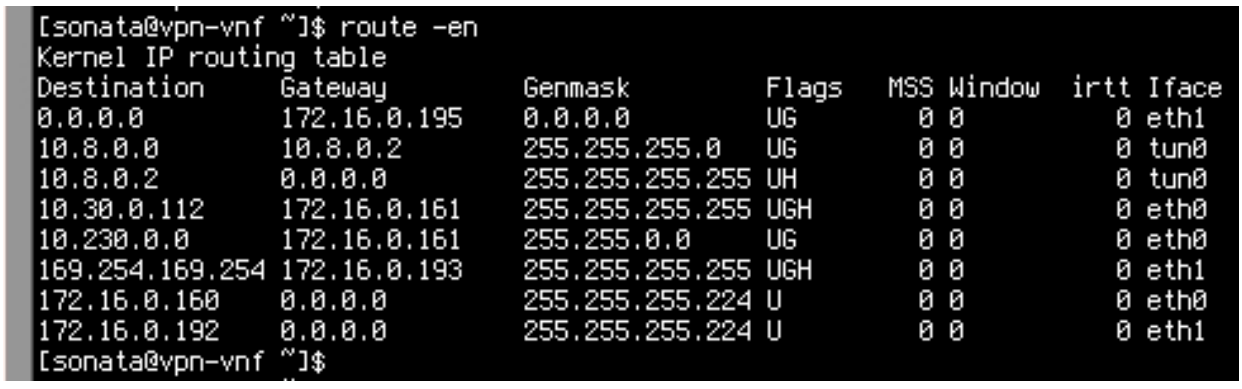

Figure 4.24: The VPN's route gateway

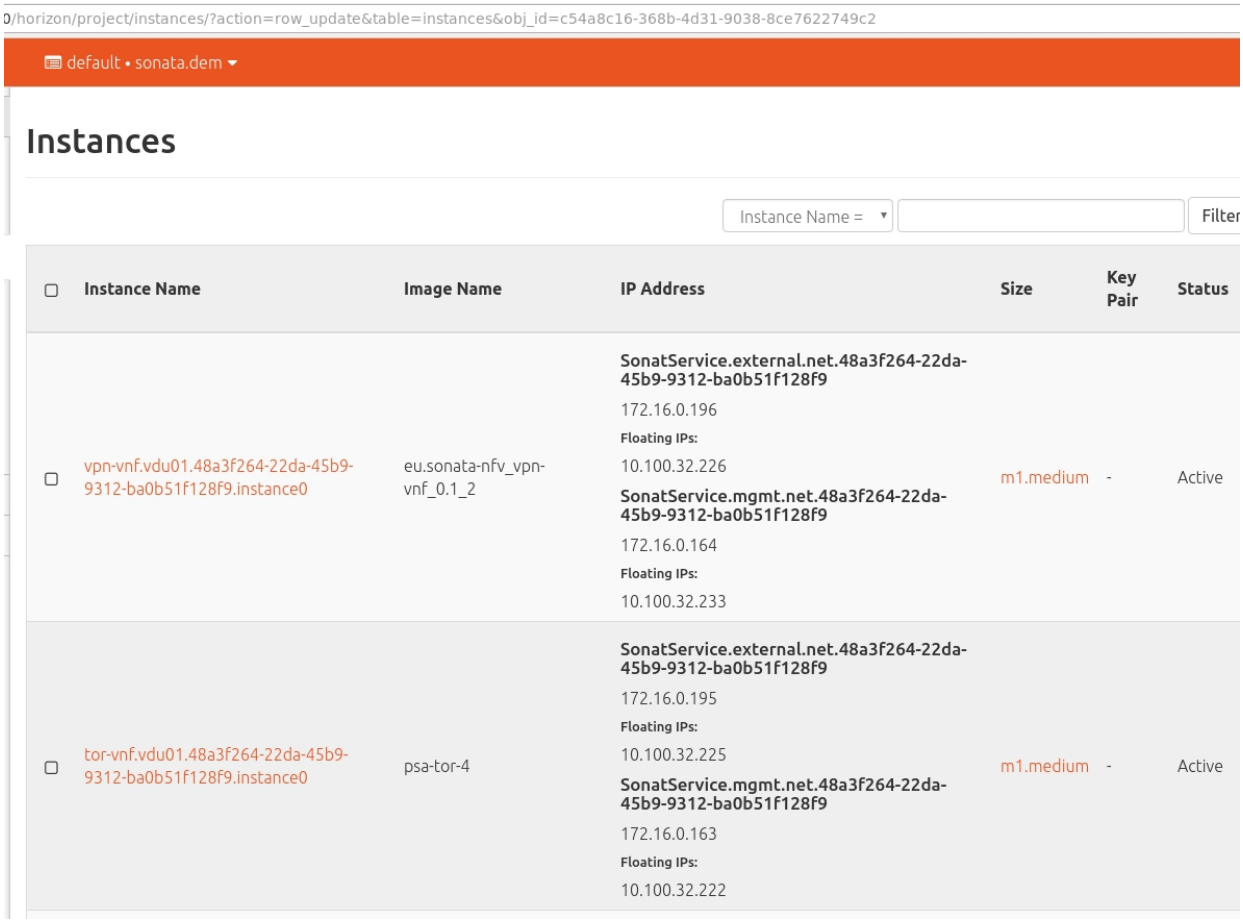

Figure 4.25: The PSA VDUs in Horizon

```
[sonata@tor-vnf ~]$ route -en
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         172.16.0.193    0.0.0.0         UG        0 0          0 eth1
10.30.0.112     172.16.0.161    255.255.255.255 UGH       0 0          0 eth0
10.230.0.0      172.16.0.161    255.255.0.0     UG        0 0          0 eth0
169.254.169.254 172.16.0.193    255.255.255.255 UGH       0 0          0 eth1
172.16.0.160    0.0.0.0         255.255.255.224 U         0 0          0 eth0
172.16.0.192    0.0.0.0         255.255.255.224 U         0 0          0 eth1
[sonata@tor-vnf ~]$
[sonata@tor-vnf ~]$
```

Figure 4.26: The TOR's route gateway

```
tai@Sydney:~$ curl -s http://www.whatismyip.com/ | grep -i 'Your IP'
    <span class="cf-footer-item"><span data-translate="your_ip">Your IP<
/span>: 171.25.193.78</span>
tai@Sydney:~$
```

Figure 4.27: A curl to www.whatismyip.com

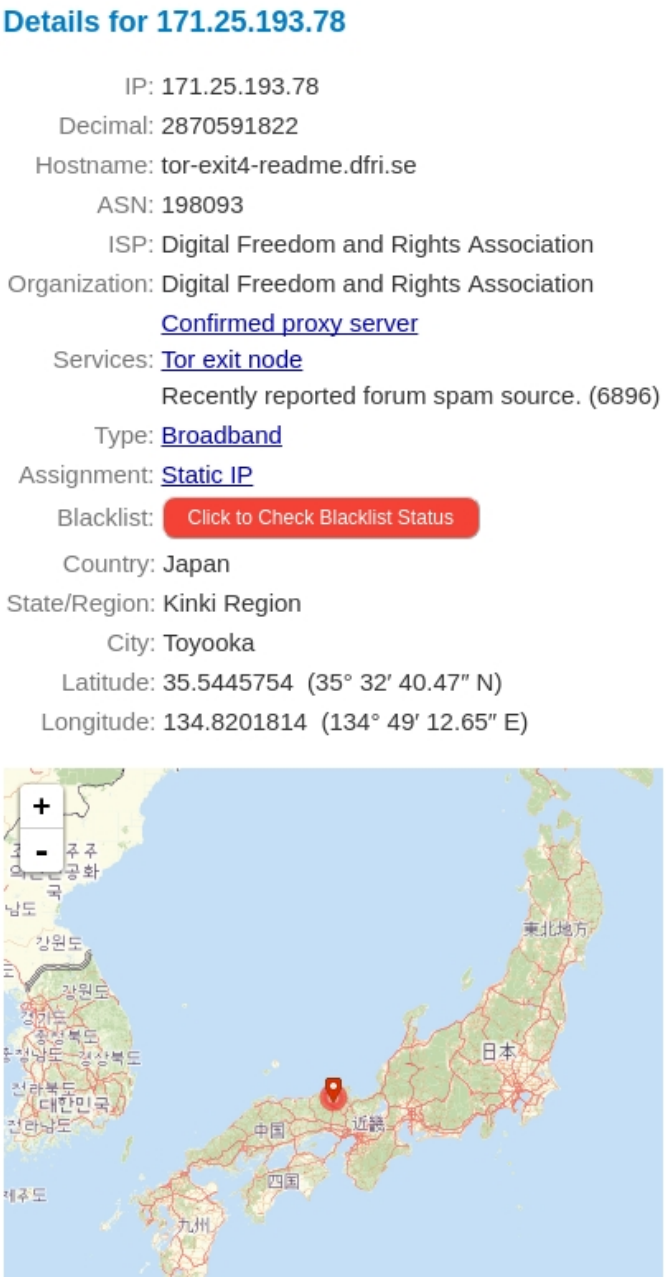

Figure 4.28: Where is 171.25.193.78

a different service in the self-service portal. This time, it is the premium service. The browser is now pointed to a site which is blocked by the configuration of the proxy, e.g. `https://www.sonata-nfv.eu` and the result is checked (access denied).

Finally, the premium service is stopped, reverting the service chain back to the default one (VPN only). This reconfiguration is verified by checking that `https://www.sonata-nfv.eu` can again be accessed and the IP address does not change with every request.

4.2.9 VPN,TOR,Firewall NS validation

The chain containing the previously tested VPN,TOR service and the firewall has been deployed and validated on the service plaform.

Once the NS onboarded, the 3 services are running into the Openstack platform. Then we connect to the VPN and make WEB browsing from another computer. Here is one of the request:

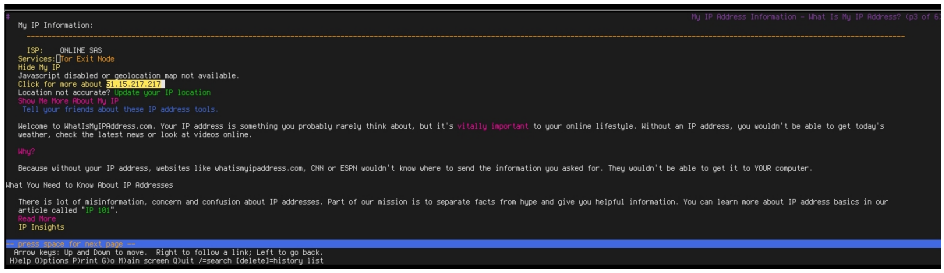

Figure 4.29: Pfsense_Web_Browsing.jpg

We can see that our IP displayed is 51.15.217.217. That is a TOR IP.

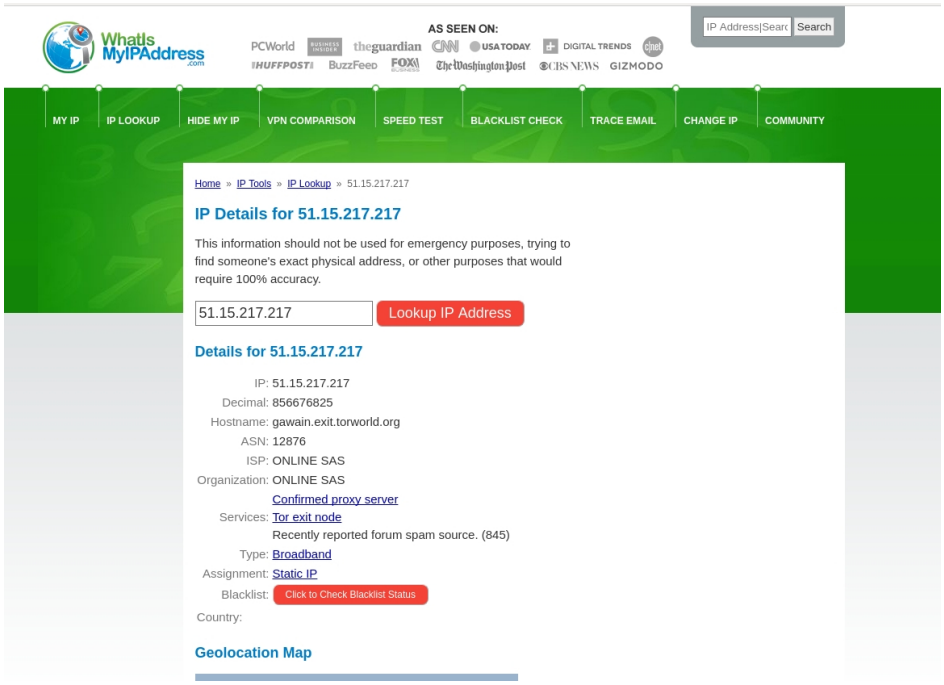

Figure 4.30: Pfsense_Web_IP.jpg

We can now check that the traffic is passing through the firewall. Here is the dashboard accessible via the management network.

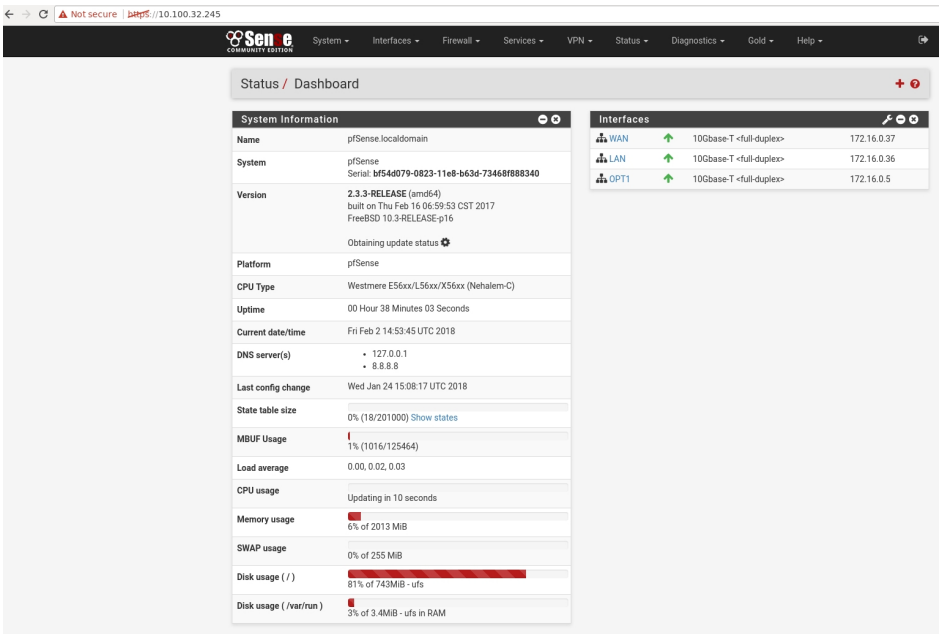

Figure 4.31: Pfsense_dashboard.jpg

We can see in the monitored connection the TOR VPN (the first two lines) with the first TOR router (78.142.19.11).

This IP, 78.142.19.11 is effectively a TOR router.

We can monitor the traffic and see the traffic going in and out.

And on other pages, we can see the blocked traffic. Here is a example of nmap to confirm that the not allowed traffic is effectively blocked:

4.2.10 PSA innovations

The PSA pilot showcases a number of innovations enabled by the SONATA platform. These are capabilities that go beyond the basic functionality of any service platform, which include on-boarding, deployment, and instantiation of services.

The first innovation is dynamic reconfiguration of the active service chain. In the PSA pilot, the VNFs which are participating in a service chain can be changed at run-time. Specifically, depending on the service chain selection, the URL-filtering proxy and/or the anonymity feature from TOR are added on-demand. This feature goes way beyond the traditional implementation of removing the active service chain and starting an updated one. Service chain switch-over times are reduced and down-times are minimized. Changes on the end-user side are not needed. The reconfiguration is therefore transparent to the end-user.

Together with the dynamic reconfiguration, automation of that reconfiguration is realized as another innovation. The trigger for reconfiguration originates with the end-user; reconfiguration then happens fully automatically, under the control of the SSM/FSM system. The automation is introduced by the developer of the service, who knows the necessary reconfigurations best and is therefore in the best position to provide such service-specific management capabilities. Compared to traditional service management, no manual intervention is needed to select and enforce service chain reconfiguration. The time to service is thereby reduced drastically.

As the service chain reconfiguration is initiated by the end-user, effectively, a limited (and very precisely confined) amount of control is transferred to the end-user, via the use of the self-service

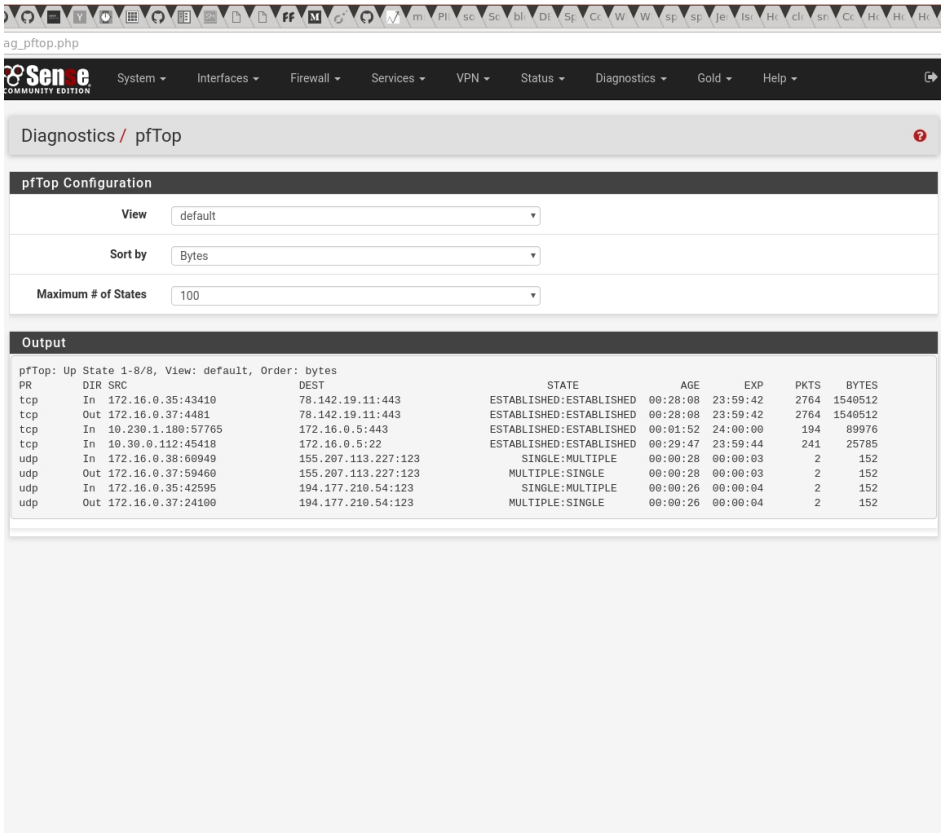

Figure 4.32: Pfsense_Monitoring.jpg

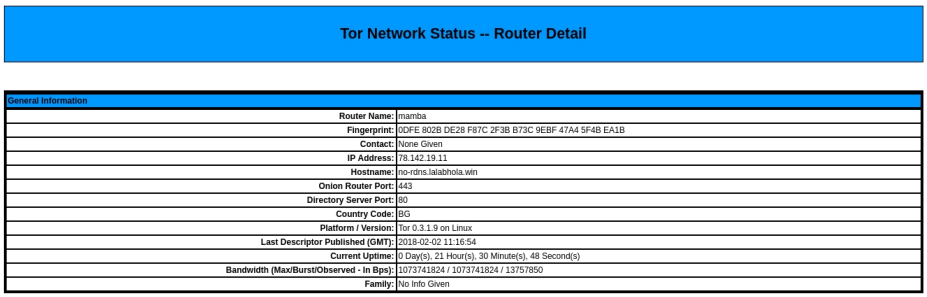

Figure 4.33: Pfsense_TOR_server.jpg

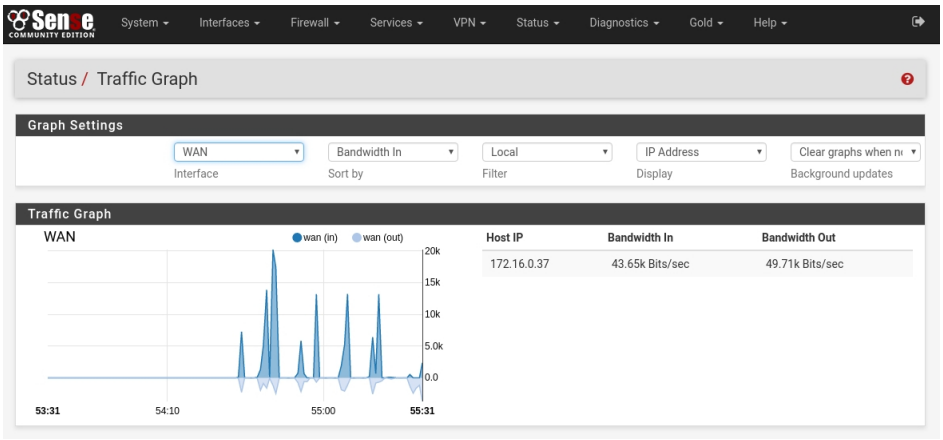

Figure 4.34: Pfsense_Traffic.jpg

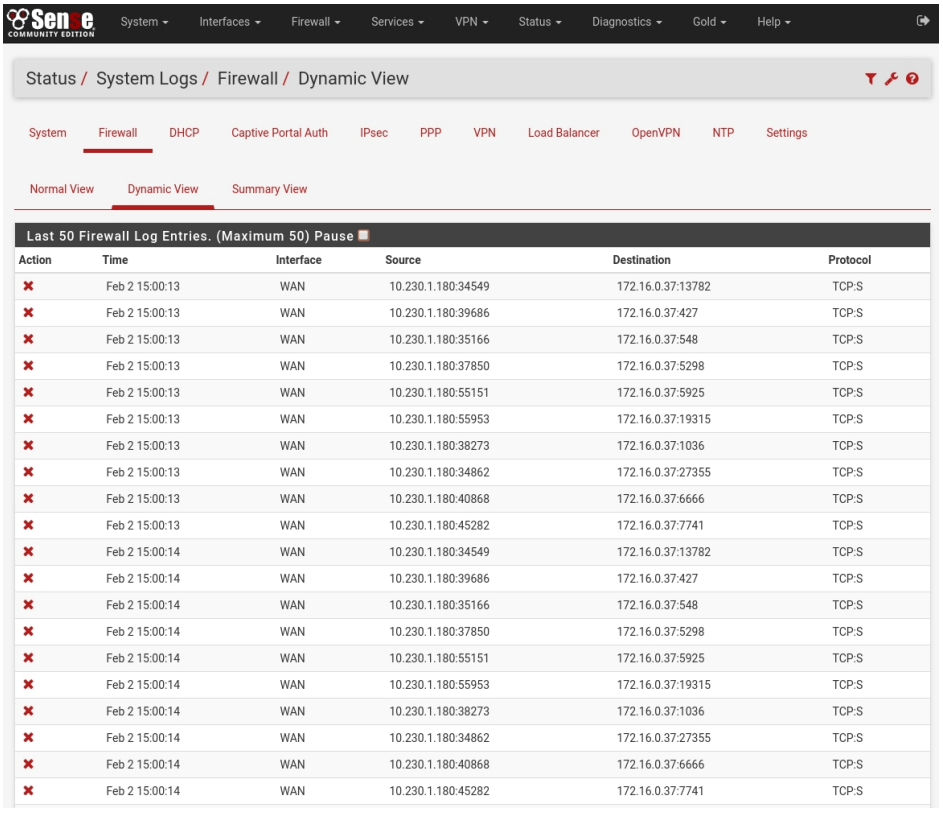

Figure 4.35: Pfsense_Firewall.jpg

portal and the MANO extensibility scheme (SSM/FSM). This is another innovation shown in the PSA pilot.

The control by the end-user is effected by the self-service portal. This GUI shows a fourth innovation realized by the SONATA platform, the capability to extend the interaction modes with the MANO system. Instead of the traditional way of only allowing staff of the operator to interact with the MANO system, SONATA allows this system to be extended in specific ways. Here, it allows the addition of an end-user-facing GUI for service management.

The combination of the above mentioned capabilities gives rise to another innovation: the provisioning of user-specific service chains. The chain can be selected for each individual user and every user can choose the chain that satisfies its needs most. Compared to traditional provisioning models, which provide the same service chain to every user who selected a particular service, the PSA pilot shows how individual users can reconfigure the chain to their taste without affecting other users which might have selected a different service chain.

4.3 Multi-Orchestrator and slicing for HSP

4.3.1 Multi-orchestrator Architecture and components

4.3.1.1 Background and motivation

In deliverable D6.2 [12] we described the motivation and the general context for the HSP pilot, but for sake of readability we include here a brief outline. This section is applies to the Hierarchical Service Providers pilot as background and motivation, as that pilot also deals with similar issue but is mostly focused around the support of recursiveness of the same SP rather than the interoperability between different orchestrators. In the Hierarchical Service Providers (HSP), we consider the scenario of a company composed of two local offices scattered across Europe, or divided into different business units, such as a head office and a telecom service provision offices for mobile services, streaming services, etc. In this scenario, one of the business units wants to deploy network services composed of some VNFs (Virtual Network Functions), but the structure of this network service is complex and needs to leverage resources or has specific requirements that a local office cannot offer on its own dedicated NFVI (Network Function Virtualisation Infrastructure). Nonetheless, the local office is able to offer to its users a MANO and a Service Platform to operate and orchestrate their service instances, which is able to meet their day-to-day requirements. For this reason, the local office MANO has to interact with other partitions of the company that can meet these complex requirements. The service deployment requests need to be sent to the head office or to other local offices, which may have a dedicated NFVI, so that the request can be completed. The NFVI which is able to meet the requirements of the local office is orchestrated by a relevant MANO operated by company. The communication between different MANOs could be realised by considering a north-south interface between them, where each north-bound MANO interacts with its south-bound counterpart, as it would do with a VIM. In SONATA, this interaction is facilitated and mediated by the SONATA abstraction layer [2] that allows exposing the MANO capabilities in a compliant way with a VIM (Virtual Infrastructure Manager) interface. By means of this MANO to MANO abstraction, this scenario can even be extended and generalised by considering a generic tree model where each orchestrator can leverage other orchestrators to access resources it cannot directly orchestrate, or to leverage software and services provided by other segments of the company. As such one SP provides a Network Service (NS) to the other SP. We term these the lower-SP and upper-SP. Essentially, as far as the lower-SP is concerned, the upper-SP is just another customer requesting service; similarly, from the upper-SP's perspective, the lower-SP is providing a component in their overall network service in a similar manner to the NFVI. The interface between

the two SPs is essentially identical to the interface between a SP and a customer (which Sonata has already defined). This approach allows the details of each SP to be hidden from the other. Hence the lower-SP could alter the way it implemented the service (as long as the service presented at the interface is maintained unchanged), and similarly the upper-SP could shift to using a different lower-SP. As an Orchestrator of an SP already has a mechanism to interact with Infrastructure Managers - namely, the wrappers for VIMs and WIMs - we can reuse this functionality and create a wrapper for another Orchestrator. Note that each SP does its own MANO functionality, including lifecycle management.

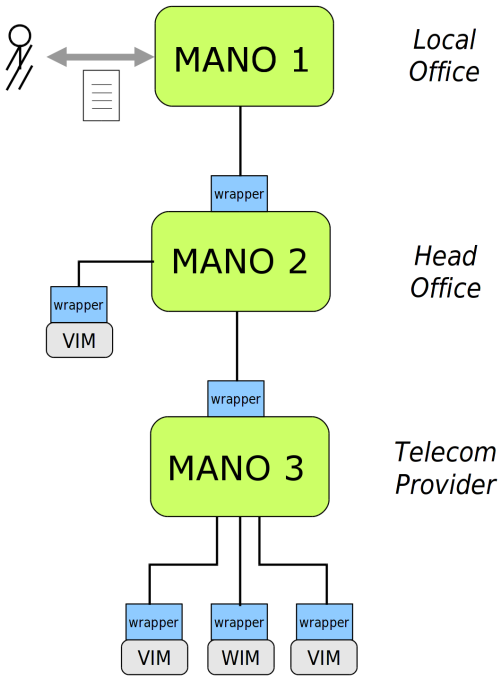

Figure 4.36: Simple SP stack

We don't want the architecture to be limited to just an upper-SP and lower-SP – we want to be able to have a stack of SPs. This allows us to combine or extend the scenarios above (imagine the operator above (#1) is bought by another operator, which wants to resell operator #1's NSs). Such a recursive approach simplifies operational management and the creation of new services. The most general case of this recursive architecture is a tree. Although this may not appear in many real deployments, it is beneficial to have such a flexible and generic model, rather than a specially defined but limited lower-SP / upper-SP model.

The use case was mentioned in D3.1[4] and D6.1[4], and recursive architectures were discussed in D2.2[2] and D2.3[3]. This is not an East-West multi-domain peer-to-peer interaction, rather it is a North - South interface. We already have a mechanism which is a North - South interface - the wrapper to a VIM or a WIM, and as stated for symmetry we can extend the North - South interface to have a wrapper for an Orchestrator. We have to consider how does an Orchestrator informs another Orchestrator what VNFs and NS are available. In this recursive architecture, there needs to be a Capability Exposure mechanism upwards, i.e. an Orchestrator can expose a set of VIMs and a set of WIMs upwards, and this is essentially the existing catalogue/registry.

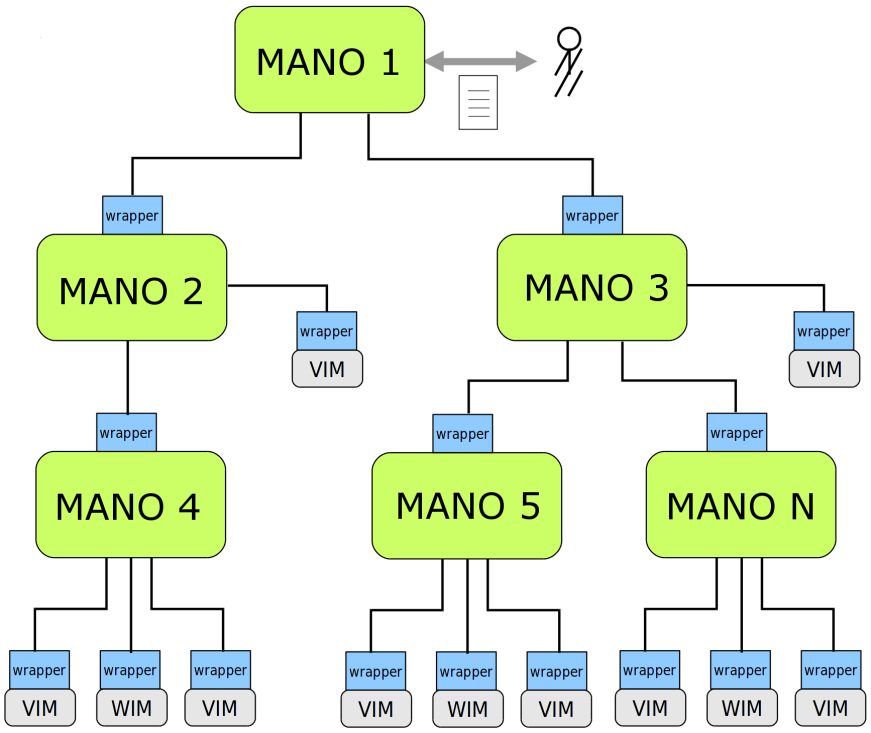

Figure 4.37: Complex SP stack

4.3.1.2 Objectives

Within the context of the Hierarchical Service Provider Pilot we aim at meeting and validating the following objectives:

- Demonstrate **service instantiation across multiple partitions** of an administrative domain (In line with Project Objective 1.1)
- Demonstrate **interface capabilities** of SONATA Service platform toward orchestration system of higher level of abstraction (other MANO/multi-domain orchestrator) (Project objective 4.2)
- Demonstrate **multi-VIM** support with a new lightweight VIM Adapter and Slice Manager interaction (Project objective 2.1)
- Validate the possibilities of the SONATA data model in terms of **recursiveness** of infrastructure, resource, services and functions
- Demonstrate the interface capabilities of SONATA with a slice-able NFV infrastructure and the possibilities to use its Infrastructure Abstraction to drive creation of slices

4.3.1.3 Pilot assumptions and context

We recapitulate here the assumption already listed in [13]:

- Two levels of SP – the recursive architecture allows an arbitrary number of levels of the hierarchy of SPs. In this pilot, we implement a scenario with two level in the hierarchy.

