D5.4 Final release of SONATA platform

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	D5.4 Final release of SONATA platform
Workpackage	WP5 Integration, system testing and qualification
Due Date	31/07/2017
Submission Date	04/08/2017
Version	0.1
Status	To be approved by EC
Editor	Dario Valocchi (UCL)
Contributors	Stuart Clayman, Alex Galis, Francesco Tusa, Dario Valocchi (UCL), Felipe Vicens, Josep Martrat (ATOS), Dani Guija, Shuaib Siddiqui, Einar Meyerson, Juan Nuñez, Javier Fernández Hidalgo (i2CAT), José Bonnet (ALB), Manuel Peuster, Hadi Razzaghi Kouchaksaraei (UPB), Panos Trakadas, Panos Karkazis (SYN), Steven Van Rossem, Thomas Soenen (IMEC), Navdeep Uniyal (NEC)
Reviewer(s)	Michael Bredel (NEC), Wouter Tavernier (IMEC).

Keywords:

prototype, integration, qualification

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

Executive Summary:

This Deliverable documents the progress made in the WP5 of the SONATA project. WP5 focuses on defining processes and methodologies for an *agile* integration process of the output of the work done in WP3 and WP4. In previous deliverables, we highlighted and illustrated how we embraced a Continuous Integration/Continuous Delivery (CI/CD) software development life-cycle. In order to satisfy the growing demand for a stable and reliable software framework providing VNF orchestration and to meet the project objectives in terms of impact and adoption, our development pipeline includes a final stage of validation and qualification, meant to assess the reliability and verify the stability of the SONATA software framework. Since this document is the last one delivered by Work Package 5, we focus on this final stage of our CI/CD pipeline, in order to document the results we obtained in terms of stability and reliability using the software management approach we described in previous documents. The document also includes references to the latest release of SONATA software framework, our analysis on the interfaces provided by SONATA for ensuring interoperability with OSS and SLA management system.

Contents

List of Figures

List of Tables

1 Introduction

This deliverable concludes the series of deliverable produced for WP5, dealing with software and system testing, integration and qualification. In this deliverable, we first document the third release of the integrated SONATA platform, which includes all the software components developed, integrated and qualified within the time horizon of the project. With this release, we also include information on the SONATA framework installation process. The central part of the deliverable illustrates the result of our qualification phase, showing some of the metrics collected and of the results obtained during the last phase of our Continuous Integration/Continuous Delivery (CI/CD) software development pipeline, which is fully documented in [8]. With these results, we want to highlight how our Agile development philosophy and the CI/CD development cycle we embraced from the very beginning of the project has helped us to produce a software framework which, although does not represent a commercial software product, is able to stand stress and stability tests. Finally, this document contains the results obtained in developing additional interfaces of SONATA framework for operational support and additional functionality, as documented in Task 5.3 of [1], as well as the integration and testing strategy used in the context of the project to develop VNFs to be used in the project pilots.

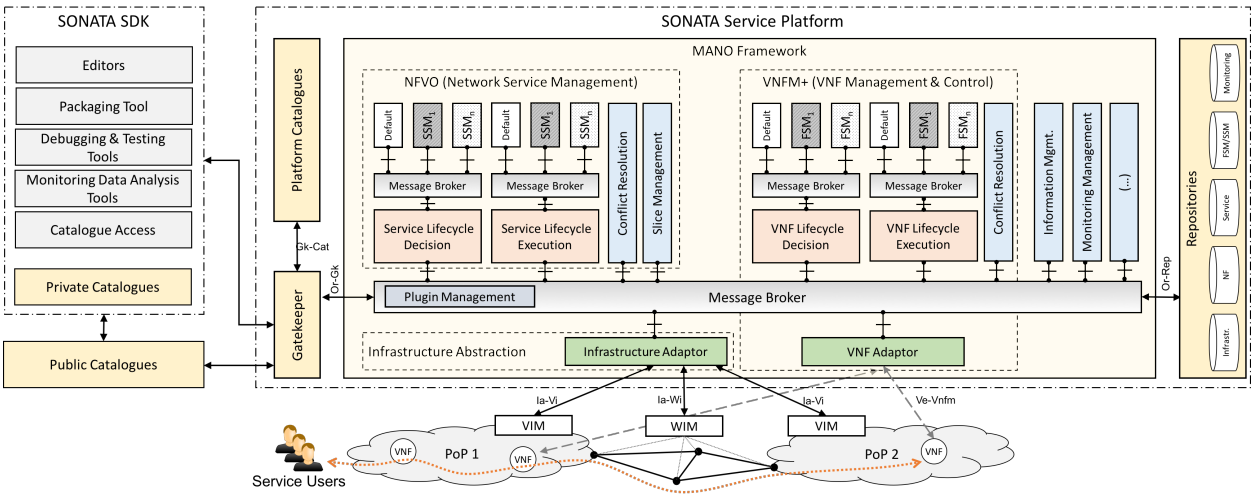

Figure 1.1: Detailed architecture of SONATA service platform

In the following of this introduction, for sake of readability, we will briefly summarise the architecture and main components of the SONATA platform. The architecture of the SONATA platform is summarised in Figure 1.1. SONATA framework consists of two parts: (1) the SONATA Software Development Kit (SDK): this part offers functionality and tools for the development and validation of virtual network functions and virtual network services, including an editor for service descriptors, an emulator to locally test developed services, monitoring data analysis tools, service storage, packaging and publishing tools; (2) the SONATA Service Platform (SP), which offers the functionality to orchestrate and manage network services during their life cycles with a MANO framework, interact with the underlying virtual infrastructure through Virtual Infrastructure Managers (VIM) and WAN Infrastructure Managers (WIM) to efficiently use a heterogeneous set of virtual resources,

store available network services and virtual network functions with dedicated catalogues, represent the status of the deployed network services and network functions, of the virtual infrastructure, and of the virtual resources through a set of Repositories, which represent the information system of the SP, and offer the functionality of the SP itself to the outside world, with a unique endpoint, called the Gatekeeper.

1.1 Deliverable structure

The document is organised as follows.

- Section 2 describes the third software release of SONATA, which includes software modules in the service platform and the SONATA SDK;
- Section 3 illustrates the outcomes of the qualification of our SONATA framework v3.0;
- Section 4 includes a description of the additional interfaces offered by the SONATA framework toward OSS systems or SLA management system, additional software modules like the self-monitoring of SONATA modules and includes a description of the software management policies and tools used in the project context to develop and test VNFs envisioned in the project's pilots.
- Section 5 concludes the deliverable.

2 SONATA framework Release 3.0

This section documents the modules composing the release 3.0 of the SONATA framework. For each module, a brief description is given, along with the specification of the new features and improvement introduced in this latest release.

Table 2.1: SONATA Framework v3

Module name	Description	Repository	Reference documentation
Infrastructure Abstraction	Allows vendor-agnostic interaction between the MANO and with virtual infrastructure	son-sp-infrabstract	D4.1, Section 5.2 D4.2, Section 6 D4.3, Section 5
Monitoring Framework	Monitoring Framework gathers, analyses performance information from NS/VNF and provides alarm notifications, based on alarm definitions which have been defined by the users. The architecture of the system is based on data exporters and a monitoring server. Data exporters send monitoring data from NS/VNFs to monitoring server which collects, analyses, stores data and generates the appropriate notifications. In general, monitoring server consisting of a rest API interface, an alerting mechanism, a time-series DB and a real time notification service.	son-monitor son-monitor-probe	D4.1, Section 5.2.4 D4.2, Section 7 D4.3, Section 6
Gatekeeper GUI	Gatekeeper GUI is designed to cover the needs of the developers and platform administrators in supporting the process of DevOps in SONATA. Gatekeeper GUI is an API management and visualisation tool that on one hand enables SONATA developers to manage their services throughout their whole lifecycle, while on the other hand offer Service Platform administrator the ability to provision, monitor and monetise platform resources.	son-gui	D4.1, Section 2.5 D4.2, Section 3.5 D4.3, Section 2.4
SDK Emulator	This module implements SONATA's novel multi-PoP NFVI emulation platform which is based on Containernet and allows to emulate arbitrary complex multi-PoP environments on a single machine. In this emulated environment, VNFs are in general packaged and executed as Docker containers. The emulation platform also offers OpenStack-like northbound interfaces to support experiments with arbitrary MANO systems, e.g., OpenSource MANO, that control the emulated infrastructure. In addition, the emulator offers a simplified gatekeeper interface to support the direct deployment of SONATA service packages on the emulated platform which can serve as a rapid prototyping environment for network service and VNF developers.	son-emu	D3.1, Section 3.4 D3.2, Section 3.4 D3.3, Section 4.5
SDK Profiler	Son-profile is part of SONATA's SDK and offers supporting functionality to do performance profiling of network services. To do so, a developer can specify benchmarking experiments in a so called *profiling experiment descriptor (PED)*. Son-profile takes these definitions and automatically runs a large set of test configurations on a target platform.	son-cli	D3.2, Section 3.5 D3.3, Section 4.6
SDK Editor	This module is a web-based, graphical editor for network service descriptors and VNF descriptors. It supports the SONATA description approach and automatically validates user inputs against the SONATA schema definitions. It also integrates with GitHub to allow developers to easily share their network service projects.	son-editor-backend son-editor-frontend	D3.3, Section 4.4

Module name	Description	Repository	Reference documentation
SDK Monitoring	This is the monitoring framework, implemented in the SONATA SDK. The module can gather monitored metrics from services deployed in the emulator and the service platform. For services deployed in the emulator, the framework allows to dynamically define the set of exported metrics via a definition file. The metrics are stored in a time-series database and can be visualised on a web-based GUI.	son-cli	D3.3 Section 4.7
Gatekeeper	The Gatekeeper is the SONATA Service Platform (SP) that controls which users or systems can access the features provided by the SP (such has onboarding or downloading packages, downloading packages, services or functions meta-data, downloading records, subscribing for (a)synchronous monitoring, etc.), onboarding packages (after successful validation) into the Catalogue, accepting service instantiation requests, etc.	son-gkeeper	D4.1, Section 2 D4.2, Section 3 D4.3, Section 2
Gatekeeper BSS	The Gatekeeper's BSS is a graphical user interface through which the SONATA's customer can instantiate network services, to acquire service licences, to update service instances, to provide location info to the new services or to check the status of the requests made to the platform.	son-bss	D4.1, Section 2.6 D4.2, Section 3.6 D4.3, Section 2.5
Gatekeeper User Management	The Gatekeeper's User Management is responsible for the authentication and authorisation processes, managing the identity, controlling the permissions and the access to the Service Platform (SP) the platform's users and micro-services, allowing or denying the requested actions.	son-gtkusr son-keycloak	D4.2, Section 3.2 D4.3, Section 2.2
Catalogue	The Service Platform Catalogue is a key component responsible for the storage of NSDs, VNFDs, PDs and SONATA packages (files called son-packages).	son-catalogue-repos	D4.1, Section 5.1 D4.2, Section 4.1 D4.3, Section 3.1
Son-sm	Son-sm is a collection of tools and examples, that is implemented to support SSM/FSM development. It contains a template that provides the basic SSM/FSM functions such as the SSM/FSM registration into MANO framework. Multiple examples such as monitoring and configuration FSMs are also provided which helps developers to accelerate FSM/SSM development.	son-sm	D3.3, Section 4.9
MANO Framework	SONATA's MANO framework is the core of SONATA's service platform and builds a flexible orchestration system. It consists of a set of loosely coupled components (micro services) that use a message broker to communicate. These components are called MANO plugins and can easily be replaced to customise the orchestration functionality of the platform. New features: termination of a running service instance, Function Life Cycle Manager and Placement Plugin as separate plugins, Start/Stop/Configure FSM support.	son-mano-framework	D4.1, Section 3, 4 D4.2, Section 5 D4.3, Section 4
SDK Command Line Interface (CLI)	The SONATA SDK command line interface tools is a component meant to assist the SONATA service developers on their tasks. It includes a series of tools to cover critical points of service development within SONATA SDK.	son-cli	D3.1, Section 3.2, 3.3, 3.5 D3.2, Section 3.2 D3.3, Section 4.2
SONATA Validator	The SONATA Validator is the SONATA tool responsible for the validation of SDK projects, packages, services and functions. It was initially developed with the purpose of aiding the development of services and functions within an SDK project. However, it was also implemented as a service, enabling its usage by third-party entities. This element has been introduced both in the SDK and on the Gatekeeper from the current release 3.0.	son-cli	D3.3, Section 4.3

2.1 SONATA Framework installation

2.1.1 Service Platform installation

SONATA service platform is one of the major components of the system. It consists of highly modular and customizable management and orchestration (MANO) framework. It consists of various loosely coupled micro-services/plugins (described above) which communicate through the message broker and assist developers in deploying and managing the network services on the infrastructure.

The SONATA Service Platform installation scripts enable users to easily deploy the Service Platform with all the required plugins to customize and manage the MANO Framework on network service as well as on the network function level. The 'son-install' allows users to Create, Manage, Upgrade and Destroy the SONATA software components for the Service Platform as a standalone machine. The latest version of the son-install allows the user to select the version of Service Platform to be deployed on the standalone machine.

SONATA SP can be installed following two procedures. The first procedure consists n an existing VM that fulfil the minimum requirements of both Software and Hardware, son-install provides a set of Ansible playbooks to perform the installation. The installation can be customised, editing configuration files to set components passwords, databases and enable/disable some functions. The second way to install SONATA SP is having an OpenStack infrastructure, son-install can be used directly to spawn a VM, installing dependencies and SONATA SP in an automated way.

2.1.1.1 Installation on Bare Metal and Virtual Machines

Prerequisites:

- **Hardware:**
  - Minimal size requirements
    - Memory 4GB RAM
    - CPU 2 vCPU
    - Disk 25GB
  - Recommended size requirements
    - Memory 8GB
    - CPU 4vCPU
    - Disk 80GB
- **Software:**
  - Ansible 2.3.0+
  - Git
  - Docker CE 17.06

Installation:

On a clean Ubuntu 16.04:

```
sudo apt-get install -y software-properties-common
sudo apt-add-repository -y ppa:ansible/ansible
sudo apt-get update
```

```
sudo apt-get install -y ansible
sudo apt-get install -y git
git clone https://github.com/sonata-nfv/son-install.git
cd son-install
echo sonata | tee ~/.ssh/.vault_pass
ansible-playbook utils/deploy/sp.yml -e \
  "target=localhost plat=sp public_ip=<your_ip4_address> \
    plat_hostname=<your_hostname> vm_user=sonata" -v
```

NOTE: as for many Ansible playbooks, you must configure passwordless sudo in your '/etc/sudoers', ie, the login running the playbooks belongs to sudo/wheel group

- for Debian systems: `%sudo ALL=(ALL:ALL) NOPASSWD: ALL`
- for Redhat systems: `%wheel ALL=(ALL:ALL) NOPASSWD: ALL`

Installation Options

The SONATA installer offers different options to install the Service Platform. Please find below a detailed description of the various options.

Default Installation

To install SONATA Service Platform v3.0 you only need to follow the steps described above taking into account that the configuration will be the following:

- Ansible target: -> `Localhost`
- The public IP is the IP in the ipv4 interface card. It is used to access the GUI, BSS and the Gatekeeper API

Custom Installation

For custom installation, a configuration file is available in the sub-folder: `group_vars/sp/vault.yml` This file is encrypted. To open it, the password should be located `~/.ssh/.vault_pass`. The default password is `sonata`. To edit the encrypted file you can use the following command: ansible-vault edit group_vars/sp/vault.yml

```
# VARs for MONITOR pgSQL database
upassword: sonata
urootpw: 1234
dbname: monitoring
dbuser: monitoringuser

# VARs for GATEKEEPER pgSQL database
gtk_db_name: gatekeeper
gtk_db_user: sonatatest
gtk_db_pass: sonata

# VARs for GATEKEEPER USER MGMT
gtk_keycloak_user: admin
gtk_keycloak_pass: admin
```

```
# VARs for IA
ia_repo_user: sonatatest
ia_repo_pass: sonata

#VARs for Monitoring
mon_db_name: monitoring
mon_db_user: monitoringuser
mon_db_pass: sonata
```

To change the default configuration of installation, the following file can be edited: `roles/sp/defaults/main.yml`

```
---
######################
# defaults file for sp
######################

nbofvms: 1
pop: alabs
proj: demo
distro: xenial
plat: sp

########################
# Docker network segment
########################
# Docker network name
docker_network_name: son-sp

##############################################
# SONATA 5G NFV SP specific version variables
##############################################
# SONATA SP Version. This installer was created to deploy version 3.0
sp_ver: 3.0

# SONATA SP Hostname. Default Public IP address
plat_hostname: "{{ public_ip }}"

# SONATA SP Domain Name. Default Public IP address
domain_name: "{{ public_ip }}"

# SONATA SP fqdn. Default Public IP address
fqdn: "{{ public_ip }}"

# SONATA SP User created by son-install
sp_user: sonata
sp_pass: "$1$SRc2ws2Z$rSdCC/UKiatagNdfsTVuf0"
```

2.1.1.2 Installation on OpenStack

The SONATA SP can be easily deployed within an Openstack environment, and this procedure is further simplified by the usage of `son-cmud.yml` Ansible playbooks. First, configure the Openstack API URL and authentication credentials for your tenant in `"~/.config/openstack/clouds.yaml"` (or in `"/etc/openstack/clouds.yaml"` if you have 'root' access). This file includes ALL the provisioned Openstack platforms that your are able to deploy SONATA resources ex:

Configuration for "SON-DEMO" project at Altice Labs PoP

```
os_alabs_demo:
  auth:
    auth_url: 'http://172.31.6.9:5000/v3'
    username: 'son-demo'
    password: '********'
    project_name: 'son-demo'
    project_domain_name: "default"
    user_domain_name: "default"
  auth_type: password
  identity_api_version: "3"
  region_name: RegionOne
```

Configuration for "SONATA.DEM" project at NCSRD PoP

```
os_alabs_demo:
  auth:
    auth_url: 'http://10.100.16.20:5000/v3'
    username: 'sonata.dem'
    password: '********'
    project_name: 'sonata.dem'
    project_domain_name: "default"
    user_domain_name: "default"
  auth_type: password
  identity_api_version: "3"
  region_name: RegionOne
```

Second, get the 'son-install' from the Github repository:

```
$ git clone https://github.com/sonata-nfv/son-install.git
$ cd son-install
```

Third, set the `"~/.ssh/.vault_pass"` file with the `'sonata'` string used as default password for user `'sonata'`: change it as you which as long as you update the hash at `"roles/sp/defaults/main.yml"`.

Finally, run the `'son-cmud.yml'` playbook with the appropriate parameters for each PoP - for example, the following command will create an Ubuntu 16.04 VM for "dem" project at NCSRD's PoP and deploy the SONATA SP to that VM:

```
$ ansible-playbook son-cmud.yml
   -e "ops=create plat=sp pop=ncsrd proj=dem distro=xenial" -v
```

This single command line spends for about 30 minutes, depending on the OpenStack performance (to create the VM) and Internet access (to install and update the common packages, libraries and tools and to retrieve the Docker images from Docker Hub).

Now, you have the SP web application available at the floating IP assigned by OpenStack - this IP address is visible in the verbose mode and is also registered at Inventory file at `"$PWD/inventory/sp/hosts"`.

2.1.2 SDK installation

The SONATA SDK is a modular set of light-weight software tools that are available from different GitHub repositories. A user can select which tools she wants to install on her development machine. As shown in Figure 2.1 does each SDK repository contain a set of tools that belong together, e.g., all command line tools for project creation are part of the son-cli repository and can thus be installed as one package. Similarly, the son-emu package contains the emulation component as well as its management CLI tool. Each of the shown packages in Figure 2.1 can be installed without requiring one of the other packages. The installation procedure for the SDK is described in the following and documented in more detail in D3.3 Section 3.

