D5.2 Integrated Lab-based 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	D5.2 Integrated Lab-based SONATA Platform
Workpackage	WP5 Integration, system testing and qualification
Due Date	30th June, 2016
Submission Date	13th July 2016
Version	0.1
Status	Final
Editor	Dario Valocchi (UCL),
Contributors	Santiago Rodríguez (Optare), Aurora Ramos, Felipe Vicens (ATOS), Michael Bredel (NEC), Shuaib Siddiqui (i2CAT), Daniel Guija (i2CAT), José Bonnet, Alberto Rocha (PTIN), Manuel Peuster (UPB), Sharon Mendel-Brin (NOKIA), Luís Conceição (UBI), Stuart Clayman, Alex Galis, Francesco Tusa, Dario Valocchi, Zichuan Xu (UCL), Geoffroy Chollon (TCS), Stavros Kolometsos (NCSRD), George Xilouris (NCSRD), Steven Van Rossem, Thomas Soenen, Wouter Tavernier (iMinds), Theodore Zahariadis, Panos Trakadas, Panos Karkazis, Sotiris Karachontzitis (SYN)
Reviewer(s)	Stuart Clayman, Alex Galis, Zichuan Xu (UCL), Thomas Soenen (iMinds), Panos Trakadas (SYN)

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 Work Package 5 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. The current target is the delivery of the first lab based prototype of the SONATA service platform.

With the end of the first year of the project, four important and related documents linked with the WP3, WP4, WP5 and WP6 work are issued by the consortium in a very short time interval, so it is crucial to provide a clear description of the content separation of these four documents, clarifying their scope and mutual relation.

At the end of month 10 of the project, Deliverables D3.1[12] and D4.1[13] were submitted to the Commission. These documents contain implementation details of the modules developed for the SONATA SDK and the SONATA SP respectively. These include the internal module architecture, APIs definitions, unit and module tests definitions.

Together with this document, deliverable D6.1 "Definition of pilots infrastructure setup and maintenance report"[14] is planned to be issued at the end of month 12. This companion document contains the detailed description of the infrastructure used during the development of the first SONATA prototype, along with some details on the pilot definition.

This deliverable is based on the recommendations elaborated in deliverable D5.1 [11] "Continuous Integration and Testing Approach", and it targets the description of the software development management aspects of the **SONATA prototype**. D5.2 scope is the specification of the first integrated, lab-based prototype of the SONATA environment. It also documents the **Prototype modules** that are part of this first integrated prototype, extending their characterisation with details on their interfaces and inter-working functionality. It lists and details the **Integration tests**, which have been designed and developed to ensure the functionalities expected from the first prototype. It also describes the **instruments** and **workflows** used to ensure code stability and to facilitate inter-working across the consortium. Furthermore, it details the **test framework** used to validate the outputs of the prototype development.

Finally, it has the objective to outline the outcomes of applying the **CI/CD approach** to the development of the SONATA prototype, thus underlining the innovation SONATA targets to bring into network service development.

Contents

List of Figures

List of Tables

1 Introduction

This document covers the interworking and integration aspects of the modules of the SONATA environment, this introduction will give a brief overview of the SONATA architecture from D2.2 [10].

Figure 1.1 presents the main components of the SONATA architecture. This is conceptually divided into two parts.

On the left part of Figure 1.1 is the SONATA Software Development Kit (SDK). This offers functionalities 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.

On the right part of Figure 1.1 is the SONATA Service Platform (SP), which offers the functionalities to:

- Orchestrate and manage network services during their lifecycles with a MANO framework, interacting 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;
- Richly 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
- Offer the functionalities of the SP itself to the outside world, with a unique endpoint, called the *Gatekeeper*