- The NS provided to the customer consists of two VNFs chained together. The upper-SP provides one VNF itself, and the lower-SP provides one VNF and the NFVI. (It would be simpler if the upper-SP devolved the entire service provision to the lower-SP (like a virtual operator), but our choice allows us to explore more challenging issues.)
- The SPs identify services in the same way (in fact as a triple of: vendor name, service name and version number). In a real-world situation, this corresponds to the assumption that the SPs share a catalogue of services (perhaps it is global). (In situations where this is not possible, then the best approach is probably for the lower-SP to publish its catalogue, most likely with a publish-subscribe model, so upper-SPs are aware of changes in the lower-SP's catalogue; the upper-SP would also be responsible for putting a request in the format so that the lower-SP can understand it.)
- The SP API is a request/response for service. This is also likely in the real world, as it is simple and fits with the OpenStack approach. One could imagine more complicated models, where for an NS requiring several elements, several tentative requests are made and, if they're all positive, then a firm request is sent. In the simple approach, if one request fails then roll-back (i.e. delete) requests must be sent to the other elements.
- The requests are sent at the same time. It is possible that, for speed reasons, upper-SPs would make the request to lower-SPs and then to its own NFVI. This is a second-order issue that we don't plan to explore in the pilot.
- Resources are always granted. This is for simplicity. In the real world, each SP tracks its own resources and would check that it has sufficient to meet the request. A solution for NFVI resource tracking and representation in a hierarchy of MANO is nevertheless presented as a validation flow for this pilot.

4.3.1.4 Virtual Network Functions for HSP pilot

The focus of this pilot is not really on specific network functions, but on the overall inter-working between multiple MANOs. Therefore we consider simple services composed of multiple VNFs. In this scenario, a VNF developer wants make available a set of VNFs he developed to the customer of the upper-SP, and for this reason, he wraps it into an NSD containing only those VNFs. A service developer of the same organisation, but belonging to another business unit which uses the lower-SP, wants to include those VNFs in his network service and wants his service to span also over the infrastructure served by the lower-SP.

4.3.1.5 New Components

To facilitate the interaction that will be described in the following sections, a subset of the software components of SONATA had to be extended with extra functionality, plus some new modules has been produced on the infrastructural level. In what follows is an extensive list of the extended modules and new modules developed for this pilot:

- SONATA Infrastructure Abstraction
  - **Compute Wrapper module:** for interfacing with VLSP and offering VNF deployment capabilities
  - **Networking Wrapper module:** for interfacing with VLSP and offering networking capabilities

- **WIM Wrapper module:** for offering inter-PoP networking configuration capabilities.
- **SONATA SP Wrapper:** to allow a SONATA platform to interface with the SONATA Gatekeeper of another SONATA platform

- **SONATA Domain Adaptor:** to allow 5GEx resource orchestrator to interface to the SONATA Gatekeeper

4.3.2 Demonstration scenario

This pilot has been designed to assess the feasibility of considering one of the service platform mentioned in the previous sections with MANO of a different technology, so to assess the issue arising from such a scenario.

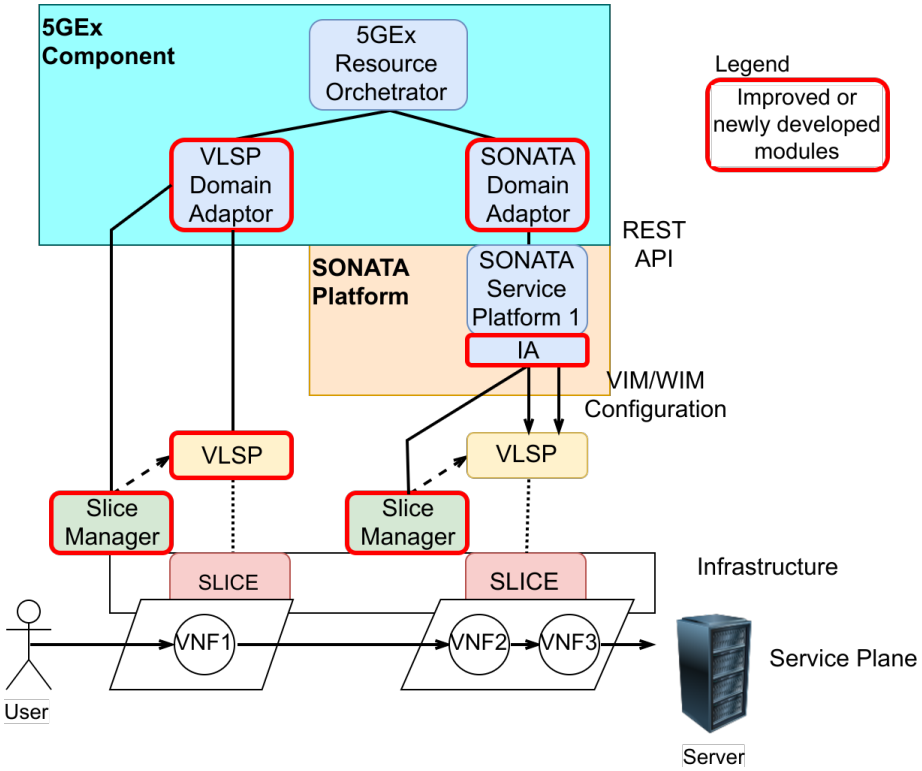

Figure 4.38: Scenario interaction

As an example of proof of concept of this scenario we designed and developed a demonstrator that provides a 5G PPP cross-projects interaction. The Figure 4.38 depicts a high-level view of the interaction in this scenario, that is described in what follows. Two Service Platforms co-operate for rapid and dynamic service provisioning in an NFV environment. The company has segmented its NFVI in order to meet the demands of separate organisations/departments/business units. Therefore, the company has deployed a hierarchy of service platforms that collaborate in order to deploy NFV-based end-to-end services across the network. A generic SP/orchestrator can leverage on a segment of NFVI or on other SPs to instantiate functions and services. We selected the 5GEx SP [1] as the orchestrator at the higher level, as it was designed and built to be a multi-domain orchestrator. It operates over a lightweight VIM domain (VLSP) [23] with slicing capabilities and over a SONATA SP which was designed and built to be a single domain orchestrator, through a newly developed adaptation layer ("SONATA domain adapter"). A REST

interface between the 5GEx domain adapter and the remote SONATA Gatekeeper facilitates the communication. The SONATA SP operates on top of another partition of the NFVI, again using the slicing capabilities and the VLSP VIM to manage it. The SONATA Infrastructure Adapter drives the creation of slices with an on-demand VIM interacting with the Slice Manager and offers to the MANO the interface to interact with the VIM. The End-to-end service and its composing VNFs are on-boarded as services in SONATA Service Platform and exposed as Domain Capabilities to the 5GEx orchestrator through the "SONATA domain adapter" together with the available abstracted resource view.

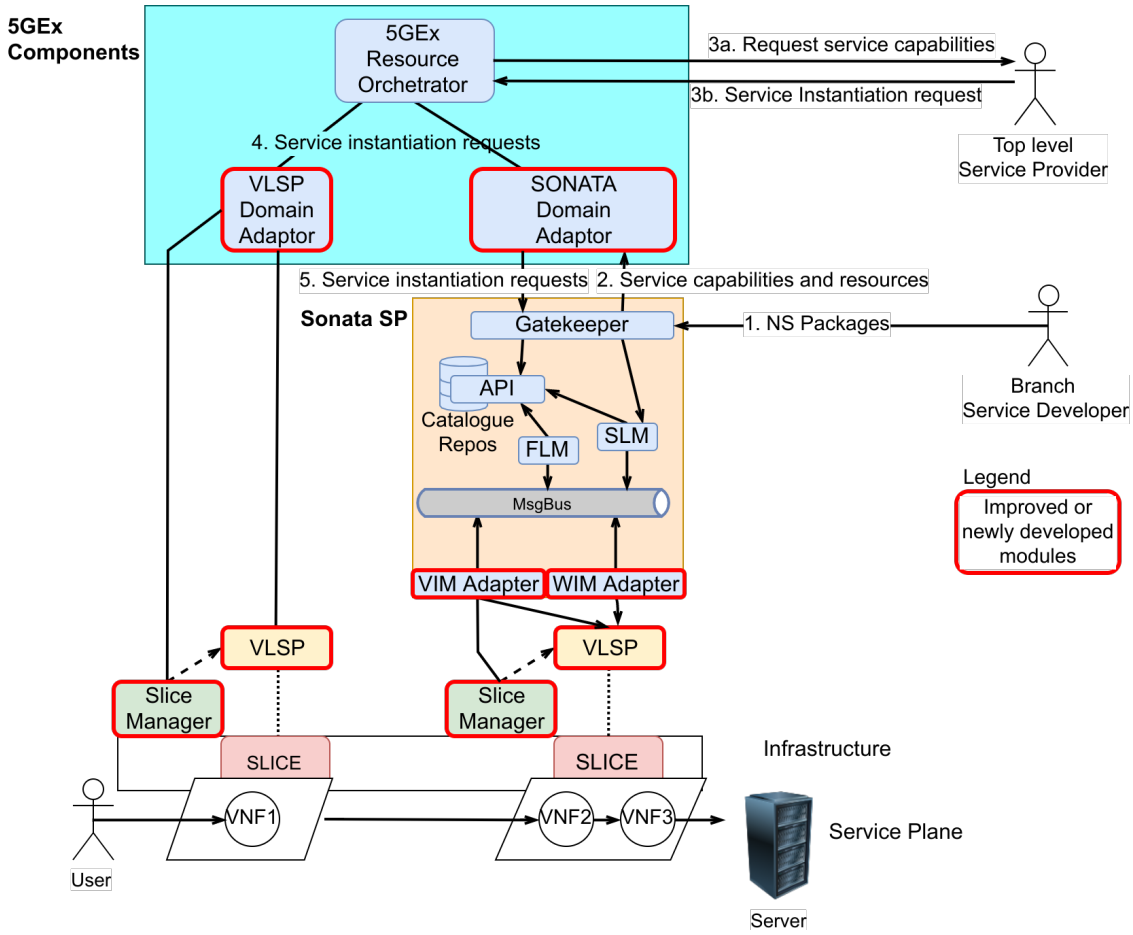

Figure 4.39: Interaction between SONATA and 5GEx

The Figure 4.39 details the interaction between the involved components of SONATA and 5GEx. A service developer in one of the branches of the company develops a service composed of one or more VNFs to be used on the SONATA platform of his division. Therefore it packages it using the SONATA SDK and onboard it in the SONATA SP (Step 1 in Figure Figure 4.39). In the 5GEx domain, the SONATA adaptor is used to retrieve available service and functions from the SONATA catalogue through the Gatekeeper API (Step 2 in the Figure 4.39) A Service provider on another division of the company (or in the top level) leverages 5GEx orchestrator ESCAPE to retrieve the available functions and services across the connected domains, and embeds one of the services available in the SONATA domain in its overall service requests (step 3a in the Figure 4.39). At this point a request is issued by the service provider to ESCAPE (step 3b in the Figure 4.39), that can use its domain adapters to split the service according to the exposed capabilities and

resource availability, and finally send the service instantiation request (step 4 in the Figure 4.39). The SONATA platform will receive a service instantiation request through the relevant Gatekeeper API (step 5 in the Figure 4.39). The normal service instantiation flow will follow the request, instantiating the entities as described in the package on-boarded by the service developer in Step 1. WAN configuration between the two NFVI PoP could be achieved by using the NAP feature of the Gatekeeper API to attach the entry-point of the service deployed in the SONATA domain to the other branches of the overall end-to-end service, or it can be left to other subsystems of the company. In our scenario, the functionality of the VLSP VIM allows for this configuration to be done at the deployment time by using the aforementioned NAP and a specific WIM adaptor. Overall, the richness and flexibility of both the SONATA APIs and the 5GEx APIs have been clearly demonstrated in the presented proofs of concepts.

4.3.2.1 Multiple SONATA platforms

A second scenario is represented by a generalization of the one depicted in the previous section. In this generalisation, we extend the number of SONATA platform interconnected to the 5GEx resource orchestrator. The Figure 4.40 depicts the scenario.

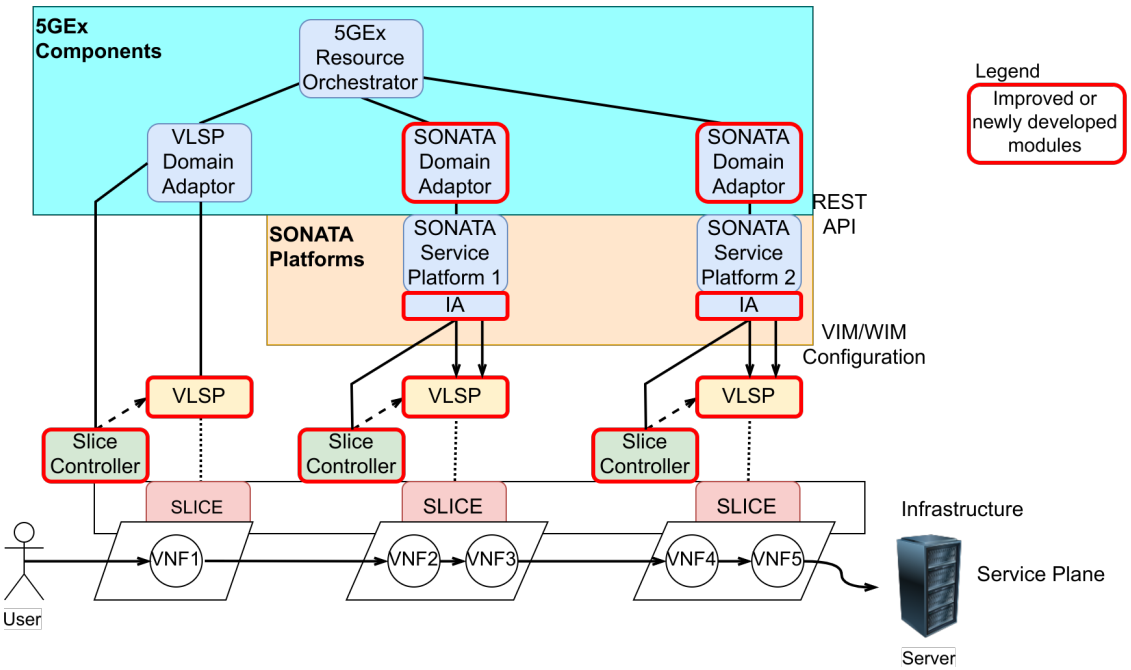

Figure 4.40: Interconnection between SONATA and 5GEx

This scenario follows a very similar flow with respect to the one described in the previous section. The VLSP VIM and WIM configuration models have been extended with respect to the previous scenario in order to facilitate the enforcement of forwarding rules across the virtual topology, as well as inside the VLSP PoP, in a consistent way, independently of the number of SONATA platforms connected to the 5GEx resource orchestrator.

4.3.3 Pilot validation flow

As the final part of this section, we report an extensive list of the several steps involved in the validation process of the afore mentioned scenarios. We focus on the first scenario presented in this

deliverable, with one 5GEx platform operating on a sliceable NFVI-PoP, offering VLSP VIM on demand, and on top of a SONATA service platform, which in turn operates on another sliceable NFVI-PoP, offering VLSP VIM on demand. The testbed configuration for the demonstration flow is shown in the Figure 4.41.

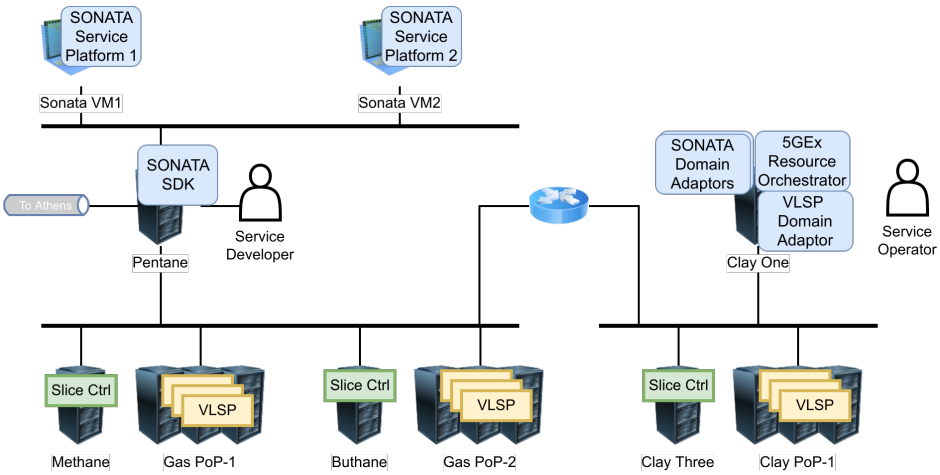

Figure 4.41: Testbed configuration

4.3.3.1 Testbed installation

- Site-1
  - A virtual machine is created to host the SONATA SP
  - Slice controller is running on a host in the site and one or more hosts are allocated and configured for slice creation
  - A VLSP Visualiser is installed and running for inspection and validation. This software artefact allows the operator to visualise the resulting topology of a virtual service deployment.
  - The VM downloads son-install code base and dependencies (ansible, docker) or the SONATA stand-alone VM is used
  - **son-install** CMUD is used to install the SONATA SP in the VM
  - Users are created and configured for *admin*, *service_developer*, *service_operator* and *service_platform*
- Site-2
  - A virtual machine is created to host the needed components of the 5Gex environment
  - Slice controller is running on a host in the site and one or more hosts are allocated and configured for slice creation
  - VLSP Visualiser is installed and running for inspection and validation. This software artefact allows the operator to visualise the resulting topology of a virtual service deployment.

4.3.3.2 Test-bed configuration

- Site-1
  - The admin user configures a VLSP WIM, providing the Slice controller endpoint and configuration, through the relevant GUI page
  - The admin user attaches the VLSP VIM, providing the Slice controller endpoint and configuration, through the relevant GUI page, attaching them to the VLSP WIM
- Site-2
  - A 5GEx VLSP Domain Adapter is instantiated in the 5GEx virtual machine, and it is triggered to collect resource availability and available functions from the assigned NFVI-PoP. This triggers the relevant Slice controller to create a slice on the assigned infrastructure and instantiate a VLSP VIM to manage it.

4.3.3.3 NSD and VNFDs on-boarding

- Site-1
  - A package is created containing an NSD and the VNFDs for the VNF developed by the SONATA service_developer using the SONATA SDK

The package is on-boarded in using the SONATA Gatekeeper API

4.3.3.4 Service instantiation

Step 0 - *Initial Condition*

- The NSD is available in the SONATA SP catalogue. It references two VNFDs.
- SONATA SP is configured to operate on the Slice controller running in Site-1 for what concerns both VIM and WIM abstraction.

Step 1 – *SONATA capabilities request*

- The 5GEx SONATA Domain Adaptor is instantiated in the 5GEx virtual machine. It is triggered to use the */vims*, */wims* and */services* API offered by the Gatekeeper to retrieve both resource availability, and the so called capabilities, which are extracted from the */services* and interpreted as available VNFs in the SONATA domain platform.
  - Triggering the */vims* API makes the SONATA Gatekeeper retrieves from the SONATA Infrastructure Abstraction the information on the available NFVI-PoP and on their resource availability.
  - Inside the SONATA Infrastructure Abstraction, the sliceable VLSP Wrapper issues a request to the Slice controller for a new slice to be allocated and receives back VIM configuration parameters for the allocated VIM.
  - The resource availability of such a slice are sent back to the Gatekeeper and to the 5GEx SONATA Domain Adaptor.

Step 2 - *Deployment Orchestration*

- The 5GEx Resource orchestration is triggered in the 5GEx Virtual Machine through a REST call to instantiate a service, based on VNFs and resources available in the VLSP Domain and in the SONATA Domain
- The service chain is split in the 5GEx resource orchestrator according to the VNFs offered by the VLSP and the SONATA domain. Also, the WIM configuration information retrieved from by the SONATA Domain Adapter are used to logically interconnect the section of the service to be deployed in the VLSP domain, and the one to be deployed in the SONATA domain, through the so called Service Attachment Point (SAP)
- The two domain adapters issue two instantiation requests for their relevant domain.

Step 3 – *SONATA deployment*

- The SONATA Gatekeeper receives an instantiation request for the NSD, also specifying an ingress NAP and an egress NAP for WIM configuration. These nap will be used to steer traffic through the chain from other VLSP domain(s) and toward other service endpoints.
- The Gatekeeper interact with the SLM to start a new Service Instantiation
- The SONATA Service Lifecycle Manager (SLM) and Function Lifecycle Manager (FLM) trigger the Infrastructure Abstraction to deploy the two VNFs which compose the NSD using the usual infrastructure.service.prepare call and one infrastructure.function.deploy call
- The SONATA Infrastructure Abstraction uses the VLSP Wrapper to instantiate VLSP virtual routers and virtual links according to the VNF structure described in the VNFDs using the VLSP VIM previously allocated for the slice.
- The SONATA SLM sends a infrastructure.service.chain.configure call to the infrastructure abstraction which completes the service graphs in VLSP with inter-VNF virtual links.
- In parallel, the 5GEx VLSP Domain adapter follows its internal logic for deploying a third VNF on the VLSP domain directly orchestrated by 5GEx

Step 4 – *SONATA WAN configuration*

- The SONATA SLM sends an infrastructure.service.wan.configure call to the infrastructure abstraction. The infrastructure abstraction will use the VLSP WIM wrapper to configure VLSP ingress and egress interfaces by which traffic will be received, injected in the VLSP network and ejected toward the next PoP.
- In parallel, the 5GEx VLSP Domain adapter follows its internal logic to provide a similar configuration also on the NFVI-PoP directly orchestrated by 5GEx.

Step 5 – *User validation of service deployment*

- After the service deployment process is completed an both domains, traffic from server to user is steered through the two NFVI-PoP using the concatenation of ingress and egress interfaces of each VLSP domain.
- VLSP Visualisers are used to inspect the resulting topology, as well as to inspect monitoring data about the volume of traffic going through the virtual topology.

4.3.4 Innovation

The HSP Pilot is an exploratory work on the very wide subject of MANO platforms inter-working, a subject that is going to be very relevant in the close future of software networks, with so many competing service platforms and MANO implementations already available or coming soon. In this pilot, we scratched the surface of this problem by providing a first solution for an inter-MANO communication interface based on the SONATA platform, with the rich and flexible API provided by its **Gatekeeper**, the configurable and extensible southbound interface provided by the **Infrastructure Abstraction**, and the relevant Service Model. In our demonstration, a new VIM and WIM wrappers for VLSP VIM have been developed, also demonstrating how SONATA can be adapted to an implementation of an NFV Infrastructure with Slicing capabilities. As a final remark, it is worth noting that the 5Gex-to-SONATA demonstration flow marks the 1st demonstrated interaction between two MANO/Service platforms, and also inter-working of two 5G-PPP projects.

4.4 Hierarchical Service Providers pilot on recursiveness

This section describes the Hierarchical Service Providers (HSP) pilot scenario used to demonstrate the recursiveness capability of the SONATA Service Platform.

This Pilot extends our Virtual CDN (vCDN) Pilot to show the Hierarchical Service capabilities that SONATA offers when using two different Service Platforms. More information about HSP Pilot can be found in its own section in deliverable D6.2 [12].

The main HSP scenario will distribute or split the original vCDN Network Service (NS) in two NSs to demonstrate recursiveness and the resource allocation. When the vCDN NS is deployed in the first SP (SP1) as composition of three VNFs, two VNFs will have the infrastructure resources allocated in SP1 while a third VNF will use the second SP's infrastructure resources to be deployed as a second NS, instead of using the first SP's infrastructure resources. This second NS is basically composed by the third VNF. This way, the first NS input will be allocated in the first SP and its output allocated in the second SP.

As Figure 4.42 shows, the Virtual CDN network service will be deployed using two different SONATA Service Platforms interconnected through the new Infrastructure Abstraction Service Platform Wrapper connector. With this configuration, the virtual network functions that compose the overall service will be distributed in the following way:

- Virtual Cache (vCC): deployed in the higher Service Platform (SP1) and using infrastructure resources from SP1.
- Virtual Traffic Classifier (vTC): deployed in SP1, using infrastructure resources from SP1.
- Virtual Transcoding Unit (vTU): deployed in SP1 using a service deployed in the lower Service Platform (SP2) like infrastructure resource. In SP2, the vTU network service only contains the vTU network function.

4.4.1 HSP's architecture and components

In SONATA, we have implemented two approaches of MANO recursivity. The first approach was based on the recursivity of one SONATA Service Platform on top of a second SONATA Service Platform. The second approach was designed in collaboration with 5GEx Project that includes a **Slice Controller** that is a module able to create data-center slices by allocating VIMs on demand

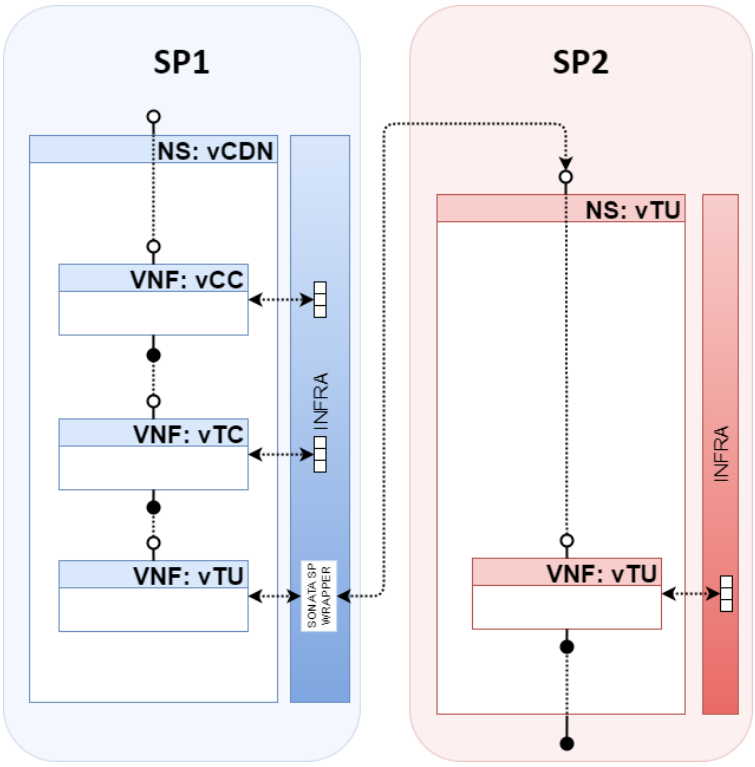

Figure 4.42: vCDN in HSP scenario

on top of elastic partitions of data-center resources. More details on this components have been given in Section Section 3.2.4.

4.4.1.1 HSP - SONATA SP on top SONATA SP

The SONATA architecture in HSP Pilot remains almost the same as the standalone SONATA SP. In order to enable recursiveness, and connect a SONATA Service Platform with another SONATA Service Platform, a new module called SONATA Wrapper is introduced. The following picture Figure 4.43, shows the architecture in a HSP scenario with two Service Platforms properly connected. The SONATA Wrapper is located at the Infrastructure Abstraction level, and directly communicates the upper Service Platform with the Gatekeeper API in the lower Service Platform.

4.4.1.2 HSP - 5GEx Collaboration

One of the step on the path to a fully inter-working hierarchy of SONATA service platform is the achievement of a recursive data model for the NFVI and the relevant resource exposed. For sake of readability, we briefly recapitulate the NFVI model used by SONATA.

The Figure 4.44 shows an example of NFVI according to the SONATA model. The NFVI is composed by several NFVI-PoP, which can be though of as datacenters which offers computational as well as storage and networking resources exposed through one or more VIMs. The NFVI-PoPs are connected through a WAN which offers connectivity between PoPs, access to external users and access to the internet. The Wan can be segmented in areas, and each area is managed by a WIM. In our scenario we demonstrate that such an infrastructural data model can be managed in a hierarchical way by leveraging SONATA platforms.

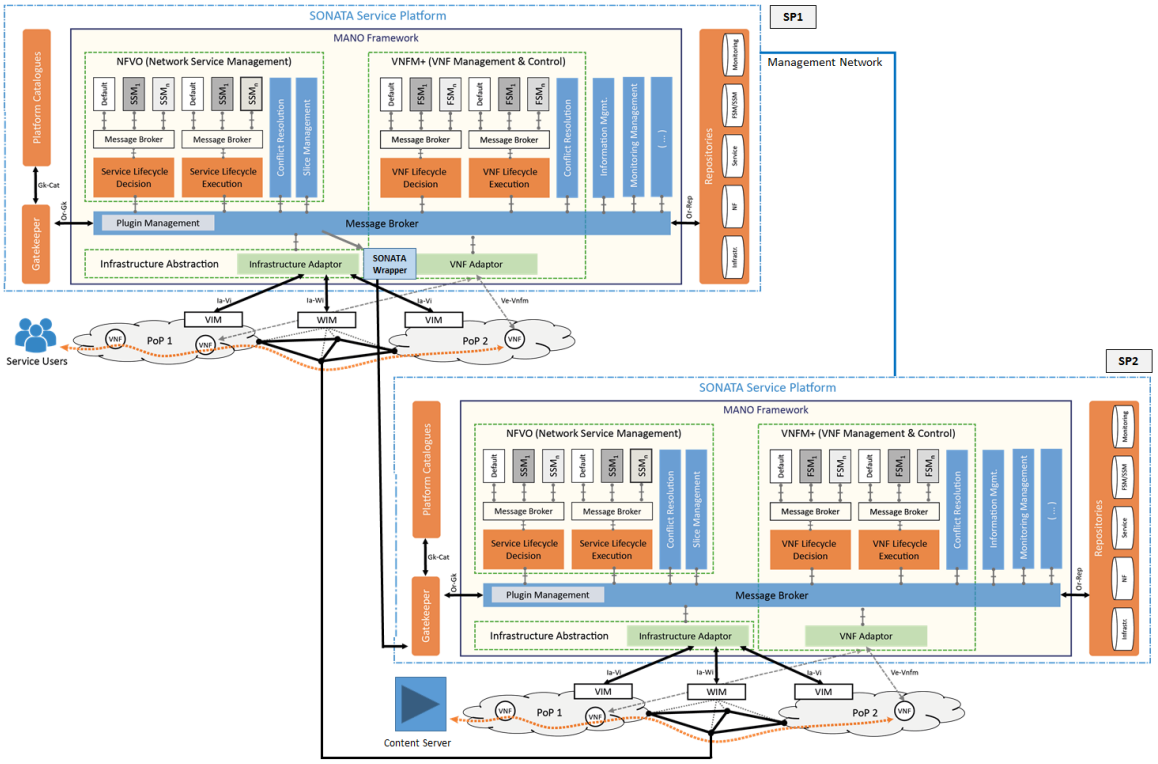

Figure 4.43: HSP architecture

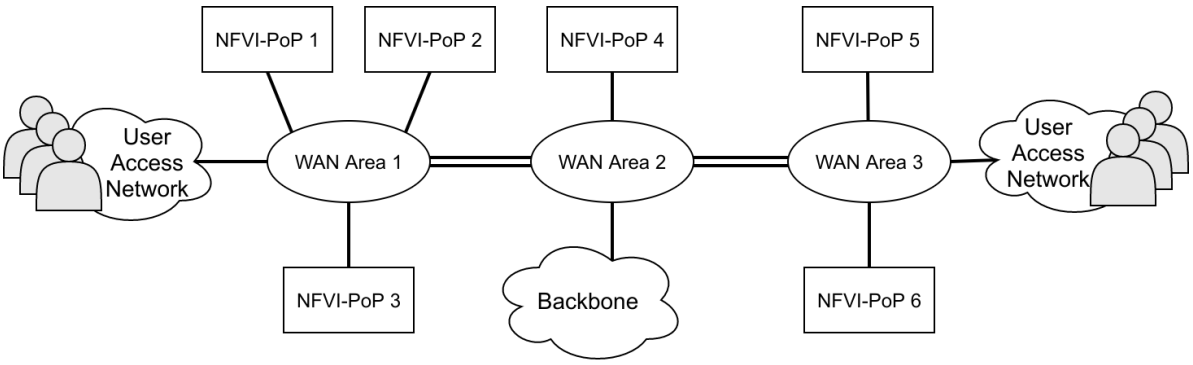

Figure 4.44: NFVI example

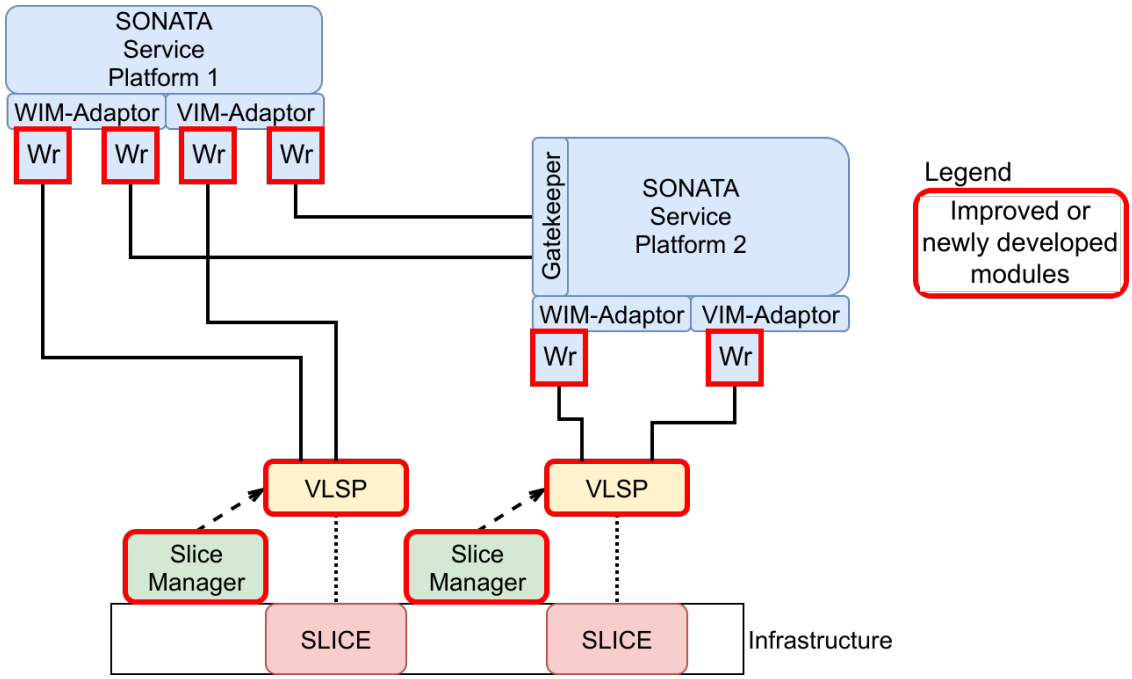

Figure 4.45: A NFVI scenario

The Figure 4.45 shows the considered scenario. The NFVI is composed by 2 NFVI-PoPs and two WAN area. Both this entities are represented by the two VLSP VIM/WIM provided by the two Slice managers. The SONATA Gatekeeper API, through its subcomponent **gtkvim**, offers two endpoints which are particularly useful in this scenario. The */vims* and */wims* endpoints allow accessing the information that the SONATA Infrastructure Abstraction uses to keep track of the NFVI composition, as well as its status in terms of resource availability and configurations. It is worth noting that, likewise any other system call that is relevant to the infrastructure, these API call are mediated by the Infrastructure Abstraction. Also, the Abstraction Layer itself allows the service platform to adapt the behaviour of the service platform in response to call relevant to the infrastructure. A composition of these features is the key concept facilitating the following scenario. A service operator accesses the upper layer service platform and wants to retrieve information about the composition of the NFVI and of the available resources. He uses the gatekeeper API to retrieve this information using the relevant API mentioned above. This call triggers the API to retrieve information on all NFVI-PoPs configured in the infrastructure repository. Thanks to the flexibility of the infrastructure abstraction layer, this can be configured to list the lower level Service Platform as a PoP connected to the upper layer Service Platform. This way the call to retrieve resources from this special PoP can be mapped in the abstraction layer to a request to a request to the lower layer SP to list available PoPs configured in it. The Figure 4.46 and the Figure 4.47 shows the internal configuration of the database used by the abstraction layer of the two service platforms, while the Figure 4.48 shows the data structures retrieved from the **/vims** API by means of the above mentioned mechanism.