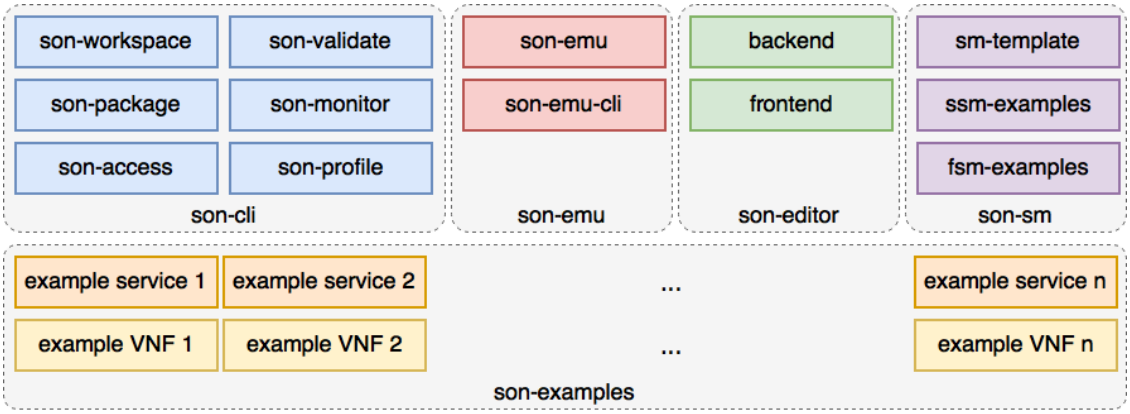

Figure 2.1: SONATA SDK installation packages and the contained modules

2.1.2.1 son-cli

This repository installs a set of command line tools are meant to aid the SONATA service developers on their tasks. The tools are briefly described as follows:

- `son-workspace` creates, configures and manages development workspaces and projects.
- `son-package` packages a project, containing services and functions, to be instantiated in the SONATA Service Platform. All project components are syntactically validated and external dependencies are retrieved to produce a complete service package.
- `son-validate` can be used to validate the syntax, integrity and topology of SONATA service packages, projects, services and functions. Son-validate can be used through the CLI or as a micro-service running inside a Docker container.
- `son-access` enables authenticating users to the Service Platform and integrates features to push and pull resources from the Service Platform Catalogues. It is used to upload the

service package to the SDK emulator or the Service Platform Gatekeeper, so the service can be deployed in the respective environment.

- `son-monitor` provides tools to easily monitor/generate metrics for debugging and analyzing service performance.
- `son-profile` supports network service developers to automatically profile their network services and network functions.

The install procedure and tool usage are documented and maintained in the GitHub repository.
`git clone -b 'v3.0' https://github.com/sonata-nfv/son-cli.git`
The needed dependencies can be installed via an Ansible script:

```
sudo apt-get install ansible git aptitude
sudo vim /etc/ansible/hosts
Add: localhost ansible_connection=local
cd son-cli/ansible
sudo ansible-playbook install.yml
```

Afterwards, son-cli can be installed via the Python setup script:

```
cd son-cli
python3 setup.py install
```

An alternative way is to install son-cli using the OS package distribution system. We currently support Ubuntu Trusty (14.04) and Ubuntu Xenial (16.04).
Begin by adding the GPG key:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 8EC0731023C1F15B
```

For Ubuntu Trusty add the repository: `echo "deb http://repo.sonata-nfv.eu ubuntu-trusty main" | sudo tee -a /etc/apt/sources.list`
For Ubuntu Xenial add the repository: `echo "deb http://repo.sonata-nfv.eu:8080 ubuntu-xenial main" | sudo tee -a /etc/apt/sources.list`
Finally, install the CLI tools:

```
sudo apt-get update
sudo apt-get install sonata-cli
```

2.1.2.2 son-emu

There are two ways to install and use son-emu. The simple one is to use Vagrant [17] to create a VirtualBox-based VM on your machine that contains the pre-installed and configured emulator. The bare-metal installation requires a freshly installed Ubuntu 16.04 LTS and is done by an Ansible playbook. It is important to note that the emulation platform needs `root` access to the host machine to emulate arbitrary complex networks and their topology. Thus, it should always be installed on a dedicated machine or VM.

Installation using Vagrant

- Request VirtualBox and Vagrant to be installed on the system.
- `git clone -b 'v3.0' https://github.com/sonata-nfv/son-emu.git`
- `cd ~/son-emu`
- `vagrant up`
- `vagrant ssh` to enter the new VM in which the emulator is installed.

Installation using Ansible

To install `son-emu` from scratch on a physical machine or an empty VM, an Ansible script is available in its repository. This installation requires a fresh Ubuntu 16.04 LTS installation. To prepare the machine, the user has to do the following steps:

- `sudo apt-get install ansible git aptitude`
- `sudo vim /etc/ansible/hosts`
- Add: `localhost ansible_connection=local`

After this, Containernet needs to be installed on the machine, because `son-emu` builds up on this fork of the well known Mininet network emulation platform. The following steps install and prepare Containernet for the `son-emu` installation:

- `cd`
- `git clone -b 'v3.0' https://github.com/containernet/containernet.git`
- `cd ~/containernet/ansible`
- `sudo ansible-playbook install.yml`

Finally, `son-emu` can be installed on the machine by doing:

- `cd`
- `git clone  -b 'v3.0' https://github.com/sonata-nfv/son-emu.git`
- `cd ~/son-emu/ansible`
- `sudo ansible-playbook install.yml`

2.1.2.3 son-editor

The SONATA descriptor editor (`son-editor`) is composed of a front-end and a back-end component which are both located in independent repositories. The editor can be installed and deployed as single Docker container by using a docker-compose script.

Since the editor uses OAuth to authenticate its users, a OAuth application token is required to run it. To retrieve such a token (from GitHub), go to GitHub `Settings` > `OAuth applications and 'Register a new application'`:

- Chose an application name: `SONATA Editor`

- Configure the URL of your installation: `http://localhost/` or `http://your-domain.com`
- Configure the authentication callback URL:
  - `http://localhost/backend/login` or
  - `http://your-domain.com/backend/login`
- Save and collect the generated `ClientID` and `ClientSecret` for later use

Once the OAuth token is available, the editor code can be downloaded from GitHub and the OAuth credentials have to be added to the configuration file.

- `git clone -b 'v3.0' https://github.com/sonata-nfv/son-editor-backend`
- `cd son-editor-backend/`
- Add GitHub OAuth `ClientID` and `ClientSecret` to `config.yaml`

Finally, the Docker container that runs the editor can be built and deployed using `docker-compose` it the root directory of the editor backend:

- `docker-compose up`

Open your web browser and point to your server / local machine, e.g., `http://localhost/` and log-in to the editor using your GitHub account.

2.1.2.4 son-examples

The `son-examples` repository contains several example network services and their corresponding VNFs that can be used to test SONATA's SDK tools and SONATA's service platform. It does not contain any *installable* software component, which means that a user can simply clone the repository and start to use the examples.

- `git clone -b 'v3.0' https://github.com/sonata-nfv/son-examples.git`
- `cd son-examples`

To build the Docker-based VNFs that are part of the examples, the following script can be called:

- `vnfs/build.sh`

To pack all service projects that are contained in this repository, the following script can be called:

- `service-projects/pack.sh`

This will create the corresponding SONATA packages in `service-projects`.

2.1.2.5 son-sm

The `son-sm` repository includes a collection of tools and examples, implemented to support SSM/FSM development. It contains a template that provides the basic SSM/FSM functions like the SSM/FSM registration in the MANO framework. SSM/FSM examples such as monitoring and configuration FSMs are also included in the son-sm repository, which help developers to accelerate FSM/SSM development.

Son-sm does not contain any installable software, therefore, can be used simply by cloning the repository as follows:

- `git clone --recursive https://github.com/sonata-nfv/son-sm.git`

3 Final Release Qualification and System Performance Assessment

This section illustrates the results obtained by our CI/CD development process in integrating and qualifying our software framework. We document the results obtained with the tests we designed for out qualification phase, showing results for several key components of the SONATA framework. Finally, as this represents the last deliverable for the SONATA WP5, we close this section drawing some conclusions on the use of CI/CD approach in SONATA, highlighting the insights it gave us on the effectiveness and efficiency of our design and leading us to the several improvements provided in the past and present releases.

3.1 System performance assessment

In this section, we present the some of the results obtained during the qualification of SONATA Service Platform and SDK components. We identified main macro-components to stress with relevant tests:

- Gatekeeper API, as the main entry point to the SONATA Service Platform
- Service Platform Catalogue API, as the main storage element for SONATA file packages and descriptors
- Monitoring framework, as the main feedback loop in the SONATA DevOps approach.
- Infrastructure Abstraction, as the single point of communication with the virtual infrastructure
- SDK Emulator, as the main performance evaluation tool of the SONATA SDK

We also provide results for qualification tests aimed at assessing the installation process and the most important End-to-End flows of the SONATA framework.

3.1.1 Installation tests

3.1.1.1 SONATA SP over OpenStack

The goal of this test is to perform the SONATA SP installation over an OpenStack environment to test the installation scripts provided by son-install. The SP deployment time depends on the following factors:

- the OpenStack platform performance
- the PoP bandwidth to the Internet
- the Docker Hub download time (currently, 30 Docker images of different size)

Environment:

This test is composed of the following components:

- Jenkins
- OpenStack Mitaka
- son-install repository at `https://github.com/sonata-nfv/son-install.git`
- SP Docker images at `https://hub.docker.com/sonata-nfv/` or at SONATA Registry

The environment components can be observed in Figure 3.1

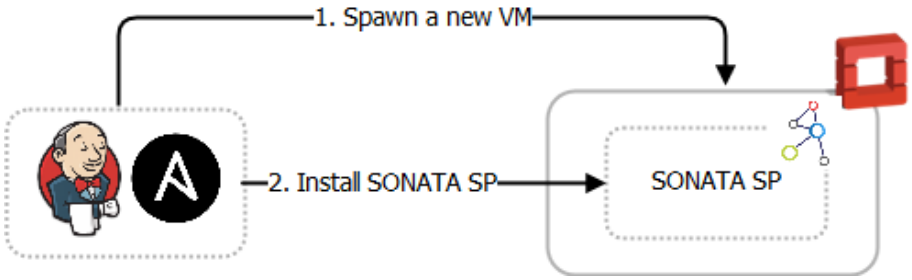

Figure 3.1: SONATA SP installation over Openstack environment

Workflow:

The Jenkins plays the Ansible Control Center (ACC) role: its job will run the 'son-cmud.yml' to create a VM on a provisioned VIM and deploy the SP.

Results:

Taking into account the following hardware parameters:

- VM Flavor: m1.small (2vCPU 4GB RAM 40GB Disk)
- Internet Connection: 50Mbps symmetric

The measurement of SP deployment times per PoP are shown in Table 3.1:

Table 3.1: Deployment times per PoP

PoP	prepare ACC	create VM	upgrade VM	install SP	Total
Alabs	2m 32s	34m 11s	7m 54s	32m 06s	1h 16m 43s

3.1.1.2 SONATA SP over existing VM

The goal of this test is to perform the SONATA SP installation over VM environment to test the installation scripts provided by son-install. The SP deployment time depends on the following factors:

- the VM performance
- the VM bandwidth to the Internet
- the Docker Hub download time (currently, 30 Docker images of different size)

Environment:

This test is composed of the following components:

- Jenkins
- Pre-installed Ubuntu xenial 16.04 VM
- son-install repository at `https://github.com/sonata-nfv/son-install.git`
- SP Docker images at `https://hub.docker.com/sonata-nfv/` or at SONATA Registry

The environment components can be observed in Figure 3.2.

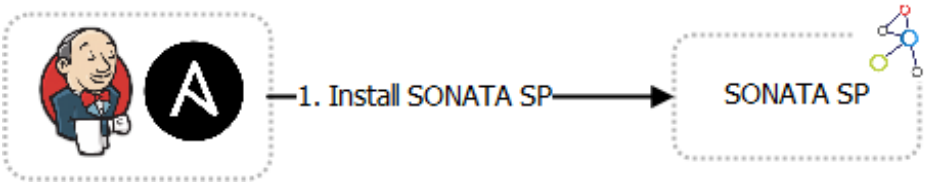

Figure 3.2: SONATA SP installation VM environment

Workflow:

The Jenkins plays the Ansible Control Center (ACC) role: its job will run the 'son-cmud.yml' to install the SONATA SP over a pre-created VM.

Results:

The measurement of SP installation times can be observed in Figure 3.3.

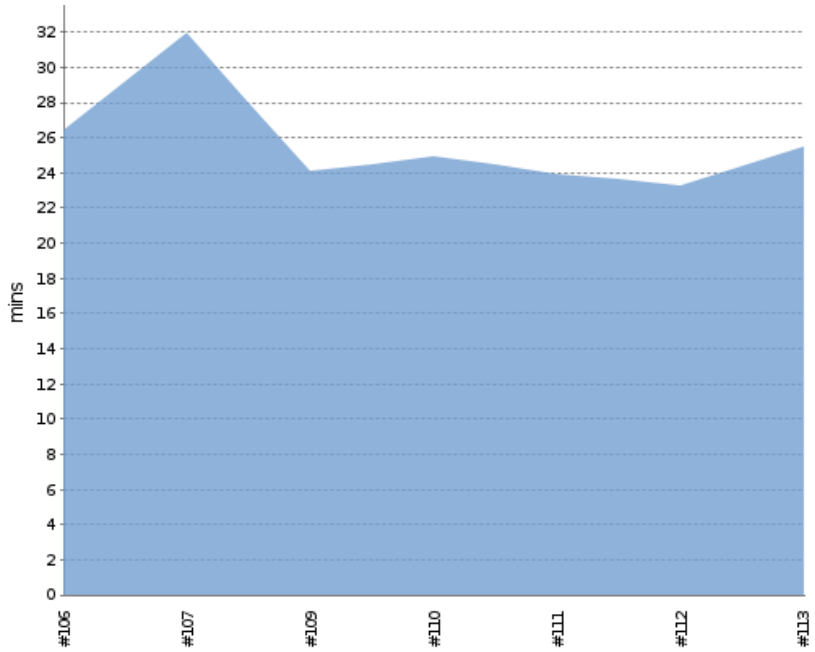

Figure 3.3: SONATA SP installation times VM environment

Taking into account the following hardware parameters:

- VM Flavor: m1.small (2vCPU 4GB RAM 40GB Disk)
- Internet Connection: 50Mbps symmetric

3.1.2 Stress test for the Gatekeeper API

This subsection lists the stress tests designed to assess the performance of the main entry point of the SONATA service platform, the Gatekeeper. Each endpoint of the rich API of the Gatekeeper is tested through a relevant test which is divided workflows to test the API performance from different angles, like considering or not latency introduced by user management, or by the back-end. It is important to note that the tool used for this tests is Gatling [14]. Gatling is a highly capable load testing tool. It is used to stress the gatekeeper APIs with multiples virtual users to get performance metrics.

The Gatekeeper API stress tests that will be described in what follows are:

1. Stress Services API test
2. Stress Functions API test
3. Stress Packages API test
4. Stress Records API test
5. Stress Requests API test
6. Stress VIM API test
7. Stress Users API test
8. Stress Micro-services API test

3.1.2.1 Stress Services API test

The Gatekeeper API exposes the **services** present in the Service Platform. This test intends to stress the Gatekeeper API in order to evaluate the number of **requests per second** to retrieve the **services** that can be handled by the system.

Environment:

For this test the components showed in the Figure 3.4 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper services (**son-gtksrv**)
- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Catalogues and repositories (**son-catalogue-repos**)
- SONATA Keycloak database (**mongodb**)
- SONATA Catalogues and repos Database (**mongodb**)

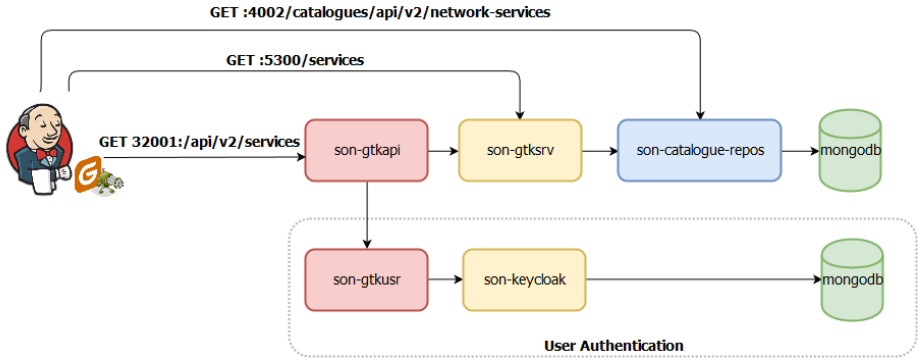

Figure 3.4: Gatekeeper API stress test - services module

Workflow:

Three procedures are considered to accomplish this task. Following this approach, time from the border component and internal component are calculated in order to get detailed data to detect internal delays that can be improved.

Gatekeeper API: This is the normal path that contains the complete chain. Jenkins creates requests to gatekeeper API specifically to `:32001/api/v2/services`. Internally the **son-gkapi** authenticates each request with the user token using **son-gtkusr**. The **son-gtkusr** is the component in charge of validating the user against the **son-keycloak** and its database, **mongodb**. Once the user is validated, the **son-gkapi** passes the request to **son-gtksrv**. After that **son-gtksrv** creates a request to **son-catalogue-repos** to query the **services** in the **mongodb** database. Once the data is available, it is returned from **son-catalogue-repos** to Jenkins.

Gatekeeper services: The gatekeeper **services** module (**son-gtksrv**) is used directly by Jenkins in order to bypass the user authentication, performed by **son-gtkapi** and **son-gtkusr**. In this case Jenkins creates requests to **son-gtksrv** specifically to `:5300/services`. Once the request is received by **son-gtksrv**, it is processed and **son-gtksrv** creates requests to **son-catalogue-repos** in order to retrieve the **services** information.

SONATA Catalogues and Repositories: Jenkins will retrieve the information of the **services** directly from **son-catalogues-repos** using the API: `:4002/catalogues/api/v2/network-services`.

3.1.2.2 Stress Functions API test

The Gatekeeper API exposes the **functions** present in the Service Platform. This test intends to stress the Gatekeeper API in order to evaluate the number of requests per second that can support when retrieving the **functions**.

Environment:

For this tests the components showed in the Figure 3.5 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper functions (**son-gtkfnct**)

- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Catalogues and repositories (**son-catalogue-repos**)
- SONATA Keycloak database (**mongodb**)
- SONATA Catalogues and repos Database (**mongodb**)

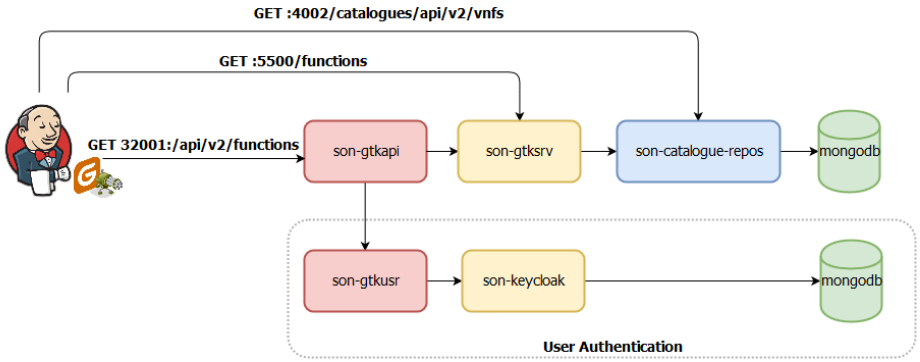

Figure 3.5: Gatekeeper API stress test - functions module

Workflow:

Three procedures are considered to accomplish this task. Following this approach, time from the border component and internal component are calculated in order to get detailed data to detect internal delays that can be improved.

Gatekeeper API This is the normal path that contains the complete chain. Jenkins creates requests to gatekeeper API specifically to `:32001/api/v2/functions`. Internally the **son-gkapi** authenticates each request with the user token using **son-gtkusr**. The **son-gtkusr** is the component in charge of validating the user against the **son-keycloak** and its database, **mongodb**. Once the user is validated, the **son-gkapi** passes the request to **son-gtkfnct**. After that **son-gtkfnct** creates a request to **son-catalogue-repos** to query the **functions** in the **mongodb** database. Once the data is available, it is returned from **son-catalogue-repos** to Jenkins.

Gatekeeper functions The gatekeeper **functions** module (**son-gtkfnct**) is used directly by Jenkins in order to bypass the user authentication, performed by **son-gtkapi** and **son-gtkusr**. In this case Jenkins creates requests to **son-gtkfnct** specifically to `:5500/functions`. Once the request is received by **son-gtkfnct**, it is processed and **son-gtkfnct** creates requests to **son-catalogue-repos** in order to retrieve the **functions** information.

SONATA Catalogues and Repositories Jenkins will retrieve the information of the **functions** directly from **son-catalogues-repos** using the API: `:4002/catalogues/api/v2/vnfs`

3.1.2.3 Stress Packages API test

The Gatekeeper API exposes the **packages** present in the Service Platform. This test intends to stress the Gatekeeper API in order to evaluate the number of **requests per second** that can be supported when retrieving or uploading a **package**.