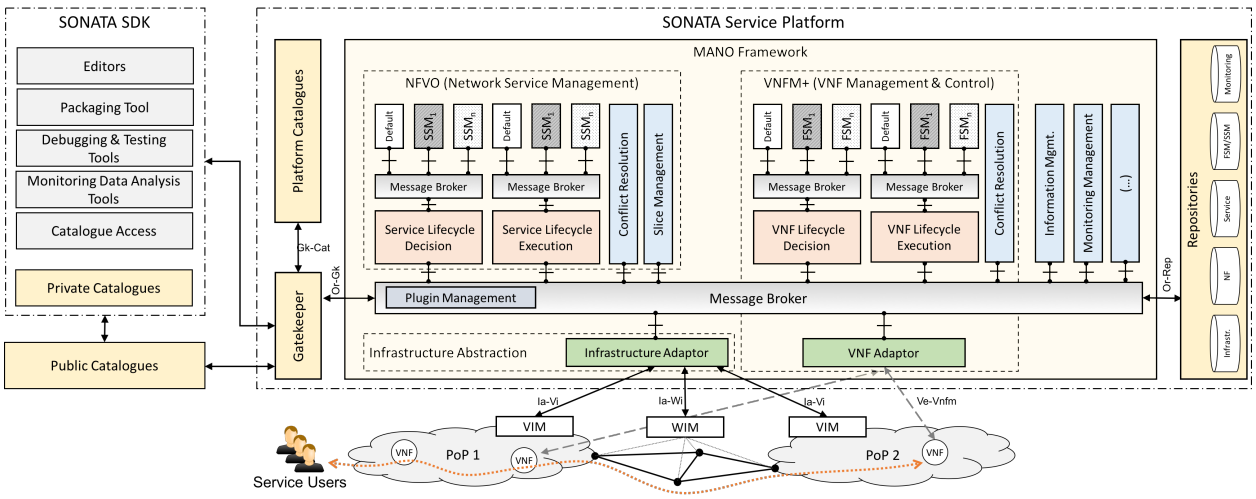

Figure 1.1: Detailed architecture of SONATA service platform

1.1 Softwarization in the 5G context

Recently, the 5G-PPP Architecture Working Group with input from 19 current 5G projects released a white paper on the architectural view of 5G [1]. It stresses that softwarization on all network segments is among the most important enablers for the native dynamicity and flexibility foreseen for 5G networks. The mechanisms facilitating network softwarization, as shown in Figure 1.2, are foreseen as a separate plane of artefacts and functions interworking with the other 5G planes: Applications and Business Services, Multi-Service Management, Integrated Management and Operations, Control and Data/Forwarding Planes.

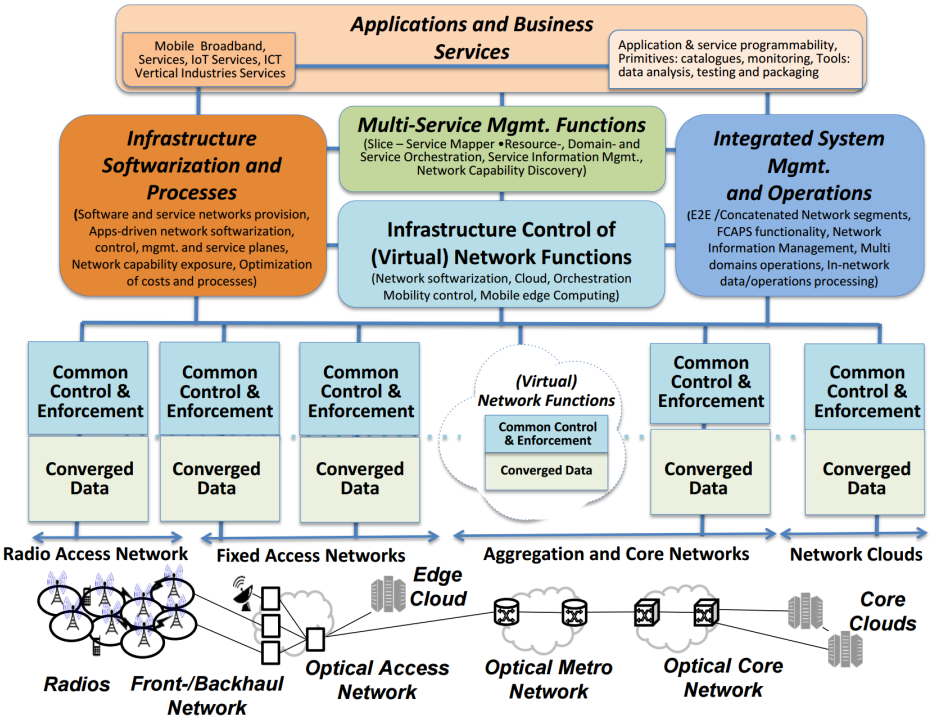

Figure 1.2: 5G Planes

An entire section of this architectural white paper is dedicated to the software technologies which are considered the driver for introducing a reduction in OPEX and CAPEX, a decrease in service deployment time, increase resource and energy efficiency and improve the quality of experience. Figure 1.3 shows the network segments and the different layer in which the so called *softwarization* process is called to push the boundaries in terms of capabilities and flexibility.