The same mechanism is equally valid for the WAN information contained in the abstraction layer, as shown in the Figure 4.49.

Figure 4.46: Database internal configuration 1

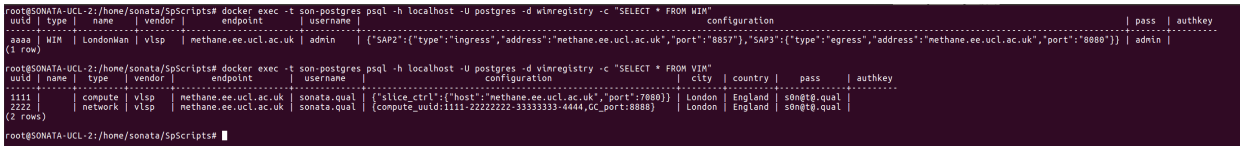

Figure 4.47: Database internal configuration 2

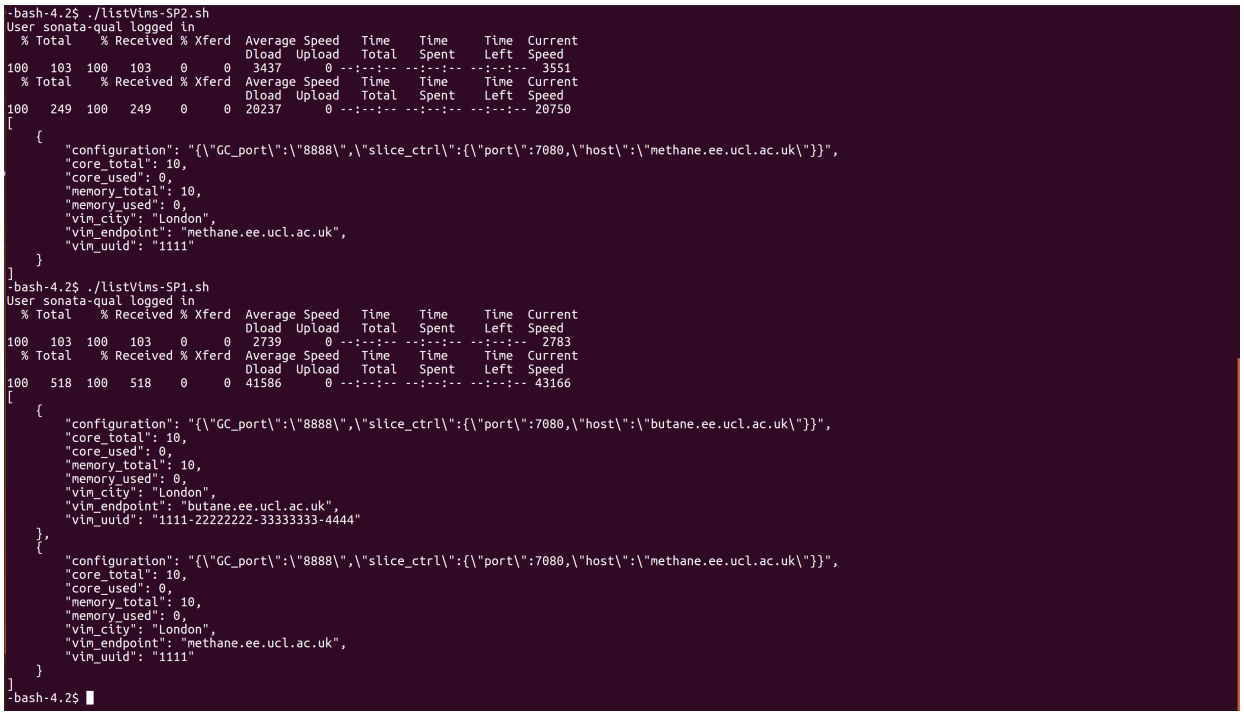

Figure 4.48: Data structures from the **/vims** API

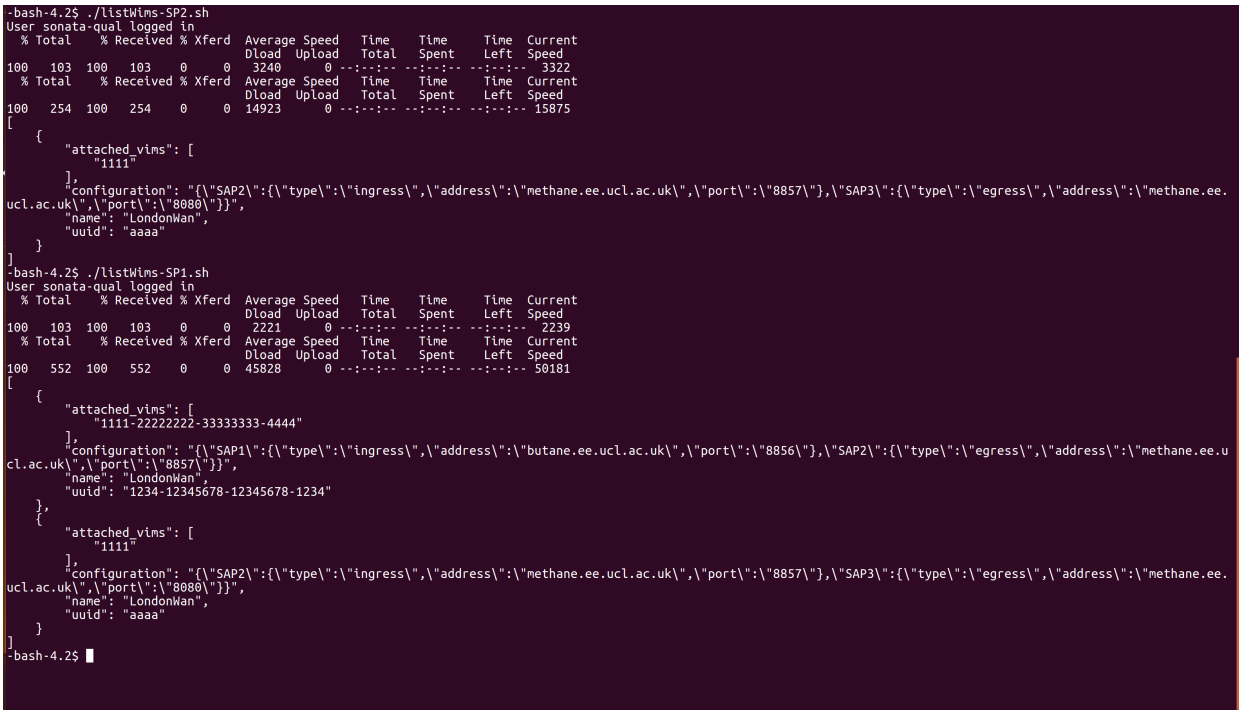

Figure 4.49: Data structures from the **/wims** API

4.4.2 HSP demonstration scenarios

As was mentioned before, the HSP Scenario consist in a "splitted" network service whose components (network functions) will be instantiated in two different service platforms interconnected in a hierarchical way.

Below is shown the steps that will be followed to instantiate the overall service performed by the two service platforms. For reasons of visibility, the message sequence chart of the entire process was divided in two different diagrams, each of them associated to the instantiation of the service since the point of view of each service platform.

The Figure 4.50 shows the steps followed to instantiate the overall network service from the SP1 point of view. In these steps we can highlight:

1. the invoker user performs a vCDN network service instantiation request filling in the BSS the ingress and egress information which will be used to establish the placement of the service components

2. the gatekeeper receives the request and fetch the network service descriptor that contains the properties of the service in the catalogue

3. the gatekeeper invokes to the service lifecycle management to instantiate the service and passes it the service descriptor

4. the service lifecycle management asks placement plugin for the future placement of the network service components

5. the placement plugin using the ingress and egress information, responses to the service lifecycle management that the vCDN network functions will be instantiated as follows: vTC and vCC

in the first service platform (SP1) and vTU in the second service platform (SP2). All network service management and control will be in SP1

6. with this information, the service lifecycle management makes the functions instantiation requests to the function lifecycle management
7. function lifecycle management deploy de FSMs in the SP1 and calls the Infrastructure Adaptor (IA) to instantiate the network functions
8. the IA determines which wrapper will it use: Openstack wrapper if destination is SP1 or SP Wrapper if destination is SP2
9. IA requests the function instantiation to the right wrapper (vTC and vCC use OpenStack wrapper and vTU uses SP wrapper)
10. the wrapper requests the network function instantiation to the VIM and obtains the instance identifier and the status when the function is instantiated
11. the instance identifier and the status are sent to the function lifecycle management through the IA
12. the function lifecycle management stores the VNF record associated to the network function instance in the repository and informs the service lifecycle management about the network function instance status
13. once all the components are instantiated, the IA configures the service chain and informs the service lifecycle management about the readiness of the service
14. the function lifecycle management stores the NS record associated to the network service instance in the repository and informs the gatekeeper about the network function instance status
15. the gatekeeper informs the BSS about the readiness of the service
16. the invoker user can check in the BSS that the network service is ready

The Figure 4.51 shows the steps followed to instantiate the overall network service from the SP2 point of view. In these steps we can highlight:

1. the SP Wrapper in SP1 requests the vTU NS instantiation to the SP2's gatekeeper that receives the request and fetch the network service descriptor that contains the properties of the service in the catalogue
2. the gatekeeper invokes the service lifecycle management to instantiate the service and passes it the service descriptor
3. the service lifecycle management asks placement plugin for the future placement of the network service components
4. the placement plugin responses to the service lifecycle management that the vTU network service will be instantiated entirely in the SP2
5. with this information, the service lifecycle management makes the function instantiation requests to the function lifecycle management

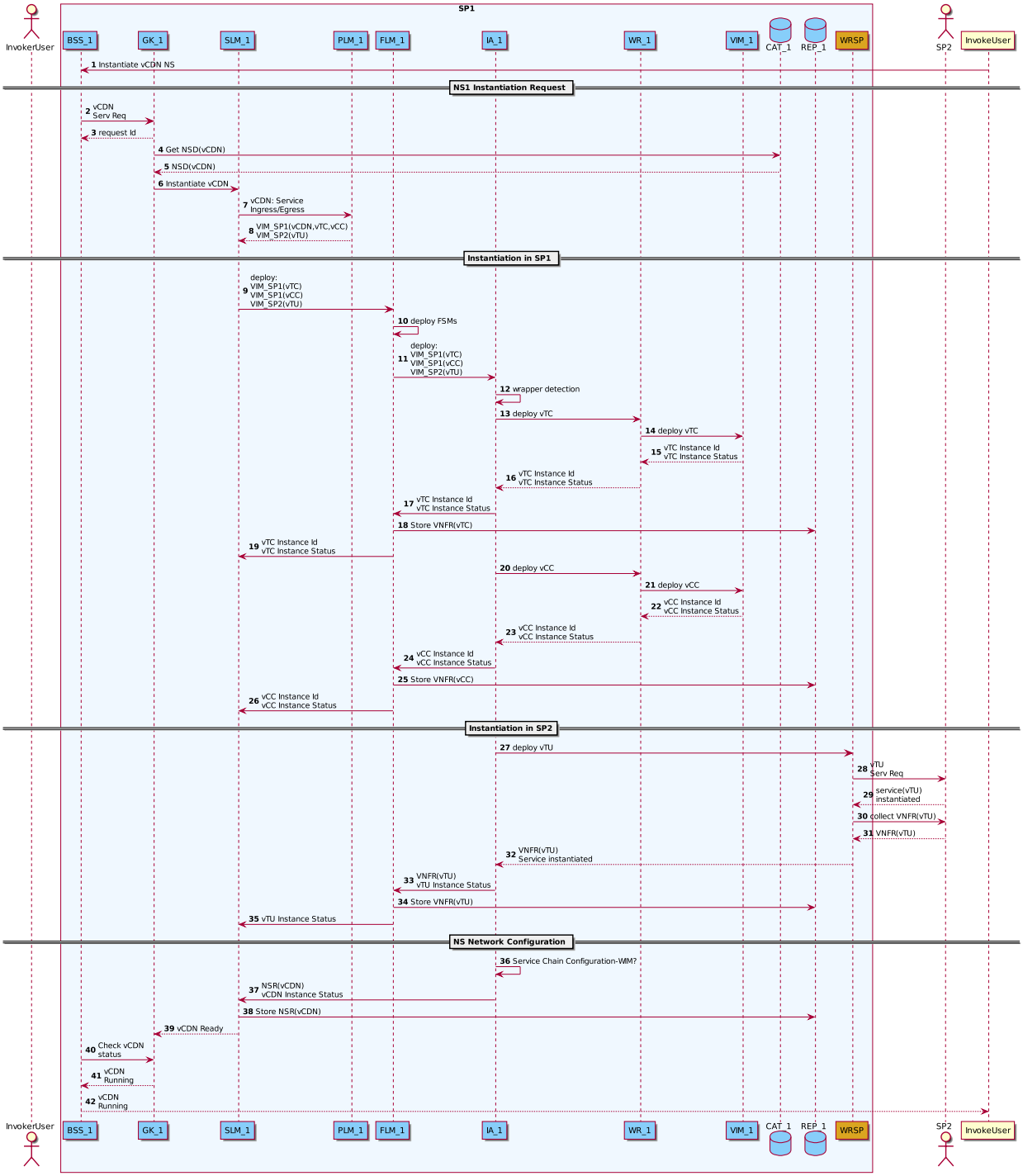

Figure 4.50: Service Instantiation on SP1

6. function lifecycle management calls the Infrastructure Adaptor (IA) to instantiate the network function
7. the IA determines which wrapper will it use: Openstack wrapper. It could be possible to use another SP wrapper if a third service platform was connected to SP2 (recursive way)
8. IA requests the function instantiation to the wrapper
9. the wrapper requests the network function instantiation to the VIM and obtains the instance identifier and the status when the function is instantiated
10. the instance identifier and the status are sent to the function lifecycle management through the IA
11. the function lifecycle management stores the VNF record associated to the network function instance in the repository and informs the service lifecycle management about the network function instance status
12. the IA informs the service lifecycle management about the readiness of the service
13. the service lifecycle management stores the NS record associated to the network service instance in the repository and informs the gatekeeper about the network function instance status
14. the gatekeeper informs the SP1 about the readiness of the service
15. the SP Wrapper in SP1 requests the VNF record to the gatekeeper

4.4.3 HSP SSM and FSM components

Since the recursiveness of the SONATA SP was designed in such a way that it is transparent for the MANO Framework (for the MANO Framework, the lower SP appears as just another VIM attached to the Infrastructure Adaptor and the upper SP as just another user requesting a NS), no additional functionality is required from the MANO to support the HSP case. Since we showcase our HSP feature using the vCDN service introduced in Section 4.1, and all SSMs and FSMs are extensions and thus components of the MANO Framework, the developer does not need to alter any of the SSMs and FSMs. The used SSMs and FSMs in this pilot are thus identical to those in the vCDN pilot and their description can be found in Section 4.1.3.

4.4.4 HSP NS validation

This section covers the screenshots and logs of the vCDN service deployed in HSP mode.

4.4.4.1 BSS

Each service platform has its own BSS module that shows the available services that can be instantiated, the running instances and the performed requests. In the Hierarchical Service Platform scenario, the vCDN network service (composed by vCC, vTU and vTC network functions) is instantiated in the SP1 using resources from SP1 and SP2.

Figure 4.52 shows the service descriptor on-boarded in SP1, the vCDN that is composed by vCC, vTU and vTC. The service on-boarded in SP2 that can be seen in Figure 4.53 is the vtu-vnf,

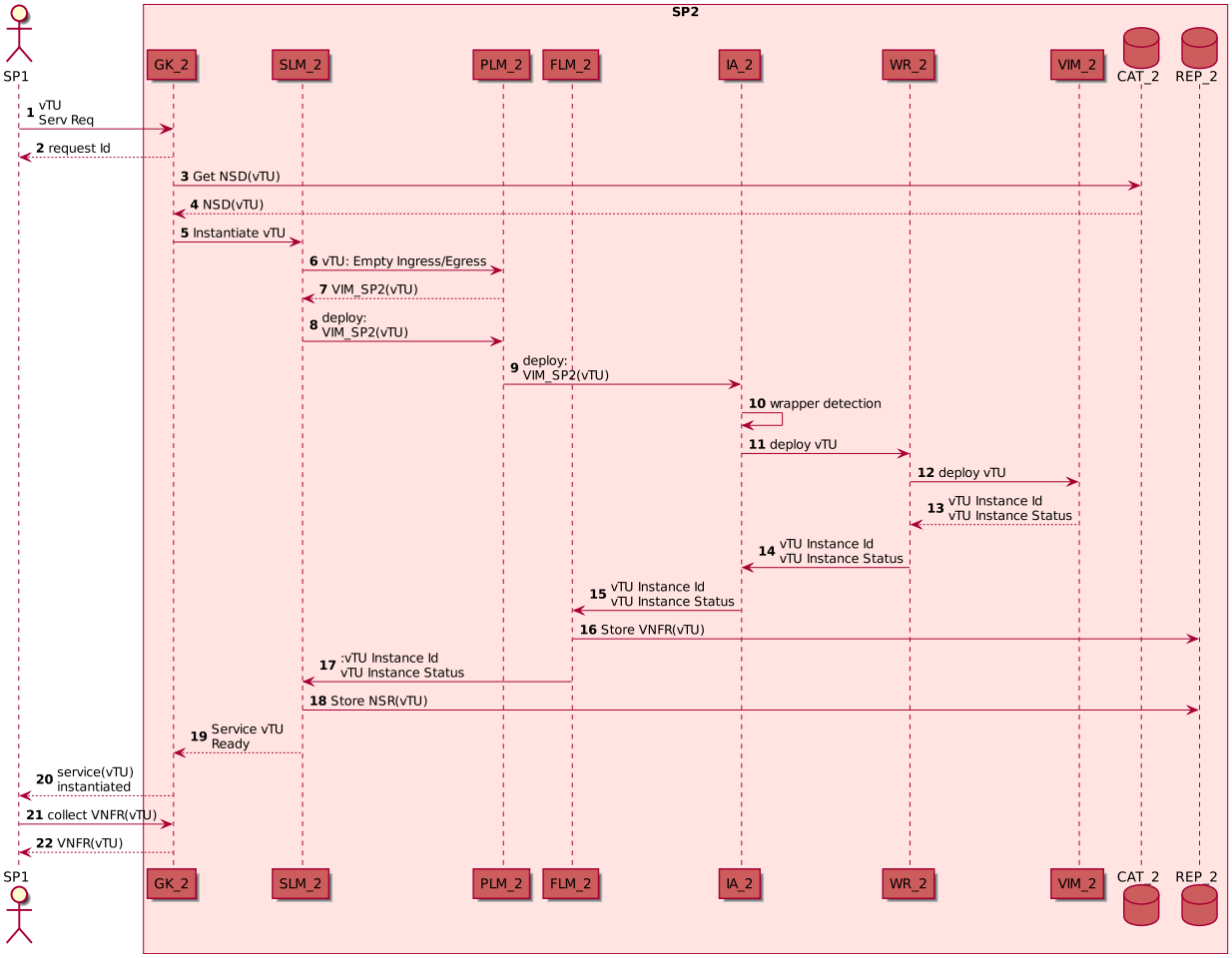

Figure 4.51: Service Instantiation on SP2

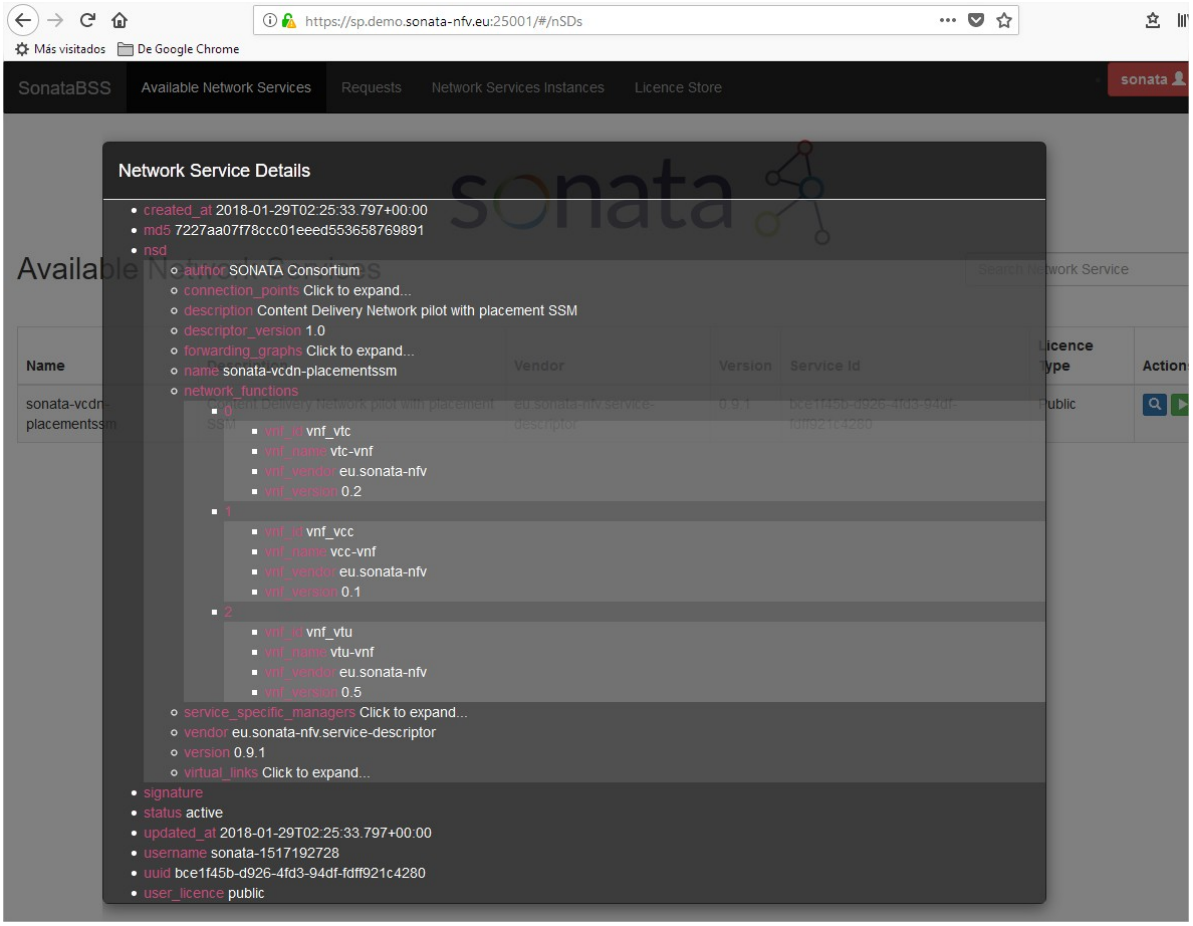

Figure 4.52: HSP Pilot BSS service in SP1

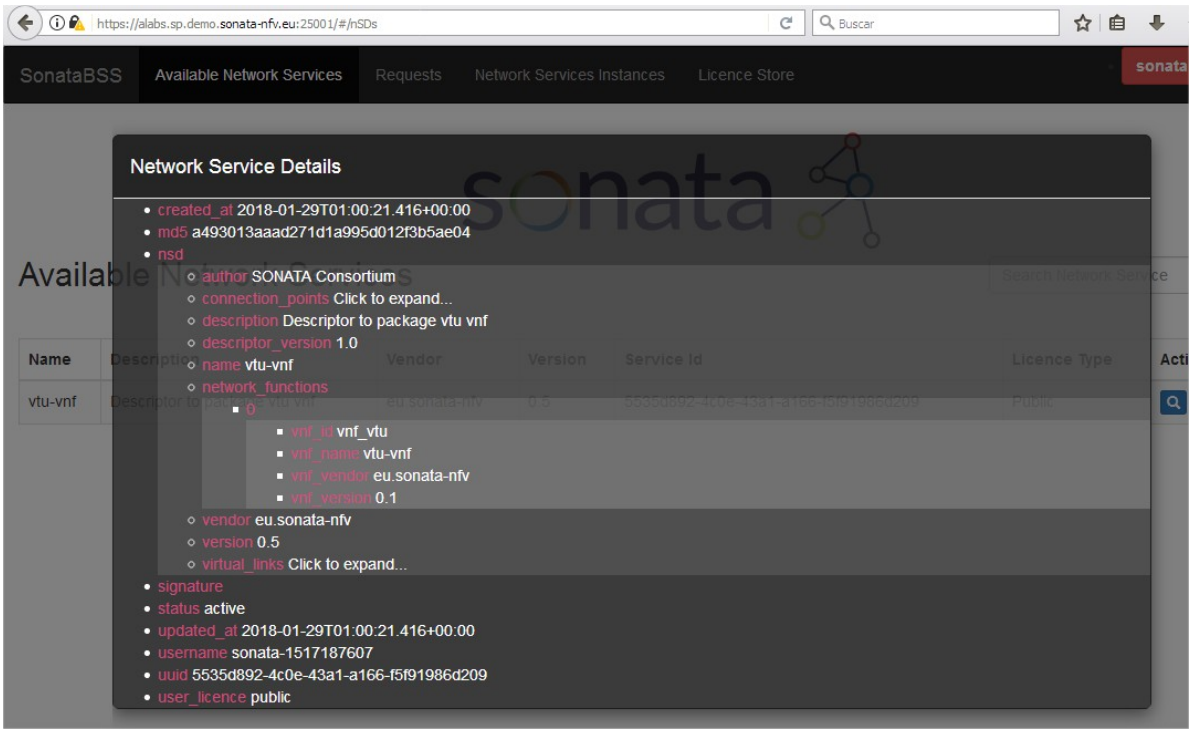

Figure 4.53: HSP Pilot BSS service in SP2

a network service that only contains one network function, the vtu-vnf (that's the reason of the network service name).

When the ingress and egress fields are filled as shown in Figure 4.54 the SP1 detects that the service instantiation requires resources from SP1 and SP2, and make the corresponding requests.

Figure 4.55 shows the ID assigned to the instantiation request in SP1.

Figure 4.56 and Figure 4.57 show that instantiation request in SP1 changes its status from INSTANTIATING to READY.

Figure 4.58 and Figure 4.59 show that instantiation request in SP2 changes its status from INSTANTIATING to READY.

The instantiation request in READY status means that the lifecycle of the request is finished and the Service is instantiated as can be seen in Figure 4.60 and Figure 4.61, that show the "Normal Operation" instances in both Service Platforms.

4.4.4.2 GUI

In the Graphical User Interface, as is shown in Figure 4.62 and Figure 4.63 can be listed the VNF instances deployed in SP1 and SP2. As we mentioned before, in SP1 there are three VNF instances: vCC, vTC and vTU, where vTU uses the resources from SP2. In SP2 only appears the vTU instance.

4.4.4.3 OpenStack

Figure 4.64 shows the VNF instances deployed in the SP1's OpenStack: vCC and vTC.

Figure 4.65 shows the VNF instance deployed in the SP2's OpenStack: vTU.