Environment:

For the test to retrieve packages from the Service Platform, the components showed in the Figure 3.6 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper packages (**son-gtkpkg**)
- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Catalogues and repositories (**son-catalogue-repos**)
- SONATA Keycloak database (**mongodb**)
- SONATA Catalogues and repos Database (**mongodb**)

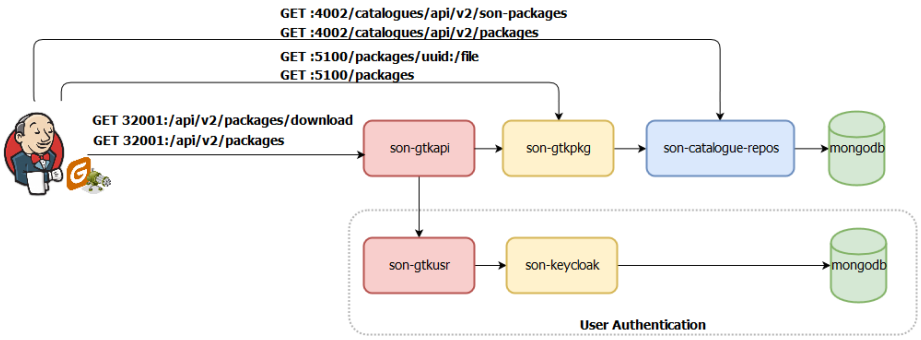

Figure 3.6: Gatekeeper API stress test - packages module - GET

For the test to upload packages to the Service Platform, the components showed in the Figure 3.7 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper packages (**son-gtkpkg**)
- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Catalogues and repositories (**son-catalogue-repos**)
- SONATA Keycloak database (**mongodb**)
- SONATA Catalogues and repos Database (**mongodb**)
- SONATA Package Validator (**son-validator**)
- SONATA Validator database (**redis**)

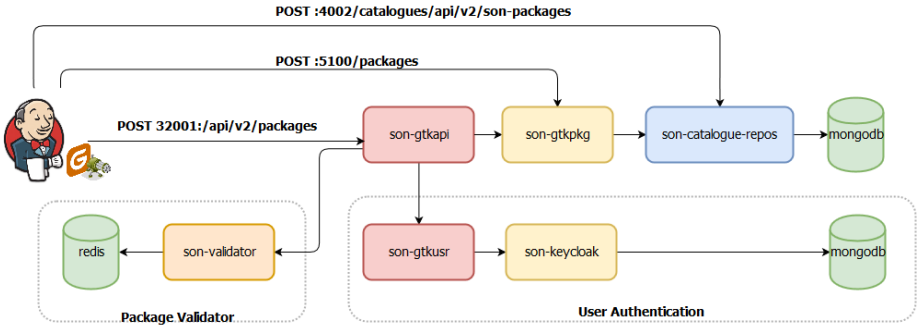

Figure 3.7: Gatekeeper API stress test - packages module - POST

Workflow:

Retrieve package descriptors case:

Three procedures are considered to accomplish this task. Following this approach, time from the border component and internal component are calculated in order to get detailed data to detect internal delays that can be improved.

Gatekeeper API This is the normal path that contains the complete chain. Jenkins creates requests to gatekeeper API specifically to `:32001/api/v2/packages` (package descriptors) and `:32001/api/v2/packages/download` (binary package). Internally the **son-gkapi** authenticates each request with the user token doing use of **son-gtkusr**. The **son-gtkusr** is the component in charge of validating the user against the **son-keycloak** and its database, **mongodb**. Once the user is validated, the **son-gkapi** passes the request to **son-gtkpkg**. After that **son-gtkpkg** creates a request to **son-catalogue-repos** to query the **package descriptors** or **binary package** in the **mongodb** database. Once the data is available, it will be returned from **son-catalogue-repos** to Jenkins.

Gatekeeper package The gatekeeper **packages** module (**son-gtkpkg**) is used directly by Jenkins in order to bypass the user authentication, performed by **son-gtkapi** and **son-gtkusr**. In this case Jenkins creates requests to **son-gtkpkg** specifically to `:5100/packages` and `:5100/packages/:uuid/file`. Once the request is received by **son-gtkpkg**, it is processed and **son-gtkpkg** creates requests to **son-catalogue-repos** in order to retrieve the **package descriptors** or the **binary package**.

SONATA Catalogues and Repositories Jenkins will retrieve the information of the **packages** directly from **son-catalogues-repos** using the API: `:4002/catalogues/api/v2/packages` for package descriptors or
`:4002/catalogues/api/v2/son-packages` for binary packages.

Package on-boarding case:

Three procedures are considered to accomplish this task. Following this approach, time from the border component and internal component are calculated in order to get detailed data to detect internal delays that can be improved.

Gatekeeper API This is the normal path that contains the complete chain. Jenkins POST a SONATA package to gatekeeper API specifically to `:32001/api/v2/packages`. Internally the **son-gkapi** authenticates each request with the user token doing use of **son-gtkusr**. The **son-gtkusr**

is the component in charge of validating the user against the **son-keycloak** and its database, **mongodb**. Once the user is validated, the **son-gkapi** passes the package to **son-validator** in order to check the SONATA package. After that if the package is validated, **son-gtkpkg** creates a request to **son-catalogue-repos** to upload the **package** to the **mongodb** database.

Gatekeeper package The gatekeeper **packages** module (**son-gtkpkg**) is used directly by Jenkins in order to bypass the user authentication, performed by **son-gtkapi** and **son-gtkusr**. In this case Jenkins POST a SONATA package to **son-gtkpkg** specifically to `:5100/packages`. Once the package is received by **son-gtkpkg**, it is processed and **son-gtkpkg** forwards it to **son-catalogue-repos** in order to save the **package** in the **mongodb** database.

SONATA Catalogues and Repositories Jenkins POSTs a SONATA **packages** directly from **son-catalogues-repos** using the API: `:4002/catalogues/api/v2/son-packages`.

3.1.2.4 Stress Records API test

The Gatekeeper API exposes the **records** present in the Service Platform. This test intends to stress the Gatekeeper API in order to evaluate the number of **requests per second** that can support when retrieving the **records**.

Environment:

For this test the components showed in the Figure 3.8 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper services (**son-gtkrec**)
- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Catalogues and repositories (**son-catalogue-repos**)
- SONATA Keycloak database (**mongodb**)
- SONATA Catalogues and repos Database (**mongodb**)

Workflow:

Three procedures are considered to accomplish this task. Following this approach, time from the border component and internal component are calculated in order to get detailed data to detect internal delays that can be improved.

Gatekeeper API This is the normal path that contains the complete chain. Jenkins creates requests to gatekeeper API specifically to `:32001/api/v2/records/services`. Internally the **son-gkapi** authenticates each request with the user token using **son-gtkusr**. The **son-gtkusr** is the component in charge of validating the user against the **son-keycloak** and its database, **mongodb**. Once the user is validated, the **son-gkapi** passes the request to **son-gtkrec**. After that, **son-gtkrec** creates a request to **son-catalogue-repos** to query the **records** in the **mongodb** database. Once the data is available, it is returned from **son-catalogue-repos** to Jenkins.

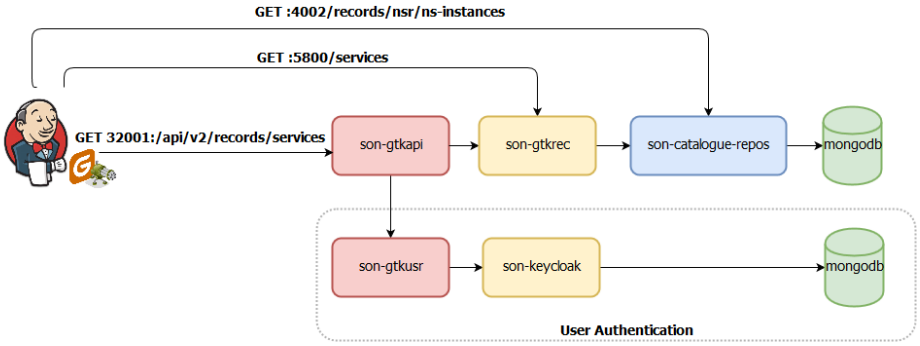

Figure 3.8: Gatekeeper API stress test - records module

Gatekeeper services The gatekeeper **records** module (**son-gtkrec**) is used directly by Jenkins in order to bypass the user authentication, performed by **son-gtkapi** and **son-gtkusr**. In this case, Jenkins creates requests to **son-gtkrec** specifically to `:5800/services`. Once the request is received by **son-gtkrec**, it is processed and **son-gtkrec** creates requests to **son-catalogue-repos** in order to retrieve the **records** information.

SONATA Catalogues and Repositories Jenkins retrieves the information of the **records** directly from **son-catalogues-repos** using the API: `:4002/records/nrs/ns-instances`.

3.1.2.5 Stress Requests API test

The Gatekeeper API exposes the **requests** present in the Service Platform. This test intends to stress the Gatekeeper API in order to evaluate the number of **requests per second** that can support when retrieving the **requests**.

Environment:

The components that are part of this API stress tests are the same as the Stress Services API test. The Catalogue (`son-catalogue-repos`) is a key component that stores the network service descriptors (NSDs) required by the requests API in order to create a new service instantiation. For this test the components showed in the Figure 3.9 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper services (**son-gtksrv**)
- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Catalogues and repositories (**son-catalogue-repos**)
- SONATA Keycloak database (**mongodb**)
- SONATA Catalogues and repos Database (**mongodb**)

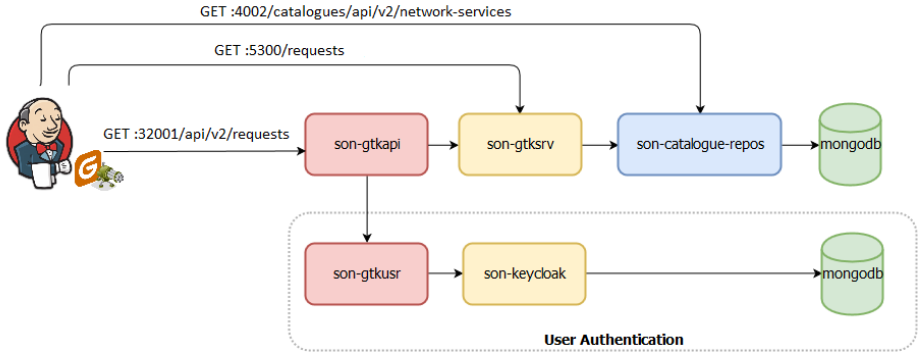

Figure 3.9: Gatekeeper API stress test - requests module

Workflow:

Three procedures are considered to accomplish this task. Following this approach, time from the border component and internal component are calculated in order to get detailed data to detect internal delays that can be improved.

Gatekeeper API This is the normal path that contains the complete chain. Jenkins creates requests to gatekeeper API specifically to `:32001/api/v2/requests`. Internally the **son-gkapi** authenticates each request with the user token using **son-gtkusr**. The **son-gtkusr** is the component in charge of validating the user against the **son-keycloak** and its database, **mongodb**. Once the user is validated, the **son-gkapi** passes the request to **son-gtksrv**. After that **son-gtksrv** creates a request to **son-catalogue-repos** to query the **services** in the **mongodb** database. Once the data is available, it is returned from **son-catalogue-repos** to Jenkins.

Gatekeeper services The gatekeeper **services** module (**son-gtksrv**) is used directly by Jenkins in order to bypass the user authentication, performed by **son-gtkapi** and **son-gtkusr**. In this case Jenkins creates requests to **son-gtksrv** specifically to `:5300/requests`. Once the request is received by **son-gtksrv**, it is processed and **son-gtksrv** creates requests to **son-catalogue-repos** in order to retrieve the **services** information.

SONATA Catalogues and Repositories Jenkins will retrieve the information of the **services** directly from **son-catalogues-repos** using the API: `:4002/catalogues/api/v2/network-services`.

3.1.2.6 Stress VIM API test

The Gatekeeper API exposes the **VIMS** present in the Service Platform. This test intends to stress the Gatekeeper API in order to evaluate the number of **requests per second** that can support when retrieving the **compute vims**.

Environment:

For this test the components showed in the Figure 3.10 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper services (**son-gtkvim**)

- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Catalogues and repositories (**son-catalogue-repos**)
- SONATA Keycloak database (**mongodb**)
- SONATA Message Broker (**son-broker**)
- SONATA Infrastructure Abstraction (**son-sp-infrabstract**)
- SONATA Infrastructure Abstraction Database (**postgres**)

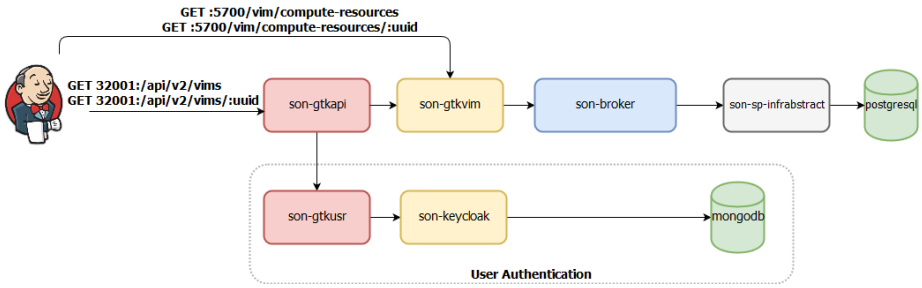

Figure 3.10: Gatekeeper API stress test - vims module

Workflow:

Two procedures are considered to accomplish this task. Following this approach, time from the border component and internal component are calculated in order to get detailed data to detect internal delays that can be improved.

Gatekeeper API This is the normal path that contains the complete chain. Jenkins first creates requests to gatekeeper API specifically to `:32001/api/v2/vims` and then extract the uuid from the payload and creates a second request to `:32001/api/v2/vims/:uuid` . Internally the **son-gkapi** authenticates each request with the user token doing use of **son-gtkusr**. The **son-gtkusr** is the component in charge of validating the user against the **son-keycloak** and its database **mongodb**. Once the user is validated, the **son-gkapi** passes the request to **son-gtkvim**. After that **son-gtkvim** creates a request to **son-sp-infrabstract** through the message broker. This happens in an asyncronous way. The reply from the son-sp-infrabstract goes through the message broker and is available in the reply generated by son-gtkapi when the :uuid generated for the request is used.

Gatekeeper vims The gatekeeper **vim** module (**son-gtkvim**) is used directly by Jenkins in order to bypass the user authentication, performed by **son-gtkapi** and **son-gtkusr**. In this case Jenkins creates requests to **son-gtkvim** specifically to `:5700/vim/compute-resources`. Once the request is received by **son-gtkvim**, it is processed and **son-gtkvim** creates requests through **son-broker** to **son-sp-infrabstract** in order to retrieve the **vims** information. Once the data has been generated by **son-sp-infrabstract**, it is sent through **son-broker** to **son-gtkvim**. This information is going to be available for the next request to **son-gtkvim** `:5700/vim/compute-resources/:uuid`

3.1.2.7 Stress Users API test

The Service Platform User Management API is the communication point between the Gatekeeper API and the User Management - Keycloak components. These components are critical for authentication processes such user identity and access management. This stress test is responsible for taking performance measurements of the component feature to register new users to the Service Platform.

Environment:

The Gatekeeper API and the User Management API stress tests are performed in the SONATA Qualification environment. This environment is the proper scenario to deploy the required components to apply the quality tests and take measures from relevant indicators of performance. In this case, Jenkins and the Gatling Stress tool to are set to configure and perform the stress tests to the User Authentication APIs.

For the User management API User related stress tests, the main actors are the Gatekeeper API (`son-gtkapi`), the User Management API (`son-gtkusr`), the Keycloak tool and the MongoDB support database. For this test the actors and environment showed in the Figure 3.11 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Keycloak database (**mongodb**)

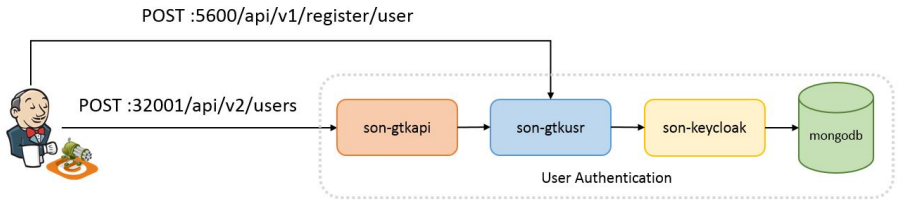

Figure 3.11: User Management API stress tests environment

Workflow:

The first API stress test is based on the user registration feature of the User management. This focus on the measurement of the capability to register a variable high number of users in a short time intervals. Results will indicate which is the impact of this scenario and how the API performs. The workflow for this API stress tests basically perform two main operations:

- First it randomly generates the registration form (JSON object) that is required in order to create a new User Account, with the critical user information.
- Then, it submits the generated JSON object to the registration endpoint in the Gatekeeper API.

Once the registration submission reaches the Gatekeeper API, it forwards the registration message to the User Management API, which performs a series of processes to create a new User Account and store user information.

A second API stress test will be responsible for measuring the performance of the user sign in or login endpoint. This will focus on the capability of the API to support different loads of users signing in to the Service Platform in short time intervals.