This architecture foresees different frontiers to be reached and pushed using software, like:

- Uniform programmability of all (virtual) network functions
- Uniform Inter-working with control, management and service planes
  - Southbound: network OS facilities & enablers
  - Northbound: multi domain orchestration facilities & enablers
  - East/West frontiers: Uniform inter-working facilities with the management, control and application planes
- Internal frontiers:

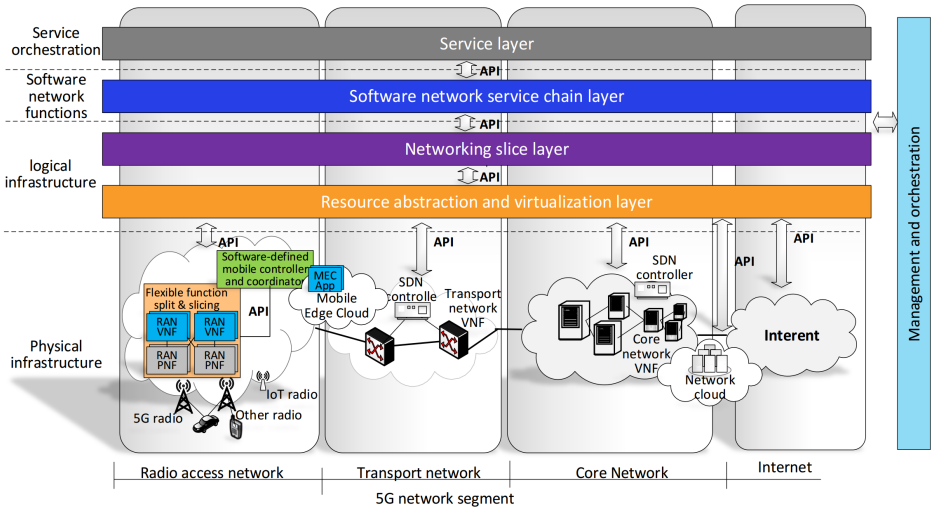

Figure 1.3: Software network technologies in 5G overall architecture

- Uniform computing, networking, service orchestration and management
- Software based closed control loops and orchestration loops
- *Softwarization* as a Service: Exporting network capability descriptions for use by other software networks
- Elasticity: facilities & enablers for efficient network functions rapid (re)deployment between network segments including edge networks
- Service & Resource Slicing: uniform partitioning and use of *Softwarized* plane resources: facilities and enablers for support of multi-domain service and resource slices
- Light Weight Facilities: Low foot-print and simplified S/W for integrated systems

Given the complexity and diversity of the tasks that are software dependent, it is crucial and mandatory for the network world to borrow from the software world, not only its flexibility and dynamicity, but also some fundamental aspects of the Software Development theory:

- Software Development Methodologies
- Software Development Tools
- Software Management

These will be necessary to handle the level of complexity in a structured, organised and controlled way, averting the risk of a network software crisis, where software introduces instabilities and decrease maintainability.

It is with this view in mind that SONATA plans to bring DevOps approach in the heart of virtual network service provisioning. It is not a case that a software development environment, tightly coupled with a service platform, are at the very core of SONATA. These represent the level of inter-dependency that is envisioned by the SONATA approach between Network Service Development and Operation. The additional step we highlight on this document, is how the same path envisioned for network service development and operation, has helped the development of the first SONATA prototype in such a large, heterogeneous and distributed development team like the SONATA consortium.

1.2 Structure of the deliverable

The rest of this document is structured as follows: Section 2 outlines the scenario the SONATA lab based prototype is dived into, describing the involved actors and how they are supposed to interact with the system. Section 3 describes the Continuous Integration and Continuous Delivery workflow and tool-set used during the development and integration of the SP and SDK modules. Section 4 extends the outcome of WP3 and WP4, giving a brief description of the developed artefacts, the interactions between them, also highlighting the advantages brought by the use of the CI/CD approach. Section 5 contains an extensive lists of the integration tests, which have been designed and implemented to validate the integration of the SONATA prototype. Finally, in Section 6 we draw our conclusions.

2 Integrated SONATA Prototype Flow

In deliverable D2.2[10] we gave a high level view of the actors involved in the SONATA ecosystem. We summarize the roles sketched in Figure 2.1:

- End User: a private person, a company or even a network service or content provider that wants to consume a network service.
- Network Service Developer/Provider: a person or a company developing and/or offering a VNF or a complete network service. It interacts with the SDK to build its service and uses the SP to run it.
- SONATA SP Operator: a person or a company running the SONATA service platform. It receives requests to on-board and run network services and VNFs from developers and uses the platform to manage them. In order to run services, it resorts to an infrastructure operator for resources.
- Infrastructure Operator: a person or a company that manages the computing, storage and networking resources. It offers its set of virtual resources through a set of virtual resource managers (VIM).

It is worth to mention that the SP operator and the infrastructure operator are the same actor in most of the cases.