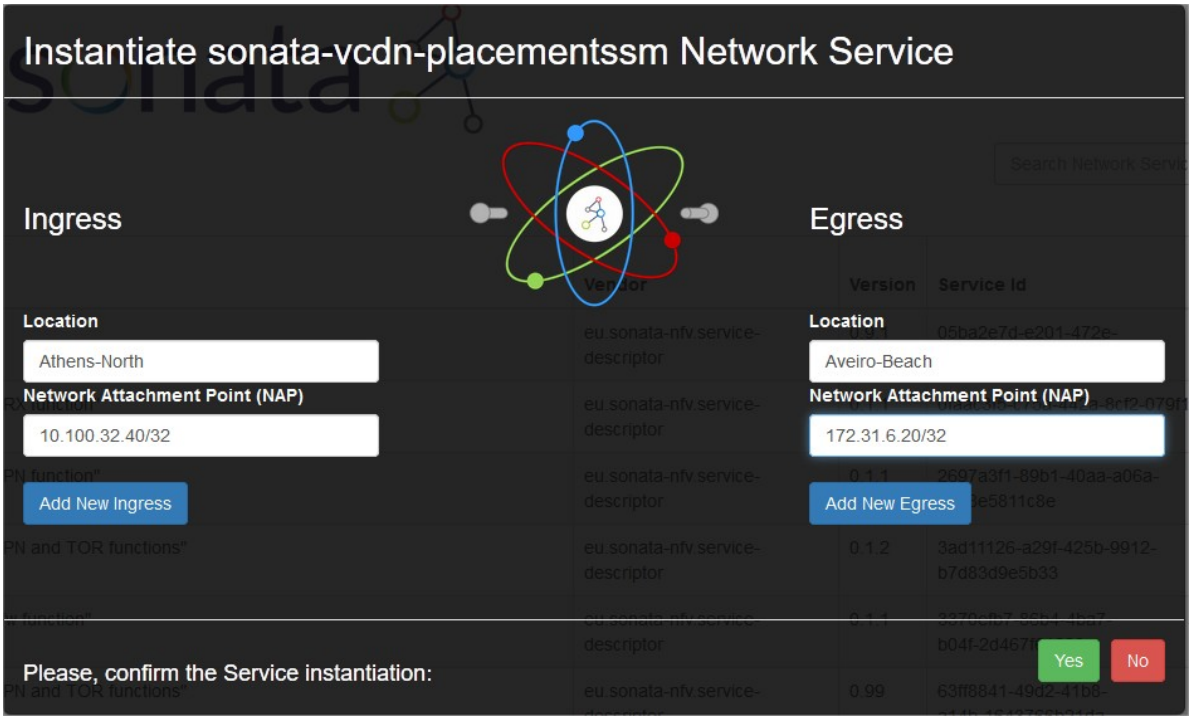

Figure 4.54: HSP Pilot BSS service ingress and egress

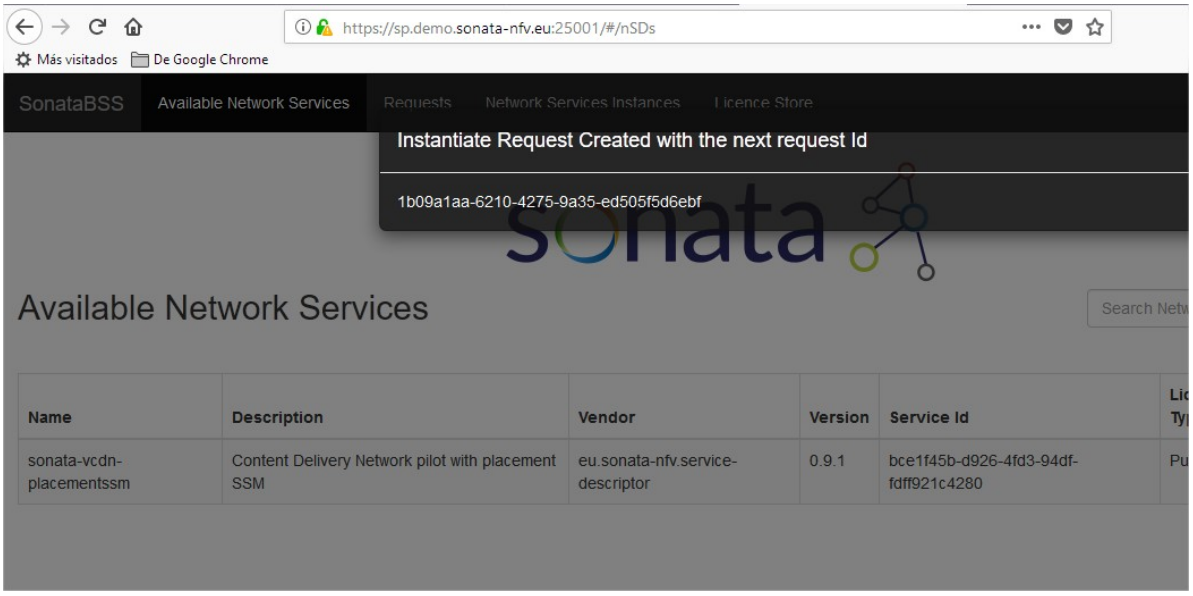

Figure 4.55: HSP Pilot BSS service instantiation request id

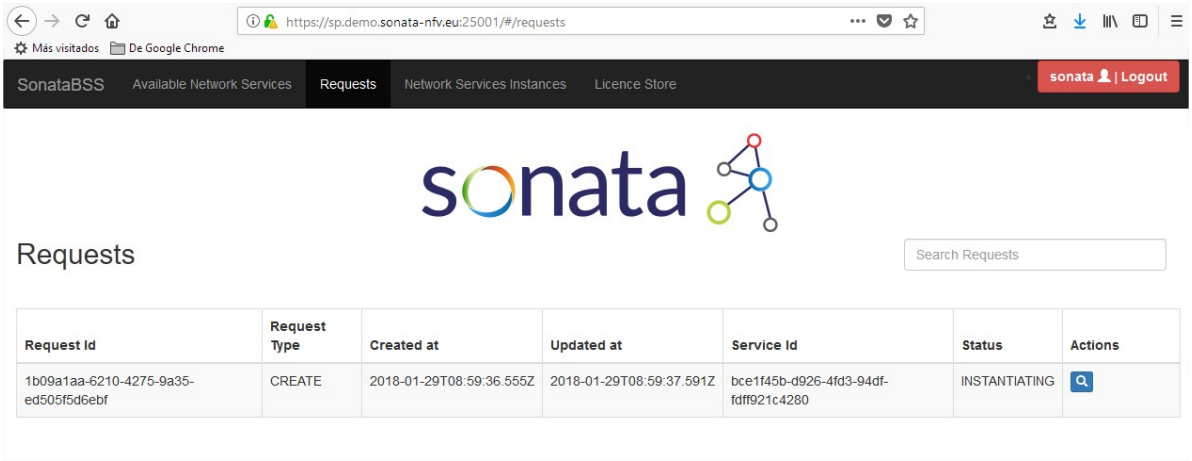

Figure 4.56: HSP Pilot BSS request instantiating status in SP1

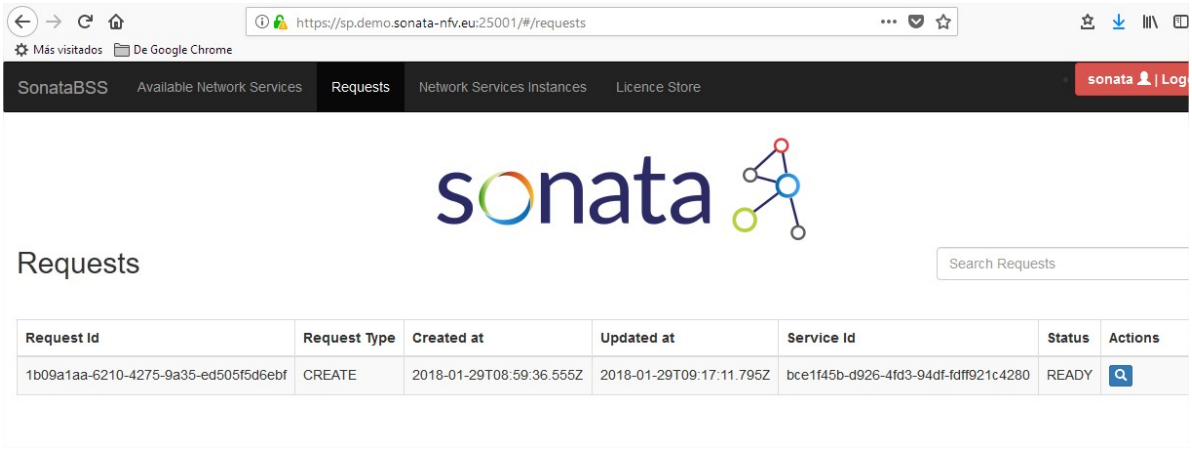

Figure 4.57: HSP Pilot BSS request ready status in SP1

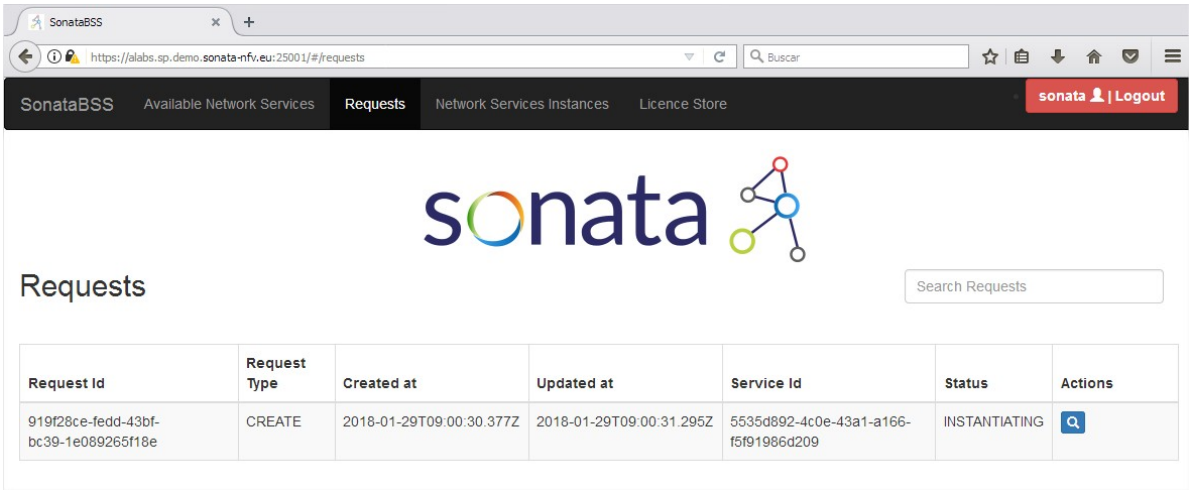

Figure 4.58: HSP Pilot BSS request instantiating status in SP2

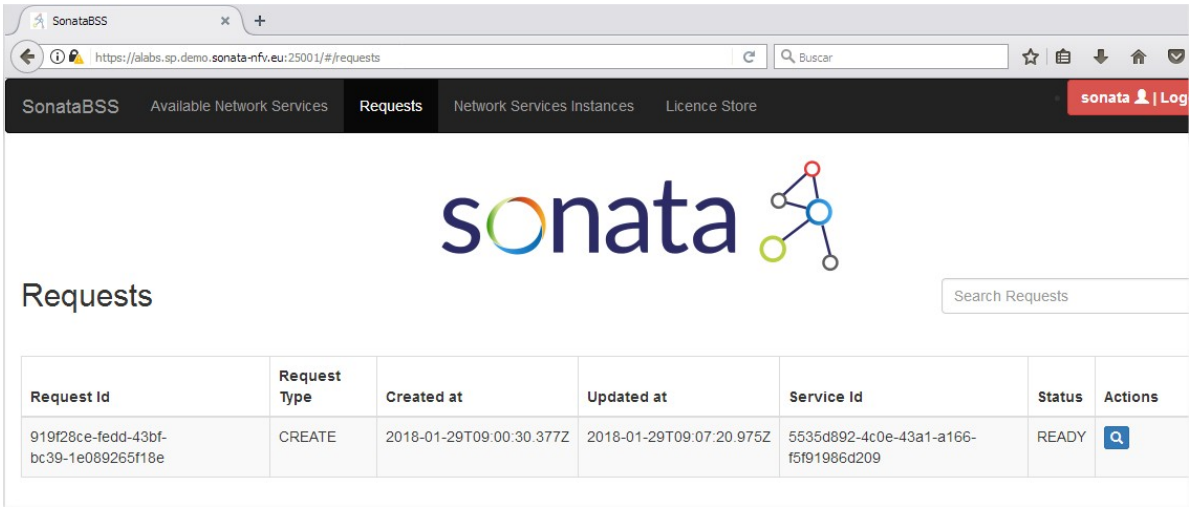

Figure 4.59: HSP Pilot BSS request ready status in SP2

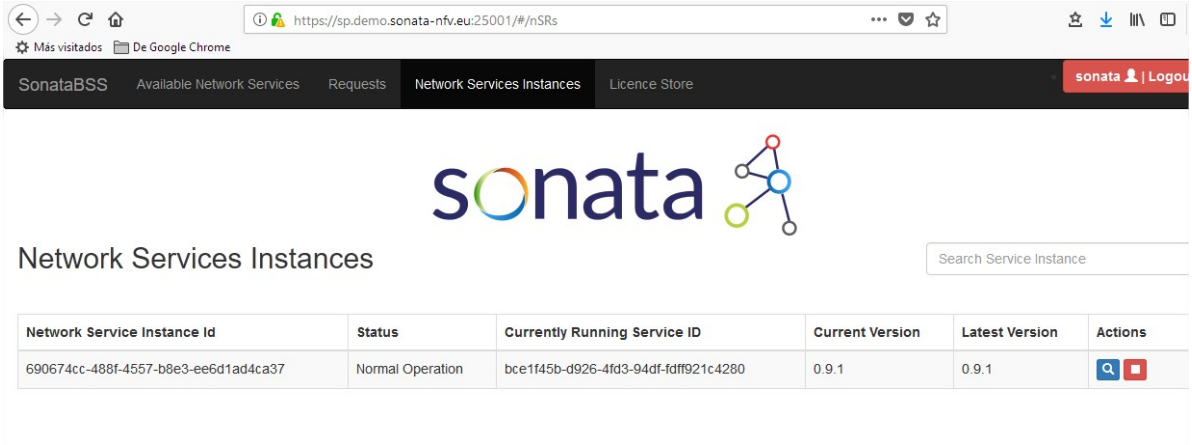

Figure 4.60: HSP Pilot BSS instance in normal operation SP1

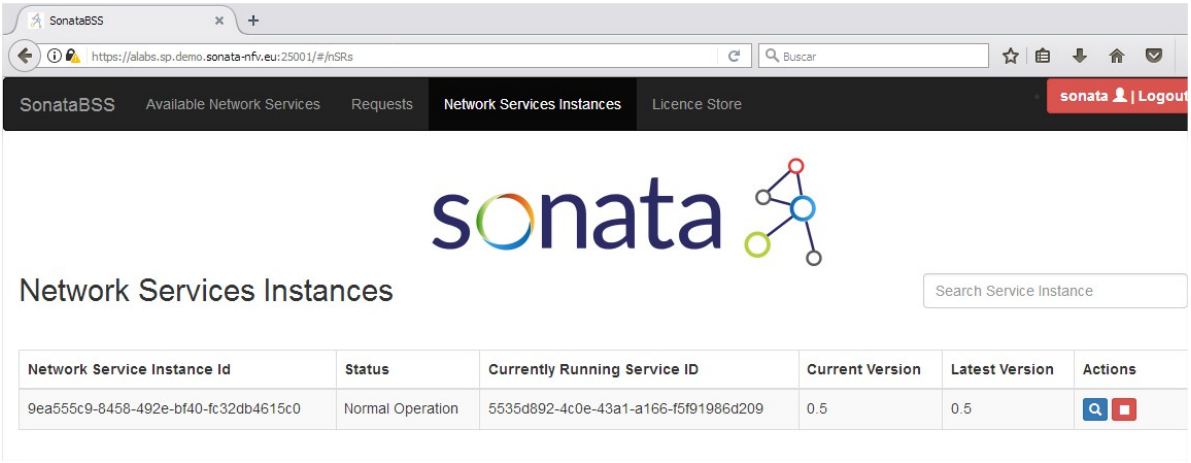

Figure 4.61: HSP Pilot BSS instance in normal operation SP2

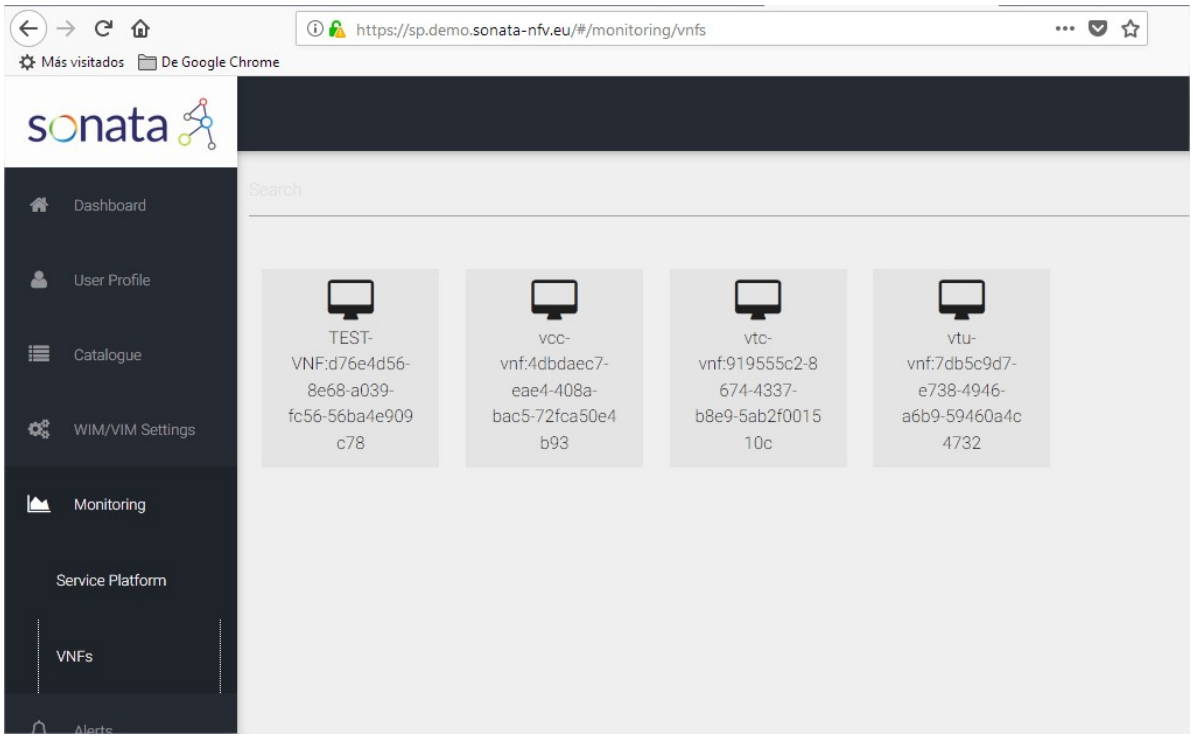

Figure 4.62: HSP Pilot GUI VNF instances in SP1

4.4.4.4 Monitoring Framework

Figure 4.66 shows some of the metrics collected from the Monitoring Framework with respect to the performance of the vTC instance in SP1.

Additionally, Figure 4.67 provides information on vCC performance monitoring.

It must be stressed that these figures show that monitoring framework works as the glue between the Dev and the Ops part in the SONATA context, by providing valuable information to users in order to optimize the performance of the running services.

4.4.4.5 Logs of IA SONATA GK Client

This section contains the trace logs about the communication between the Infrastructure Abstraction module from SP1 and the Gatekeeper API from SP2.

Get Active service From SP2

SP1 collects the list of available services in SP2 that can be instantiated:

```
[DEBUG] [SONATA-GK-CLient] Retrieving active services:
[DEBUG] [SONATA-GK-CLient] /services endpoint request (Request Object):
[DEBUG] GET http://172.31.6.41:32001/api/v2/services?status=active HTTP/1.1
[DEBUG] [SONATA-GK-CLient] /services endpoint response (Response Object):
[DEBUG] HttpResponseProxy{HTTP/1.1 200 OK Content-Type: application/json, Link:
    <http://172.31.6.41:32001/api/v2/services?offset=0&limit=10>;
    rel="first",<http://172.31.6.41:32001/api/v2/services?offset=0&limit=10>;
    rel="last",
    Record-Count:1, X-Content-Type-Options: nosniff, Content-Length: 1083]
```

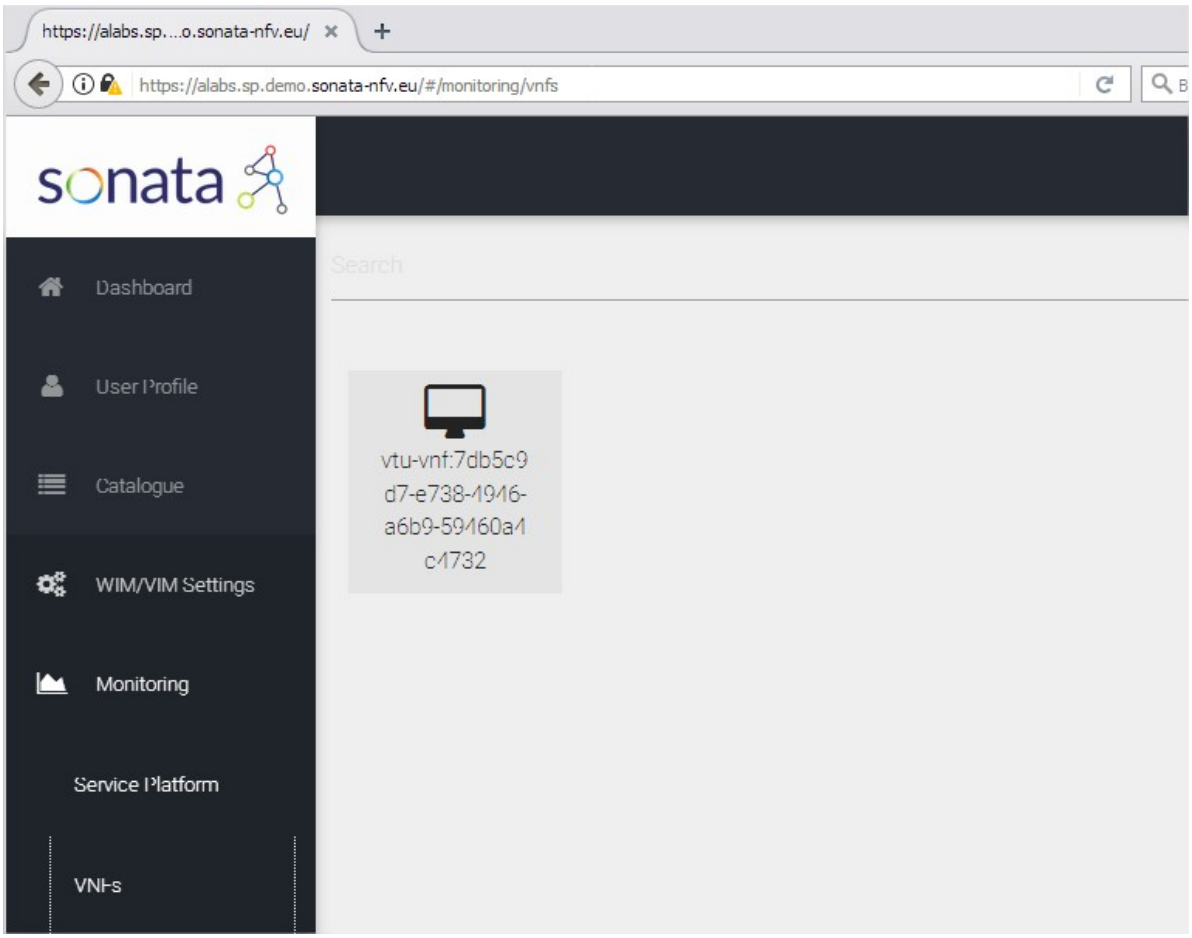

Figure 4.63: HSP Pilot GUI VNF instances in SP2

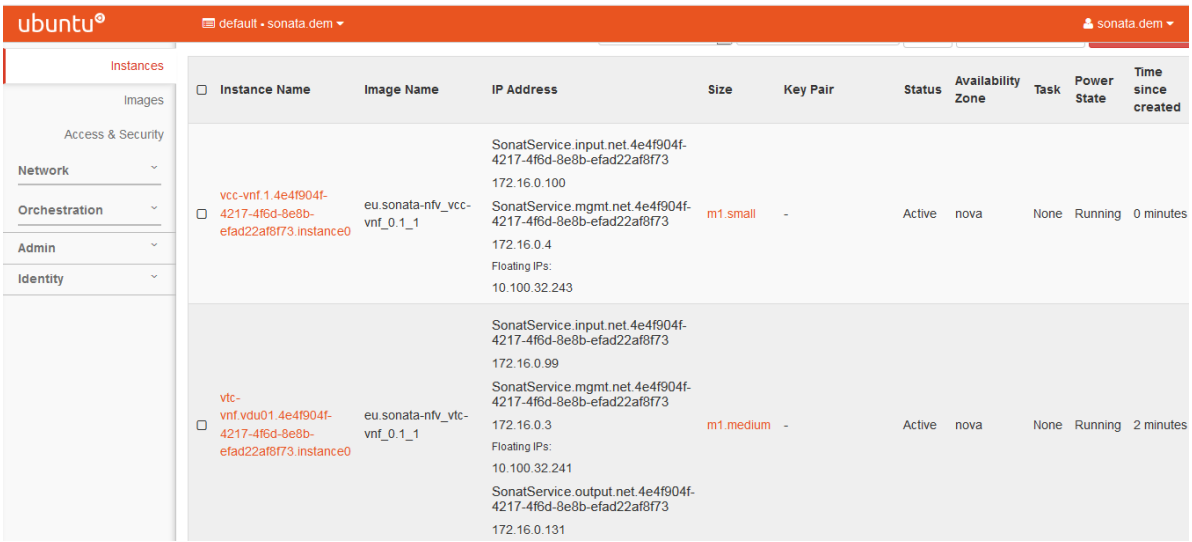

Figure 4.64: HSP Pilot - Instances deployed in VIM attached to SP1

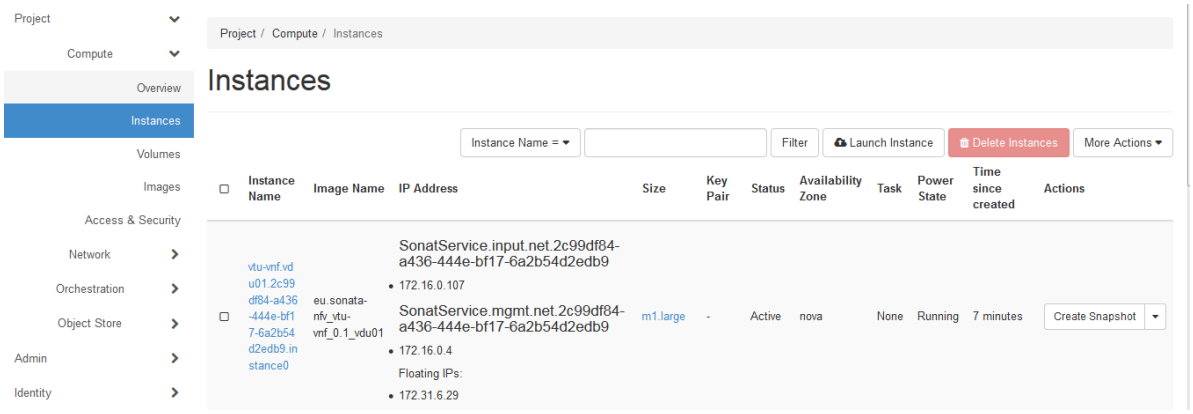

Figure 4.65: HSP Pilot - Instances deployed in VIM attached to SP2

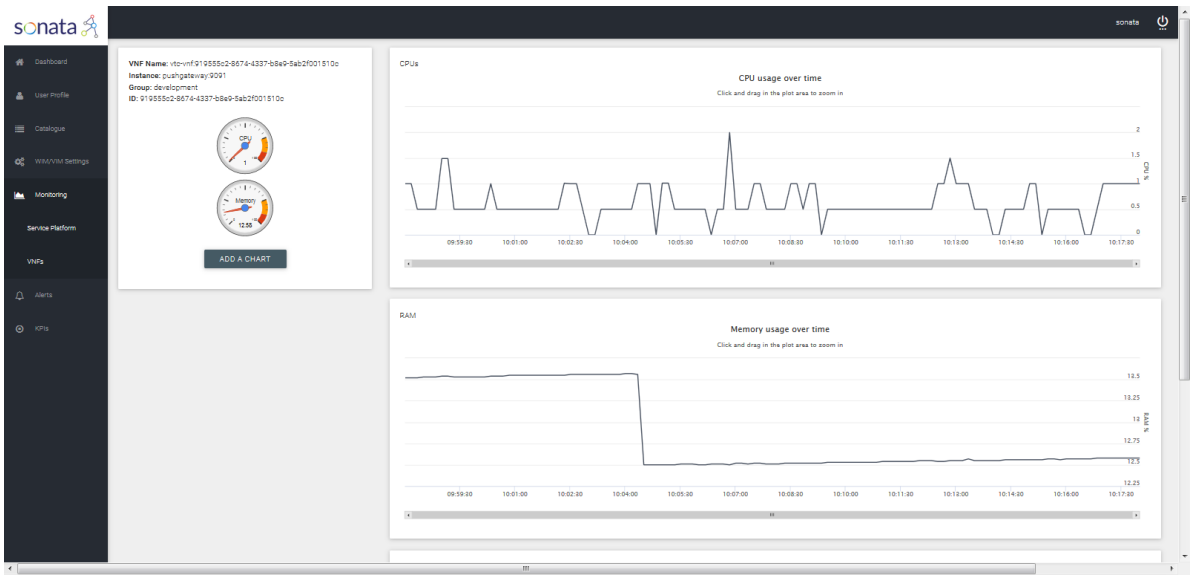

Figure 4.66: Monitoring view of vtc

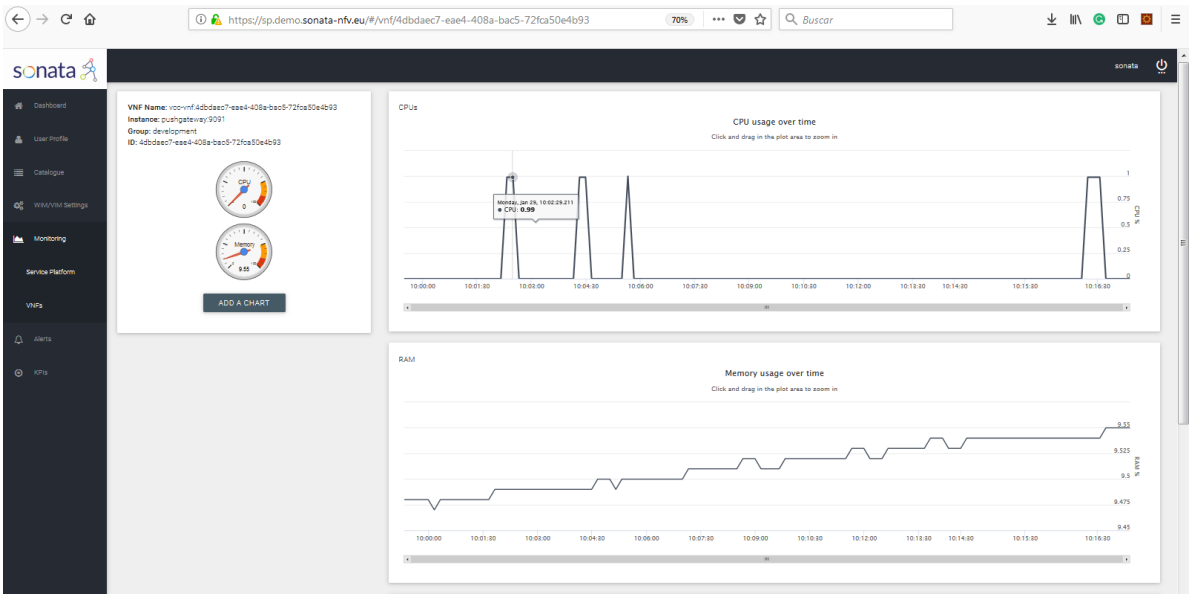

Figure 4.67: Monitoring view of vCC


```
    ResponseEntityProxy{Content-Type:application/json,Content-Length: 1083,
    Chunked: false]}}
[DEBUG] Response Received with Status: 200
[DEBUG] {"created_at":"2018-01-29T01:00:21.416+00:00","md5":
    "a493013aaad271d1a995d012f3b5ae04","nsd":{"author":"SONATA Consortium",
    "connection_points":{"id":"nscpmgmt","interface":"ipv4","type":"management"},
    {"id":
    "nscpinput","interface":"ipv4","type":"external"},{"id":"nscpoutput",
    "interface":
    "ipv4","type":"external"}],"description":"Descriptor to package vtu vnf",
    "descriptor_version":"1.0","name":"vtu-vnf","network_functions":{"vnf_id":
    "vnf_vtu",
    "vnf_name":"vtu-vnf","vnf_vendor":"eu.sonata-nfv","vnf_version":"0.1"}],
    "vendor":
    "eu.sonata-nfv","version":"0.5","virtual_links":{
    "connection_points_reference":
    "vnf_vtu:cpmgmt","nscpmgmt"],"connectivity_type":"E-LAN","id":"vlmgmt"},
    {"connection_points_reference":"nscpinput","vnf_vtu:cpinput"],
    "connectivity_type":
    "E-Line","id":"vlinput"},{"connection_points_reference":"vnf_vtu:cpinput",
    "nscpoutput"],
    "connectivity_type":"E-Line","id":"vlvccout"}]},"signature":null,"status":
    "active",
    "updated_at":"2018-01-29T01:00:21.416+00:00","username":"sonata-1517187607",
    "uuid":
    "5535d892-4c0e-43a1-a166-f5f91986d209","user_licence":"public"}]
[DEBUG] VNF: eu.sonata-nfv::vtu-vnf::0.5
```


Instantiating in SP2

SP1 requests the vTU service instantiation in SP2:

```
[DEBUG] [SONATA-GK-CLient] Creating a new instantiation request
[DEBUG] [SONATA-GK-CLient] /requests POST request:
[DEBUG] POST http://172.31.6.41:32001/api/v2/requests HTTP/1.1
[DEBUG] Response Received with Status: 201
[DEBUG] [SONATA-GK-CLient] /requests POST response:
[DEBUG] {"id":"919f28ce-fedd-43bf-bc39-1e089265f18e","created_at":
    "2018-01-29T09:00:30.377Z",
    "updated_at":"2018-01-29T09:00:30.377Z","service_uuid":
    "5535d892-4c0e-43a1-a166-f5f91986d209","status":"NEW","request_type":"CREATE",
    "service_instance_uuid":null,"began_at":"2018-01-29T09:00:29.959Z","callback":
    "http://son-gtkkpi:5400/service-instantiation-time"}
[DEBUG] [SONATA-GK-CLient] Getting request information object
[DEBUG] [SONATA-GK-CLient] /requests request:
[DEBUG] GET http://172.31.6.41:32001/api/v2/requests/919f28ce-fedd-43bf-bc39-\
    1e089265f18e
    HTTP/1.1
[INFO] Received message on infrastructure.function.deploy
[DEBUG] message SID-ca40072e-2885-4dbc-9bee-0cbfcaddbdb6
[INFO] Deploy function call received by call processor.
```

Get status for the service deployment = INSTANTIATING

SP1 detects that the request status changes from NEW to INSTANTIATING:

```
[DEBUG] [SONATA-GK-CLient] Getting request information object
[DEBUG] [SONATA-GK-CLient] /requests request:
[DEBUG] GET http://172.31.6.41:32001/api/v2/requests/919f28ce-fedd-43bf-bc39-\
    1e089265f18e HTTP/1.1
[DEBUG] Response Received with Status: 200
[DEBUG] [SONATA-GK-CLient] /requests response:
[DEBUG] {"id":"919f28ce-fedd-43bf-bc39-1e089265f18e","created_at":
    "2018-01-29T09:00:30.377Z","updated_at":"2018-01-29T09:00:31.295Z",
    "service_uuid":"5535d892-4c0e-43a1-a166-f5f91986d209","status":
    "INSTANTIATING",
    "request_type":"CREATE","service_instance_uuid":null,"began_at":
    "2018-01-29T09:00:29.959Z","callback":
    "http://son-gtkkpi:5400/service-instantiation-time"}
[INFO] Status of request 919f28ce-fedd-43bf-bc39-1e089265f18e: INSTANTIATING
[DEBUG] [SONATA-GK-CLient] Getting request information object
[DEBUG] [SONATA-GK-CLient] /requests request:
[DEBUG] GET http://172.31.6.41:32001/api/v2/requests/919f28ce-fedd-43bf-bc39-\
    1e089265f18e HTTP/1.1
[DEBUG] Response Received with Status: 200
[DEBUG] [SONATA-GK-CLient] /requests response:
[DEBUG] {"id":"919f28ce-fedd-43bf-bc39-1e089265f18e",
    "created_at":"2018-01-29T09:00:30.377Z",
```

```
    "updated_at":"2018-01-29T09:00:31.295Z","service_uuid":
    "5535d892-4c0e-43a1-a166-f5f91986d209","status":"INSTANTIATING",
    "request_type":
    "CREATE","service_instance_uuid":null,"began_at":"2018-01-29T09:00:29.959Z","
    callback":"http://son-gtkkpi:5400/service-instantiation-time"}
[INFO] Status of request 919f28ce-fedd-43bf-bc39-1e089265f18e: INSTANTIATING
```