A third API stress test will be responsible for measuring the performance of the authorization process for any submitted request to the Gatekeeper API. When a user with an active session sends any request to the Gatekeeper API, the User Management is responsible for validating the identity of the user and process the request to authorize it.

3.1.2.8 Stress Micro-services API test

The Service Platform User Management API is the communication point between the Gatekeeper API and the User Management - Keycloak components. These components are also critical for authentication processes of micro-services identity and access management. This stress test is responsible for taking performance measurements of the component feature to register new Service Platform components as micro-services of the platform.

Environment:

As the User API stress tests, the Gatekeeper API and the User Management API stress tests are performed in the SONATA Qualification environment. This environment is to deploy the required components and apply quality tests in order to take measurements of performance indicators. In this case, Jenkins and the Gatling Stress tool to are set to configure and perform the stress tests to the Micro-service Authentication APIs.

For the User management API Micro-service related stress tests, the main actors are also the Gatekeeper API (`son-gtkapi`), the User Management API (`son-gtkusr`), the Keycloak tool and the MongoDB support database.

For this test the actors and environment showed in the Figure 3.12 are:

- Jenkins + Gatling
- SONATA Gatekeeper API (**son-gtkapi**)
- SONATA Gatekeeper user (**son-gtkusr**)
- SONATA Keycloak (**son-keycloak**)
- SONATA Keycloak database (**mongodb**)

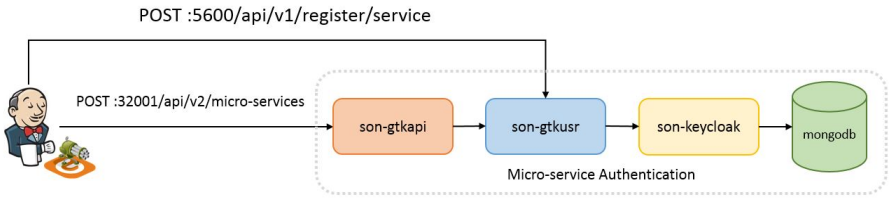

Figure 3.12: User Management API micro-services stress tests environment

Workflow:

This API stress test is based on the micro-service registration feature of the User management. This focus on the measurement of the capability to register a variable high number of micro-services in a short time intervals. Results will indicate which is the impact of this scenario and how the API performs. The workflow for this API stress tests basically perform two main operations:

- First it randomly generates the registration form (JSON object) that is required in order to create a new Service Account, with the critical micro-service information.
- Then, it submits the generated JSON object to the registration endpoint in the Gatekeeper API.

Once the registration submission reaches the Gatekeeper API, it forwards the registration message to the User Management API, which performs a series of processes to create a new Service Account and store user information.

A second API stress test will be responsible for measuring the performance of the micro-service sign in or login endpoint. This will focus on the capability of the API to support different loads of micro-services signing-in to the Service Platform in short time intervals.

In this case, a third API stress test is not required for the authorisation process, as it is the same for the users' case. There is no difference in that point of the process between users or micro-services.

3.1.3 Stress test for Service Platform Catalogue API

The Service Platform Catalogue API is a key communication point between the Gatekeeper and the Catalogue component. This API is responsible for servicing and attending different types of requests such read, submit or update various elements as packages and descriptors. The purpose of this test is to stress the Catalogue API with a series of bursts of requests for the different supported elements in the Catalogue. This way, it is possible to measure the performance of the API.

The Catalogue API stress tests are performed in the SONATA Qualification environment. This environment allows a proper scenario to deploy the required components and configure different settings in order to run different stress tests and take performance measurements. This environment holds Jenkins and the bash scripts that are responsible for configuring test variables and load the Python scripts that perform the stress processes and take measurements from the test actors. Those test actors are the Catalogue API component and the MongoDB database.

The stress tests are implemented in Python. These tests are basically scripts divided by element type, and perform post/submit package and descriptors operations. In order to fill the Catalogue and stress the API, each stress test submits elements at different burst rates simultaneously. The Python code implements several methods, which creates and populates the request body elements with random values and then submits the request using threads. The Python code is called by the bash scripts that are set under the Jenkins configuration. The Figure 3.13 represents a global view of the stress tests and the environment.

One of the elements used to perform the stress tests is the SONATA file package (son-packages), a binary file which includes:

- 1 SONATA Package descriptors (PDs)
- 1 SONATA Network Service descriptors (NSDs)
- 1 SONATA Virtual Network Function descriptors (VNFDs)

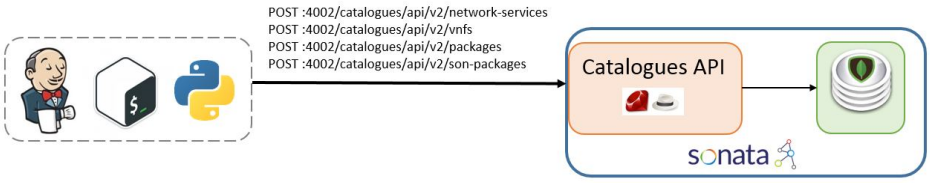

Figure 3.13: Global view of the catalogue stress test

Three tests are performed, one for each type of stored element, and for each test different burst rates of requests are applied:

- 10 descriptors
- 100 descriptors
- 1000 descriptors

While the son-packages binary files are submitted to the Catalogue API, the measurements for performance are taken from the contents of the files, so the following results show the API performance for the descriptor elements of the Catalogue. The measurements are based on the elapsed time to complete the submission and the rate of submit or post per second. Three test rounds have been applied with the same conditions to consider variability of the measures. Tests results are presented in the following charts Table 3.2, Table 3.3 and Table 3.4:

Table 3.2: Stress Catalogue API results (PDs)

Element	Request entries	Elapsed time	Rate	Round
Package descriptors	10	0.20131778717 s	49.6727097022 posts/s	1
Package descriptors	100	0.829627990723 s	120.535952401 posts/s	1
Package descriptors	1000	7.45192098618 s	134.193586037 posts/s	1
Package descriptors	10	0.108062982559 s	92.538626671 posts/s	2
Package descriptors	100	0.726701974869 s	137.607992627 posts/s	2
Package descriptors	1000	6.78188800812 s	147.451564933 posts/s	2
Package descriptors	10	0.0844738483429 s	118.379832293 posts/s	3
Package descriptors	100	0.634788990021 s	165.528467965 posts/s	3
Package descriptors	1000	6.0412569046 s	165.528467965 posts/s	3

Table 3.3: Stress Catalogue API results (NSDs)

Element	Request entries	Elapsed time	Rate	Round
Network service descriptors	10	0.39290189743 s	25.4516459844 posts/s	1
Network service descriptors	100	3.77645492554 s	26.4798606025 posts/s	1
Network service descriptors	1000	34.2661888599 s	29.1832863026 posts/s	1
Network service descriptors	10	0.347470998764 s	28.7793802521 posts/s	2
Network service descriptors	100	3.45433592796 s	28.9491242558 posts/s	2
Network service descriptors	1000	34.6839900017 s	28.8317462885 posts/s	2
Network service descriptors	10	0.354546070099 s	28.2050792361 posts/s	3
Network service descriptors	100	3.32253408432 s	30.0975091488 posts/s	3
Network service descriptors	1000	31.9779469967 s	31.271550988 posts/s	3

Table 3.4: Stress Catalogue API results (VNFDs)

Element	Request entries	Elapsed time	Rate	Round
Network function descriptors	10	0.20876789093 s	47.9000863372 posts/s	1
Network function descriptors	100	1.93367195129 s	51.7150801785 posts/s	1
Network function descriptors	1000	20.7767319679 s	48.1307648163 posts/s	1
Network function descriptors	10	0.218561172485 s	45.7537809039 posts/s	2
Network function descriptors	100	1.95065498352 s	51.2648319897 posts/s	2
Network function descriptors	1000	23.5728750229 s	42.421639237 posts/s	2
Network function descriptors	10	0.235634088516 s	118.379832293 posts/s	3
Network function descriptors	100	0.634788990021 s	157.532662935 posts/s	3
Network function descriptors	1000	6.0412569046 s	165.528467965 posts/s	3

3.1.4 Stress test for the Monitoring Framework components

The target of this section is two-fold: first, to describe in brief the stress tests that have been executed in order to prove the robustness of the Monitoring Framework primary components, and secondly to present some of the results.

Figure 3.14 depicts a general overview of the components comprising the Monitoring Framework under stress test.

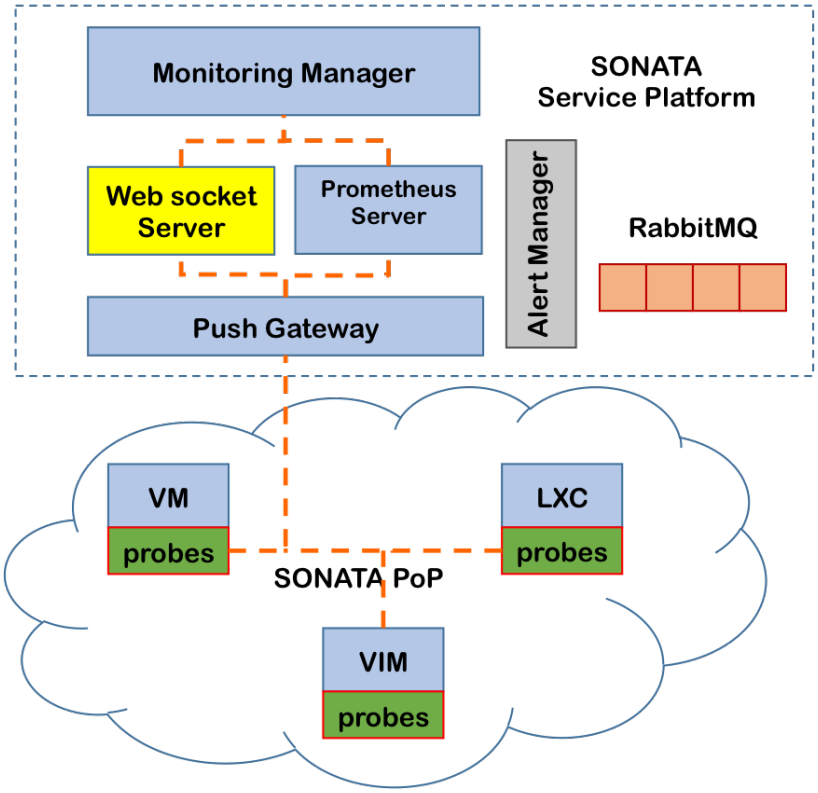

Figure 3.14: SONATA Centralised Monitoring Y1

It is noticed that the tests are bash scripts that can even run automatically in a periodic manner.

The following Figure 3.15 shows the execution of the tests in the Jenkins server used by SONATA partners.

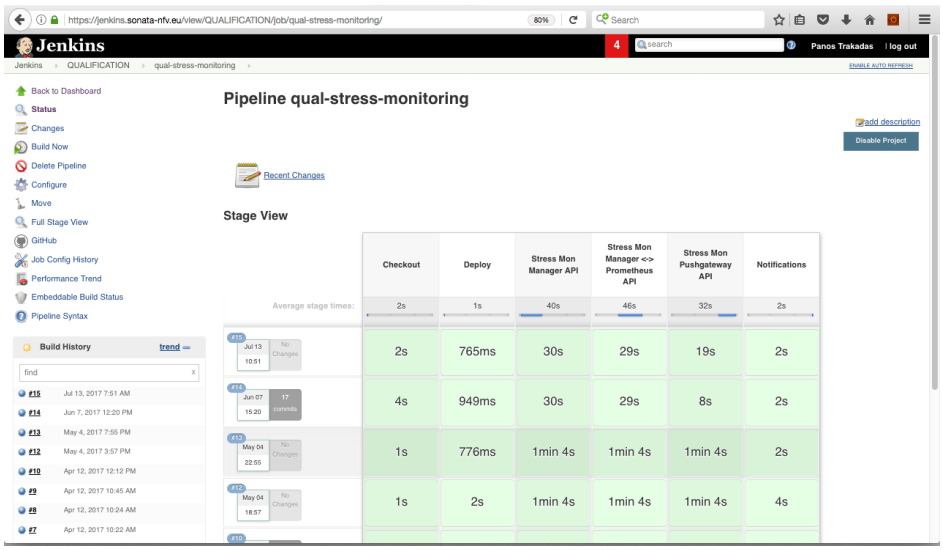

Figure 3.15: Test execution in Jenkins server

3.1.4.1 Stress test for Monitoring Manager

The purpose of this test is to stress the API of the Monitoring Manager with bursts of HTTP GET methods.

Two tests are performed, as described below:

1. Send concurrently 10 requests until the total number reaches 1000.
2. Send concurrently 100 requests until the total number reaches 1000.

The results of the second test can be observed in Figure 3.16 and Figure 3.17.

3.1.4.2 Stress test between Monitoring Manager and Prometheus Server

The purpose of this test is to stress the network connectivity between the Monitoring Manager and the Prometheus Server with bursts of HTTP GET methods received at the Monitoring Manager.

In this test, the Monitoring Manager must retrieve information from Prometheus Server in order to serve the request.

Two tests are performed, similar to the previous case:

1. Send concurrently 10 requests until the total number reaches 1000.
2. Send concurrently 100 requests until the total number reaches 1000.

The results of the second test can be observed in Figure 3.18 and Figure 3.19.

3.1.4.3 Stress test for Push Gateway

The purpose of this test is to stress the Push Gateway with bursts of requests.

Two tests (harder than the previous ones) are performed, similar to the previous case:

1. Send concurrently 100 requests until the total number reaches 10000.
2. Send concurrently 1000 requests until the total number reaches 10000.

The results of the second test can be observed in Figure 3.20 and Figure 3.21.

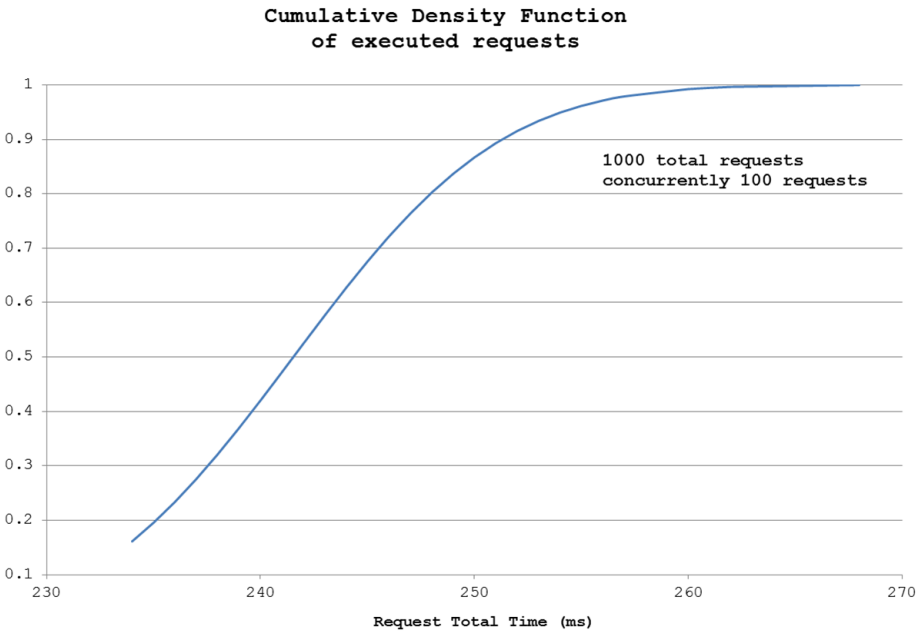

Figure 3.16: Monitoring Manager stress test results 1

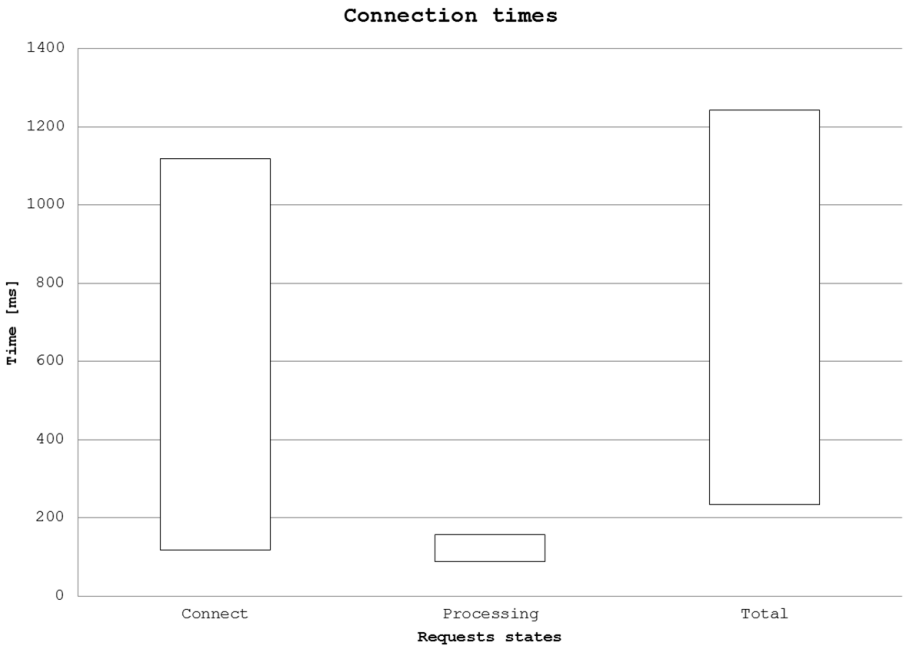

Figure 3.17: Monitoring Manager stress test results 2

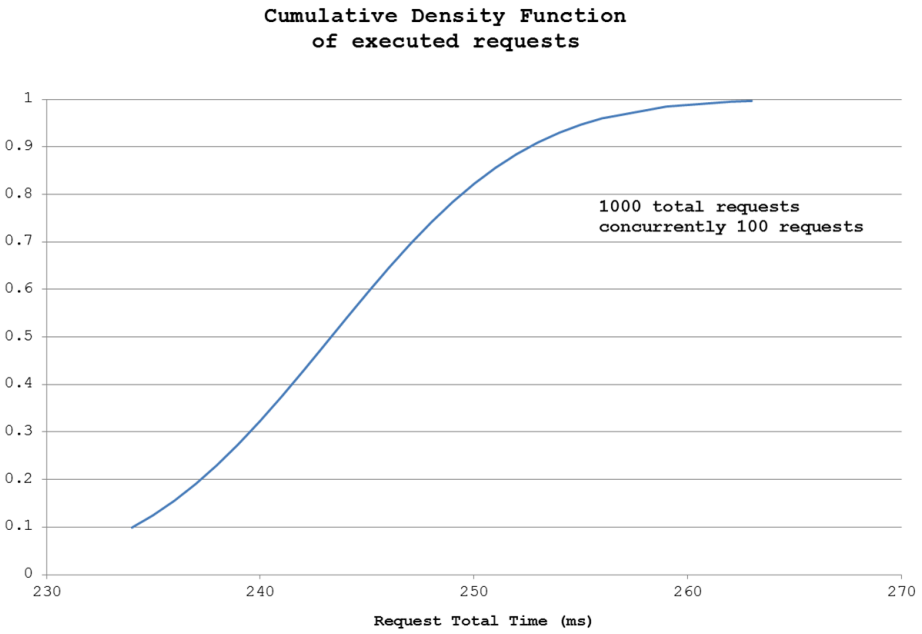

Figure 3.18: MonMan - Prometheus stress test results 1

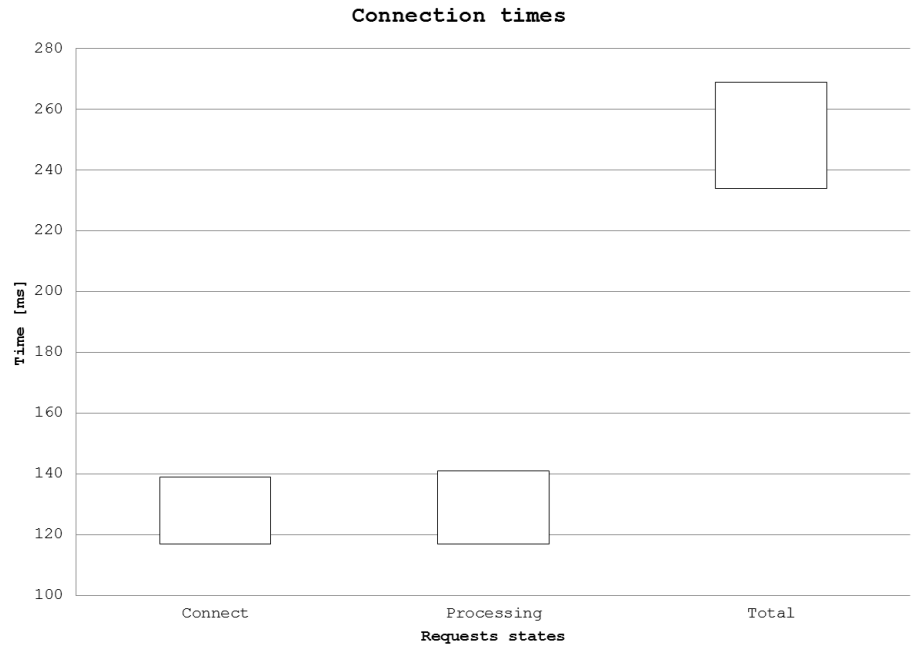

Figure 3.19: MonMan - Prometheus stress test results 2

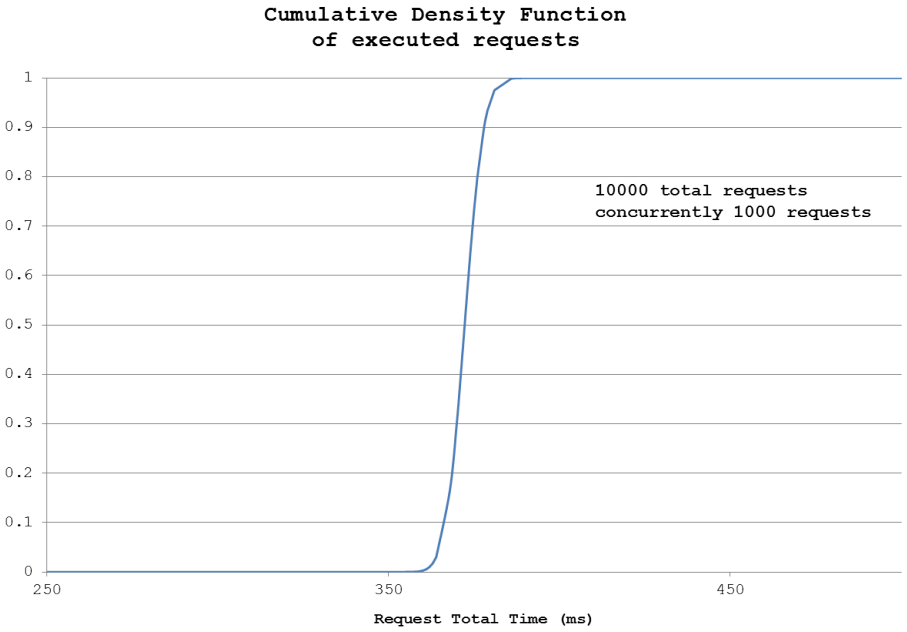

Figure 3.20: Push Gateway stress test results 1

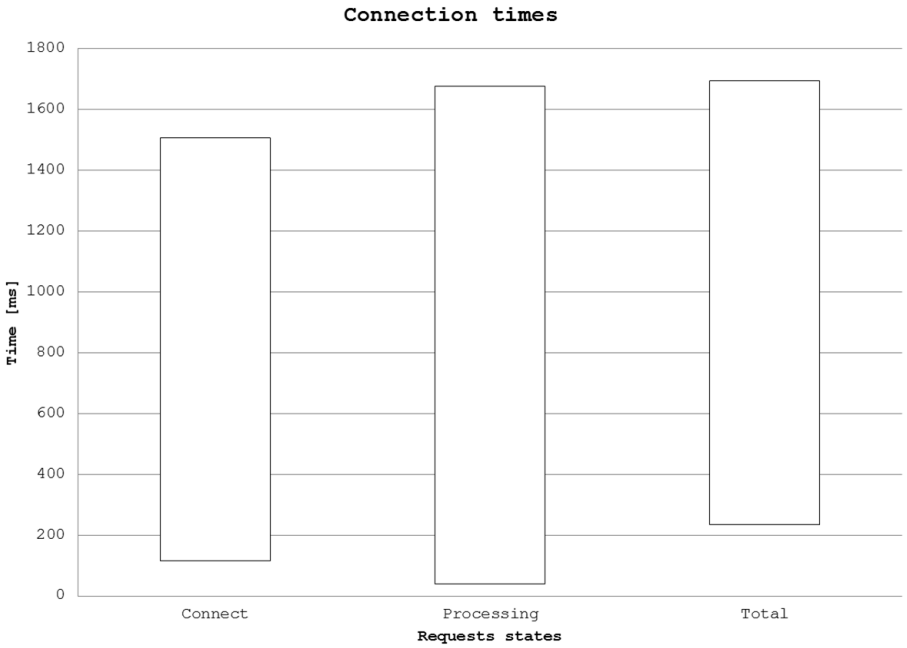

Figure 3.21: Push Gateway stress test results 2

3.1.5 Stress test for the Infrastructure Abstraction

This set of tests assesses the ability of the Infrastructure Abstraction layer to buffer requests and ensure their completion in spite of the delay introduced by the underlying processes needed by the VIM, thus ensuring an effective decoupling between the MANO processes and the virtual infrastructure layer. The tests produce a burst of requests for service preparation, which involves the pre-provisioning of VM images and the creation of virtual networks and other infrastructural virtual elements, and function deployment, which involves translating VNF descriptors and actual deployment on the VIM. Relevant parameters of the test are the number of parallel clients sending requests to the IA, the number of sequential requests each client will send to the IA before shutting down, and the timeout after which the request is considered to be expired. In order to perform the test without actually stressing the underlying infrastructure, which is shared with all the other development, integration and qualification processes and tasks of the project, the IA has been provided with a Mock VIM Wrapper (see [6], [7] and [11] for more information on VIM wrappers) which is able to serve the above mentioned requests without using a real VIM. In order have a realistic setting for the test, we used our integration environment to collect data on the delay introduced by the actual VIMs and used the collected statistics to configure the Mock VIM wrapper. A service platform instance has been created and configured to use 30 of these mocked NFVI-PoPs for the following set of tests.

3.1.5.1 Infrastructure Abstraction load test

The first set of tests has been performed with different levels of parallelism. Each client has been set to send 10 sequential messages, and to wait no more than 60 seconds for a response. Tests have been performed with a number of client varying from 10 to 5000.

Figure 3.22, Figure 3.23, Figure 3.24, Figure 3.25, Figure 3.26 and Figure 3.27 show on the left the estimated probability density function of the response delay for successfully completed requests, and on the right the bandwidth, expressed in requests per second, of the completed requests and failed requests, along with the overall average value in dotted lines.

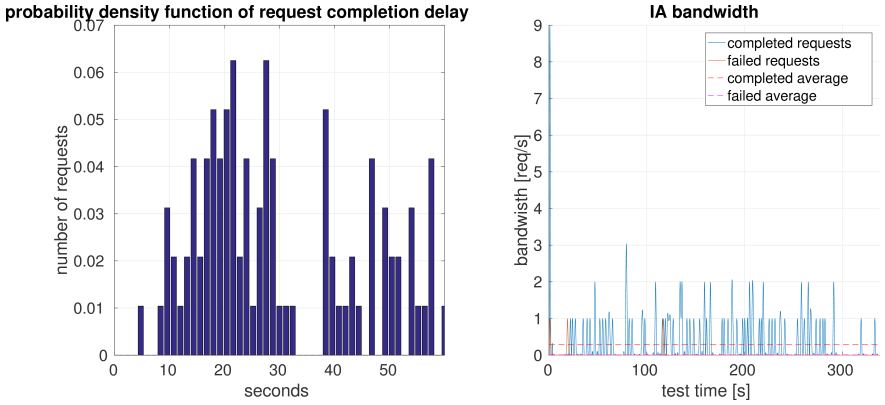

Figure 3.22: Average delay PDF and request bandwidth with 10 clients

3.1.5.2 Infrastructure Abstraction stability test

The second set of tests has been performed to assess performances and stability on a longer time horizon, we considered 500 clients sending 500 sequential messages, for a total load of 250000

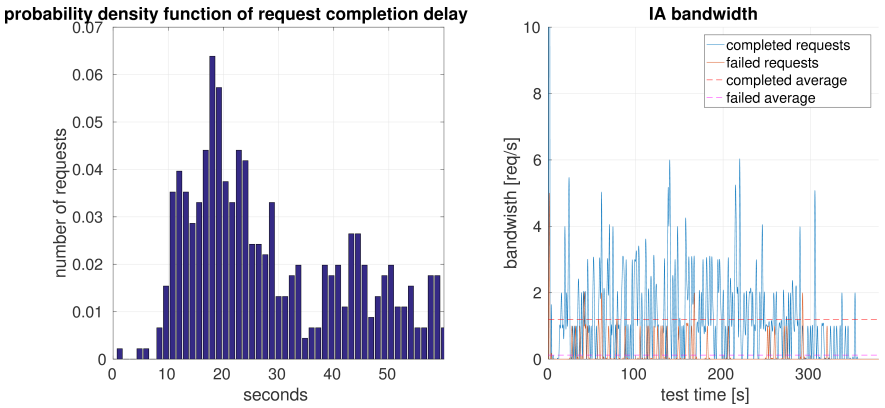

Figure 3.23: Average delay PDF and request bandwidth with 50 clients

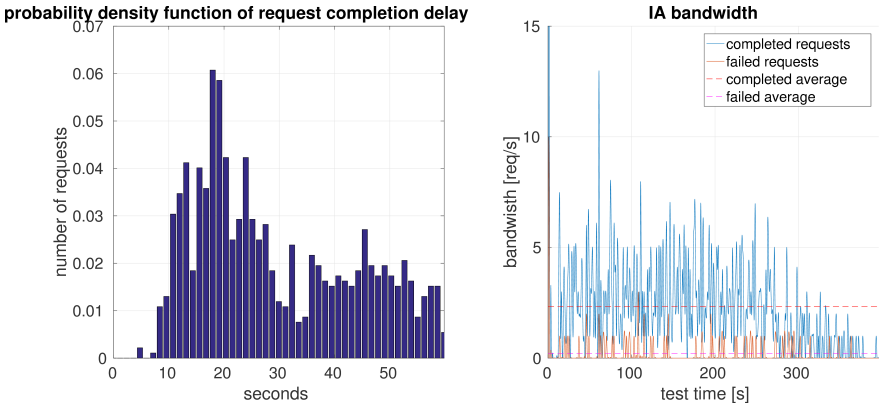

Figure 3.24: Average delay PDF and request bandwidth with 100 clients

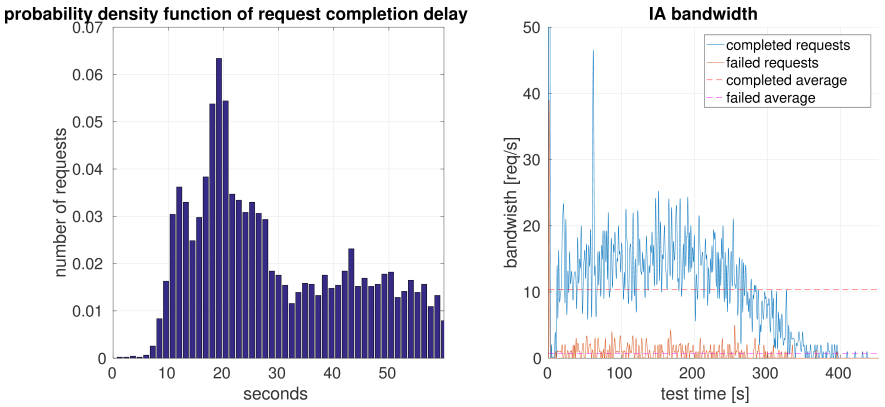

Figure 3.25: Average delay PDF and request bandwidth with 500 clients

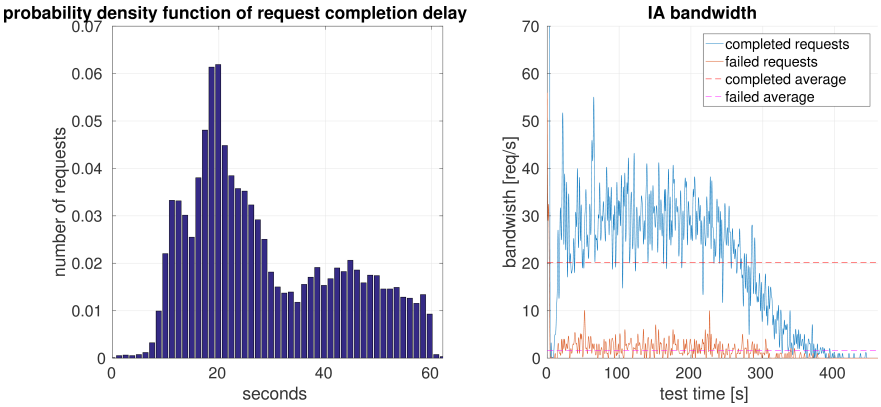

Figure 3.26: Average delay PDF and request bandwidth with 1000 clients

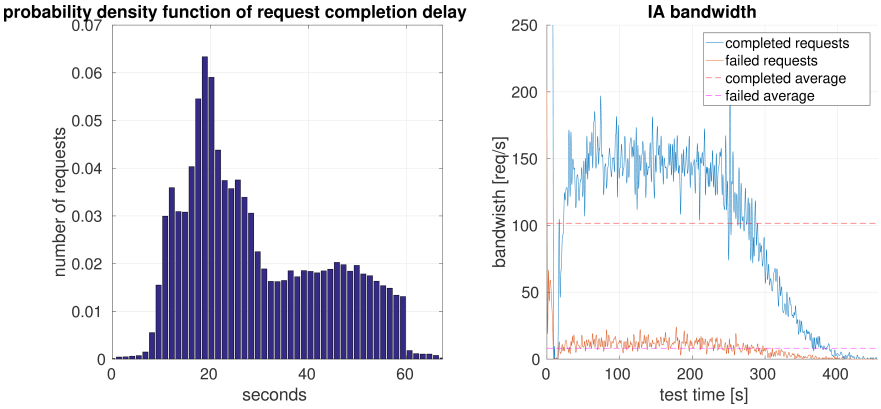

Figure 3.27: Average delay PDF and request bandwidth with 5000 clients

requests. For this test, the timeout has been set to the higher value of 120 seconds. Figure 3.28 and Figure 3.29 show the collected result for this test.

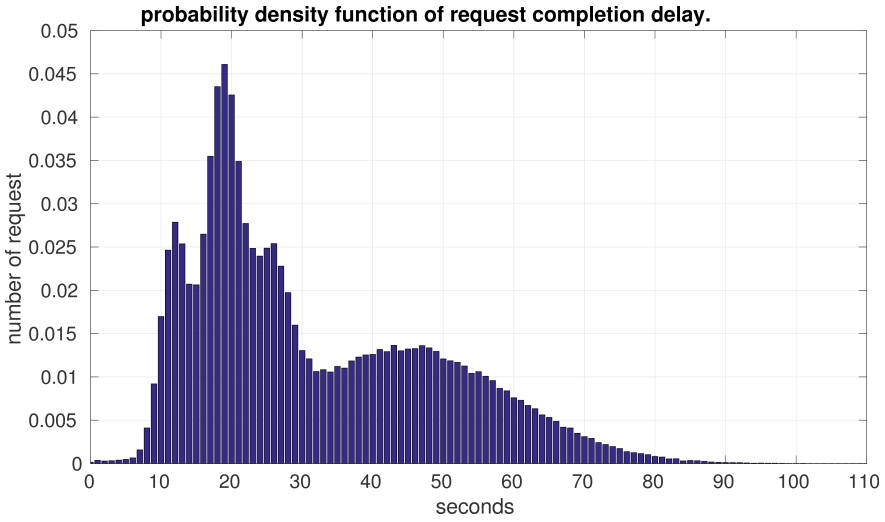

Figure 3.28: Request delay with 500 clients sending 500 requests, timeout set at 120s

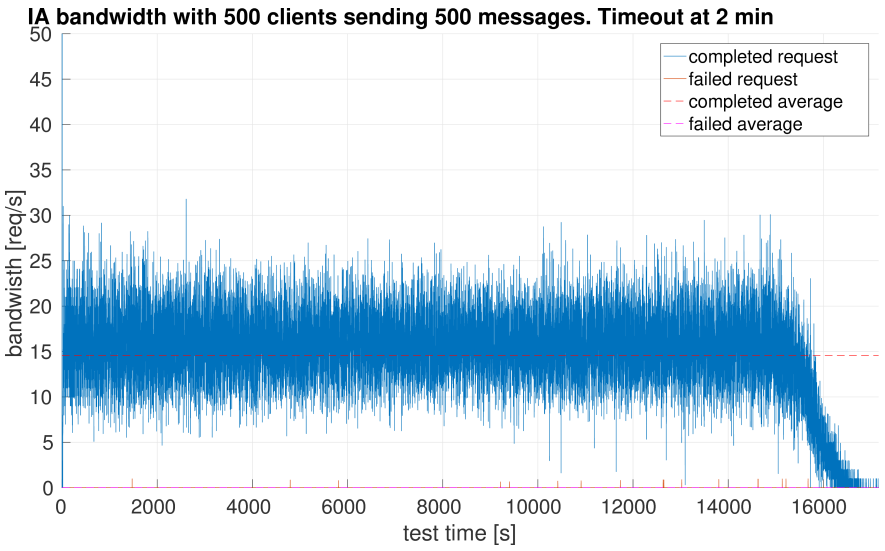

Figure 3.29: Request bandwidth with 500 clients sending 500 requests, timeout set at 120s

Also in this test, where a higher number of total requests are served, the IA is able to handle the load with an average request completion bandwidth of 15 requests per second and a negligible failure rate of 0.000064, where a request is considered as failed if the client timeout expires.

3.1.5.3 Infrastructure Abstraction stress test final remarks

Finally, the result presented above have also been compared to highlight performance dependency from the level of parallelism used by the IA client.

Figure 3.30 shows the box plot of the response delay for completed requests as the number of parallel clients vary from 10 to 5000. Red lines in the box plot represent the median, black asterisk

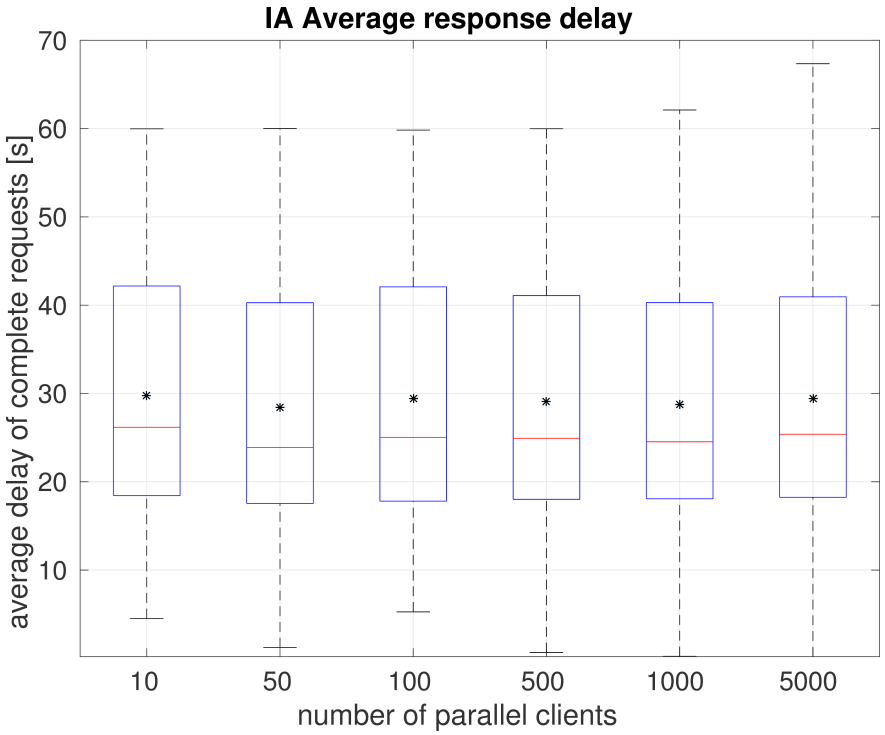

Figure 3.30: Box plot of delay for completed request as a function of the number of parallel clients

average values. As shown, the IA performance in terms of response delay remains unaffected by the amount of load, also when it increments of one order of magnitude.

Figure 3.31 shows how the success rate of the IA is affected by the number of parallel clients interacting with it. It is evident how an increment of the number of clients of one order of magnitude generated a limited increment in the failure rate.

3.1.6 Stress test for the SDK Emulator

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 (see [4], [5] and [10]. To evaluate its performance under stress, we conducted a set of experiments which investigate the emulator's behaviour in differently sized usage scenarios. All experiments described in this section have been done on a bare-metal machine with only the emulation platform installed on top a fresh Ubuntu 16.04 LTS installation. The used machine has an Intel(R) Core(TM) i5-4690 CPU running at 3.50GHz with 24GB memory and 200GB disk size. We evaluated the emulator performance in two important categories: Emulated topology setup and service deployment as described in the following two sections.

3.1.6.1 Emulated Topology Setup

The first experiment category investigates the behaviour of the emulation platform for different sizes of the topology that is emulated. To do so, the emulator was started with 1 to 100 PoPs to be emulated. For each number of PoPs three different kinds of topology were tested:

1. Linear topology: All PoPs are connected linearly to form a long chain of PoPs, e.g., PoP1

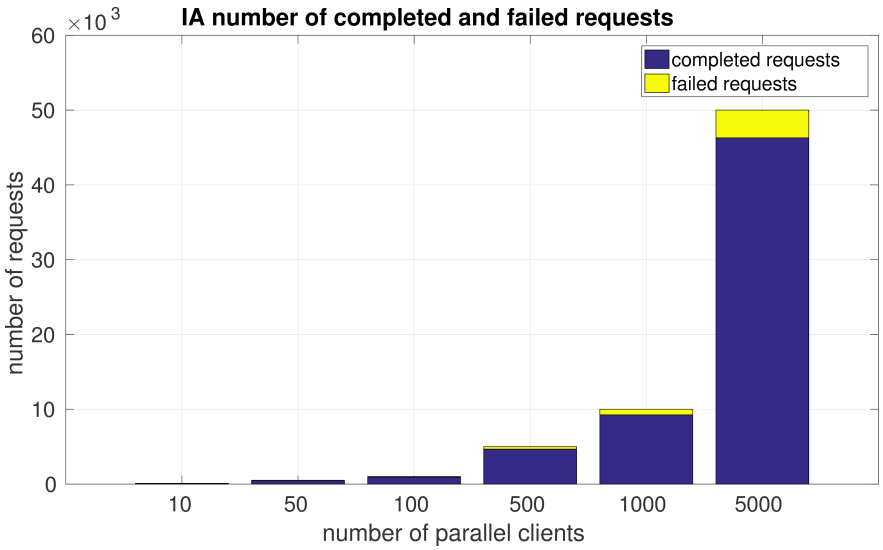

Figure 3.31: Completed VS failed requests number as a function of the number of parallel clients

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.

All experiments in this section have been repeated ten times to get statistical confidence about the results.

The first metric that is investigated with this experimental setup is the total setup time of the emulation platform. This is the time it takes from starting the emulator until the complete topology is booted, configured and ready to be used. Figure 3.32 shows the results for the *linear* and *star* topologies. It shows that the start-up time of the emulation platform scales nearly linearly with the number of emulated PoP when these topologies are used. In contrast to this, the system behaves differently if the *full mesh* topology is used, as shown in Figure 3.33. Here the number of emulated links between the PoPs growth exponentially which causes the platform to be limited to 50 emulated PoPs. It can also be seen that the start uptime heavily increases when a *full mesh* topology is used. However, such a topology clearly defines an edge case that will be rarely discovered in typical emulation setups.

To further investigate the start-up behaviour of the platform, the total start-up 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 creation: Time taken to create all PoPs that should be part of the emulated topology.

3. PoP interconnection: 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.

Figure 3.32: Start-up times for different numbers of emulated PoPs for linear and star topologies

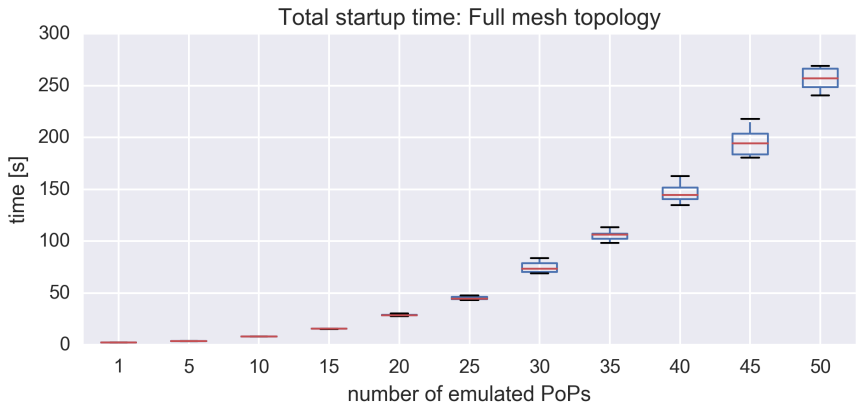

Figure 3.33: Start-up times for different numbers of emulated PoPs for full mesh topologies

Figure 3.34 shows this breakdown for the three different topology setup and 1 to 100 PoPs. It can be seen that the *environment boot* process is constant and is not affected by the size of the emulated topology. The *PoP creation* and *PoP interconnection* 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 *full mesh* topology case. This is again due to the exponentially growing number of required links to form the *full mesh* topology.

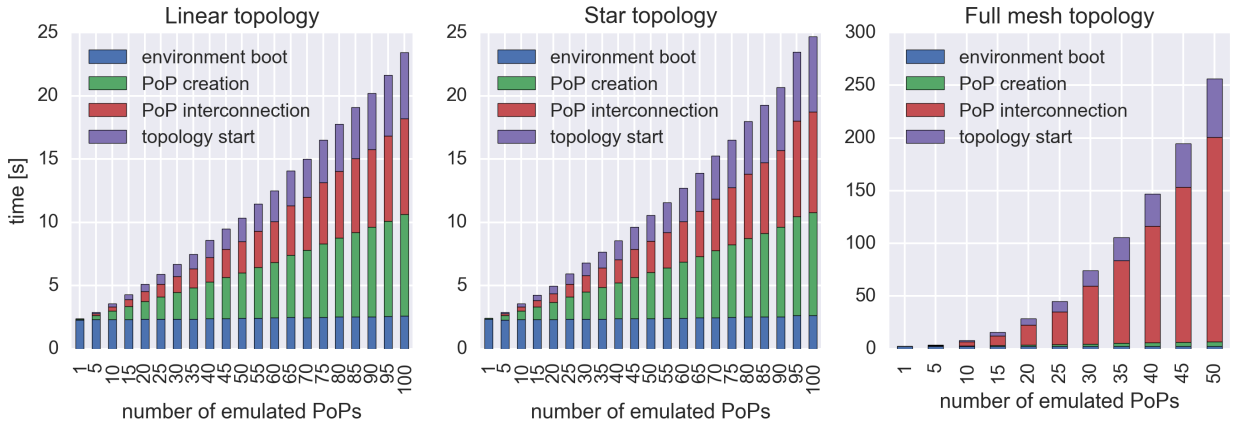

Figure 3.34: Emulation startup time breakdown

In addition to the time metric, we also investigated the memory consumption of the emulator for these experiments. Figure 3.35 shows the used memory for the *linear* and *star* topology case and Figure 3.36 shows it for the *full mesh* case. It shows that the memory consumption scales linearly with the number of emulated PoPs and links.

3.1.6.2 Service Deployment Times

The second experiment category investigates the behaviour of the emulation platform when services, consisting of a different number of VNFs, are deployed in a single emulated PoP. To do so, services consisting of 0 to 100 VNFs are deployed on the emulation platform. Each of these experiments is again repeated ten times. The services are built of empty VNFs that are all running the *Ubuntu:trusty* Docker base image without any additional software installed in the containers. The first metric that is investigated is the *start-up time* of a service as shown in Figure 3.37. It can easily be seen that the required time to deploy a service in the emulator linearly scales with the number of involved VNF containers. It also shows the great advantage of our lightweight emulation platform: A user can deploy extremely large network services (e.g., 100 VNFs) in a couple of seconds on a single physical machine.

The second metric that is investigated is the memory consumption during this experiment. Figure 3.38 shows again a linear relationship. However, it is important to note that this heavily depends on the used VNF software that is executed inside the containers and can barely be generalised.

3.1.7 End-to-End tests

In [12] we documented and described some qualification tests aimed at qualifying the SONATA environment as a whole. In this section, we present again some details of these tests for the sake of completeness and document the results obtained in terms of completion time of the complete test and of the single phases it is composed of.

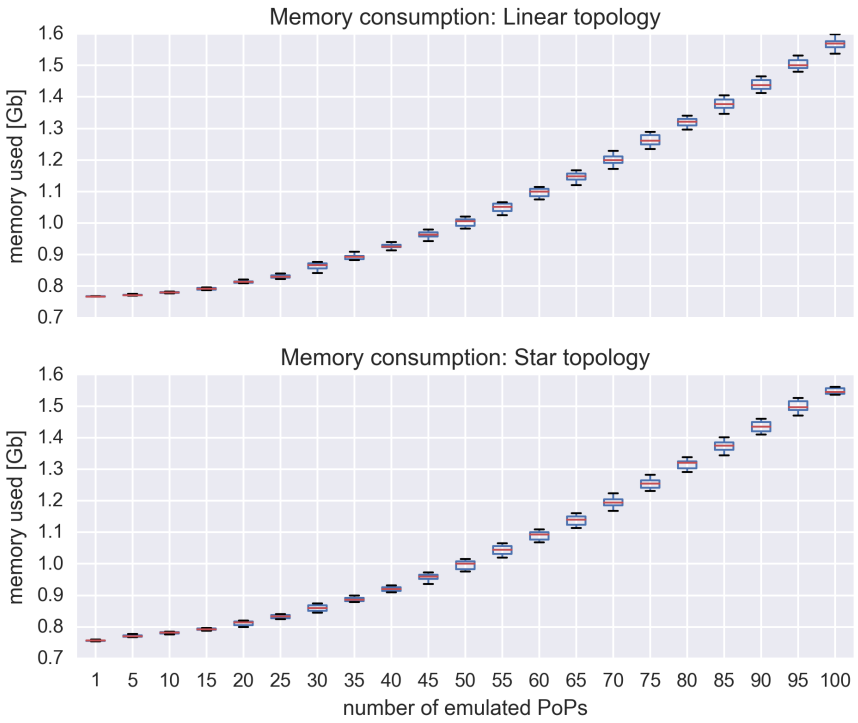

Figure 3.35: Memory consumption for different numbers of emulated PoPs for linear and star topology

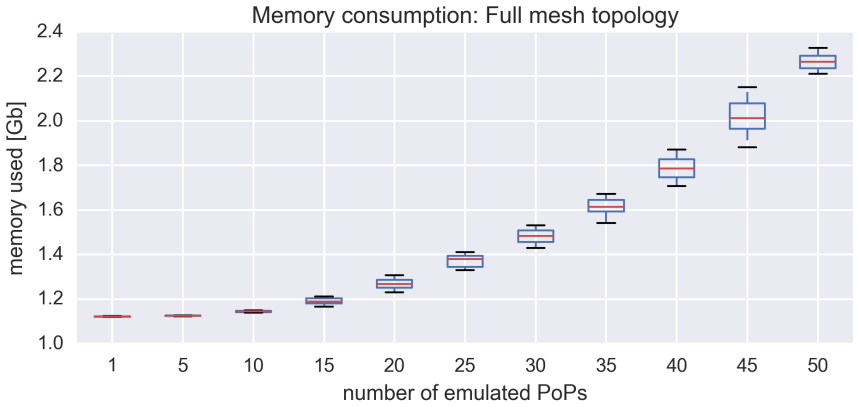

Figure 3.36: Memory consumption for different numbers of emulated PoPs for full mesh topology

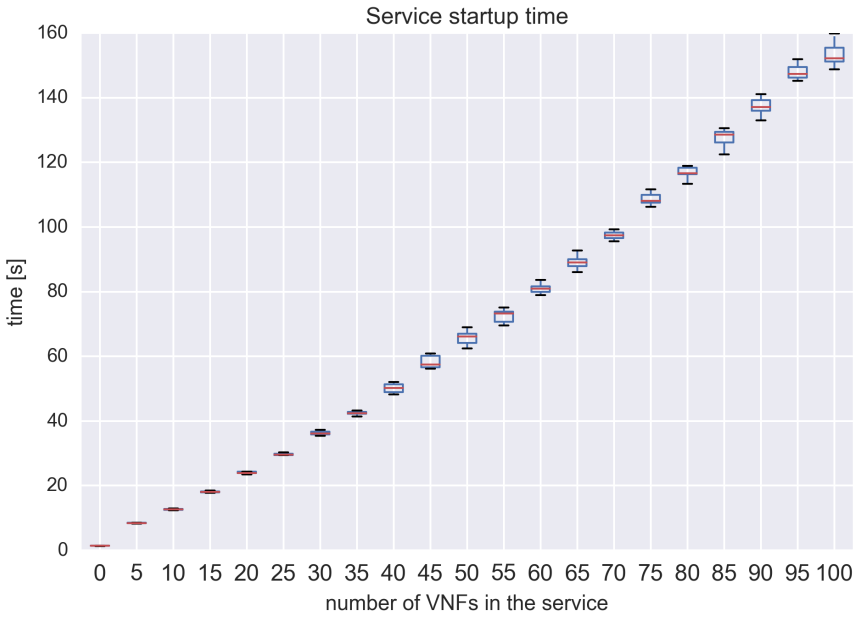

Figure 3.37: Service start-up times for changing number of deployed VNFs

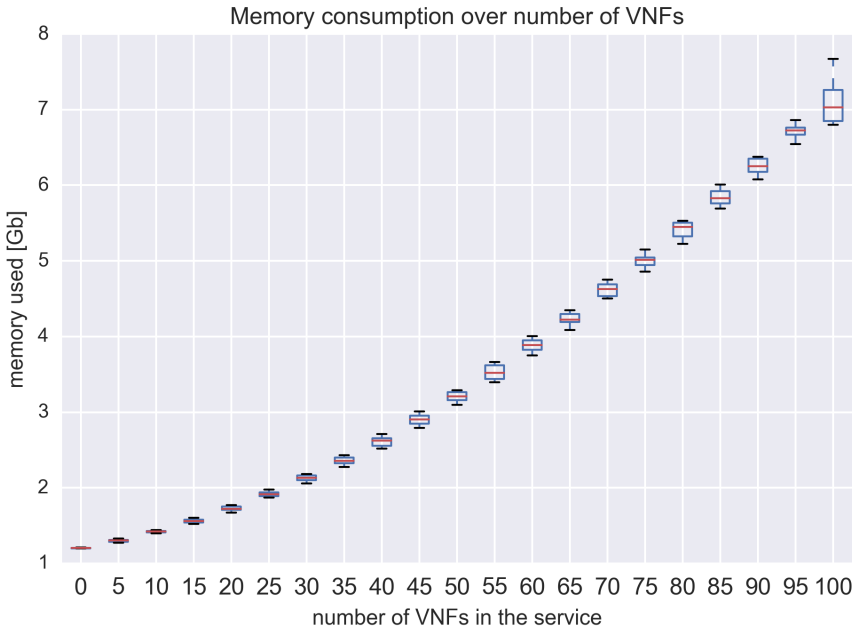

Figure 3.38: Memory consumption for changing number of deployed VNFs

3.1.7.1 1 VNF 1 PoP

The goal of this test is to deploy a Network Service composed by 1 VNF to be deployed in an NFVI-PoP in order to tests the functionalities of the SONATA Service Platform.

Environment:

For this test the following components and tools are needed:

- Jenkins
- SONATA-SDK
- SONATA Descriptors
- SONATA-SP
- SDN switch
- 1x vRING VNF (Dummy VNF)
- 1x Openstack PoP
- Traffic generator (Iperf)
- Generator and Sink Machine

Workflow:

The Jenkins uses the SONATA SDK to build a package that contains the NS under test. Once the package is created, Jenkins automatically pushes it to the Service Platform and trigger the instantiation of this Network Service. In order to measure the results of the tests, a probe was developed inside the VNF. Jenkins wait until the NS is deployed making requests to gatekeeper API. When Jenkins receives the **"READY"** status of the service deployed, the environment is ready for the test. Jenkins contact the generator machine and trigger the Iperf. After that, Jenkins makes requests to the VNF in order to get the results of the tests. If the Iperf traffic is detected going through the VNF, then the probe installed report "true" or "false".

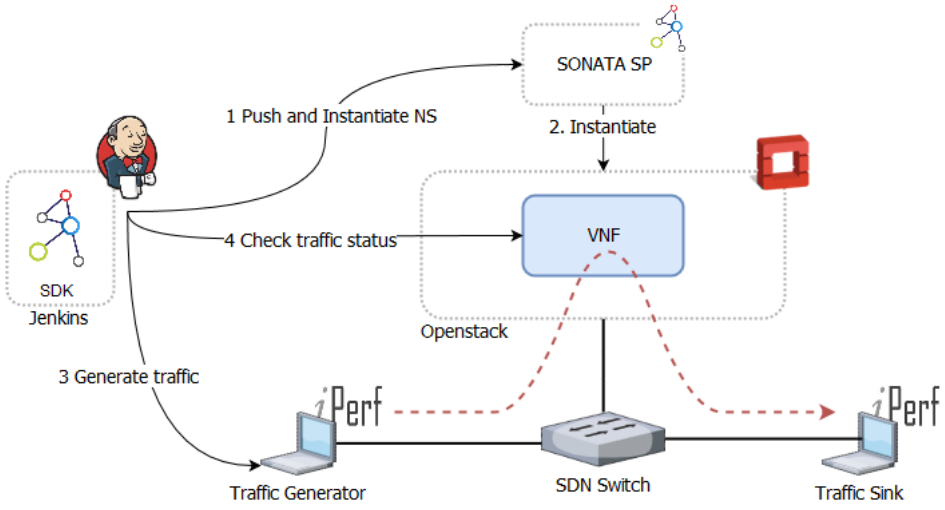

Figure 3.39: Diagram of the 1 VFN 1 PoP test with the relevan elements

Results:

The sections of the test are represented in the table below.

Package Creation	Platform Preparation	Instantiation	Traffic Generation	Traffic Detection	Total
7:24	9:07	1:00	1:15	0:01	18:47

From the table above we can conclude The test takes in average 19 minutes,

3.1.7.2 Two VNFs - One PoP

Environment:

For this test the following components and tools are needed:

- Jenkins
- SONATA-SDK
- SONATA Descriptors
- SONATA-SP
- SDN switch
- 2x vRING VNF (Dummy VNF)
- 1x Openstack PoP
- Traffic generator (Iperf)
- Generator and Sink Machine

Workflow:

This is a similar test described above (1VNF 1PoP) but with the addition of deploy 2 VNFs. The reason of this test is to qualify the SFC between 2 VNFs inside the PoP. Jenkins uses the SONATA SDK to build a package that contains the NS under test. Once the package is created, Jenkins automatically push it to the Service Platform and trigger the instantiation of this Network Service. In order to measure the results of the tests, a probe was developed inside the VNF. Jenkins wait until the NS is deployed making requests to gatekeeper API. When Jenkins receives the **"READY"** status of the service deployed, the environment is ready for the test. Jenkins contacts the generator machine and trigger the Iperf. After that, Jenkins makes requests to the two VNFs in order to get the results of the tests. If the Iperf traffic is detected going through the VNF, then the probe installed report "true" or "false".

Results:

The sections of the test are represented in the table below.

Package Creation	Platform Preparation	Instantiation	Traffic Generation	Traffic Detection	Total
7:34	9:06	1:25	1:15	0:05	19:25

From the table above we can conclude The test takes in average 20 minutes,

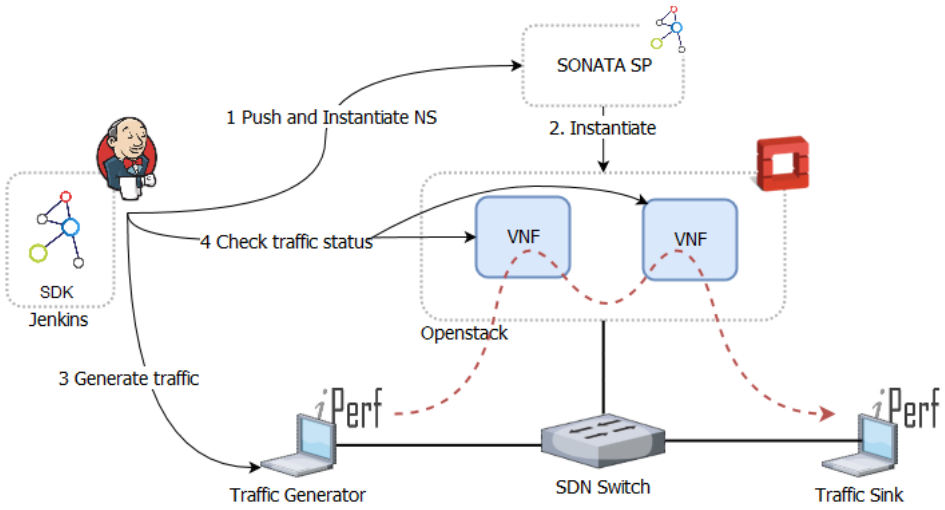

Figure 3.40: Diagram of the 2 VFN 1 PoP test with the relevan elements

3.1.7.3 Two VNFs - Two PoPs

Environment:

- Jenkins
- SONATA-SDK
- SONATA Descriptors
- SONATA-SP
- SDN switch
- 2x vRING VNF (Dummy VNF)
- 2x Openstack PoPs
- Traffic generator (Iperf)
- Generator and Sink Machine

Workflow:

The Jenkins uses the SONATA SDK to build a package that contains the NS under test. Once the package is created, Jenkins automatically pushes it to the Service Platform and triggers the instantiation of this Network Service. In order to measure the results of the tests, a probe was developed inside the VNF. Jenkins wait until the NS is deployed making requests to gatekeeper API. When Jenkins receives the **"READY"** status of the service deployed, the environment is ready for the test. Jenkins contacts the generator machine and trigger the Iperf. After that, Jenkins makes requests to the VNF in order to get the results of the tests. If the Iperf traffic is detected going through the VNF, then the probe installed report "true" or "false".

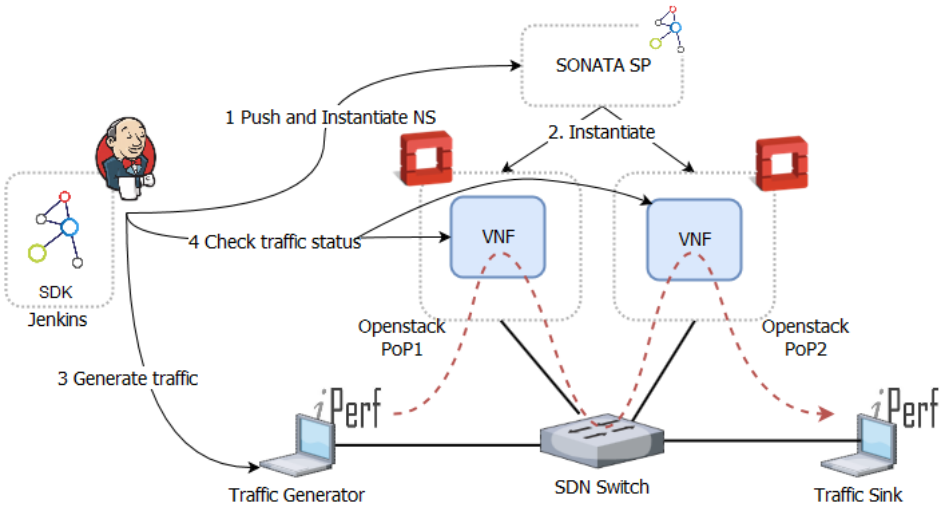

Figure 3.41: Diagram of the 2 VFN 2 PoP test with the relevan elements

3.2 Final remark on CI/CD and Agile development in SONATA

In this section we draw some conclusions on the adoption of the agile development philosophy and the CI/CD development model from the early stage of the SONATA project. It is of high importance to remark that the outcome of SONATA is not to produce a commercial software system ready for the market, but to push the boundaries of the state of the art by providing an innovative application of the DevOps strategies to the world of network services and of NFV in the context of network softwarisation. Nonetheless, the project set as a top priority the adoption of these strategies and agile methods for the development of SONATA software artefacts themselves. This decision was driven by two main objectives:

- Assess the effectiveness and efficiency of these development and management methods in the distributed and heterogeneous context of an EU project consortium
- Provide software artefacts that were more solid, reliable, tested and smoothly integrated from the very early stage of the project.

Since WP5 finalises its task on test infrastructure and system testing in M24 (contextual to the submission of this document), we can now highlight to which extent these objectives have been met and draw some lessons learnt from this experience.

As a first general consideration, we want to highlight that these development methods don't come for free. As documented in [3], they need a transient phase where a study of the state of the art in terms of protocols and practices and a careful selection of the most appropriate software tools must be carried out. This phase is of paramount importance, since the selection of the wrong tool or the adoption of the wrong practice can slow down features development and jeopardise the quality of the produced artefact, and a late switch in adopted technologies and practices is highly expensive. After this phase, a second transition is needed to train developers to the selected procedures, also considering the necessary time to climb the learning curve of the management tools adopted. This time delay introduced by this approach, given its importance and its impact on the effectiveness of the approach itself, can represent one of the main disadvantages of its adoption. Nevertheless, even considering the above mentioned delays introduced by the adoption of these methodologies and as we described in details in [8], our implementation of the CI/CD philosophy and the adoption of an

automated software development cycle, facilitated by the set of tools we selected in the first phase of the project, allowed us to complete the implementation and integration of the first prototype of the SONATA framework within one year from the beginning of the project.