Get status for the service deployment = READY

SP1 detects that the request status changes from INSTANTIATING to READY:


```
[DEBUG] [SONATA-GK-CLient] Getting request information object
[DEBUG] [SONATA-GK-CLient] /requests request:
[DEBUG] GET http://172.31.6.41:32001/api/v2/requests/919f28ce-fedd-43bf-bc39-\
    1e089265f18e HTTP/1.1
[DEBUG] Response Received with Status: 200
[DEBUG] [SONATA-GK-CLient] /requests response:
[DEBUG] {"id":"919f28ce-fedd-43bf-bc39-1e089265f18e",
    "created_at":"2018-01-29T09:00:30.377Z",
    "updated_at":"2018-01-29T09:07:20.975Z","service_uuid":
    "5535d892-4c0e-43a1-a166-f5f91986d209","status":"READY","request_type":
    "CREATE",
    "service_instance_uuid":"9ea555c9-8458-492e-bf40-fc32db4615c0","began_at":
    "2018-01-29T09:00:29.959Z","callback":
    "http://son-gtkkpi:5400/service-instantiation-time"}
[INFO] Status of request 919f28ce-fedd-43bf-bc39-1e089265f18e: READY
```


Get records to store them in VNFR

SP1 obtains information about SP2's service from the function and service records, to store it in the VNFR:


```
[DEBUG] [SONATA-GK-CLient] /requests request:
[DEBUG] GET http://172.31.6.41:32001/api/v2/requests/919f28ce-fedd-43bf-bc39-\
    1e089265f18e
    HTTP/1.1
[DEBUG] Response Received with Status: 200
[DEBUG] [SONATA-GK-CLient] /requests response:
[DEBUG] {"id":"919f28ce-fedd-43bf-bc39-1e089265f18e","created_at":\
    "2018-01-29T09:00:30.377Z",
    "updated_at":"2018-01-29T09:07:20.975Z","service_uuid":
    "5535d892-4c0e-43a1-a166-f5f91986d209","status":"READY","request_type":\
    "CREATE",
    "service_instance_uuid":"9ea555c9-8458-492e-bf40-fc32db4615c0","began_at":
    "2018-01-29T09:00:29.959Z","callback":
    "http://son-gtkkpi:5400/service-instantiation-time"}
[DEBUG] [SONATA-GK-CLient] Getting request information object...
[DEBUG] [SONATA-GK-CLient] /record/services/id request:
[DEBUG] GET http://172.31.6.41:32001/api/v2/records/services/
```

9ea555c9-8458-492e-bf40-fc32db4615c0 HTTP/1.1
[DEBUG] Response Received with Status: 200
[DEBUG] [SONATA-GK-CLient] /record/services/id response:
[DEBUG] {"created_at":"2018-01-29T09:07:10.480+00:00","descriptor_reference":
"5535d892-4c0e-43a1-a166-f5f91986d209","descriptor_version":"nsr-schema-01",
"network_functions":{"vnfr_id":"2a67392d-5d55-4c7d-b6eb-239e876c6ab5"}],\
"status":
"normal operation","updated_at":"2018-01-29T09:07:10.480+00:00","version":"1",
"virtual_links":{"id":"vlmgmt","connectivity_type":"E-LAN",
"connection_points_reference":"vnf_vtu:cpmgmt","nscpmgmt"]},{"id":"vlinput",
"connectivity_type":"E-Line","connection_points_reference":"nscpinput",
"vnf_vtu:cpinput"]},{"id":"vlvccout","connectivity_type":"E-Line",
"connection_points_reference":"vnf_vtu:cpinput","nscpoutput"]}],
"uuid":"9ea555c9-8458-492e-bf40-fc32db4615c0"}
[DEBUG] [SONATA-GK-CLient] Getting request information object...
[DEBUG] [SONATA-GK-CLient] /records/functions/ request:
[DEBUG] GET http://172.31.6.41:32001/api/v2/records/functions/
2a67392d-5d55-4c7d-b6eb-239e876c6ab5 HTTP/1.1
[DEBUG] Response Received with Status: 200
[DEBUG] [SONATA-GK-CLient] /records/functions/ response:
[DEBUG] {"created_at":"2018-01-29T09:07:10.319+00:00","descriptor_reference":
"5a5aa4d9-75fb-4c8b-9823-c01807be572e","descriptor_version":"vnfr-schema-01",\
"status":
"normal operation","updated_at":"2018-01-29T09:07:10.319+00:00","version":"2",
"virtual_deployment_units":{"monitoring_parameters":{"name":"vm_cpu_perc",\
"unit":
"Percentage"},{"name":"vm_mem_perc","unit":"Percentage"},{"name":\
"vm_net_rx_bps",
"unit":"bps"},{"name":"vm_net_tx_bps","unit":"bps"}],"id":"vdu01",\
"vnfc_instance":
{"id":"0","connection_points":{"id":"eth0","type":"management","interface":
{"address":"172.31.6.47","hardware_address":"fa:16:3e:8f:c8:7a"}},{"id":\
"input",
"type":"internal","interface":{"hardware_address":"fa:16:3e:0d:e9:6d",\
"address":
"172.16.0.105","netmask":"255.255.255.248"}}],"vc_id":
"7db5c9d7-e738-4946-a6b9-59460a4c4732","vim_id":\
"1111-22222222-33333333-5555"}],
"vm_image":"http://files.sonata-nfv.eu/son-vcdn-pilot/vtu-vnf/\
sonata-vtu.qcow2",
"vdu_reference":"vtu-vnf:vdu01","resource_requirements":{"cpu":{"vcpus":4},\
"memory":
{"size":8,"size_unit":"GB"},"storage":{"size":15,"size_unit":"GB"}},
"number_of_instances":1}],"virtual_links":{"id":"mgmt",
"connectivity_type":"E-LAN","connection_points_reference":"vdu01:eth0",\
"cpmgmt"]},
{"id":"input","connectivity_type":"E-LAN",
"connection_points_reference":"vdu01:input","cpinput"]}],

```
    "uuid":"2a67392d-5d55-4c7d-b6eb-239e876c6ab5"}
[INFO] Response created
[INFO] Received an update from the wrapper...
```

Deployment succeed
The service deployment was successfully:

```
[INFO] Deploy 7d965cd7-75d3-4039-bd83-f1021ab62b44 succeed
```

4.4.5 HSP innovations

The HSP Pilot is an initial implementation that explores how to orchestrate orchestrators and how to interwork MANO platforms, i.e., MANO recursivity. In Sonata we believe that a recursive approach is very powerful, as it allows independence of implementations and of commercial decisions, providing there is a common understanding of an API. In this Pilot we look at the case where the Sonata API is the common inter-MANO communication interface, and where there are two service platforms that together deliver the vCDN service. The previously described Pilot (Section 4.3) explores another angle of this problem space, and emphasizes different aspects. Another innovation showcased by HSP is the seamless deployment split of a network service over two service platforms. SONATA SP does that based on the resource cognition to optimize the performance of the network service.

4.5 ETSI ISG NFV Participation

As part of SONATA's strategy in Task 6.4, two activities were considered of added-value for project results dissemination. The first in the proposal and the carry out of an ETSI ISG NFV PoC. The second part was the participation with SONATA MANO solution to the ETSI NFV Plugtest. This section summarises on the above efforts.

4.5.1 ETSI ISG NFV Proof of concept

The ETSI Industry Specification Group on Network Functions Virtualization (ETSI ISG NFV) is operating a Proof-of-Concept (PoC) programme for showcasing advancements in the NFV domain. SONATA contributed a variant of the vCDN (virtualized content delivery network) pilot to this programme.

The SONATA PoC is called "Dynamic Service-specific VNF Management" and it focuses on two scenarios. Scenario 1 ("VNF Placement") is dealing with the initial placement of the individual VNFs of the vCDN service taking explicit placement instructions and resource availability into account. The second scenario ("On-demand Network Service Scaling") is based on scenario 1 and shows how monitoring and topology information gathered from the infrastructure and the service can trigger changes in the service at runtime. Specifically, a new cache VNF is added at a location where new users access the CDN.

The PoC is based on demonstrations of the service at two events:

- at IEEE Netowork Softwarisation (NetSoft) 2017, held in Bologna, Italy, on 3-7 July 2017 (`http://sites.ieee.org/netsoft-2017/`)
- at the Network of the Future 2017 conference, held in London, UK, on 22-24 November 2017 (`https://nof17.lip6.fr/`).

The ETSI NFV PoC proposal has been submitted in December 2017 and we received initial feedback in early February 2018. The feedback was very positive, but a few clarifications, in particular regarding future demonstrations were requested.An updated proposal is submitted and we expect acceptance in mid-February. Once it has been accepted, a report will be created taking into account the experience with the vCDN pilot as well as the discussions at the demo presentations.

4.5.2 ETSI Plugtest

The SONATA project, represented by ATOS, participated in one of the most prominent NFV interoperability events of the moment, the ETSI Plugtest. In addition to SONATA, in this second edition of the ETSI Plugtest, fifteen VNFs developers, ten MANOs providers, and seven NFVI suppliers have participated. In the pre-testing phase, the participants had the opportunity to make the required connections to the HIVE (ETSI VPN), as well as to do the first interoperability tests that consisted in generating descriptors and deploy, through the MANO, a VNF in an NFVI. SONATA passed several satisfactory tests in this preparatory pre-testing phase. During the Plugtest, it was necessary to create Network Services and VNF descriptors and their corresponding FSMs and SSMs for each testing session. These descriptors were developed in constant communication with the VNF vendors since the implementation strategy was different particular in each particular case.

As a result of the efforts carried out during an intense week of tests in Sophia-Antipolis, we can highlight the verification of functionalities that have been carried out between SONATA MANO, suppliers of NFVI, and VNF Vendors. SONATA started before the definition of the standards proposed by ETSI, so various features offered by SONATA are not strictly aligned with these definitions, and therefore some differences are reasonable. The exercise accomplished in the ETSI Plugtest has allowed us to draw a roadmap for future work and improvement of SONATA.

Some of the key takeaways are:

- The APIs defined between interfaces are not the same as those proposed by ETSI.
- The integration with external VNFM is different. In SONATA, this is done through Function Specific Managers (FSMs) and Service Specific Managers (SSMs) that are not defined in the ETSI.
- There are some low-level parameters of the VIM that have not been considered in SONATA, as the CPU Pinning or the SR-IOV.

5 Validation and evaluation

This section presents all the validations, apart from those conducted indirectly via the deployment and operation of the pilots. The section is divided in to parts. The first part attempts an additional system level testing to some components of the SONATA SP. The second part presents the validation of certain KPIs that relate to dominant features of the SONATA Service Platform.

5.1 System level evaluation

This section presents new results for the system level evaluation for some components as well as some steps and solutions towards performance enhancements.

5.1.1 Gatekeeper

This section describes the results of the Qualification tests for the Gatekeeper performance under heavy load. Further information about Qualification stress tests can be found at Deliverable [11] Stress tests for the Gatekeeper API section, where different sets of tests were defined to assess the performance of the main entry point to the SONATA Service Platform. Due to the huge number of variables that can interfere in the tests' results, these have been implemented and delimited to only analyse API performance based on the number of requests despite of the volume of data that responses might return. It is also important to note that some components have been implemented and integrated in later stage to the project, thus some Qualification tests might be not available.

Below you can find a list of Qualification stress tests applied to the Gatekeeper API and User Management API:

1. Test1 - Stress Functions API test (*getfunctions*)
2. Test2 - Stress Packages API test (*getpackages*)
3. Test3 - Stress Records API test (*getrecords*)
4. Test4 - Stress Requests API test (*getrequests*)
5. Test5 - Stress Services API test (*getservices*)
6. Test6 - Stress VIM API test (*getvims*)
7. Test7 - Stress Gatekeeper Public Key API test (*getpublickey*)
8. Test8 - Stress Microservices API test (*microserviceregistration*)
9. Test9 - Stress User Management Logs and Public Key API test (*umlog* and *umpublickey*)
10. Test10 - Stress Users API test (*userregistration*)

Each of the previous listed tests involve a chain of different components in order to obtain a result based on the entire process. For example, *getfunctions* test involves authentication components in order to retrieve a list of Virtual Network Function Descriptors. In the following sub-sections, performed tests are presented, with main results detailed based on the number of requests per second and showing the list of components that take part in the process chain. Then, global results of the tests are presented below.

Table 5.1: Gatekeeper stress tests and results

TestID	Test Name	Description	Components	Results (mean req/s)
Test1	getfunctions	Stress Functions API	API Gateway Function Manager User Management Keycloak Catalogue and Repositories API Keycloak DB Catalogue DB	| OK: 4.167 (100%) Total: 4.167
Test2	getpackages	Stress Packages API	API Gateway Package Manager User Management Keycloak Catalogue and Repositories API Keycloak DB Catalogue DB	| OK: 4.762 (100%) Total: 4.762
Test3	getrecords	Stress Records	API Gateway Record Manager User Management Keycloak Catalogue and Repositories API Keycloak DB Repositories DB	| OK: 6.667 (100%) Total: 6.667
Test4	getrequests	Stress Requests	API Gateway Request Manager User Management Keycloak Catalogue and Repositories API Keycloak DB Repositories DB	| OK: 5.263 (100%) Total: 5.263
Test5	getservices	Stress Services	API Gateway Service Manager User Management Keycloak Catalogue and Repositories API Keycloak DB Catalogue DB	| OK: 4.0 (100%) Total: 4.0
Test6	getvims	Stress VIM	API Gateway VIM API User Management Keycloak Keycloak DB Message Broker Infrastructure Abstraction API Infrastructure Abstraction DB	| OK: 4.0 (100%) Total: 4.0
Test7	getpublickey	Stress Gatekeeper Public key	API Gateway User Management Keycloak	| OK: 9.0 (100%) Total: 9.0
Test8	microserviceregistration	Stress Gatekeeper Public key	API Gateway Record Manager User Management Keycloak Keycloak DB	| OK: 2.5 (100%) Total: 2.5
Test9A	umlog	User Management Logs	User Management	| OK: 2.041 (100%) Total: 2.041
Test9B	umpublickey	Public Key	User Management Keycloak	| OK: 20 (100%) Total: 20

TestID	Test Name	Description	Components	Results (mean req/s)
Test10	userregistration	User Registration	API Gateway Record Manager User Management Keycloak Keycloak DB	| OK: 4.384 (100%) Total: 4.384

The following graphs show the performance trend as global results of the Qualification stress tests. These results are basically three: Mean response time, 95 percentile response time and percentage of requests KO (that failed). Test *getfunctions1* is the same as described for *getfunctions*, which is the entire process chain and the relevant test. Subsequent numbered tests are just parts of the main test, focused on isolating involved components. Usually, tests ended in *3* just involve one component. Results in Figure 5.1 show how the mean response time is really affected by different loads upon running processes on the platform, as results are variable and differ in each row, due to other tasks running concurrently in the Service Platform while performing the stress tests. The higher peaks belong to User registration requests (*userregistration*) and User Management log request (*umlog*). None of these tests, in the final stage of the project, showed KO results (see KO graph), despite long delay times. It can be observed that the number of tests here exceeds the initial list of tests in Table 5.1. This is the case, for example, for *getfunctions* that includes *getfunctions1*, *getfunctions2* and *getfunctions3*.

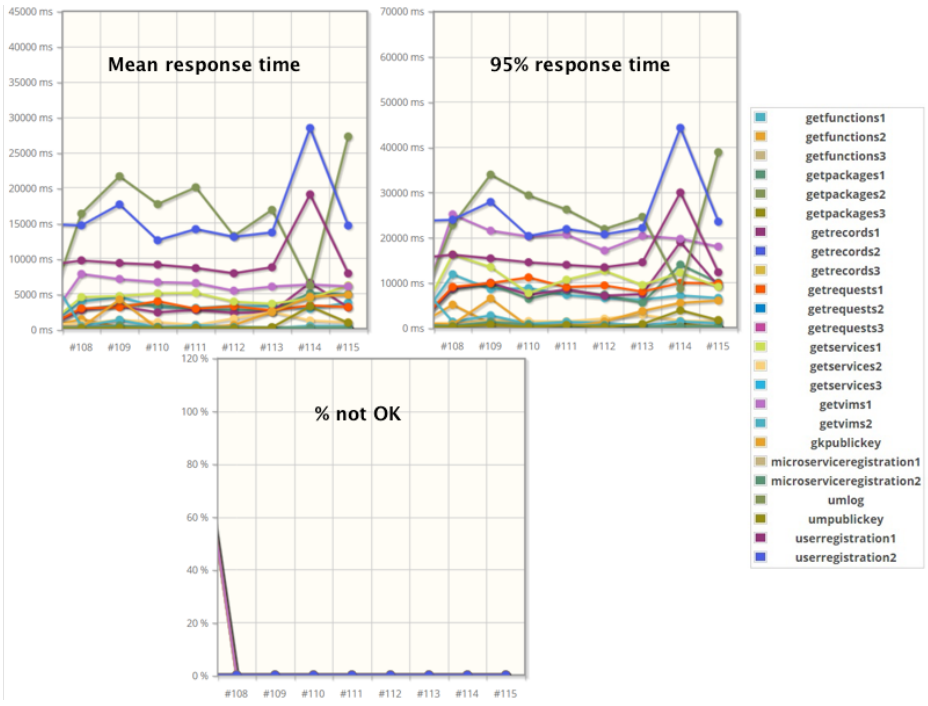

Figure 5.1: Gatekeeper stress tests

5.1.1.1 Performance improvement

The Gatekeeper is the SP component that is hit by external requests (see [2]), it is of paramount importance that its processes are of the highest efficiency possible, without compromising the needed flexibility.

There are two distinct areas where we think we can improve the Gatekeeper:

1. **Architecture simplification:** now that the needed features have been implemented, we know where we can simplify the architecture, making it more efficient;
2. **Caching of results:** an improvement that is listed in the improvements to be implemented since the very beginning of the development is the usage of a more aggressive cacheing strategy.

Each one of these two strategies is further detailed below.

Architecture simplification In Figure 5.2 (see [6]) shows the high-level architecture of SONATA's Gatekeeper API. It shows the unique access point to the whole API and how it uses different models to access the other micro-services.

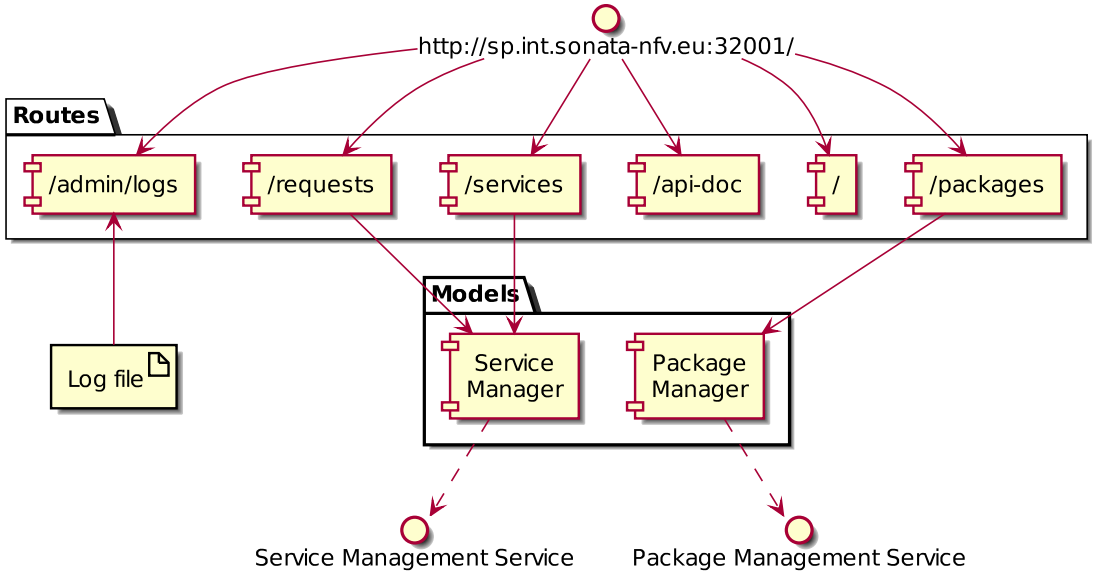

Figure 5.2: Gatekeeper's API high level architecture

The number of components grew up with the number of needed features grew (see Figure 5.3, adapted from [7]).

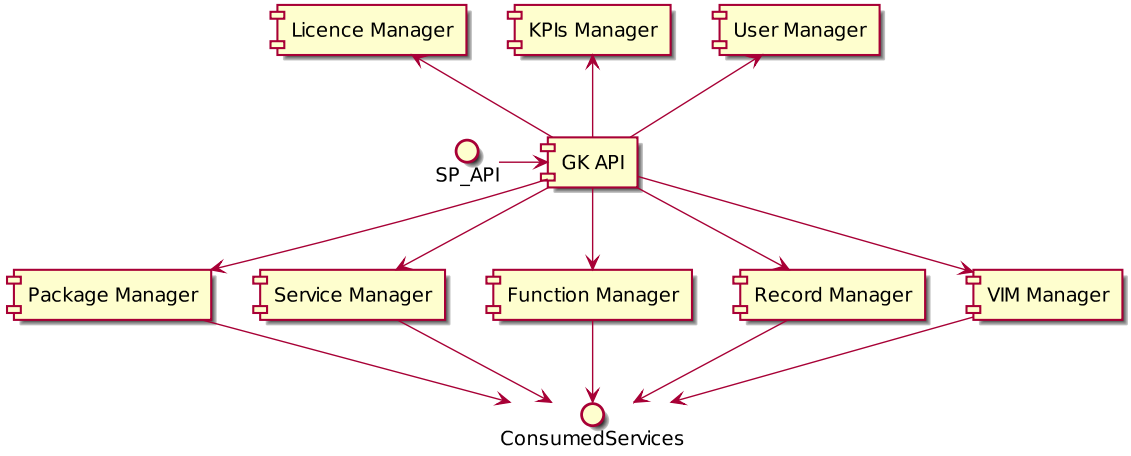

Figure 5.3: Modules with which the Gatekeeper's API has to interact with

But after using the Gatekeeper in testing and specially in building pilots, we have concluded that we can change a bit this architecture, minimising the number of containers supporting services that

clearly will grow in parallel and at the same time, thus minimising resource usage.

So, we're re-designing the Gatekeeper, to support an architecture such as the one shown in Figure 5.4.

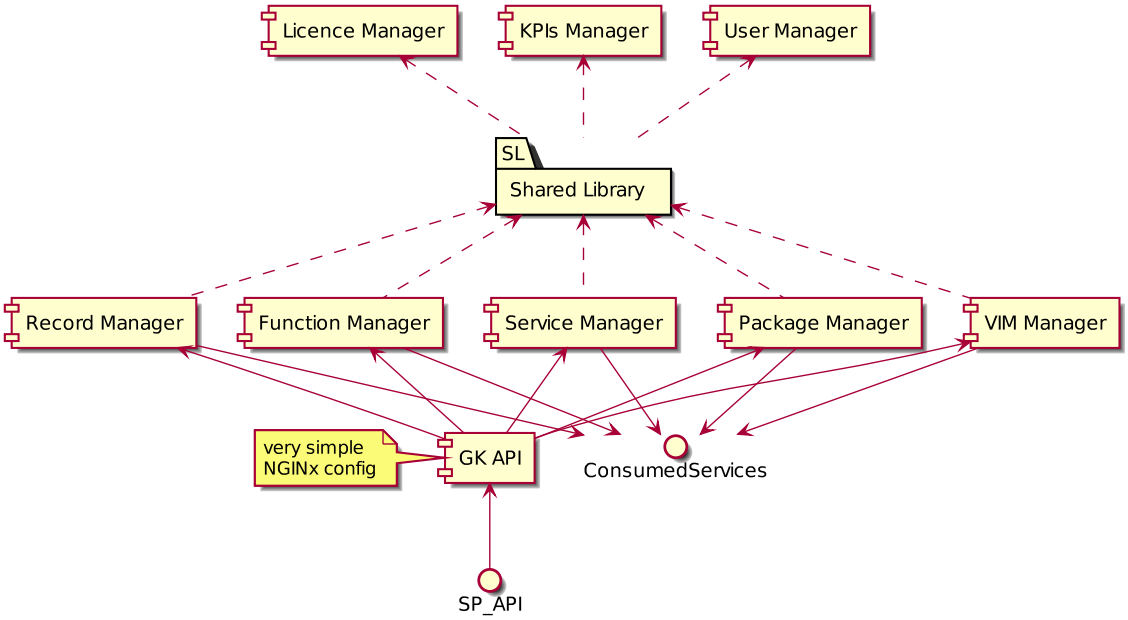

Figure 5.4: Gatekeeper's proposed evolution

This new architecture:

1. makes the **Gatekeeper API** a much lighter and easy to change component;

2. maintains each container's isolation, thorough the usage of a common library, to which a common set of classes and methods were moved.

Caching of results

Another commonly used strategy in web apps to increase performance is to **cache** commonly read results, so that the (frequent) requests do not need to hit the database rather than use local, preferably, in memory storage.

Caching data is more advantageous when it changes little (or even better, not at all) and is read often. In the SONATA's Service Platform we have such data:

- **Packages** (files and descriptors): being identifiable by the **vendor/name/version** triplet (see [2]), packages are not updatable (when there's a change, at least the version has to be updated and a new package generated);

- **User access tokens:** user access tokens are not changed and are read very often (at least one per operation), so cacheing it will provide significant time savings to all processes.

We can repeat the performance tests after having implemented these changes and, if considered needed, proceed to optimize the slowest operation found.

5.2 KPI evaluation

The following table (Table 5.2) presents the KPI evaluations considered for the evaluations campaigns of SONATA. Some KPIs are evaluated after specific testing (e.g. SDK) some other KPIs are evaluated throughout the work during pilot development and implementation.

Table 5.2: List of KPIs for evaluation

id	KPI	Description	Metric	Comments	Relevant Pilots
SDK.1	Network Service implementation and creation effort	Time required to create service descriptor(s)	developer effort (steps)	compare service creation workflows of SONATA, OSM, ONAP	
SDK.2	Network Service packaging time	Time required to package a service	time (sec)	test via various size of NS packages (different number of VNFs), compare to OSM packaging	
SDK.3	Testing environment setup time	Time required to setup a multi-pop testbed using the emulator	time (secs)	Extended D5.4. measurements: Set up 100 PoPs max, Figures about setup of realistic topologies taken from Internet TopologyZoo	
SDK.4	Test platform scalability	Memory consumption to emulate m PoPs	memory GB	Extended D5.4 measurements: memory consumption of emulator	
SDK.5	Test service deployment times	Time required to start n VNFs in selected real-world topologies, deploy hundreds of empty VNFs	time (secs)	New measurements	
SP. ORCH.1	Network Service Deployment time	Time required for a service to be instantiated	time (sec)	instantiate the service from the BSS and measure time	vCDN, vPSA, HSP
SP. ORCH.2	Self-contained logical administration of services	Time (sec) for a certain alert issued for a particular service component for the SSM to react and enforce and new policy	time (sec)	See below	vCDN, PSA
SP.GATE KEEPER.1	Network Service onboarding time	Time required to onboard a service	time (sec)	NSs are on-boarded within a package, so the KPI is measured in 'On-boarded Packages' (and not services)	vCDN, vPSA, HSP (all NSs are onboarded within a package)
SP.GATE KEEPER.2	Network Service number of on-boarding requests	Requests per min supported	number	NSs are on-boarded within a package, so the KPI is measured in 'On-boarded Packages' (and not services)	vCDN, vPSA, HSP (all NSs are onboarded within a package)
SONATA.1	Features SONATA Platform supports	Number of features	number	See below	vCDN, vPSA, HSP

id	KPI	Description	Metric	Comments	Relevant Pilots
SP.IA.1	Dynamic SFC update	Capability to update dynamically the SFC of a service functional, reconfiguration time (sec), measure time between SFC change requests	time (sec)		vCDN
SP.IA.2	Dynamic traffic steering	Enable the traffic steering between the endpoints and the PoPs selected for VNF placement	time (sec)	See below	vCDN
SP.MON.1	Monitoring of concurrent operational measurements and requests	Execution of tests measuring the scalability of monitoring framework with respect to the concurrent requests to the server asking data and the load of data that can be collected from the network services on the Push Gateway	time (ms)		vCDN, vPSA

5.2.1 SDK.1: Network Service implementation and creation effort

The SDK is one of the key innovations of SONATA and is one of the first of its kind. The overall goal of the SDK approach is to reduce the development efforts for service and VNF developers. This is required because highly complex, distributed network services, as they are envisioned for future 5G networks, are composed of many complex software components that need to work together in a jointly managed environment. Accordingly, the required effort to develop and compose network services as a network service developer is one of the key KPIs of an NFV SDK.

However, since development times heavily depend on the individual service developer and his skills, we chose to use the number of steps required to build a service as the key metric of this KPI. Using this, we modelled the service composing, packaging, and on-boarding workflow of SONATA's SDK. We compared this to similar workflows of other service platform solutions, namely OSM and ONAP. Even though, OSM does not come with its own SDK, the descriptor creation processes and packaging processes are similar to SONATA but without the availability of the SDK's supporting tools. On the one hand, a OSM network service developer packages, for example, a single VNF by manually creating a TAR file with the correct folder structure. ONAP, on the other hand, requires some pre-onboarding steps to generate the required metadata files of a VNF or service. These steps are automated in SONATA's SDK.

Table 5.3: Required network service development and on-boarding steps

Step	SONATA	OSM	ONAP
VNF pre-onboarding steps	no	no	yes
VNF image creation	yes	yes	yes
VNF descriptor creation	yes	yes	yes
VNF FSM / JuJu / GVNFM-API creation	yes	yes	yes
VNF packaging	no	yes	yes
VNF on-boarding	no	yes	yes
NS pre-onboarding steps	no	no	yes
NS descriptor creation	yes	yes	yes
NS SSM creation	yes	no	no
NS packaging	yes	yes	yes
NS on-boarding	yes	yes	yes

Another difference between SONATA and the other approaches is that SONATA packages the entire network service, including the VNFs, into a single artifact - a package. This simplifies the on-boarding procedure since a network service can be on-boarded in a single step. Based on these findings we modelled the steps of the network service development flows according to the Table 5.3. The Figure 5.5 evaluates this model for network services with up to 25 VNFs. It clearly shows that the number of steps required to develop a network service and on-board it can be significantly reduced by the use of SONATA's SDK.

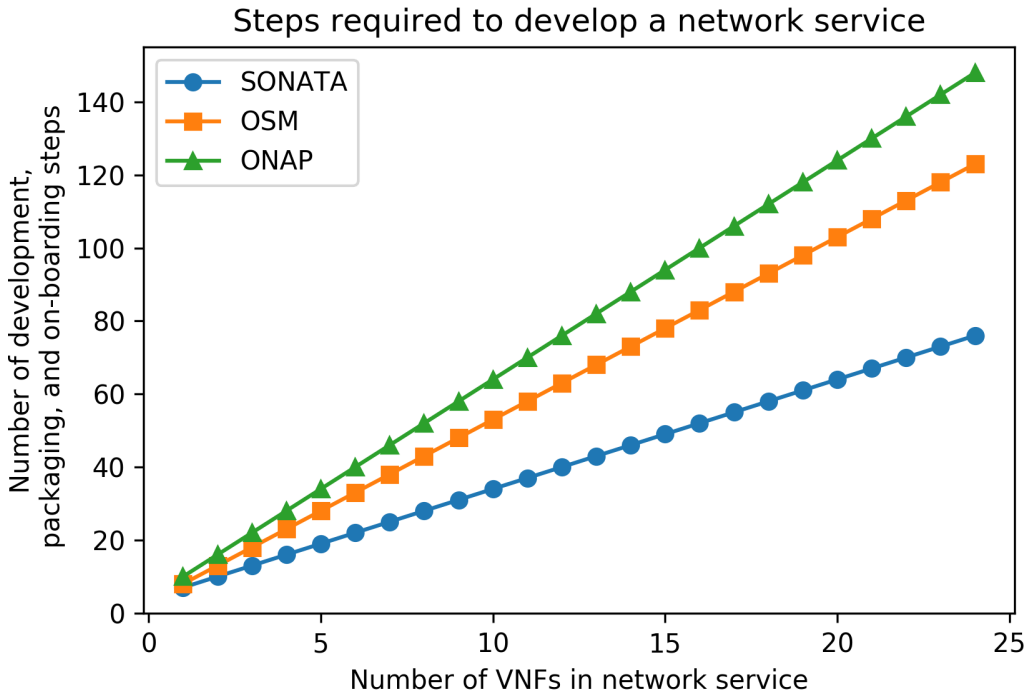

Figure 5.5: Comparison of required network service development steps for SONATA, OSM, and ONAP

5.2.2 SDK.2: Network Service packaging time

The key difference between SONATA SDK's packaging procedure and the packaging procedure of OSM is that SONATA's packaging tool automatically performs a series of validation steps to ensure that the packaged service and artifacts are aligned to SONATA's information model (son-schema). This obviously causes some overheads compared to a simplified packaging procedure without any additional checks, like done by OSM's packaging scripts. To quantify this overhead, we executed a series of packaging experiments for network services with up to 25 VNFs. The services have been packaged as slim packages, so only descriptors were included into the packages, no disk images. We used the `son-package` tool in version 3.0 as well as the packaging scripts provided by OSM 3.0 `generate_descriptor_pkg.sh`. All of these experiments (and the experiments of the following SDK KPIs) were executed on a machine with Intel(R) Core(TM) i5-4690 CPU running at 3.50GHz with 24GB memory and 200GB disk size (no SSD) and repeated 5 times. The Figure 5.6 shows the results of these experiments given as packaging time in seconds over the number of VNFs contained

in the packaged service. The results show that the SONATA packaging process is slightly slower than the OSM packaging scripts. However, this performance penalty of about 1 - 2 seconds allows SONATA to validate the packaged services which is a clear benefit. The measurements also show that the additional time for validation does only linearly increase with the number of involved VNFs.