During our development, our software development approach also helped us in underlining some shortcoming of our initial design. First, during our development, thanks to our integration phase, we were able to identify how son-schema [4][6], the schema for our manifests (NSD, VNFD, NSR and VNFR), had two issues: on one side it was too verbose on some aspects, being inefficient to parse for machines (micro-services) and difficult to read and write for humans (developers), and on the other side, it was lacking some important information that was generated during the service instantiation or orchestration. Second, the presence of a local catalogue in the SDK (see [4]) resulted in data duplication, with the inefficiency this brings. Moreover, not only our CI/CD approach helped us to correct the design on these aspects, but also facilitated the seamless integration of the changes in the software that this new design brought, allowing us to rely on our wide set of tests to ensure stability during this enhancement process.

There is another critical point we want to highlight: also after the initial transient phase of set-up, the CI/CD methodology and the associated practices need a constant effort to be maintained and extended as the scope of the requirements extends and as the number of features and software interfaces increases. In this regard, in our context where a consistent subset of the software developer was also in charge of maintaining the set of integration and qualification tests used in the different phases of our CI/CD pipeline, this maintenance effort represented an overload of the available development capabilities of the consortium, also considering that this effort is spent in parallel with the actual software development effort.

As a final consideration, we want to stress that the two critical points highlighted above must be read in the light of one final and most important advantage of these methodologies: the software environment that comes out as the product of this pipeline is more stable and more reliable. Embracing automated testing in each step of the software development process inherently produces stronger software. Furthermore, this automated testing system, coupled with the micro-service design of the SONATA framework, facilitates contribution from developers coming from other software communities. At the time of writing this document, several contributions from external developers have been received on our open source repositories, and through our system, we have been able to test them and integrate them very easily.

In conclusion, although *breaking the silos* is an expensive procedure in terms of effort and time in the first phase of the project, and it needs for constant care and dedicated effort to be maintained and updated, the adoption of the DevOps approach and the agile development philosophy have been major drivers for the positive outcome of SONATA.

4 Additional Interfaces and VNF Development for pilots

Along the design and maintenance of the testing, integration and qualification procedures for the SONATA project, Work Package 5 also deals with the development of additional building blocks and components that may be required during the integration and validation process. In particular, its tasks include to provide Service Platform self-monitoring facilities allowing critical operational information to be shared with the platform owner; realise the interface to the OSS in order to be able to trigger various situations for testing the system behaviour; develop the appropriate modules for SLA management; and develop VNFs that will be used for system testing. In this section, we document the design consideration and the outcome of our development effort on these aspects.

4.1 SLA Management interface

This sub-section addresses the **Service Level Agreement (SLA) Management** Interface the SONATA Service Platform provides. We start by giving a high-level overview of the issue and then provide a possible solution to integrate an SLA Management system with the SP.

4.1.1 Overview

With the low barrier to entry claimed by the 5G standards, many more actors will have access to common infrastructures, such as SONATA's Service Platform (see [11], and [12] for a discussion on the different kinds of actors that might be involved in an SLA) or the physical infrastructure. Naturally, some of these actors, either by the nature of its business (e.g., a health monitoring devices service provider, who will want its devices to have a stable access to the network connecting them, so that monitored people would not run unnecessary risks) or as an added value/competitive advantage that is worth selling (e.g., a simple WiFi provider that wants a certain bandwidth to be guaranteed to its customers) will require guarantees on the level of service they get from and through this infrastructure. SONATA does not go into this complex eco-system, but we believe we provide a set of tools that, connected to the adequate (external) systems that manage these SLAs, allow the implementation of such business models.

Figure 4.1 shows the various actors that can be considered when implementing SLAs in a scenario such as the one we have in SONATA.

In real/concrete scenarios, these actors may play more than a role. These actors are:

- **Service Developer**: this kind of users provide packages containing service and function descriptors, as well as, all files that allow the deployment of that service and/or functions, preferably by using the SONATA SDK;
- **SP Owner:** owns and manages the SP (might be the same as the **Infrastructure Owner** or not);
- **Infrastructure Owner:** owns the Infrastructure, i.e., the hardware and the VIMs on top of it (might be the same as the **SP Owner** or not);

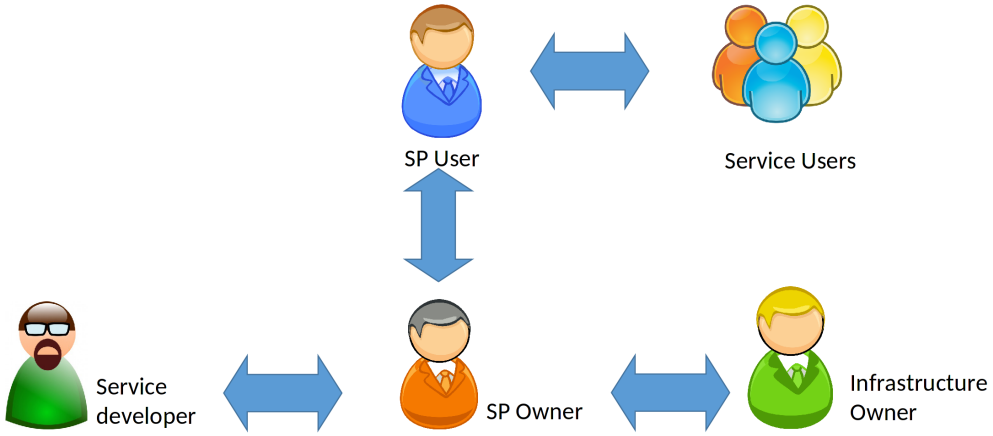

Figure 4.1: Different SLA actors

- **SP User:** or **Customer**, has access to the BSS, where services can be instantiated, updated, stopped, etc.;
- **Service Users:** in a Business-to-business (B2B) scenario, these are the final users, who use the service the **SP User** has 'sold' to them.

Any of these different actors may require the definition of at least one SLA with the other actors. And the agreement of a certain service level with one of the actors might imply some changes in other SLAs with the remaining actors.

Therefore, we left the concrete implementation of an SLA Management system out of the scope of the project, but we think we provide all the tools it can be integrated or built with, as we describe in the following sub-section.

4.1.2 Possible solution

Any solution for SLA Management that complements the SONATA SP will use the following modules, as suggested:

- **User Management:** to know who the user is, with a degree of confidence that is needed. In certain scenarios, the current implementation of the user management module can evolve to accept, e.g., multi-factor-authentication [16], such that higher and required security standards are met. Knowing the user is crucial to be able to manage SLAs with those users. **User creation**, **logging-in/out** and **user profile** are APIs that are already available through the **Gatekeeper** (see [11]), which can be used out-of-the-box from the SLA Management system;
- **Licence Management:** different types of licences can be created and managed, and then connected to specific SLAs, therefore maintaining a tight control of what can and can not be used by a given user within the SP. Licences can be stored out side of the SP (e.g., in a more traditional App Store), thus supporting already common business models, but also opening the door to more advanced business models suggested by 5G;
- **KPIs Management:** based on all the (usage) KPIs that are collected and on the type of licence a given user has, which can be exported, SLA breaks can be calculated and verified with high precision, and therefore monetization of contracts can be built upon.

So, depending on the concrete scenario to be implemented, the SONATA SP supports the adequate mechanisms to manage SLAs in a very flexible way.

4.2 Interface toward OSS systems

In this subsection, we describe how can the SONATA Service Platform (SP) be used to in the whole **Operations Support Systems** eco-system of the SP owner. We first give a short overview and scope definitions of the problem, and then suggest how a SONATA SP can be integrated into the OSS of the SP Owner.

4.2.1 Overview

ETSI has defined the interface between this new generation of Service Platforms and Orchestrators to which the SONATA SP belongs to a long time ago (see [15] and [2]). In Figure 4.2 this interface (or **reference point**, which supports several interfaces) is called **Os-Ma**.

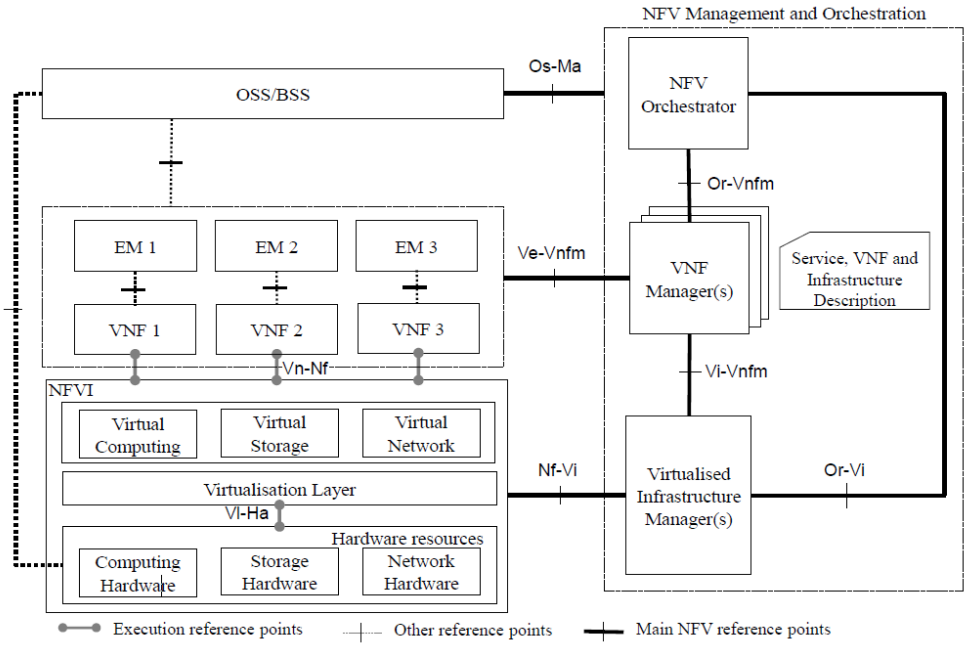

Figure 4.2: OSS interface in the ETSI MANO architecture

The **Os-Ma** reference point described in [15] must support the whole lifecycle of a service and its functions. These interfaces are ([15]):

1. support the NSD Management interface produced by the NFVO;
2. support the NS Lifecycle Management interface produced by the NFVO;
3. support the NS Lifecycle Change Notifications interface produced by the NFVO;
4. support the NS Performance Management interface produced by the NFVO;
5. support the NS Fault Management interface produced by the NFVO;
6. support the VNF Package Management interface produced by the NFVO;

7. any interaction on this reference point concerning a VNF shall be associated with at least one NS instance;

This kind of interfaces are very specific to the concrete solution chosen by a SONATA SP Owner, and therefore we have decided to consider the implementation of a unique interface with external OSSs as being out-of-scope of the project. Nevertheless, in the following sub-section, possible solutions for such an implementation is described in detail.

4.2.2 Possible solution

In this subsection, we describe how SONATA's SP can be used in an integration with the OSS/BSS eco-system of the SP owner.

Taking into account the SP's architecture ([6], [7] and [11]) and the above presented list of requirements, a possible solution could be the following:

- use the **Gatekeeper's API**'s (GK API) ability to:
  - accept, validate and on-board packages containing service and function descriptors, as well as all the artefacts needed for an automatic deployment of those services and functions, to implement requirements (**#1**);
  - accept requests for service's and/or function's meta-data (**#1**);
  - accept requests for service instantiation, update or termination (**#2**), and then pass to the MANO Framework and its (default) **SLM**/**FLM** (and receive feedback from them) the events that imply changes in the lifecycle. If non-default behaviour is needed, **Specific Service** or **Function Managers** can be provided by the service developer, which can securely interact with the rest of the platform;
  - accept requests for **synch/asynch monitoring data** and user registration/profile update requests which includes the user's email and mobile phone, thus supporting notifications (**#3**);
  - accept, validate and on-board packages containing either links to or the effective (VM) images/containers (**#6**);
- use the **BSS**'s ability to:
  - accept user registration/profile update requests which includes the user's email and mobile phone (**#3**);
  - collect user actions to instantiate, update or terminate a service (**#2**);
- use the **GUI**'s ability to:
  - accept user registration/profile update requests which includes the user's email and mobile phone (**#3**);
  - provide user's views on performance (**#4**) and faults (**#5**), both through detailed monitoring data and the platform's KPIs;
- use the **Monitoring Manager**'s ability to:
  - collects, store and provide monitoring data either automatically (**#3**) or upon request (**#4** and **#5**);

Requirement **#7** above is guaranteed by design, namely through the validation of the NSD/VNFD schema ([10], [11]).

Therefore, we think that the whole SONATA eco-system can easily be integrated with any OSS/BSS solutions in the market, given its **current and off-the-shelf features** and **very flexible architecture**.

4.3 Service Platform monitoring

Although the main objective of the Monitoring Framework is to collect, process and store data related to the performance of the network services and the virtual network functions, it also provides the ability to monitor the performance of the Service Platform itself, including the components it consists of.

To this end, SONATA Monitoring Framework is using monitoring probes on several parts of the Service Platform for collecting data. This process follows the same approach as the data collected from NS and VNF, including the Monitoring Server which collects, analyses and stores the data, the Monitoring Manager providing an API for accessing data and the Alert Manager to notify the users in case of events generation, as described in [6], [7] and [11].

In particular, the Monitoring Framework collects data from 1) the Virtual Machine that hosts the Docker Engine, where the Service Platform components will be deployed, 2) the different NFVIs being part of the Service Platform infrastructure and 3) the containers (deployed on top of the Docker Engine) that comprise the Service Platform.

The metrics collected from the Virtual Machine are shown in Figure 4.3 and include metrics related to computation, memory and network resource utilisation, such as the percentage of CPU utilisation, the free memory in MB, the number of transmitted and received packets per second, etc.

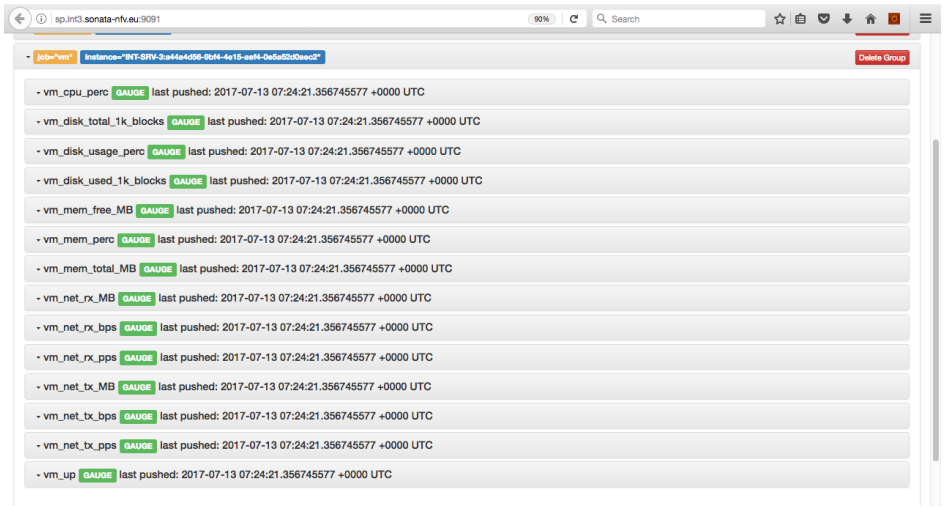

Figure 4.3: List of metrics collected from VM

Similar is the list of metrics collected from the containers being part of the Service Platform functionality, again related to network, memory and computation resources, as shown in Figure 4.4.

Finally, SONATA Monitoring Framework collects information from different NFVIs, as part of the Service Platform. In this case, the list of metrics collected from an OpenStack-based NFVI is shown in Figure 4.5, including (per tenant) the number of Security Groups, the number of floating IPs in use, the memory usage limit, etc.

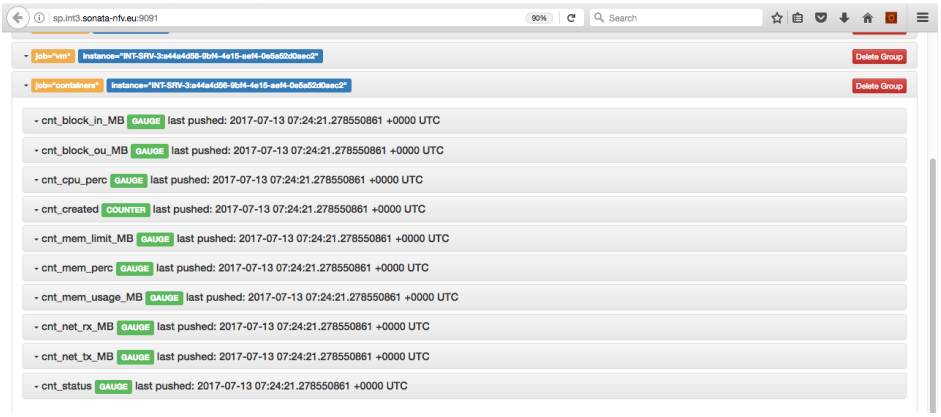

Figure 4.4: List of metrics collected from containers

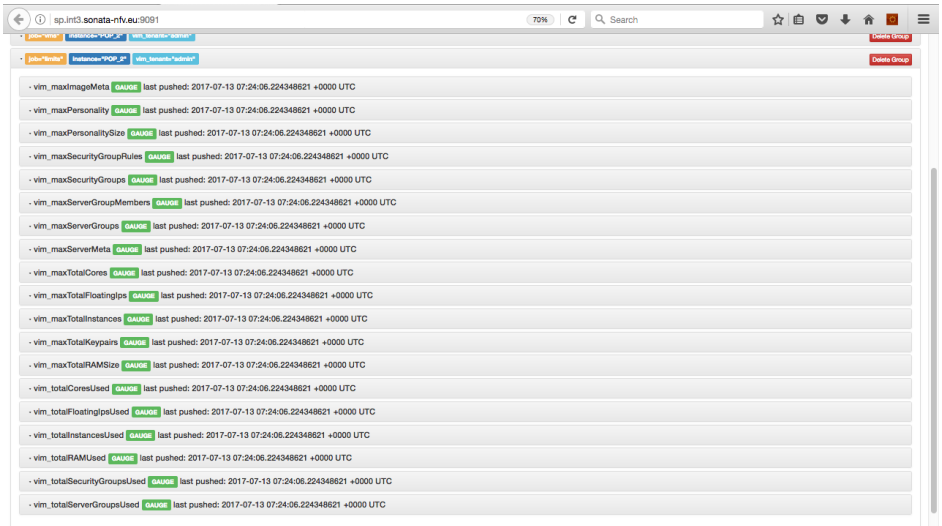

Figure 4.5: List of OpenStack limits

The collected set of data becomes available to the user through the Gatekeeper Graphical User Interface (GK-GUI) and the respective functionality of the Monitoring Manager.

Figure 4.6 depicts performance data of the Service Platform, Figure 4.7 the status (along with other information) of all the containers deployed within the Service Platform, while Figure 4.8 shows performance data related to Prometheus server container, deployed within the Service Platform.

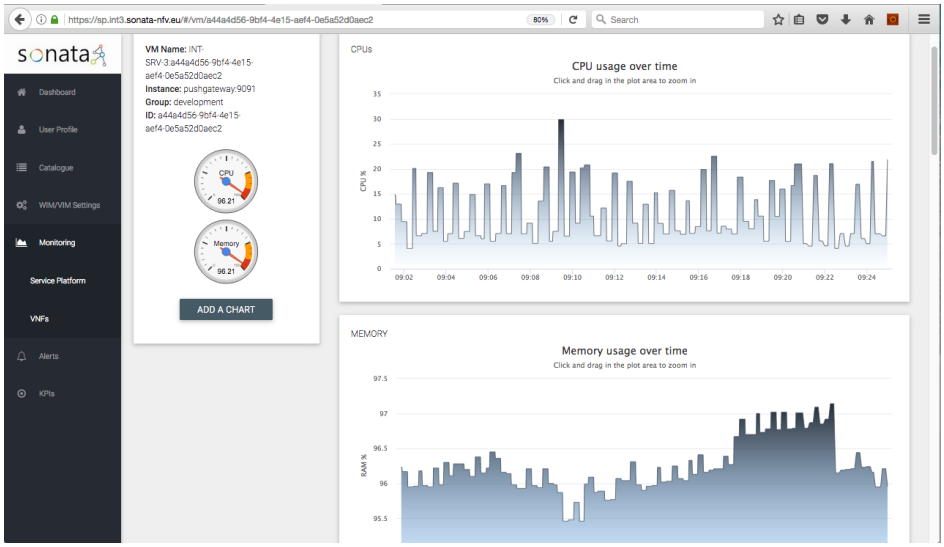

Figure 4.6: Service Platform performance

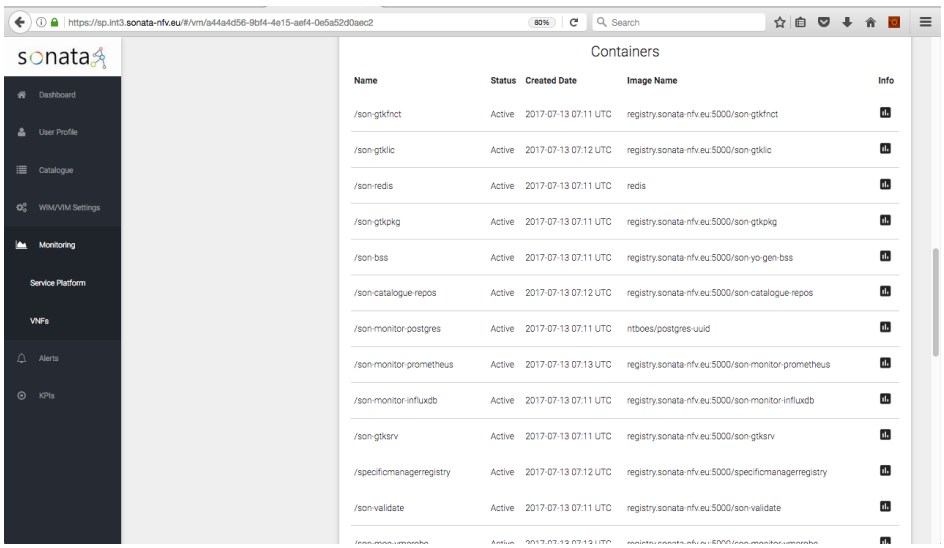

Figure 4.7: List of deployed containers

4.4 VNF development

4.4.1 VNF Development cycle and software management

As we approached the end of the project, outcomes validation by means of pilot implementation has occupied a substantial portion of the spent effort. In order to obtain the maximum results in

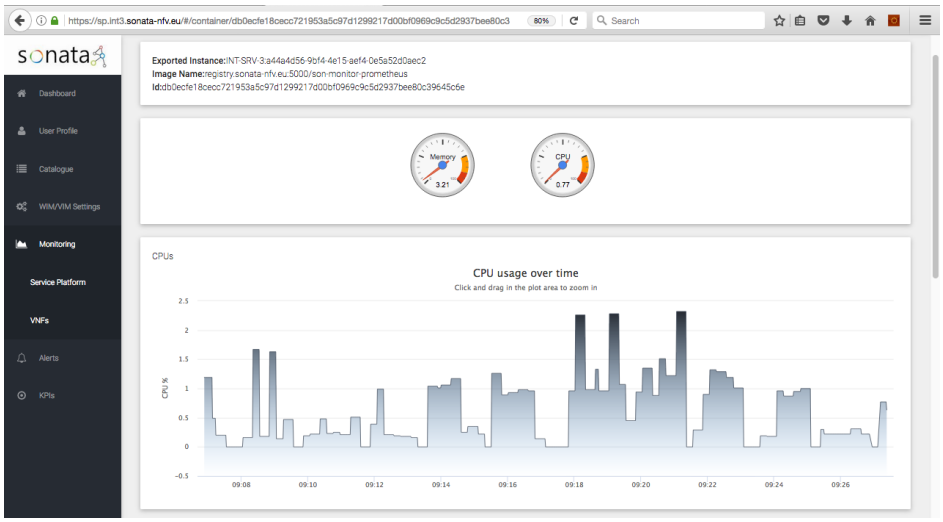

Figure 4.8: Container performance

terms of effectiveness and efficiency of our validation process, software developed to be used as a test case, especially software to be embedded in VNFs and in Specific Managers, needs processes to ensure its stability and reliability. For this reason and thanks to the lessons learned throughout the implementation of the SONATA framework, we resort to the toolset and processes we used to develop modules in the SONATA environment also for developing VNFs, descriptors, specific managers and auxiliary software used in the pilots illustrated in details in [13]. Extremely useful in this process has been the use of **Github**, including its software management internal toolset and its issue tracking system.

In practical terms, separate Github software repositories have been provisioned in the SONATA Github. Each one of this repository has been internally divided with a standard structure, so to allow automatic deployment and tests of components. Github issues regarding each pilot have been created in the relevant repository to highlight critical tasks or problems and track the relevant activity. Separate teams of developers have been allocated to each pilot, and internally to each pilot, small sub-teams dealt with specific VNFs or software module. An example snapshot of this tool is shown in Figure 4.9.

In order to have a clear understanding of the actual set-up needed for each pilot, the first step of their implementation consisted of a manual deployment and configuration of the service on the SONATA NVFI, described in [9] and described in [13]. During this manual deployment, important information for SSM/FSM development and descriptor redaction were elicited, together with a further insight into the expected service dynamics and operations. This preliminary step also allowed generating feedback from WP5 to WP3 and WP4, helping to prioritise the development of the features for the latest software release.

In the following section, we give an example of the outcome of this process, describing the case of the vTU VNF of the virtual CDN pilot.

4.4.2 A VNF development example, the Virtual Transcoding Unit (vTU)

4.4.2.1 Description

The implementation of the vTU VNF is greatly influenced by the vCDN use case. With that context in mind, the one of the vCDN Pilot, the player, sends a request to the vTU when it can not consume the content. In other words, that happens when the player can not handle any of the

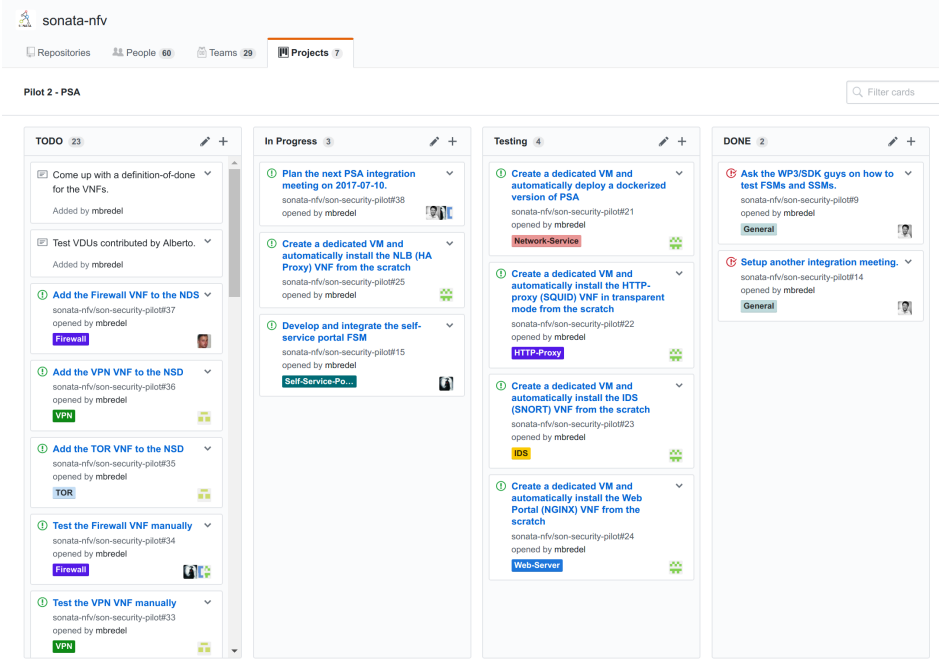

Figure 4.9: Snapshot of Github project for Pilot 2 vPSA

available transcoded formats for that particular content. In the case of the vCDN pilot, all the requests the player sends will pass through the vTC, who will, in turn, redirect them either to:

- the vTU
- the vCache going towards the content server

When the player requests a new trans-codification, the request will contain the appropriate resolution (the one the player can handle) and a reference to the particular content to be transcoded. vTU will take care of producing a new MPEG-DASH trans-codification with the requested resolution. The master content to be transcoded is in the content server. When the transcoding is ready, the newly produced content will also be stored in the content server, together with all the other transcodications for that same content.

4.4.2.2 Architecture

The vTU VNF is a VM formed by three main blocks: the API Rest (Server), the Database and the Transcoder. The latter two, the database and transcoder, are connected through a REST API. The REST API acts as a manager for all the requests, both for content consumption or content transcoding. Each of the previous components has its own docker image. Docker Compose will take care of launching all those Docker images at the same time and linking them with each other. When the player wants to request a new trans-codification, it has to send an HTTP request (POST) to the REST API. That request will trigger the addition of a new trans-codification job to the database. All the information related to transcoding jobs and content information gets stored in the database. In order to reduce the time needed for transferring the content from the vTU to the DASH server, the latter is exposing its storage via NSF, which is mounted over the network by the vTU.

4.4.2.3 Technologies

The technologies used for the API REST are Node.js + Express.js. MongoDB is used as a database. The Transcoding Unit has been developed in Python. A transcoding job can be divided into three steps:

- Firstly, the original media le is transcoded to MP4 using mpeg
- Secondly this transcoded le is fragmented to MPEG-DASH with the GPAC multi media framework
- Finally the new resolution, along with other information like the bit rate, is added to the MPEG-DASH MPD le

The vTU in order to transcode the content needs the content as close as possible, this is the main reason why between the vTU and the Content Server there is a folder using Network File System (NFS). The NFS allows accessing les over a computer network as if it was a local storage access.

4.4.2.4 Interfaces

- External Interfaces
  - vTU VNF exposes a network interface used for receiving transcoding requests. The requests are sent to the vTU exposed API REST interface.
- Internal Interfaces
  - Server to Transcoder
  - Server to MongoDB: As previously discussed all the info related to transcoding jobs and content information, gets stored in the Mongo DB and that information gets updated or consumed via queries.

4.4.2.5 Function Specific Management (FSM)

FSMs, or function specific managers, are processes that allow the developer of a VNF to customise the life cycle of his/her VNF. The code of an FSM should be embedded in a Docker container, which gets instantiated once the SONATA SP receives a request to deploy the VNF it is related to. Once instantiated, each FSM should attach itself to a virtual host of the SP message bus (for which it got connection details through the ENV variables) and register itself with the Plugin Manager. An FSM can be used to customise any of the following life cycle events (more event type details are provided below): start a VNF, stop a VNF, configure a VNF, monitor a VNF and scale a VNF. The FSM screens the message bus to receive requests to perform such life cycle events by subscribing to a message bus topic related to VNF. These requests are sent by the Function Life Cycle Manager (FLM), which implements the generic life cycle events for all the VNFs, and knows when the developer wants to replace a generic event with a customised event. The topic for the communication is chosen as follows: `'generic.fsm.<vnf_instance_uuid>'`. The VNF instance uuid is passed to the FSM through the ENV variables during instantiation. Once such a request is received, the FSM should check the `'fsm_type'` field in the payload, as it will indicate whether which type of life cycle event this request is for. Once the FSM completed the life cycle event, it responds to the FLM on the same topic with a `'status':'COMPLETED'` or `'status':'FAILED'` message, possibly extended with relevant information resulting from the event. FSMs can be developed per task, but it is also possible to have an FSM that implements multiple types of life cycle events. In that case, the FSM should listen on all the relevant topics.

son-start An FSM that implements a **start** life cycle event for a VNF should look for requests where the payload contains a `'fsm_type':'start'` key-value pair. The received payload should at least include the record of the VNF, which indicates its deployment and connection characteristics. This data can then be used to connect to the VNF and perform the tasks that start the VNF.
In our example start, will mean that FSM will have to issue a `docker run (container_name)` command, which will restart the stopped containers. This assumes that containers were stopped before receiving the start command, if containers are running and the system receives start request, the system will produce an error that the FSM will handle. One must not confuse the start procedure with the initialisation procedure, which it is done during the VM instantiation and boot. Since the start procedure must be executed over an initialised system, the FSM is able to poll the status of the booted VM and check whether or not the system is correctly initialised and ready to start functioning.

son-stop An FSM that implements a **stop** life cycle event for a VNF should look for requests where the payload contains a `'fsm_type':'stop'` key-value pair. The received payload should at least include the record of the VNF. This data can then be used to connect to the VNF and perform the tasks that stop the VNF.
In our example, when the virtual machine containing the vTU receives a stop command, the containers will automatically stop.

son-configure An FSM that implements a **configure** life cycle event for a VNF should look for requests where the payload contains a `'fsm_type':'configure'` key-value pair. The received payload should at least include the records of all the VNFs in the service and the service record. This data can then be used to connect to the VNF and configure it, e.g. import connection details of the other VNFs in the service.

son scaling An FSM that implements a **scale** life cycle event for a VNF should look for requests where the payload contains a `'fsm_type':'scale'` key-value pair. The trigger for a scaling request originates in a Monitoring Service Specific Manager (SSM). A Monitoring SSM receives all the monitoring data related to a service. Based on this data, it decides when a VNF scaling event of one of the VNFs involved in the service should take place. The Monitoring SSM prepares a message intended for the scaling FSM containing at least the record of the VNF as a payload. Custom fields and information depending on the specific service or VNF could be added as well to the request. The Scaling FSM uses this message and the record information to calculate how the VNF should be scaled to satisfy the new load. As an example, an FSM could consider to scale-out one or more VNFCs that represents a bottleneck for the VNF operation and configure a Load balancer VNFC to share the load among these scaled-out VNFC. With this policy, the FSM can then instructs the Infrastructure Adaptor (through the FLM) to increment the number of instances for specific VNFCs and report their configuration back (e.g. IP addresses). Note that the Monitoring SSM and the Scaling FSM are not in direct communication. The Monitoring SSM message is forwarded to the Scaling FSM through first the SLM and then the FLM.

5 Conclusion

This document concludes the work of SONATA Work Package 5. Its main objective has been the integration of the development results obtained in WP3 and WP4, in order to provide a stable and reliable software framework. Far from the purpose of producing a commercial software product, we resorted to state-of-the-art agile development practices and implemented an automated CI/CD development cycle that allowed us to produce, at the moment of this submission, three integrated software releases of the SONATA framework, plus an intermediate release 2.1 with updates and upgrades. It is worth noting that automated system tests on the SONATA release v3.0 and future delta ones will continue according to our CI/CD integration process to guarantee a high-quality outcome of the software that will be produced in the future months.

The resulting software framework will be used in the final phase of the project, for validating our results through the pilots described in [13]. With regard to this last aspect, this document also described some of the auxiliary software interfaces and components developed in the context of WP5, as well as the procedures and strategies adopted for VNFs development.

In this deliverable we documented the latest software release of SONATA, also detailing the automated installation process developed in the framework of WP5, which is a crucial aspect for facilitating adoption and increasing impact. For this release, we also documented some of the results of system performance tests which are constantly carried out during the qualification phase of our development pipeline and provided a conclusive analysis on the impact of the above mentioned agile methods on the project and its outcomes. We stressed that, although these methodologies come with a cost in terms of time and effort, these costs are widely amortised by the **higher quality** of the produced software artefacts in terms of **reliability** and **stability** we demonstrated through the stress test presented in this document, as well as an improved maintainability, and finally by the way they ease the effort of integrating software produced by a large distributed development team.

A Bibliography

[1] SONATA consortium. H2020-ict-2014-2 - proposal submission forms. Website, November 2014.

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

[3] SONATA consortium. D5.1 continuous integration and testing approach. Website, December 2015. Online at http://www.sonata-nfv.eu/content/d51-continuous-integration-and-testing-approach.

[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. D5.2: Integrated lab based sonata platform. Website, June 2016. Online at http://www.sonata-nfv.eu/content/d52-integrated-lab-based-sonata-platform.

[9] 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.

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

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

[12] SONATA consortium. D5.3: Integrated and qualified public release of sonata platform. Website, December 2017.

[13] 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.

[14] Gatling performance testing tool. Online at http://gatling.io/docs/current/.

[15] ETSI European Telecommunications Standards Institute. Network functions virtualisation (nfv); management and orchestration v1.2.1. Website, December 2014. Online at http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf.

[16] Multi-factor authentication. Website. Online at https://en.wikipedia.org/wiki/Multi-factor_authentication.

[17] Vagrant - development environments made easy. Online at https://www.vagrantup.com/.