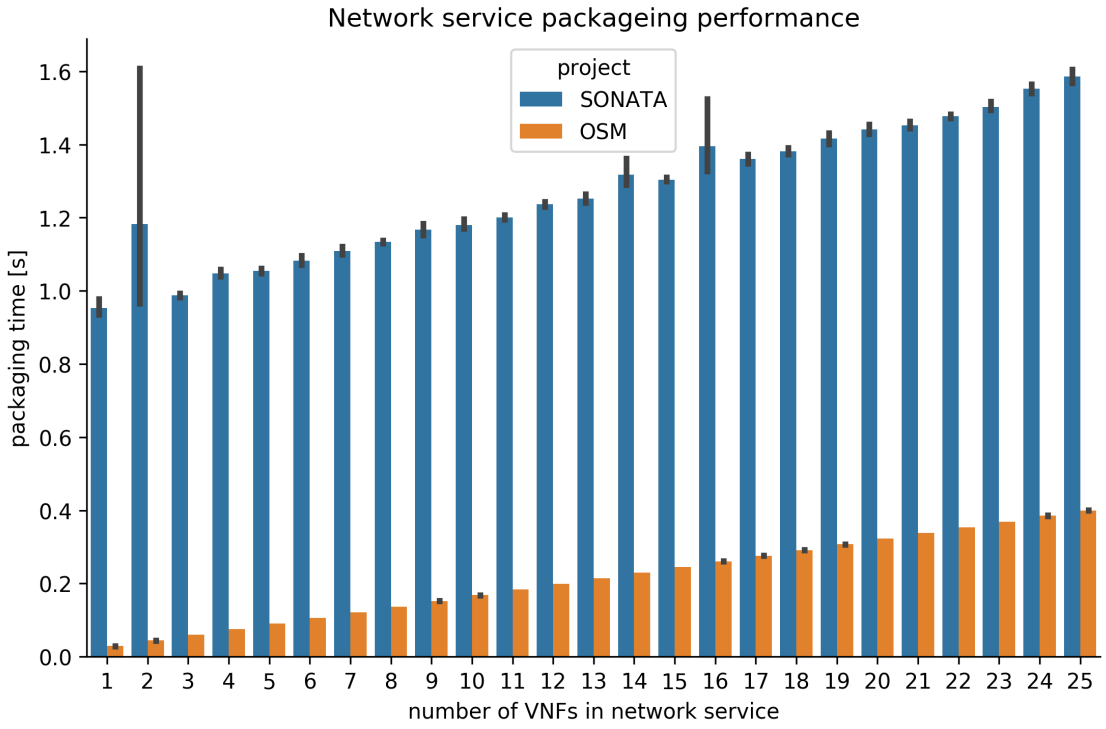

Figure 5.6: Comparison of packaging times for different network service sizes

5.2.3 SDK.3: Testing environment setup time

The emulator is one of the key components of the SDK and allows to emulate complex multi-PoP networks with an arbitrary topology, in which VNFs and services can be deployed as Docker containers. For more information about the detailed architecture of the emulator (see [4], [5] and [9]). One of the core use cases of this emulation platform is to quickly setup an environment for rapid service prototyping. This can be done by defining the multi-PoP topology to be emulated with a Python script that defines the PoPs and the links between them. After this, the emulation platform can be started and brings up the requested topology. Each PoP in this topology can have OpenStack-like APIs attached to it so that it behaves similar to a real OpenStack installation. Using this, the great benefit for the developer is the extremely fast setup time of the emulation platform compared to an installation of a federated OpenStack testbed which would take days (if not weeks) and requires substantial amounts of available infrastructure. Compared to this, this KPI evaluates the setup times of the emulation platform for different topologies to give the user an insight about the substantial time savings that can be achieved by using this tool.

The first experiment investigates the behaviour of the emulation platform for different sizes of the topology that is emulated. This experiment is based on results that have been presented in D5.4 [11] in more detail. To do so, the emulator was started with 1 up to 100 PoPs to be emulated. For each number of PoPs three different topologies were tested:

1. Linear topology: All PoPs are connected linearly to form a long chain of PoPs, e.g., PoP1 connects to PoP2, PoP2 connects to PoP3, and so on.

2. Star topology: The first PoP becomes the central PoP to which all other PoPs are connected in a star-like topology.

3. Full mesh topology: Each PoP has one dedicated link to every other PoP in the emulated topology. This simulates topologies with an extreme high number of links between the PoPs.

Again, our experiments have been executed on a single physical machine with Intel(R) Core(TM) i5-4690 CPU running at 3.50GHz with 24GB memory and 200GB disk size and have been repeated 5 times.

The Figure 5.7 shows the results of these experiments and shows that the total setup time of the platform for 100 PoPs is around 40 seconds for the linear and start topology and less than 250 seconds for 50 PoPs in a full mesh topology which has a considerable larger amount of links that need to be set up between the PoPs. To further investigate the setup behaviour of the platform, the total setup times can be broken down into the four phases:

1. Environment boot: Time taken to perform the basic boot procedure of the emulation platform.

2. PoP setup: Time taken to create all PoPs that should be part of the emulated topology.

3. Connection setup: Time to setup links between the PoPs to form the final network topology.

4. Topology start: Time taken to finally spin up and configure the emulation platform.

It can be seen that the *environment boot* process is constant and is not affected by the size of the emulated topology. The *PoP setup* and *Connection setup* phases, in contrast, are mostly responsible for the overall startup time required. The final phase linearly increases with the number of PoPs and interconnection links. The figure also shows that the time taken to setup the inter-PoP links heavily increases in the *mesh* topology case which is caused by the exponentially growing number of links.

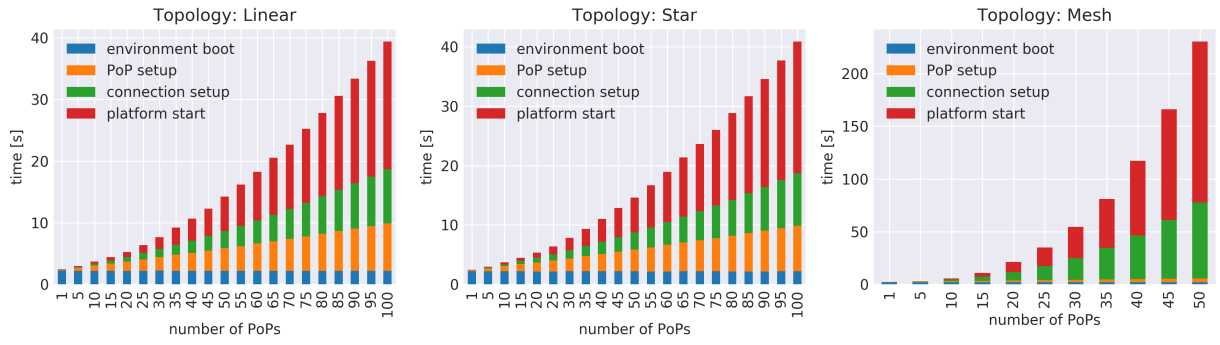

Figure 5.7: Emulator setup time breakdown for linear, star, and mesh topologies and different numbers of emulated PoPs

The second experiment does also evaluate the setup time of the emulation platform but focuses on real-world network topologies, like the famous *Abilene* network or operator networks of BT and Deutsche Telekom. These topologies have been taken from the *Internet Topology Zoo* project [16]

and are given as graphs that contain geographic coordinates for each of its nodes. The emulator can read such graphs and setup the topology accordingly. In this process, each node of the given graph becomes an emulated PoP. The capacities of the links between these PoPs is then automatically limited if bandwidth information are available in the given graph. Finally, the emulated links are configured with realistic delays, which can be calculated based on the geographic locations of the PoPs. The Figure 5.8 shows the setup times for a selected set of topologies from the *Internet Topology Zoo* library. However, the experiment has been done for more than 200 available topologies, but we focus on a selection of them to keep the document short and the figure understandable. The figure shows that all real world topologies can be setup in a couple of seconds and even large topologies with a high number of nodes, e.g., *USCarrier Telecom* has 158 PoPs and 189 links, can be setup in under 150 seconds.

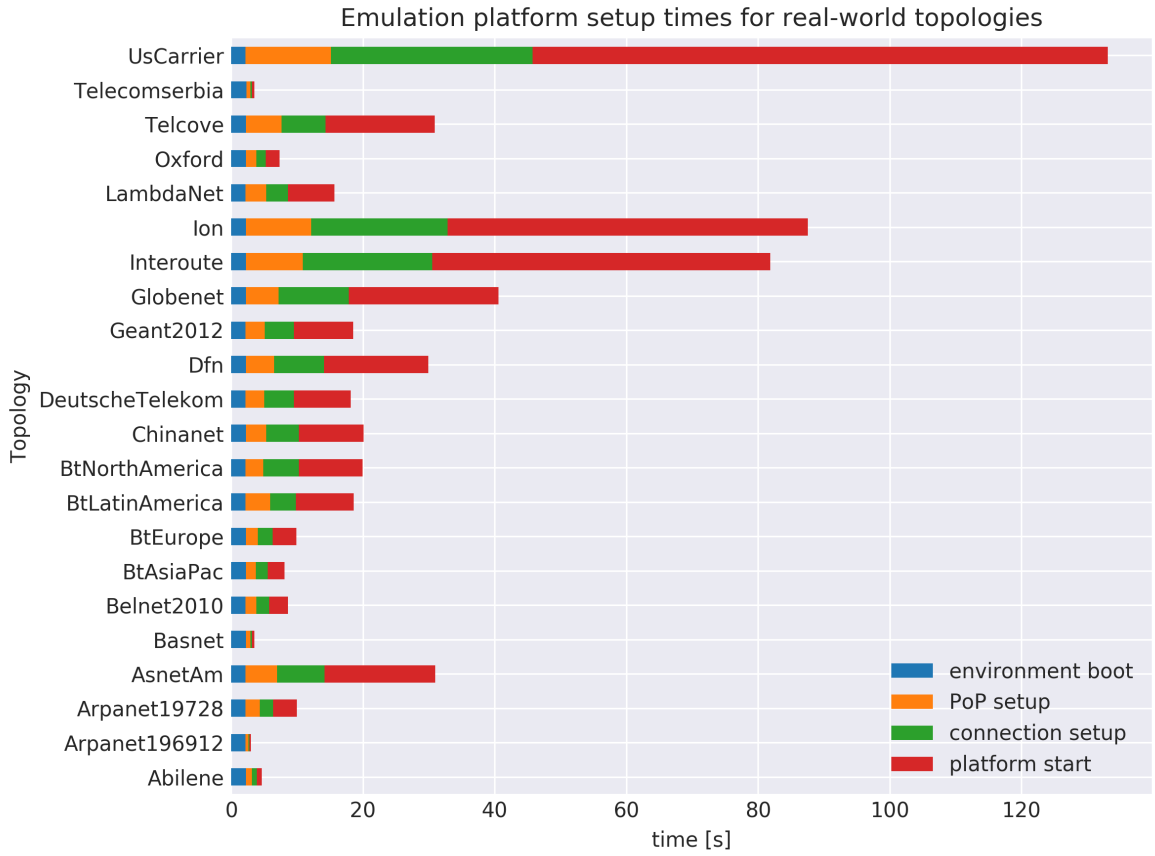

Figure 5.8: Emulator setup time breakdown for different real-world topologies

This KPI shows the benefit of the emulator as platform for rapid service development. Setup times of distributed multi-PoP testbeds can be reduced from days (or weeks) to seconds. In addition, a large variety of different real-world topologies can be considered.

5.2.4 SDK.4: Test platform scalability

This KPI focuses on the resource usage of the emulation platform as part of the SDK. As shown in SDK.3, the emulation platform is able to emulate very large topologies with more than 100 emulated PoPs. All this is done on a single physical machine which raises the question about the resource usage of the platform. To evaluate this, we chose the memory usage of the platform once

it is up and running but without services deployed on it. The reason for this is that the CPU and memory usage heavily depends on the VNFs that are executed in the emulated environment and thus do not give usable insights into the platform's resource requirements. The results shown here are based on the measurement setups described in SDK.3. The Figure 5.9 shows the memory consumption of the platform after setting up the *linear*, *star*, and *mesh* topology. It shows that the *mesh* topology requires a substantial greater amount of memory with an increasing number of PoPs. The reason for this is again the exponentially growing number of links in the topology. However, as long as PoPs and links grow with the same order of magnitude, the memory consumption of the platform scales linear in the number of used PoPs.

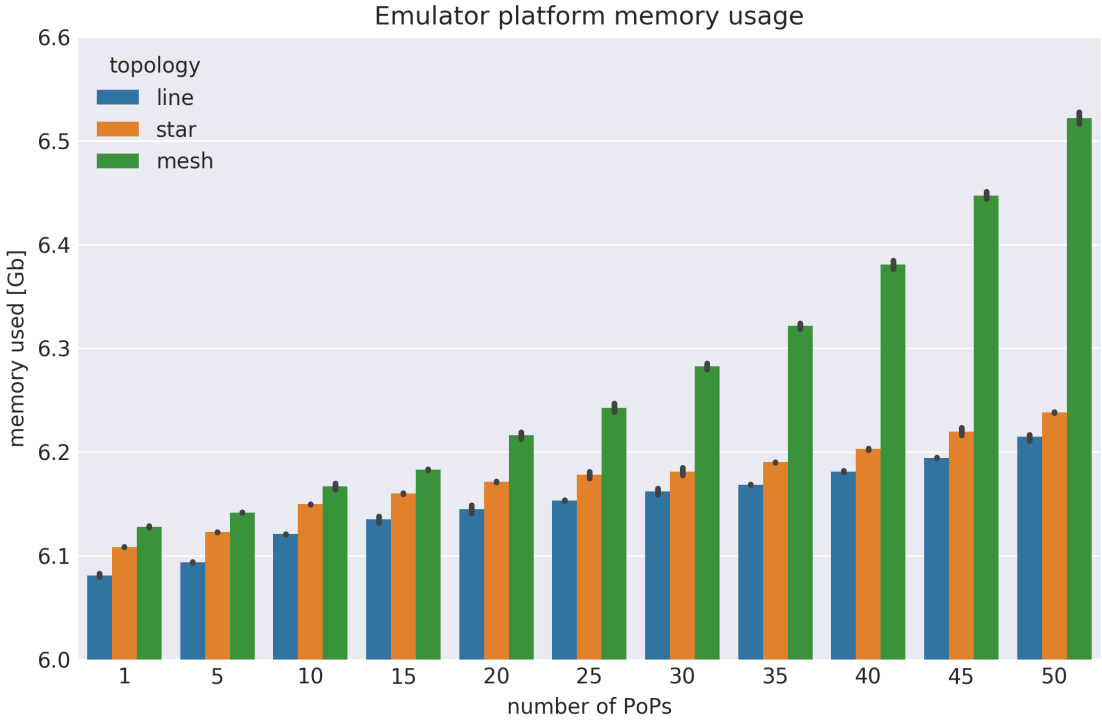

Figure 5.9: Emulator platform memory consumption for linear, star, and mesh topology

The Figure 5.10 also shows the memory consumption of the emulation platform but uses the real-world topologies that have also already been used in SDK.3. It shows that even very large topologies, like *USCarrier Telecom*, can easily be emulated with less than 10GB of RAM which is practically available in almost all modern workstations and even laptops.

5.2.5 SDK.5: Test service deployment times

The last KPI of the emulator platform that is investigated, is the time required to deploy network services of different sizes on an emulated topology. To measure this, we consider network services that consist of a huge number of container-based VNFs. These VNFs are based on an empty *Ubuntu 16.04* disk image and do not run any additional processing software, since we are only interested in the startup time of such services. For this experiment, we consider three different topologies with different sizes (*Abilene*, *Deutsche Telekom*, and *USCarrier Telecom*) taken from the *Topology Zoo* project [16] and already presented in SDK.3. All VNFs are placed randomly in these topologies when they are deployed. The Figure 5.11 shows the deployment times of these experiments for the three different topologies using a logarithmic scale. It shows that the time required to deploy a

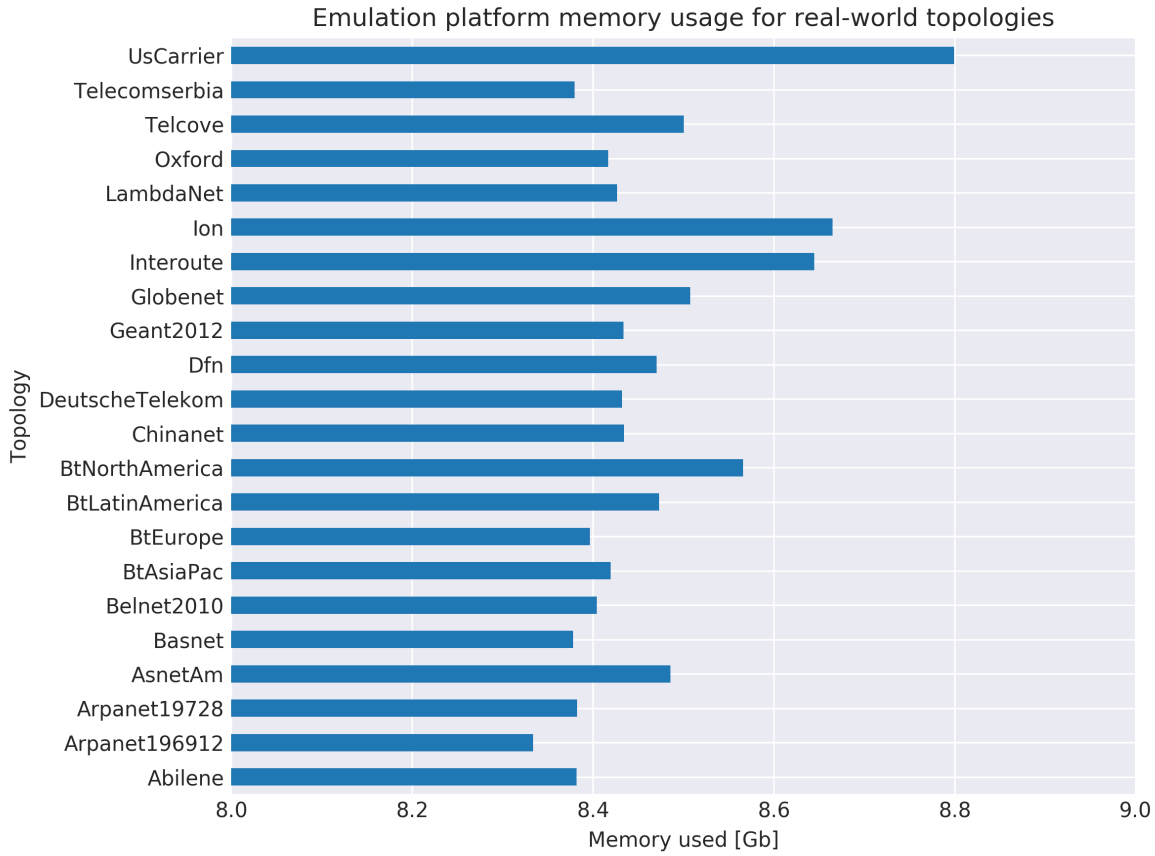

Figure 5.10: Emulator platform memory consumption for different real-world topologies

service linearly increases with the number of involved VNFs and that it is easily possible to deploy extremely large services (e.g. 256 VNFs) on our emulation platform, taking not more than 10 minutes. An interesting observation in this experiment is that the service deployment time also depends on the size of the emulated topology. This seems to be caused by the higher system load that appears when larger topologies are emulated (e.g. *USCarrier Telecom* has 158 PoPs).

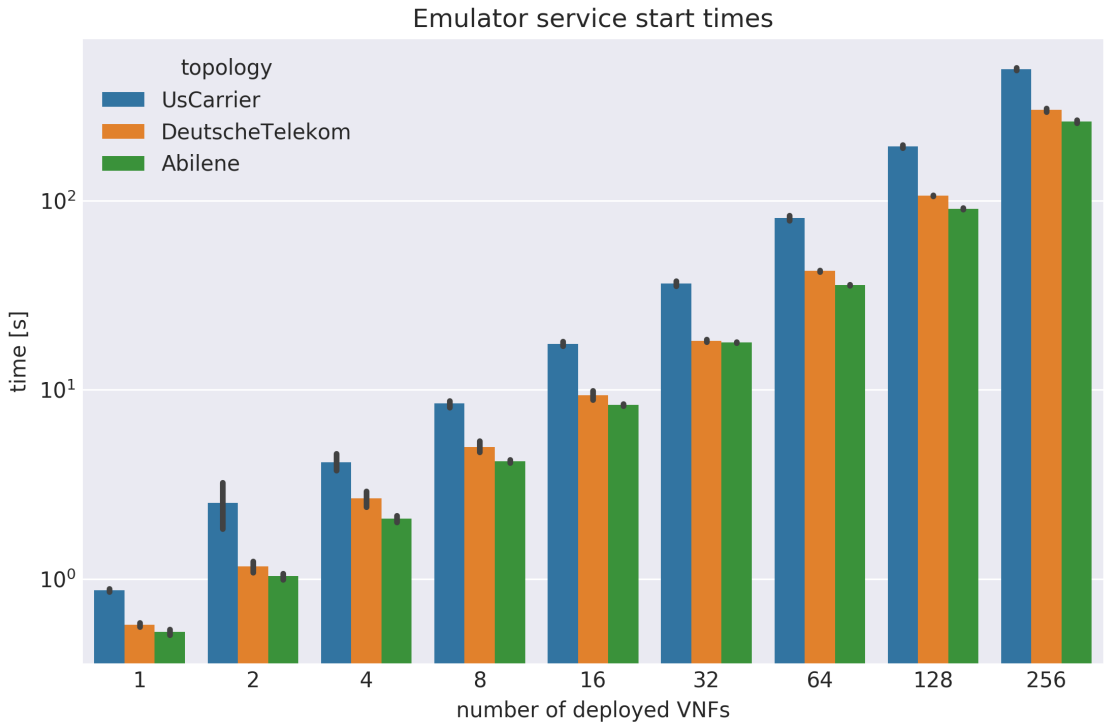

Figure 5.11: Emulator platform service deployment times over number of deployed VNFs in different real-world topologies with random placement

5.2.6 SP.ORCH.1 Network Service Instantiation time

This KPI measures the time in seconds required for a service to be instantiated, and evaluates the duration of the different steps during that network service instantiation. The resulting measurements are shown on Fig. Figure 5.12, Fig. Figure 5.13, Fig. Figure 5.14 and Fig. Figure 5.15. The network services used for these measurements are those from the SONATA pilots, and the results are deducted from the logs taken during the development of the pilots and averaged over +50 instantiation attempts.

Based on these figures, we can draw a set of conclusions:

- The biggest chunk of network service instantiation time is used to deploy the VNFs. This time is mostly spent on the VIM side. To deploy a VNF, a VM needs to be copied by the VIM from a database and then booted. Gains to reduce the deployment time of a single VNF could be obtained by increasing the hardware quality of the infrastructure (e.g. to increase the copying speed). The figures also show that the time for the VNF deploy phase increases with increasing VNFs. Since we use HEAT to orchestrate stacks in Openstack, we can't instantiate multiple VNFs of a network service on the same PoP in parallel, since HEAT only allows one stack update request at a time. Therefore, the VNF deploy phase

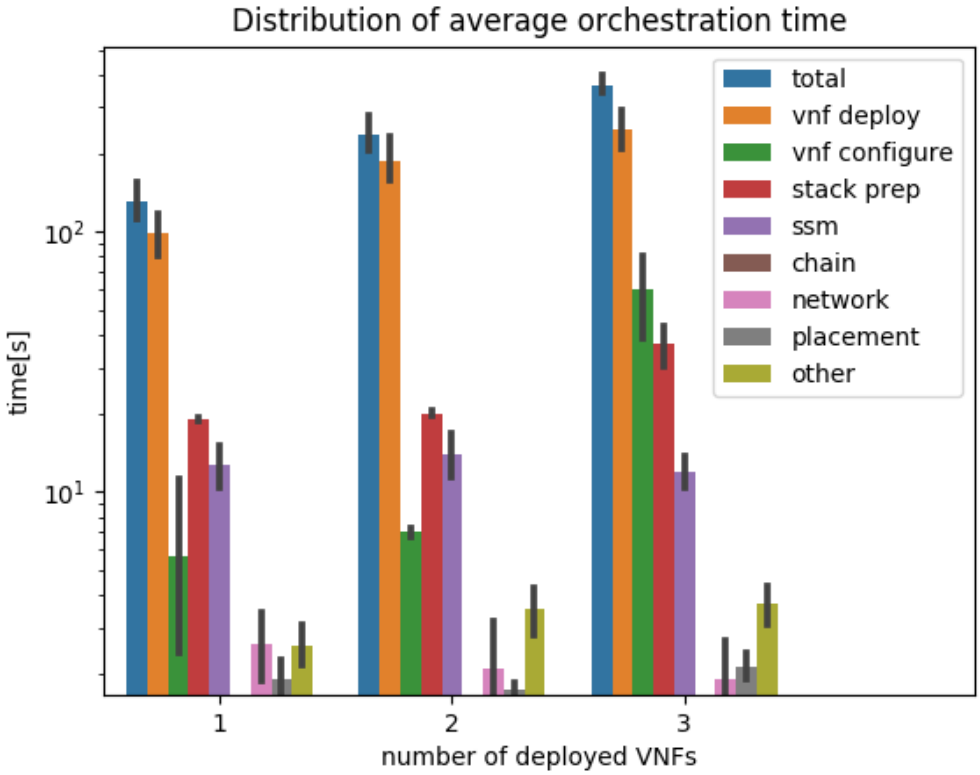

Figure 5.12: Average time taken for the different orchestration steps for a network service instantiation, with different number of VNFs

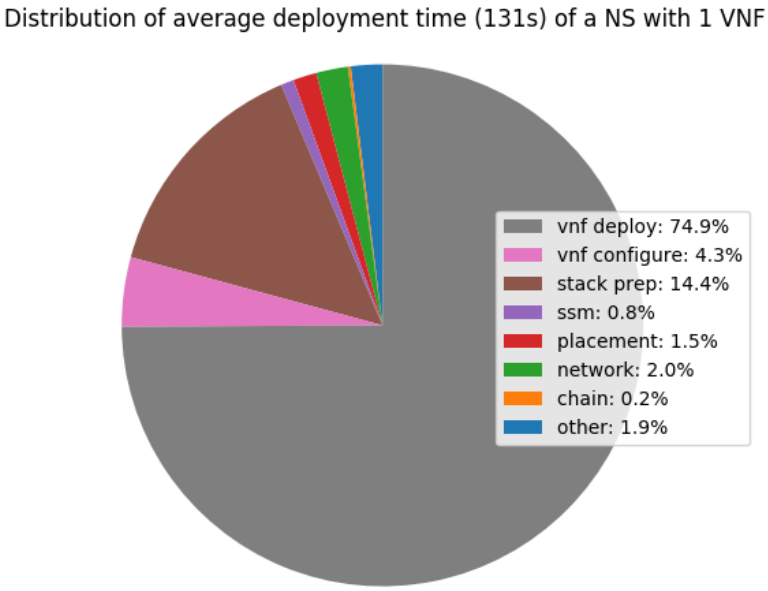

Figure 5.13: Time distribution for the different orchestration steps for a network service instantiation with 1 VNF

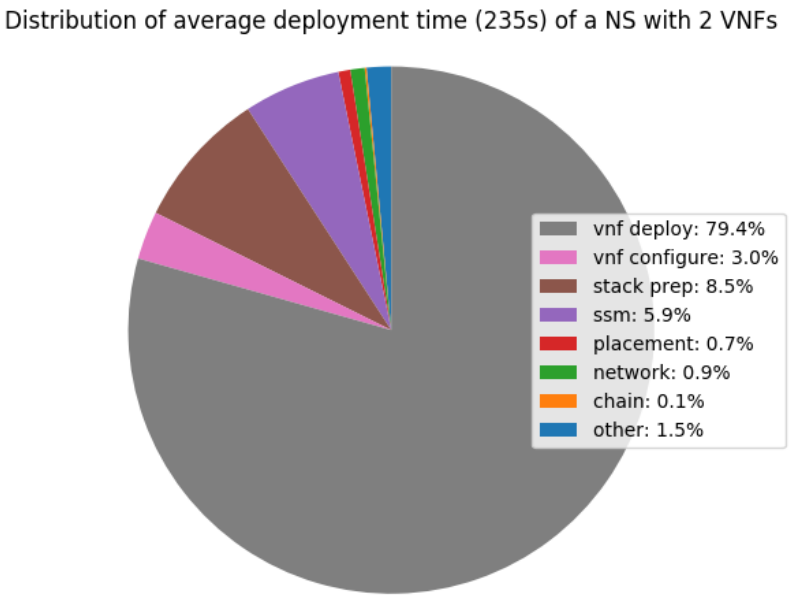

Figure 5.14: Time distribution for the different orchestration steps for a network service instantiation with 2 VNFs

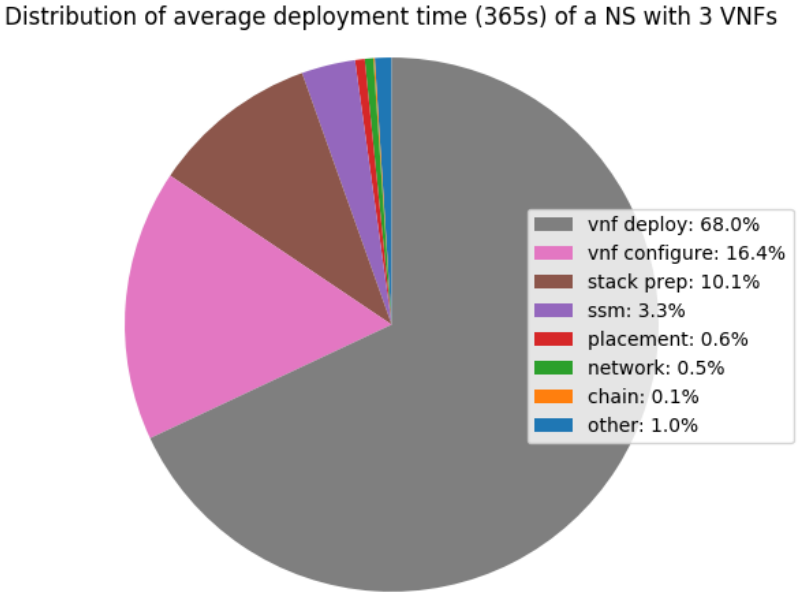

Figure 5.15: Time distribution for the different orchestration steps for a network service instantiation with 3 VNFs

will be approximately twice as long (a bit less due to some other tasks on the SP side that can be parallelized, such as FSM on-boarding and instantiation) for a network service that deploys two VNFs on the same PoP, instead of one. This can be seen when comparing Fig. Figure 5.13 and Fig. Figure 5.14, and on Figure 5.16. This trend is not valid when looking at Figure 5.15, since the duration of the VNF deploy phase per VNF is lower, compared to Fig. Figure 5.13 and Fig. Figure 5.14. This can be explained by looking at the network services used for these measurements. Among those with three VNFs, one category requires all VNFs to be deployed on the same PoP while another category spreads them among two PoPs. Therefore, only two VNFs need to be deployed serially for the second category of network services, while the third VNF can be deployed in parallel with the first two. This explains the shorter average duration of the VNF deploy phase for all network services with three VNFs. This becomes apparent when looking at Figure 5.16. The total VNF deploy time when the VNFs are instantiated over 2 PoPs is clearly lower then when all VNFs are instantiated on the same PoP. The relation between the number of VNFs (on one PoP) and the time for the VNF deploy phase appears to be linear (blue line) and closely related to the black line, which is the extrapolation of the number of VNFs multiplied with the duration of the VNF deploy phase for one VNF.

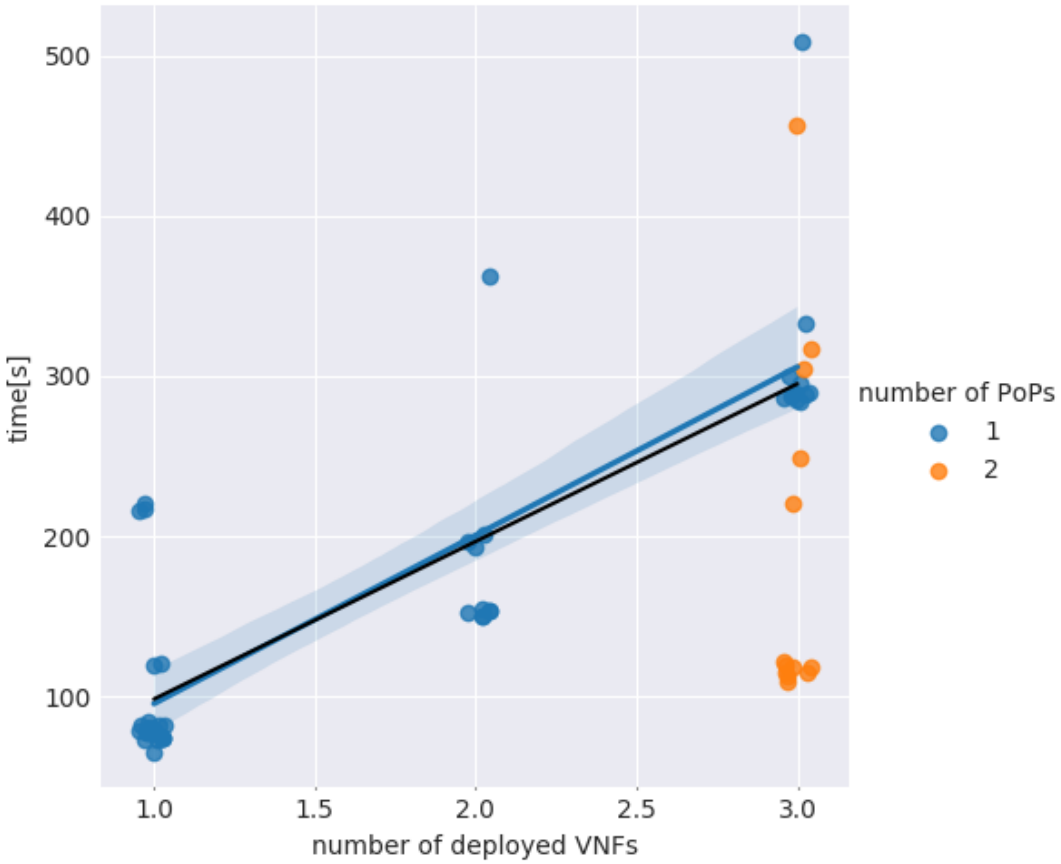

Figure 5.16: Time distribution for the VNF deploy phase for network services with different number of VNFs, deployed on 1 (blue) and 2 (orange) PoPs

- The duration of the stack preparation phase is equal for network services with one and two

VNFs, but increases when there are three VNFs. This time is mostly spent on the VIM side, where the stack is being created. The behaviour can again be explained by the used network services. Since one category of network services with 3 VNFs requires the creation of a stack on two PoPs, and the stack creation requests made by the SP to the VIMs aren't panellized, it is expected that the duration of the stack preparation phase will increase with the number of PoPs used in the network service.

- The values of the SSM phase, in which the SSMs are on-boarded, instantiated and interacted with, depend on the quality of their development by the network service developer. This makes it difficult to draw meaningful conclusions their distribution. The same goes for the VNF configure phase. During this phase, the already instantiated VNFs are configured through the FSMs that were provided by the VNF developers. These FSMs are sand boxed processes in which the VNF developers have complete operational freedom. The time these FSMs take to complete the configuration (which happens in parallel) greatly depends on how many configuration steps are needed for the VNF and how well these steps are coded in both the FSM and the VNF VM.

- Chaining VNFs, setting up the network between source, destination and involved PoPs and calculating the placement calculations (which in SONATA are basic) take up an insignificant fraction of the total network service instantiation time.

5.2.7 SP.ORCH.2: Self-contained logical administration of services

In SONATA, NS and VNF developers are allowed to customize the orchestration behaviour of the service platform. This allows them to further improve the performance of their products. Developers can add Service Specific Managers (SSM) and Function Specific Managers (FSM) to their descriptors. Such SSMs and FSMs are processes in which the developers have complete functional freedom. It is difficult to measure the quality of this specific manager mechanism, since the performance depends greatly on the quality of the SSMs and FSMs, which are coded by the developer.

Figure 5.17 shows the duration of multiple types of placement logic for a NS instantiation. The first group of NS uses the generic placement mechanism provided by the SONATA SP. The second category of NS uses a Placement SSM (i.e. a customized placement algorithm provided by the developer) to calculate how the different VNFs of the NS should be embedded on the infrastructure. The NS used for this experiment are those used in the SONATA pilots. Since the customized placement algorithm used in the Placement SSM of the second category of NS (i.e. vCDN pilot) is very similar to the generic placement algorithm used in the SP, we can use the durations of the placement phase to get a feel for the overhead caused by using an SSM to overwrite generic SP behaviour. Looking at Figure 5.17, we see that the placement phase for a customized placement algorithm is a bit longer than the generic SP placement mechanism. In the SONATA SP, Placement SSMs are not communicating directly with the SLM, but with a mediating component that forwards the requests/responses between the SLM and the SSM. The role of this mediator is to filter/block malicious intends of the SSMs. This means that, compared to the generic placement workflow, the customized placement workflow requires two additional network calls and one additional component processing the communication. This should explain the difference in duration of the generic and the customized placement phase shown on Figure 5.17.

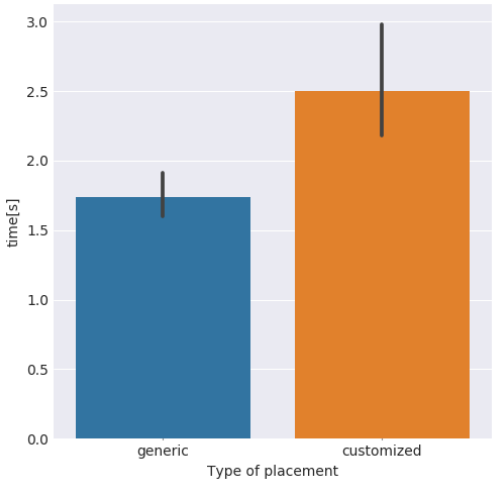

Figure 5.17: Duration of the placement phase during NS instantiation, generic vs customized

5.2.8 SP.GATEKEEPER.1: Network Service on-boarding time

This KPI measures the **time required to on-board a service**, measured in **seconds**. As explained in [2], SONATA has adopted to **package** the **Network Service Descriptor** (NSD) together with the **Virtual Network Function Descriptors** (VNFDs) of the VNFs that implement the service, the **Virtual Network Function Forwarding Graph** (VNFFG) and other related assets. We therefore measure here the time it takes to on-board the whole package.

Please check the following section for the detailed data and analysis.

5.2.9 SP.GATEKEEPER.2: Network Service number of on-boarding requests

This KPI shows how many on-boarding requests the Service Platform can handle per minute.

Using the SONATA sample package, with random generated content, a series of stress tests have been applied. In the following charts, we took 10 results samples, which are ordered as *rows*. Then, an average of these samples is presented on the last chart. In each test row, a growing number of concurrent packages are on-boarded in order to push the API to the limit. These numbers are: 10, 100, 200, 400, 450 and 500 concurrent packages. The following charts show the **on-boarded packages per second** (posts/sec), which is the rate of packages processed by the API, and the **elapsed time** in seconds (secs) which is the total time taken by the API to process all the packages.

The detailed results can be seen in the Appendix A.

The following table shows the average times for the on-boarded packages per second rate and the elapsed time, for 10 row samples. It also shows the minimum and maximum values per each timing. Looking closer to this later values, *min posts/s* values can be considered the worst package on-boarding rates. The same way as *max posts/s* values can be considered the best package on-boarding rates. Then, the minimum and maximum elapsed time values (*min secs* and *max secs*) show the best and worst times respectively, for the time taken to process the number of concurrent packages at the API.

The value for packages per minute (*packages/min*) is a theoretical approximation to the number of on-boarded packages at API during 60 seconds (a minute) at different concurrent packages numbers, which is calculated from the average on-boarded packages per seconds (*posts/s*) multiplied per 60 seconds.

Rows Summary

#Concurrent entries	avg posts/s	avg secs	min posts/s	max posts/s	min secs	max secs	packages/min
10	13.525	0.964	2.958	18.641	0.536	3.370	780
100	38.598	2.958	16.297	54.122	1.847	6.135	| 2280
200	43.895	4.775	23.780	58.615	3.412	8.410	2580
400	39.361	11.493	20.399	55.125	7.256	19.607	2340
450	27.864	19.174	11.098	49.379	9.113	40.544	1620
500	n/a	n/a	n/a	n/a	n/a	n/a	n/a

With 500 concurrent entries, the greater number of concurrent submitted packages, the API service started to fail. Figure Figure 5.18 shows the evolution of the number of posts per second and the elapsed time to process each package, with the number of submitted packages. This *hard-limit* on the number of packages still needs to be investigated, but it is typical when a system hits a pre-defined constant, like the maximum number of files opened. Nevertheless, simultaneously on-boarding this very high-number of packages (containning each one a service, as explainned) is extremely unlikely.

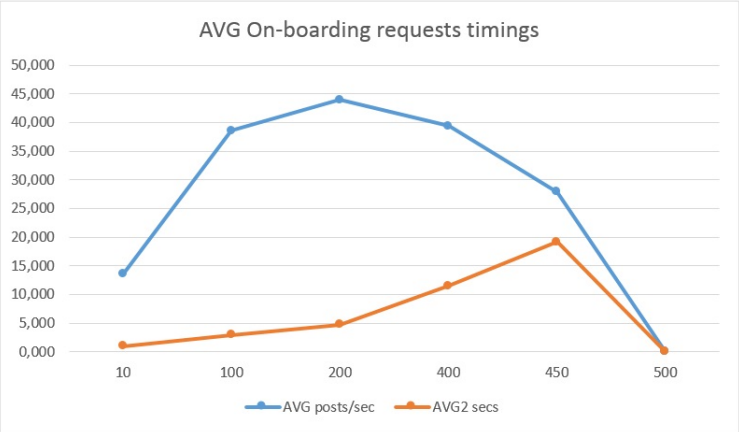

Figure 5.18: Evolution of the number of posts per second and the elapsed time to process each package, with the number of submitted packages

As can be seen in figures Figure 5.19 and Figure 5.20, in the Qualification stress tests, the best API's performance rate is achieved when the rate of concurrent on-boarded packages reaches 200, where the peak number is of 58 on-boarded packages per second. However, elapsed time results grow as concurrent on-boarded packages number grows, which is an expected result.

5.2.10 SONATA.1: Features SONATA Platform supports

Please see Appendix A for an extensive description of which features have been implemented.

5.2.11 SP.IA.1: Dynamic SFC update

This KPI measures the time in seconds between SFC change requests, given the capability to dynamically update the SFC of a service.

The chosen approach to the vPSA pilot was to implement it as a L3 service. Therefore, there is no network chaining, an no dynamic update of that chaining. At the Infrastructure Abstraction level, dynamically changing the SFC is supported, but it wasn't as heavily used as other features. Therefore, there is no data that captures the performance of dynamic SFC updating. We believe it will be a very fast operation when compared to others that have to be performed, since updating the chain is just a call to the 'scf_agent.py' script.

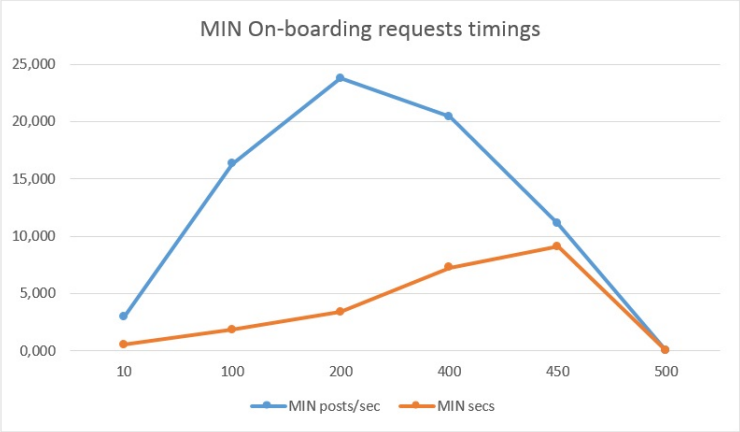

Figure 5.19: Evolution of the minimum numbers of posts per second and the elapsed time to process each package, with the number of submitted packages

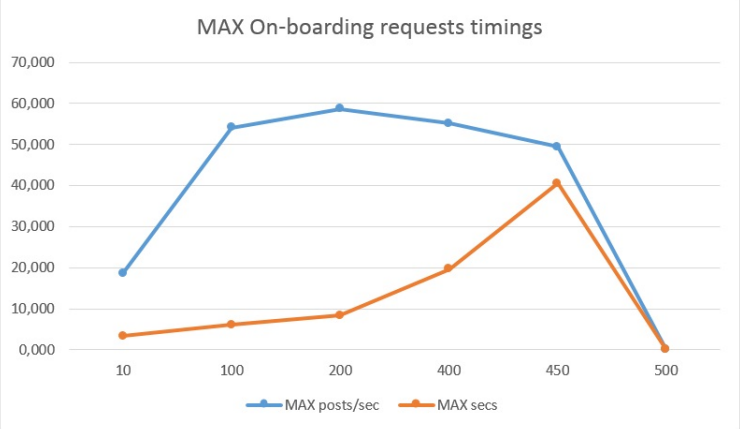

Figure 5.20: Evolution of the maximum numbers of posts per second and the elapsed time to process each package, with the number of submitted packages

5.2.12 SP.IA.2: Dynamic traffic steering

This KPI measures the time taken to fully configure the end-to-end (traffic steering between the end-points and the PoPs selected for VNF placement) network of a NS. This is the time the networking (i.e. configuring the WAN between PoPs, NS endpoints and source/destination) phase requires in SONATA's instantiation phase. For this KPI we can look into Figures Figure 5.12, Figure 5.13, Figure 5.14 and Figure 5.15 above (inspite of the data shown was measured in the SLM, the remaining time being negligible -- the time the IA takes to process the request, plus the communication time between SLM and IA, which should be milliseconds).

Take Figure Figure 5.21, a zoom in into the network part of Figure 5.12 above.

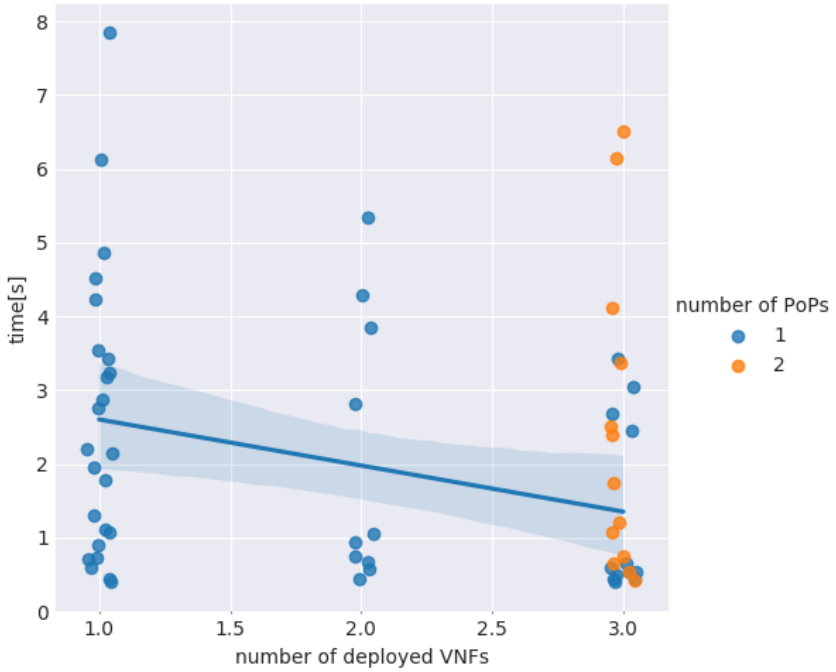

Figure 5.21: Time distribution for the network part only of different orchestration steps for a network service instantiation with 3 VNFs

In this picture one can see that:

1. for single PoP deployments, network services with a different number of VNFs all have more or less the same duration for the 'network setup'-phase;
2. For the 2-PoP deployment, it takes a bit more time compared to the single PoP deployment (for 3 VNFs).

Both things are expected, since the first is all single PoP, so no networking rules, and for the second statement, inter PoP communication requires an extra rule, so it should take a little longer.

5.2.13 SP.MON.1: Monitoring of concurrent operational NS

Monitoring is a critical service provided by the SONATA Service Platform that covers the needs of several stakeholders, including network service developers, seeking for data and information to evaluate and optimize the performance of their deployed services; infrastructure providers looking

for cloud resource optimization models; and service providers willing to assure the signed SLA. To achieve these goals, monitoring framework must be reliable and scalable in a carrier-grade sense.

This section provides insight on several tests executed on the SONATA Service Platform in order to validate the scalability of the monitoring framework. There are two sets of tests that have been executed:

1. **Tests for evaluating the performance of the Monitoring Manager**

The purpose of this set of tests is to stress the response time of the Monitoring Manager API boosted by bursts of HTTP GET methods from users asking for monitoring data. The tests provide mean value for the connection time and the processing time of the packets under different testing conditions. The following tests have been performed:

- Send concurrently 100 requests until the total number reaches 1000.
- Send concurrently 250 requests until the total number reaches 1000.
- Send concurrently 500 requests until the total number reaches 1000.
- Send concurrently 750 requests until the total number reaches 1000.
- Send concurrently 1000 requests until the total number reaches 1000.

The Figure 5.22 shows the mean value of the connection and processing time accumulated in the overall request execution time. As it can be seen, the values are linearly increased, while the transmission rate remains constant at the value of 150 packets per second.

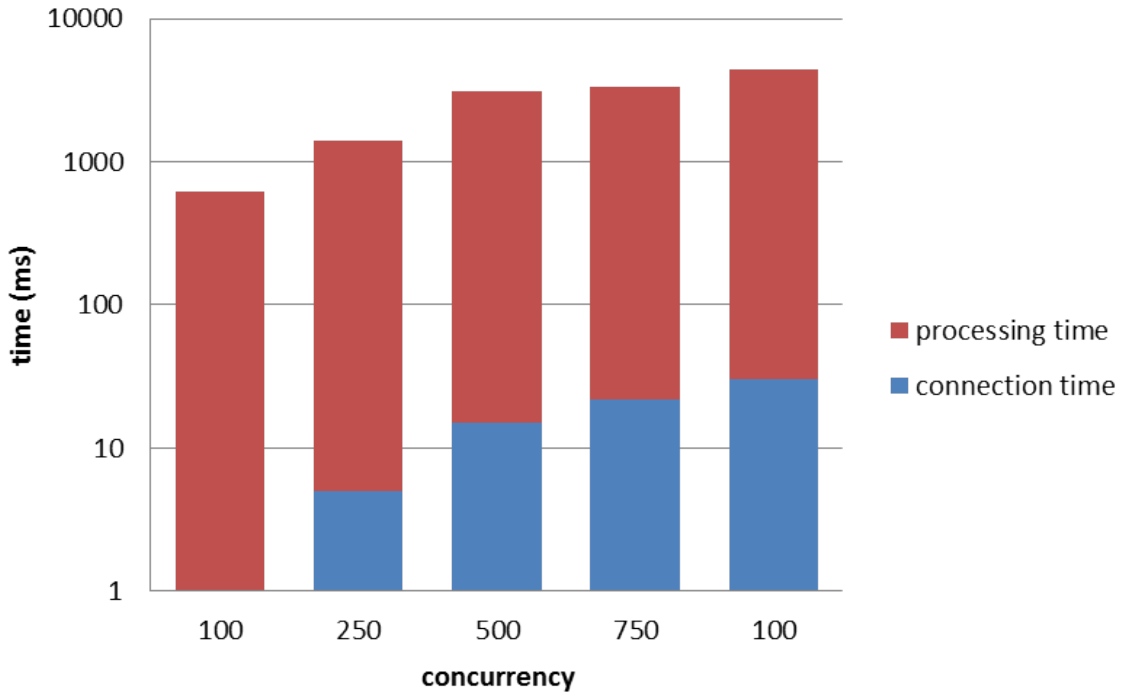

Figure 5.22: Time for executing concurrent requests

2. **Tests for evaluating the performance of the PushGateway**

The purpose of this second set of tests is to evaluate the performance of the monitoring manager with respect to the data collected from network services. As in the previous set, the tests provide

mean value for the connection time and the processing time of the packets under different (and more demanding) testing conditions. The following tests have been performed:

- Send concurrently 250 requests until the total number reaches 10000.
- Send concurrently 500 requests until the total number reaches 10000.
- Send concurrently 750 requests until the total number reaches 10000.
- Send concurrently 1000 requests until the total number reaches 10000.

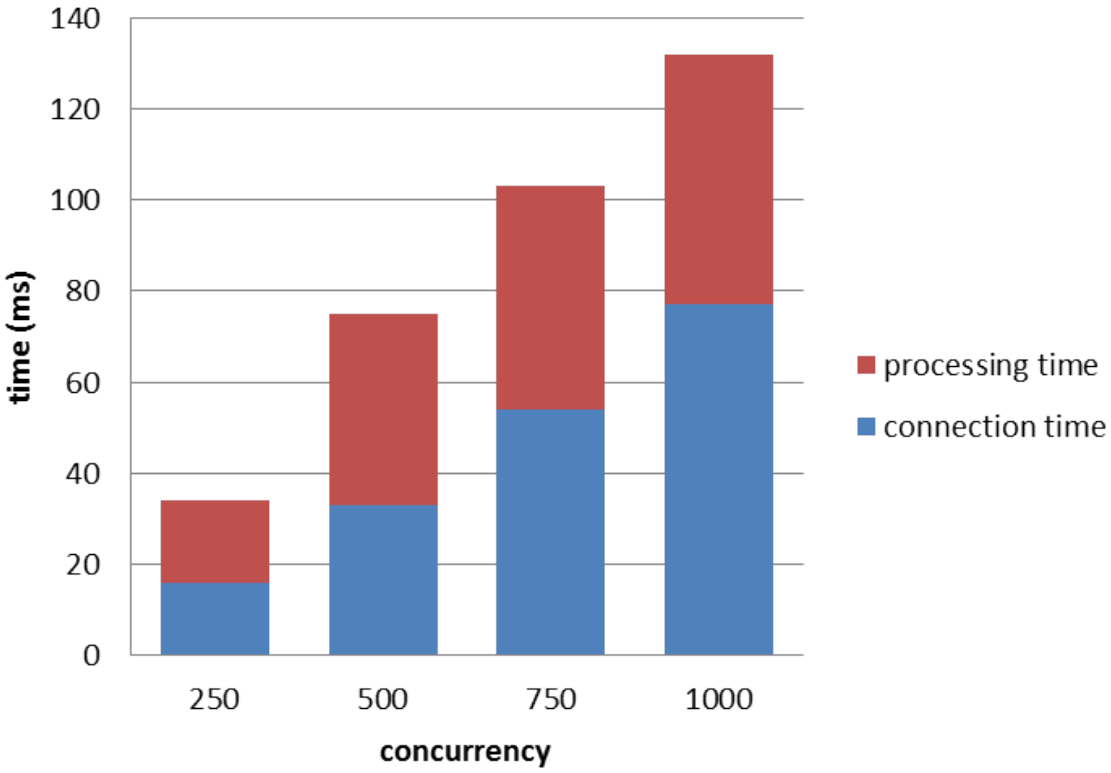

Figure 5.23: Time for collecting monitoring data concurrently

As shown in Figure 5.23, also in this case the values of connection and processing time is increasing in a linear fashion.

As a general remark, it can be concluded that the monitoring manager can serve hundreds of concurrent requests from users, while the Push Gateway is capable of collecting monitoring data from tens of thousands of VNFs concurrently.

6 Lessons learned and Future Work

This section lists a set of the encountered challenges, insights and resulting learnings gained while implementing and integrating the SONATA Service Platform using a continuous integration and delivery DevOps methodology. This is the result of a strong cooperation between prominent equipment vendors, network operators, software companies and universities, providing a set of constructive recommendations in hope of catalysing the development and deployment of NFV platforms. This section (apart from the final sub-section) is based on a paper [21] submitted by the some of the partners of the SONATA consortium to the experience track of IEEE/IFIP Network Operations and Management Symposium (http://noms2018.ieee-noms.org/). The paper has been accepted for publication in December 2017 and will be presented/published in the week of 23-27 April, 2018.

6.1 The NFV Orchestrator strongly benefits from a task-oriented implementation

To be in accordance with the ETSI, SONATA included the development of an NFV Orchestrator (NFVO). The workflows are implemented by the Service Lifecycle Manager (SLM). Looking at the service instantiation workflow, the SLM uses the placement plugin to calculate the placement, the Function Lifecycle Manager and Infrastructure Adapter to deploy VNFs and a storage component to save their records. Due to the variety in management and orchestration requirements from services, a straight-forward implementation of these workflows quickly became complex. Different services require different placement algorithms, different scaling solutions and in SONATA, services might come with Service Specific Managers (SSM), processes provided by the service developer that customise the SLM workflow. Implementing each workflow as a single process/thread overloaded the code with if-else clauses and boilerplate, and made extending them a complex task. This made the SLM an inflexible object, contradicting the SONATA requirement for a flexibility and extensibility. To this end, we re-factored the SLM into a task-based engine.

The overall functionality of the SLM was chopped into basic tasks, with each task implemented as a separate thread of control. Workflows are then established by chaining a subset of these tasks together into an ordered schedule, in accordance with the MANO requirements for the service. SSMs can now customise a workflow by overwriting the functionality of one or more generic tasks, or by adding/removing tasks in the schedule. Extending the functionality of a workflow (e.g. interacting with a newly added SP plugins) in the SLM now comes down to implementing new tasks, and inserting them in the workflow schedule. The complexity of the code base reduced, making the SLM more easily maintainable and extendible. This transition made the SLM, and thus the NFVO, the flexible component SONATA requires it to be.

6.2 The monitoring framework needs to be flexible and scalable

A service platform will collect information about many aspects of its performance - the individual elements of the physical and virtual infrastructure, the VNFs, the network services, and so on. Thus, monitoring information is of interest to several parties in the ecosystem, including: the operator of the service platform (to gain a deeper understanding about their service); the end

customer (to check that their service level agreement is being met); vendors (who are responsible for various VNFs); network service developers; machine learning specialists (to optimise algorithms that use monitoring data to automatically identify and isolate faults). These factors suggest that the monitoring framework should have two complementary features, it should be flexible and scalable.

Flexibility is needed as different parties will be interested in different information. E.g, an operator needs real time information about all the different components; a customer is only interested in summary information. So the framework needs to allow the interested party to describe what they want, and for the owner of that information to approve, adjust or reject the request. In our implementation, the monitoring framework collects and processes data from several sources (VMs, Docker containers, OpenDayLight controllers, OpenStack API, etc.), providing the interested parties the ability to activate metrics and thresholds to capture infrastructure or service-specific performance data. The user can define rules based on metrics gathered from multiple VNFs deployed in one or more NFVIs. In general, the user can subscribe to a message queue to get the real time alert notifications and monitoring data, request them through a RESTful API or directly access them through a web socket URL.

Scalability is needed because there is potentially a vast amount of monitoring information, so it cannot all be communicated or stored in full. Approaches that help scalability include: thresholds (only report a metric when its value exceeds some value); averaging, filtering and other aggregation techniques (e.g. averaging information over some time period); creating tailored alerts (for instance, so a help desk can be pre-warned that there is a problem affecting a service); and an 'emergency button' so the service/network manager can quickly reduce to 'skeleton monitoring' (e.g. if some failure means that monitoring data suddenly consumes a significant fraction of resources). To address scalability, several monitoring components in our implementation had to be distributed across NFVIs. First, each NFVI needs its own web socket server to accommodate users' requests for streaming monitoring data. Second, monitoring servers follow a federated architecture. The local servers collect and store metric data from the VNFs deployed in the NFVI, while only the alerts are sent to the federated level for further processing/forwarding to the user. Third, alerting rules and notifications can be based on monitoring data from multiple NFVIs and thus should be evaluated on a federated level. To enable 'skeleton monitoring', the design of the monitoring probe allows dynamic modification so that in cases where the difference of a monitored metric is below a threshold, it will not be sent to the monitoring server.

6.3 The concept of Network slicing is still evolving

Within the 5G concept a large amount of effort is focusing on architectures that support network slicing, implying evolution from the network sharing models towards network isolation, multi-tenancy and end-to-end resource provisioning and guarantees. In SONATA, slicing is considered at the lower service platform layer via the MANO framework, which leverage the IA and Slice Management components, and a distributed monitoring framework. Finally, issues related to service provider peering and recursive operations of the service platform (where slicing is also employed) is tackled by the Gatekeeper. In this context, SONATA considers an SDN capable WAN, managed by a WAN Infrastructure Manager (WIM) that supports slicing (by means of provisioning of isolated multi-tenant networks) plus an OpenStack based VIM integrated with an SDN controller (i.e. OpenDayLight) for the physical and network elements within the NFV infrastructure. In this view the SONATA service platform is able to create per domain slices that are interconnected constituting an end-to-end isolated network and computing resource slices.

6.4 Docker based VIMs are not yet mature enough for Service Function Chaining

SONATA set out to orchestrate network services on multiple different VIMs. Due to the growing adoption of Docker, we opted for a Docker based VIM in addition to the more commonly used Virtual Machine based VIMs. Docker is a container based virtualisation mechanism for guest operation systems that is much more lightweight than Virtual Machines (VMs), and can be faster built, tested and deployed. As VNFs are being implemented in both VMs and containers, SONATA targeted to combine container based and VM based VNFs in the same network service. To abstract the different APIs from the heterogeneous VIM landscape, an Infrastructure Adapter was introduced in the SONATA service platform to provide a streamlined API towards the MANO framework. Where VM based VIMs, with OpenStack the most used, have matured when it comes to NFV use, we came to conclude that Docker based VIMs are still missing some critical features. Both Kubernetes and Docker Swarm, two of the major Docker based VIMs, fail to provide complete isolation between the deployed containers, introducing security concerns. Docker based VIMs have been designed to host end-point services, causing them to lack features when it comes to hosting containers that are along the data path. One of these is that it is unclear how to integrate Service Function Chaining with the Kubernetes networking capabilities, which is a critical shortcoming when thinking about orchestrating NFV services. Efforts to bridge this gap are being made, e.g. by Multus. Due to these missing features, SONATA was unable to combine VM and container based VIMs in the same service.

6.5 Selecting the right software tools at the beginning of the project is crucial

It is important to automate large parts of the CI/CD pipeline, allowing the developers to focus on code development and provide them with quick feedback. To this end, a set of software tools such as GitHub and Jenkins were selected. We learned that this selection process is of the highest importance, as the selection of the wrong tool might conflict with the DevOps goals of the methodology. At one point shortly after the beginning of the project, we adopted OWASP, a tool that analyses code for security risks by scanning it for vulnerabilities, performing penetration testing, etc. Extending the Jenkins job that evaluates pull requests with an additional OWASP code check exponentially increased the time it took to analyse the updated code. The addition of OWASP significantly increased the duration of the integration cycle, contradicting our DevOps objective to quickly provide the developer with feedback, which led us to optimise the pipeline and use OWASP in a parallel job to not interrupt the first code check iteration. A learning period should be provisioned to allow developers to discover the selected software tools. This puts additional stress on the software tools selection process, as adopting a new software tool in a later stage of the project leads to a new learning period, which is both costly in terms of effort and time, especially in a distributed consortium like SONATA.

6.6 A good CI/CD and DevOps methodology allows for quick detection of design issues

As our CI/CD pipeline performs integration tests from the beginning of development, we were able to detect design issues very fast. For example, early in the project we identified a mismatch between the designed schemas, i.e templates, for service and VNF descriptors and the descriptor

data required by the service platform to correctly orchestrate the VNFs. Detecting descriptor issues late in the project would have implied a huge effort to correct, as already developed services for e.g. pilots and SDK tools that aid in the development of such services would need to be changed.

As it is tough to detect such design issues before an integration cycle and the cost in terms of time and effort of fixing such issues late in the development process is high, we feel that our CI/CD methodology allowed us i) to keep a clear and complete view of the status of the project at all times and ii) to meet software delivery deadlines in an environment of limited resources. While it gives no guarantees about the quality of the software, our methodology makes us confident about its stability and reliability, since it endured numerous cycles in the integration and qualification environment before it was released.

Newly introduced features, even late in the project, quickly became stable. For example, the authentication and authorisation feature for service platform microservices was lately introduced through the User Management concept. It was supposed to have a big impact on our integrated platform, as it added a new security wall between its components, demanding extra adaptation. Following the CI/CD loop, it was successfully integrated through a continuous adaptation from the components' interfaces with no major impact. Once this feature was deployed, it started to gain stability while it was enhanced with additional features such groups, roles and permissions.

6.7 Maintenance and updating of the CI/CD pipeline takes priority over code development

The success of the used CI/CD pipeline is directly correlated with its care and enforcement. In the SONATA project, a significant subset of the software developers was also responsible for the maintenance of the integration and qualification environment, causing the description and updating of new tests to slack at times. When the outcome of integration tests is ignored or they are not updated when new features appear in the software, the DevOps feedback loop is lost causing integration issues to appear in a later stage of the development. It is therefore of the utmost importance that the CI/CD pipeline is strictly followed and enforced, and gets prioritised over code development during every stage of the project.

6.8 Service platforms should cooperate with a hierarchical, recursive architecture

Often delivery of a service to a customer will need the involvement of more than one service platform. For example, the customer may want the service in multiple geographical locations, and no operator is present in them all. Another example is where some operators specialize in end-customer-facing operations, whilst others specialise in the "wholesale" provision of infrastructure, or in providing specific types of VNF. We believe that cooperating service platforms should be organised in a hierarchical architecture, meaning that an "upper-SP" provides the end-to-end service to the customer, and it chooses to involve a "lower-SP" to deliver part of the required capability (in other words, they have a "north-south" rather than "east-west" relationship). From the customer's perspective, they only interact with, and know about, the upper-SP; from the upper-SP's perspective, the lower-SP is providing a component in their overall network service in a similar manner to the NFVI; and as far as the lower-SP is concerned, the upper-SP is just another customer requesting a service. Further, we believe that the architecture should be recursive, meaning that the lower-SP can in turn arrange for some of the service it provides to be delivered by a yet lower- SP (and so on). The advantages of such an approach are commercial and technical: it has

clear lines of responsibility, allows autonomy and flexibility in service provision (e.g. different SPs could use different orchestrators), and only a single, standardised "north-south" API is needed.

A couple of issues concern capabilities for discovery and addressing. The upper-SP needs to learn what capabilities can be provided by potential lower-SPs. Our current thinking is that this is best done by the lower-SP publishing the capabilities it can offer (similar to today's Suppliers' Information Notes about network services), instead of a query protocol for instance. On addressing, we need to ensure that packets can flow along a service function chain that spans the SPs. At the moment, we think the most likely approach is that the upper-SP tells the lower-SP constraints, so that the lower-SP makes a good choice about the virtual link address and port identifier that it uses for the VNF(s) it supplies.

6.9 Learning by doing

A theme of the Sonata project has been 'learning by doing'. As well as being intrinsic to our continuous integration and delivery DevOps methodology, it was also crucial given industry's historic lack of detailed agreement about the right way to do orchestration. Hence, rather than a unidirectional flow from architecture to spec to development to product, the development of orchestrators needed a more cyclical process. OSM, Open Source MANO, has similarly been 'learning by doing' and it is no coincidence that we have contributed strongly there. Sonata and OSM have therefore been 'learning by doing' about the strengths and weaknesses of the initial NFV standards, which is leading to improvements in those standards. For instance, more detail was needed about the semantics of the interfaces in order to achieve interoperability; and the overlap /confusion between the element manager, VNF manager and orchestrator needed resolving, which is one of the motivations for recursion). We have thus been very pleased that OSM has taken on a couple of our components, whilst also moving its thinking in our direction in terms of a micro-service, recursive architecture, using a message bus and supplying a sandbox.

The open source nature of Sonata has been ground-breaking EC project. However, Sonata is limited in time and in effort participation from its partners. We raise this as a fundamental issue for EC projects – in a world which is becoming more open source, how can EC projects be *long-term* sustainable and truly open?

7 Conclusions

The work presented in this deliverable covered the establishment of the stable SONATA deployment that has been refined over the course of the project. The SONATA deployment allows the employment of DevOps procedures aiding the development network services in the frame of NFV ecosystem. Besides the provision of and extensive SDK framework for the development and the lifecycle management of the package comprising the network services, it offers syntax and structure validators following a SONATA defined scheme for the metadata and the descriptors defining the NS. In addition, actual infrastructure based environments deployments (i.e. Integration, Qualification) were used extensively to develop, deploy and validate in a CI/CD manner all the SONATA produced artefacts that consist of the SONATA Service Platform, responsible to orchestrate the NS over a multi-tenant, multi-pop, and multi-orchestrator environment. Finally the Demonstration environment, for the realisation of the SONATA pilots is also defined and finalised, based on an extensive infrastructure, dispersed in interconnected via VPN testbeds.

The second part of the deliverable presented the pilots, focusing not on the actual composition or development of each VNF per se (reader is redirected to deliverable [12] for more details on the actual architecture and design) but rather on the innovation brought/demonstrated by each pilot and the scenarios that are used for the demonstration of these innovations.

The third part of the deliverable discusses the results of the SONATA evaluation and validation campaigns. It should be noted that as the roadmap of SONATA dictated, system level evaluations were conducted and discussed at the final deliverables of WP3/4/5 (i.e [9], [10], [11]). Specifically for the Gatekeeper component, some additional validation is contributed in this document, encapsulating results from the numerous NS deployments and activity throughout the project duration. Finally the evaluation of the KPIs declared in Table 5.2 is presented. Results demonstrate the added value brought by SONATA platform, stemming from (i) the support of DevOps approaches inherently in the platform operations; (ii) the introduction of recursive architecture for the deployment of services over cascaded Service Platforms or multi-orchestration environment; (iii) the support of Network Slicing via the Slice Manager at the SP level and (iv) the decomposition of NFVO and VNFM via the introduction of SSM and FSM plugins allowing flexible service orchestration and refined VNF lifecycle management.

Finally the document concludes with the discussion of lessons learnt from failures and successes we had during the project lifetime.

The work presented in this deliverable covered the establishment of the stable SONATA deployment that has been refined over the course of the project. The SONATA deployment allows the “employment” of DevOps procedures aiding the development network services in the frame of NFV ecosystem. Besides the provision of and extensive SDK framework for the development and the lifecycle management of the package comprising the network services, it offers syntax and structure validators following a SONATA defined scheme for the metadata and the descriptors defining the NS. In addition, actual infrastructure-based environment deployments (i.e. Integration, Qualification) were used extensively to develop, deploy and validate in a CI/CD manner all the SONATA produced artefacts that consist of the SONATA Service Platform, responsible to orchestrate the NS over a multi-tenant, multi-pop, and multi-orchestrator environment. Finally the Demonstration environment, for the realisation of the SONATA pilots is also defined and finalised, based on an extensive infrastructure, dispersed in interconnected via VPN testbeds.

The second part of the deliverable presents the pilots, focusing not on the actual composition or development of each VNF but rather on the innovation brought/demonstrated by each pilot and the scenarios that are used for the demonstration of these innovations.

The selected pilots considered for the validation and evaluation are: (i) Virtual Content Delivery Network (vCDN), (ii) the Personal Security Application (PSA), (iii) Hierarchical Service Provider (HSP) and Multi-orchestrator and slicing. All pilots provide capabilities that go beyond the basic functionality of SONATA platform that include on-boarding, deployment, and instantiation of services. vCDN pilot innovations are mainly related to the virtual CDN service deployment optimisation and service resource management during runtime. PSA pilot innovations are mainly related to additional security-related services, specifically anonymity and safe browsing. HSP pilot innovations are mainly related to the multi MANO interworkings, SP recursiveness of the service on-boarding and abstraction layer and slice management.

The third part of the deliverable discusses the results of the SONATA evaluation and validation campaigns. It should be noted that as the roadmap of SONATA dictated, system level evaluations were conducted and discussed at the final deliverables of WP3/4/5.. Specifically for the Gatekeeper component, some additional validation is contributed in this document, encapsulating results from the numerous NS deployments and activities throughout the project duration. Finally the evaluation of the KPIs declared in Table 5.2 is presented. Results demonstrate the added value brought by SONATA platform, stemming from: (i) the support of DevOps approaches inherently in the platform operations; (ii) the introduction of recursive architecture for the deployment of services over cascaded Service Platforms or multi-orchestration environment; (iii) the support of Network Slicing via the Slice Manager at the SP level and (iv) the decomposition of NFVO and VNFM via the introduction of SSM and FSM plugins allowing flexible service orchestration and refined VNF lifecycle management.

Finally the document ends with the discussion of lessons learnt from failures and successes we had during the project lifetime.

A Validation of requirements

The table will be moved to an appendix. The section here needs to discuss the requirements elicitation process (summary) i.e. how many, how many where validated how many were not and some lessons learnt from this process.

Table A.1: Mapping SONATA requirements to use cases

No.	Requirement	Pilots	Validation KPI	Validation
	Business Requirements			
1	Catalogue Specs Mappings	vCDN, PSA	Number of specification mappings	Yes
2	Instances Mappings	vCDN, PSA	Number of Instances mappings	Yes
3	Usage of Service Platform	vCDN, PSA	Network Service usage events, VNF usage events	Yes
4	Audit Service Chain Changes	vCDN, PSA	service chain components status change events	Yes
5	Isolated/Reserved Vs Sharing Services	vCDN, PSA	operations for reserve/free resources for Network Services	No
6	Definition of Policies of Service for User	vCDN, PSA	Network service SLA(definition and thresholds), Network Service ACLs, Network Service usage reaching thresholds events	No
7	Monitoring Service Usage	vCDN, PSA	Network Service Usage	Yes
8	Service Activation Management Operations	vCDN, PSA	Activation operations supported for network services	No
9	Service Management Operations - Configuration	vCDN, PSA	Configuration operations supported for network services,time to perform operations,number of operations allowed by second	Yes

No.	Requirement	Pilots	Validation KPI	Validation
10	Service Infrastructure Management Operations	vCDN, PSA	NFVI SLAs, events of reaching thresholds of virtual infrastructure, quantity of resources of each type,elasticity of each resource	Yes
11	Legal Compliant	vCDN, PSA	Legal Compliance Measures (Only if needed)	Yes
12	Multiple IoT Vendors	vCDN, PSA		No
13	Multiple IoT Tenants	vCDN, PSA	Receive traffic from at least two different sensor vendors and treat it in a uniform way	No
14	Multi-Tenancy	vCDN, PSA	Owner, SLA, At least two IoT services without any conflict	Yes
15	Support Different Modes of Management/Control	vCDN, PSA	Owners, SLA by component, VNF specific monitoring metrics	Yes
	Functional Requirements – Service Programming			
16	VNF Catalogue	vCDN, PSA	Number of VNFs available in the catalogue	Yes
17	VNF Placement	vCDN, PSA	Complexity and time	Yes
18	Service Chaining (SFC)	vCDN	Service mapping complexity and time	Yes, in single-node PoP
19	Dynamic SFC update	vCDN, PSA	Service mapping complexity and time	No

No.	Requirement	Pilots	Validation KPI	Validation
20	VNF Scaling	vCDN	Resource usage (CPU/RAM/Bandwidth), availability in the VNFD fields to define scale policies, support the interruption of VNF operations, downtime while processing the scaling	No
21	Integration with Existing VNFs / Components		Support for corresponding annotations (or primitives) in the service programming model/language, the amount of downtime required to reconfigure a running service graph	No
22	Support for Service Templates/Cardinalities		Support for corresponding annotations (or primitives) in the service programming model / language	No
23	Inter-VNF QoS constraints		Support for corresponding annotations (or primitives) in the service programming model / language	Yes

No.	Requirement	Pilots	Validation KPI	Validation
24	Placement Constraints for VNFs	vCDN	Support for corresponding annotations (or primitives) in the service programming model / language, Deployment time, Cost. Specific metrics related to the service SLA/ end-user QoS	Yes
25	Isolation constraints for VNF		Support for corresponding annotations (or primitives) in the service programming model / language	Yes
26	Security VNF Availability	PSA	Enough number of Security VNFs	Yes
27	Personalized VNF	PSA	VNF descriptors supporting this functionality, validation of uniqueness of VNFs per use	Yes
	Functional Requirements – SDK			
28	SDK edition	vCDN	Manipulate IoT traffic	Yes
29	SONATA Reliability	vCDN	Reliability Options	Yes
30	SONATA DevOps	vCDN	DevOps tools & Instruments	Yes
31	On-Demand Runtime Control	vCDN	Runtime possible modifications	Yes
32	Security Simulation Tools	PSA	Number of available security simulation tools	Yes
33	VNF Deployment	vCDN, PSA	Number of VNFs deployed	Yes
34	Support for Self-contained logical administration of services	2.3.3 D2.3	Number of VNFs deployed	Yes
	Functional Requirements – Service Platform			

No.	Requirement	Pilots	Validation KPI	Validation
35	Support for Integration with existing VNFs/Components		Support for corresponding annotations (or primitives) in the service programming model/language	No
36	Service Chaining Support Across WANs		West-east interfaces between service platforms or architectural support for service platform hierarchies	Yes
37	Manual Service Function Placement		API primitive towards the orchestrator that allows manual function placement	Yes
38	Capability Discovery in Service Platform		Capability discovery primitives in the infrastructure abstraction layer and the service platform's NBI	Yes
39	SONATA Multitenancy	vCDN	Number of tenants	Yes
40	SONATA DevOps	vCDN	DevOps tools and instruments	Yes
41	Resource Infrastructure Mapping	vCDN	Resource allocation and time	Yes
42	Application Deployment	vCDN	Time	Yes
43	Ongoing Services Scale Up\Down	vCDN	Interruption, overloading	No
44	Multi NFVI Orchestration	vCDN	Number and size of the NFVI-PoPs	No
45/p>	Analytics plugin	vCDN	Time to perform analytics in real time operations	Yes (KPIs viewable in the GUI)

No.	Requirement	Pilots	Validation KPI	Validation
46	NS management northbound interface	vCDN, PSA	Set of operations allowed, successful management of the lifecycle of NFVI service elements (VMs and VNs), sufficient abstract service parameters to meet the deployment constraints of the NS/VNFs of all other use cases	Yes
47	Distributed NFVI	vCDN, PSA	Support at least 2 Datacenters or PoPs	Yes
48	Open Interfaces Toward NFV	vCDN, PSA	Open Interface specification or Standardization Body inclusion, successful activity of NFVI host service elements (VMs and VNs) with the constraints required by the NS/VNFs	Yes
49	Legacy Support	vCDN, PSA	Integration of one legacy NF	Yes
50	VNF Resource Report	vCDN, PSA	List how many resources are in use by each service	Yes (extractable from VNF records)
51	Authorization		Access and block functionalities regarding access level	Yes
52	Traffic Simulator		Number of IoT traffic sensors, Generated traffic	No
53	VNF Integration with Service			Yes
54	NFVI Northbound API	vCDN, PSA, HSP		Yes
55	Southbound Plugin to use NFVI API	vCDN, PSA, HSP		Yes
56	No Information Duplication for NS/VNFs with Heterogeneous SPs	vCND, PSA		Yes
57	Traffic Steering among NFVI-PoPs	vCDN, PSA		Yes
58	Support for Self-contained logical administration of services	HSP		Yes

No.	Requirement	Pilots	Validation KPI	Validation
59	One level of SP recursivity	PSA, HSP	Cecursivityin SPs	Yes
60	East/West-bound API	HSP		Yes
61	Master/Slave SP Negotiation protocol **Functional Requirements – Service Monitoring**	HSP		Yes
62	Timely alarms for SLA violation	Proven performance and scalability of the selected message bus system in the service platform		No
63	Statistics Interface	vCDN	Statistic elements	Yes
64	VNF Specific Monitoring	vCDN	Availability of an API for VNFs capturing such metrics. Metrics accuracy and response time	Yes
65	VNF Real-time Monitoring	PSA	Monitoring frequency, time to process alerts	Yes
66	VNF Reporting to BSS/OSS & Subscriber	PSA	Specification of a channel for user report	Yes
67	Quality of Service Monitoring	PSA	Traffic QoS , packet loss, delays	Yes
68	VNF and Topology Validation	PSA	Attestation mechanism validation	Yes
69	VNF Scaling	vCDN	Trigger scaling	No
70	VNF SLA Monitor		Provided metrics and data visualization	Yes
71	VNF Status Monitor		Feedback detail provided	Yes
	Non-Functional Requirements			
72	Service Platform Scalability	PSA	support for numerous sensors, support for line-rate performance traffic processing in service chains across all use cases, Escalation functionality availability and provisioning times	Yes

No.	Requirement	Pilots	Validation KPI	Validation
73	Service Platform Customizability		Plug & play, configurable service platform architecture that can be trimmed to a minimum basic feature set	Yes
74	SONATA System Interface	vCDN	Use facility, using time	Yes
75	SONATA System Update	vCDN	Updates and releases interoperability	Yes
76	SONATA Security	vCDN	Security issues	Yes
77	SONATA platform high availability	vCDN	Time of downtime allowed, SLA values, down times, VNF deployment time, availability, Cost, location, shared resources	Yes
78	Authentication	vCDN, PSA	Access	Yes
79	Confidentiality	PSA	Security controls in place	Yes
80	Support Services with 5 nines SLA/Ctrl		Interruption time, availability	No
81	Support state-full services		Interruption time, availability	No
82	Integration with OSS		Owner specific KPI	Yes (APIs are public and documented)
83	Dedicated GUI or interface for any manual intervention	HSP	GUI implemented	Yes
84	SONATA SPs synchronization	HSP	Synchronization between SPs	No

B Detailed package onboarding times

This page shows the detailed data collected to evaluate two of the SONATA Service Platform KPIs:

- **SP.GATE KEEPER.1**: Network Service on-boarding time, i.e., how long does it take to on-board a package (i.e., service);
- **SP.GATE KEEPER.2**: Network Service number of on-boarding requests, i.e., how many package (i.e., service) on-boarding requests the Service Platform can handle per minute;

Using the SONATA sample package, with random generated content, a series of stress tests have been applied. In the following charts, we took 10 results samples, which are ordered as *rows*. Then, an average of these samples is presented on the last chart. In each test row, a growing number of concurrent packages are on-boarded in order to push the API to the limit. These numbers are: 10, 100, 200, 400, 450 and 500 concurrent packages.

The following tables show the **on-boarded packages per second** (posts/sec), which is the rate of packages processed by the API, and the **elapsed time** in seconds (secs) which is the total time taken by the API to process all the packages.

Row 1

#Concurrent entries	posts/sec	secs
10	15.610	0.641
100	52.02	1.922
200	54.542	3.66
400	51.348	7.790
450	45.243	9.946
500	n/a	n/a

Row 2

#Concurrent entries	posts/sec	secs
10	2.958	3.379
100	28.971	3.451
200	40.022	4.997
400	55.125	7.256
450	29.665	15.169
500	n/a	n/a

Row 3

#Concurrent entries	posts/sec	secs
10	18.641	0.536
100	45.593	2.193
200	51.469	3.885
400	51.659	7.743
450	39.264	11.460
500	n/a	n/a

Row 4

#Concurrent entries	posts/sec	secs
10	15.194	0.658
100	25.023	3.996
200	35.355	5.656
400	31.274	12.789
450	27.313	16.475
500	n/a	n/a

Row 5

#Concurrent entries	posts/sec	secs
10	12.654	0.790
100	54.122	1.847
200	58.615	3.412
400	50.350	7.944
450	22.647	19.869
500	n/a	n/a

Row 6

#Concurrent entries	posts/sec	secs
10	15.60	0.640
100	34.834	2.870
200	38	5.263
400	21.161	18.902
450	16.919	26.597
500	n/a	n/a

Row 7

#Concurrent entries	posts/sec	secs
10	13.020	0.768
100	53.674	1.863
200	44.959	4.448
400	46.346	8.630
450	23.048	19.524
500	n/a	n/a

Row 8

#Concurrent entries	posts/sec	secs
10	11.361	0.880
100	16.297	6.135
200	50.505	3.959
400	20.399	19.607
450	14.042	32.045
500	n/a	n/a

Row 9

#Concurrent entries	posts/sec	secs
10	17.104	0.584
100	38.887	2.571
200	49.278	4.058
400	33.348	11.994
450	11.098	40.544
500	n/a	n/a

Row 10

#Concurrent entries	posts/sec	secs
10	13.108	0.762
100	36.562	2.735
200	23.780	8.410
400	32.599	12.270
450	49.379	9.113
500	n/a	n/a

Analysis of these numbers is done in Section **KPI Evaluation**, above.

C SONATA Infrastructure deployment summary

C.1 SONATA SDK

Typically, the SDK is available on GitHub at (`https://github.com/sonata-nfv/`) and runs on the Developer Workstation with Git installed.

C.2 SP infrastructure

The Service Platform can be deployed in one of the three options (currently, only Ubuntu 16.04 and CentOS 7 guests are supported):

- Ansible SP deployment to the local Linux machine
- Ansible SP deployment to an OpenStack VIM
- Instantiation of a QCOW2 image

Whatever deployment method you have chosen, the SP micro-services run as independent Docker containers - version 3 instantiates 35 containers.

C.2.1 Ansible SP deployment to the local Linux machine

'son-install' Ansible playbooks simplifies the SP deployment to the local machine:

```
git clone https://github.com/sonata-nfv/son-install.git
cd son-install
ansible-playbook utils/deploy/sp.yml -v
```

The ASCIINEMA screencast of this deployment can be seen at `https://asciinema.org/a/WFrwKGn6VlqrCg5N5qXw0ggZG` (73m)

C.2.2 Ansible SP deployment to an Openstack VIM

'son-install' playbooks enable the multi-distro SP deployment to a multi-PoP Openstack VIM on a single line CLI:

```
$ git clone https://github.com/sonata-nfv/son-install.git
$ cd son-install
$ ansible-playbook son-cmud.yml -e \
  'ops=create plat=sp pop=[ncsrd|alabs] proj=dem distro=[xenial|Core]' -v
```

This process is shown in the following ASCIINEMA screencast: *SONATA SP deployment through 'son-install'* (130m).

The following table presents typical duration to deploy a SP to an Openstack VIM. ANSIBLE TIMESTAMP

Task	Duration
##### START-DEPLOYMENT 20180215-09:11:48	00:00:04
##### INSTALLING NTP 20180215-09:11:52	00:00:10
##### UPGRADE PCKGs 20180215-09:12:02	00:00:44
##### OPENSTACK-CLI 20180215-09:12:46	00:02:51
##### CREATE-VM 20180215-09:15:37	00:01:16
##### REBOOT-WAITING 20180215-09:16:53	00:03:59
##### INSTALLING NTP 20180215-09:20:52	00:00:47
##### UPGRADE PCKGs 20180215-09:21:39	00:00:10
##### UPDATING-OS-PACKAGES 20180215-09:21:49	00:09:48
##### INSTALLING-PYTHON 20180215-09:31:37	00:02:28
##### INSTALLING-OS-TOOLS 20180215-09:34:05	00:01:15
##### REBOOTING 20180215-09:35:20	00:00:47
##### OPENSTACK-CLI 20180215-09:36:07	00:03:14
##### SP-DEPLOYMENT 20180215-09:39:21	00:04:06
##### SP PGSQL DEPLOYMENT 20180215-09:43:27	00:01:14
##### SP MONGODB DEPLOYMENT 20180215-09:44:41	00:01:11
##### SP REDISDB DEPLOYMENT 20180215-09:45:52	00:00:25
##### SP PGSQL-MONITORING DEPLOYMENT 20180215-09:46:17	00:00:10
##### SP BROKER DEPLOYMENT 20180215-09:46:27	0:00:41”
##### SP INFLUXDB DEPLOYMENT 20180215-09:47:08	0:01:01
##### SP GTK-KEYCLOAK DEPLOYMENT 20180215-09:48:09	0:02:02
##### SP GUI DEPLOYMENT 20180215-09:50:11	00:02:10
##### SP BSS DEPLOYMENT 20180215-09:52:21	00:01:59
##### SP GTK DEPLOYMENT 20180215-09:54:20	00:00:02
##### SP SON-VALIDATE DEPLOYMENT 20180215-09:54:22	00:03:24
##### SP GTK-PACKAGEs DEPLOYMENT 20180215-09:57:46	00:01:23
##### SP GTK-SERVICES DEPLOYMENT 20180215-09:59:09	00:01:36
##### SP GTK-FUNCTIONs DEPLOYMENT 20180215-10:00:45	00:00:44
##### SP GTK-RECORDs DEPLOYMENT 20180215-10:01:29	00:00:23
##### SP GTK-VIM DEPLOYMENT 20180215-10:01:52	0:00:41
##### SP GTK-LICENSING DEPLOYMENT 20180215-10:02:33	00:02:13
##### SP GTK-KPI DEPLOYMENT 20180215-10:04:46	00:01:47
##### SP GTK-USER-MGMT DEPLOYMENT 20180215-10:06:33	00:01:07
##### SP GTK-RATE-LIMITER DEPLOYMENT 20180215-10:07:40	00:01:06
##### SP GTK-API DEPLOYMENT 20180215-10:08:46	00:01:07
##### SP REPOs DEPLOYMENT 20180215-10:09:53	0:01:18
##### SP MANO DEPLOYMENT 20180215-10:11:11	00:02:50
##### SP IFTA DEPLOYMENT 20180215-10:14:01	00:04:35
##### SP MONIT DEPLOYMENT 20180215-10:18:36	00:05:47
##### SP SEC-GW DEPLOYMENT 20180215-10:24:23	00:00:32
##### END-DEPLOYMENT 20180215-10:24:55	01:13:07

This picture shows a graphical view of time spent to prepare the jump host and create the guest machine to receive the SP platform.

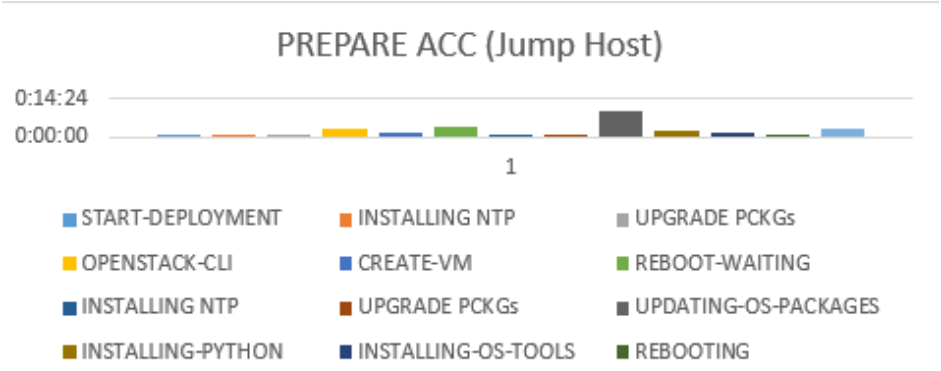

Figure C.1: ACC-Prepare-deployment-times.PNG

This picture shows a graphical view of time spent to deploy the SP to the guest machine.

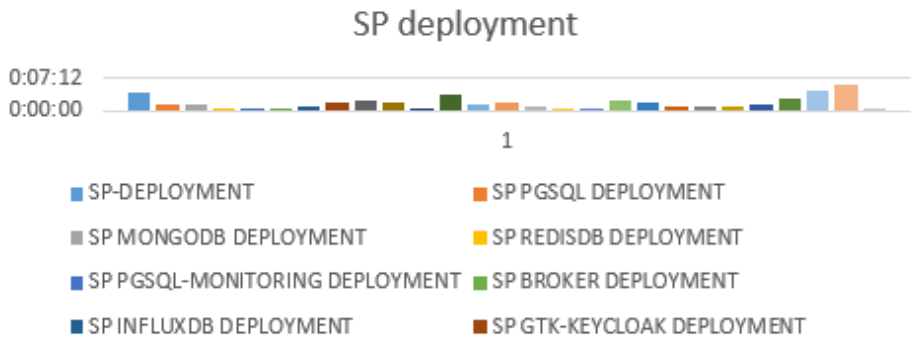

Figure C.2: SP-deployment-times.PNG

A brief interpretation of this table shows:

- the longest duration is the operating system package upgrade: 9'48"
- the long time for container's instantiation is due to the image retrieve from the Hub repository
- the typical time needed to instantiate and prepare a guest machine on an Openstack is about **30** minutes
- the typical SP deployment time spends about **45** minutes

C.2.3 Instantiation of a QCOW2 image

The **SP version 3 QCOW2 image**] is available at the SONATA FTP/HTTP Image repository. Depending on the virtualization layer, you can run the SP on the following targets:

- Openstack NFVI

```
// https://docs.openstack.org/ocata/user-guide\
//   /cli-nova-launch-instance-from-image.html guide
$ source admin-openrc
$ glance image-create --visibility public --name mySPv3 --disk-format=qcow2 \
  --container-format=bare --file son-sp-v3.qcow2 --progress
$ openstack image list && openstack security groups list && \
  openstack flavor list && openstack network list
$ openstack server create mySPv3 --image [son-sp-v3] --flavor [m1.medium] \
  --security-group [sp-secgrp] --nic net-id=[dem-netw]
$ openstack server list
```

- Linux server with 'libvirt' installed

```
# virt-install --name=mySPv3 \
--vcpus=2 \
--memory=4096 \
--cdrom=son-sp-v3.qcow2 \
--disk size=30 \
--os-variant=ubuntu16.04
```

- Hyper-V, VMware , VirtualBox (type 2 virtualization)

The conversion between image formats includes 'raw', 'vdi', 'vhd' and 'vmdk'.

```
// Ex: convert the published SP v3 image to VirtualBox format
$ qemu-img convert -f qcow2 -O vdi son-sp-v3.qcow2 son-sp-v3.vdi
```

C.3 Repositories infrastructure

SONATA provides the following repositories to the community:

- VNFs image repositories
- Docker Hub image repositories (requires Docker ID)
- NS/VNF Package repositories

C.4 NFVI infrastructure

SONATA Release 3 SP supports VIMs based only on OpenStack as the NFVI. Along the last three years, the SP has been tested against 'Liberty', 'Mitaka, 'Newton', and 'Ocata' releases. The installation of the OpenStack platforms followed one the several deployment methods and tools as well as deployment types. Summarising the SONATA SP has been tested agains deployments based on the following methods:

- From scratch, bare metal installation (both single or multi node deployments) [18]
- Via ansible scripts [17]
- Via Mirantis FUEL [14]
- OPNFV (Colorado, Danube and Euphrates releases) [20]

C.5 Deployment of pilots

In order to deploy your NS/VNF to the NFVI, first you must provision the multi-PoP infrastructure: login to the SP and assure that a VIM and WIM are available (or just add it). Then use the 'SON-CLI' SDK tools to validate and upload your NS/VNF packages. Actually, there are dedicated Jenkins jobs to help you on the instantiation of ALL NS's.

1. To clean the previous environment and populate the NFVI with new/fresh packages for NS's and VNF's just run: Pilots cleaning and package population
2. If changes occur on SSM/FSM's then you must run: Update SSM and FSM
3. To deploy ALL the environment just run: Environment/pilots deployment (all NS's)
4. Finally, go to the BSS to deploy the NS

- get token:

```
ACCESS_TOKEN=$(curl sp.int3.sonata-nfv.eu:32001/api/v2/sessions \
    -d '{"username":"sonata", "password":"1234"}' \
    | jq -r '.token.access_token')
```

- get NS ID: (from the BSS web GUI)
- Request the Instantiation of the NS:

```
curl http://sp.int3.sonata-nfv.eu:32001/api/v2/requests\
    -H "Authorization:Bearer $ACCESS_TOKEN" \
    -d '{"service_uuid": "0bf4b5de-7015-4386-8d16-7bbbb41c10b0",\
    "egresses":[], "ingresses":[]}'
```

NOTE: when there aren't changes on VNF' images neither on SSM/FSM, then you can clean your environment by deleting your Stack at Horizon (or CLI).

D Bibliography

[1] 5gex. Online at http://www.5gex.eu/.

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

[3] SONATA consortium. D2.3 updated requirements and architecture design. Website, December 2016. Online at http://www.sonata-nfv.eu/.

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

[5] SONATA consortium. D3.2 sdk operational release and documentation. Website, December 2016. Online at http://www.sonata-nfv.eu/content/d32-intermediate-release-sdk-prototype-and-documentation.

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

[7] SONATA consortium. D4.2: Service platform operational release and documentation. Website, December 2016. Online at http://sonata-nfv.eu/content/d42-service-platform-first-operational-release-and-documentation.

[8] 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.

[9] SONATA consortium. D3.3 sdk operational release and documentation. Website, June 2017. Online at http://www.sonata-nfv.eu/.

[10] SONATA consortium. D4.3: Service platform final release and documentation. Website, June 2017.

[11] SONATA consortium. D5.4: Final release of sonata platform. Website, August 2017.

[12] SONATA consortium. D6.2: Configuration of pilots and pre-validation. Website, June 2017. Online at http://sonata-nfv.eu/content/d62-configuration-pilots-and-pre-validation.

[13] SONATA consortium. D6.3: Final demonstration, roadmap and validation results. Website, January 2018. Online at http://sonata-nfv.eu/content/.

[14] Mirantis fuel. Online at https://www.mirantis.com/software/openstack.

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

[16] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[17] Openstack installation guide - ansible. Online at https://docs.openstack.org/openstack-ansible/latest/.

[18] Openstack installation guide. Online at https://docs.openstack.org/install-guide/.

[19] Inc OpenVPN Technologies. Openvpn. Website. Online at https://openvpn.net/.

[20] Opnfv installation. Online at https://www.opnfv.org/software.

[21] T. Soenen, S. Van Rossem, W. Tavernier, F. Vicens, D. Valocchi, P. Trakadas, P. Karkazis, G. Xilouris, P. Eardley, S. Kolometsos, M. Kourtis, D. Guija, S. Siddiqui, P. Hasselmeyer, J. Bonnet, and D. Lopez. Insights from sonata: Implementing and integrating a microservice-based nfv service platform with a devops methodology, April 2018.

[22] Inc The Tor Project. The tor project. Website. Online at https://www.torproject.org/.

[23] UCL. Very ligh. Website. Online at http://clayfour.ee.ucl.ac.ukhttp://clayfour.ee.ucl.ac.uk.