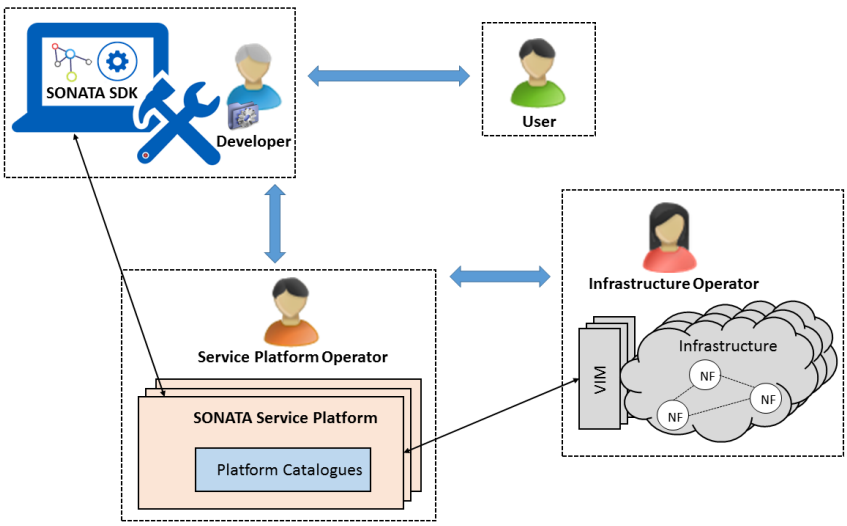

Figure 2.1: Main actors of the SONATA environment

In this section we want to expand this high level view, exploring the SDK and SP and their components and giving a view on how management and control flows go from one actor through the SONATA ecosystem to the other. Figure 2.2 gives a detailed view of the components that compose the SONATA prototype, along with the interfaces used for inter-module communication

and for the interaction with the main actors listed above. A bigger version is also included in Appendix A. Modules are represented by boxes of different colours, purple for the SDK modules, green for the SP modules, yellow for the GUI modules, light blue for the infrastructure components, and on the leftmost side of the picture the monitoring entities of the service platform are shown in grey. A description of these modules, with further details on the offered functions, is given in Section 4.

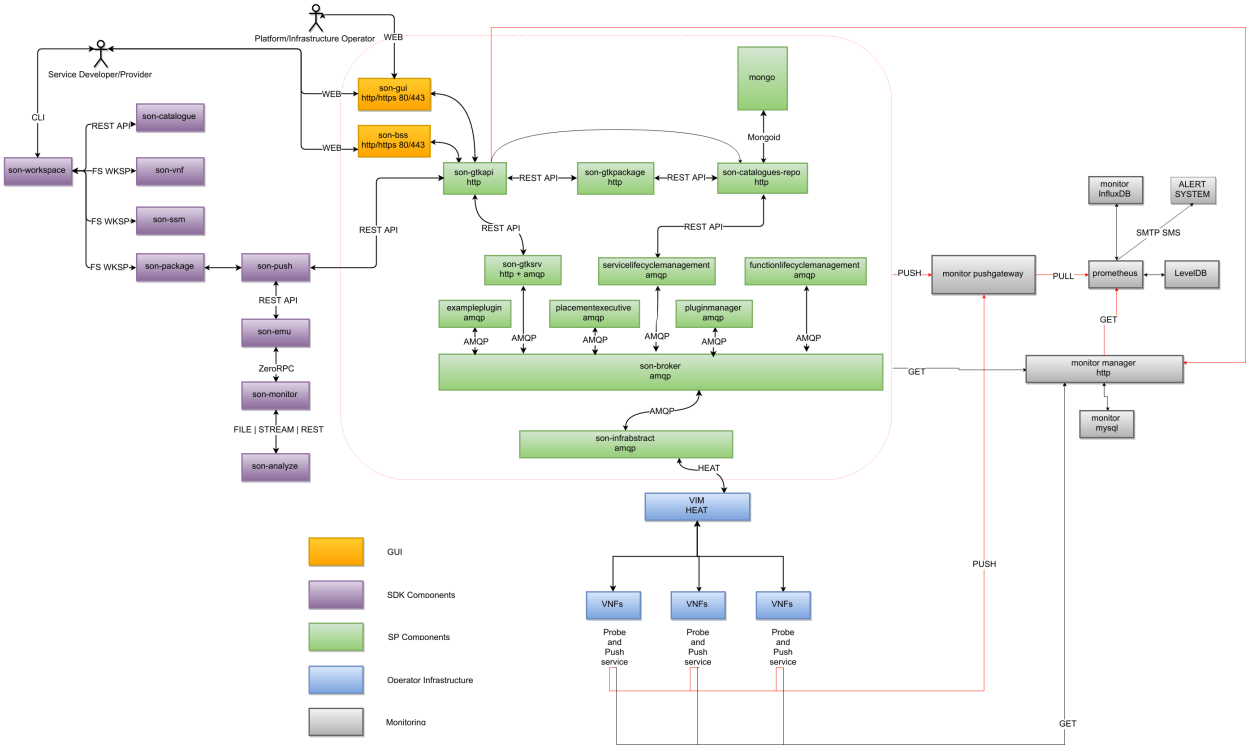

Figure 2.2: The SONATA prototype big picture

2.1 Service Platform Instantiation

Figure B.1 describes the actions that the service platform operator needs to carry out to install and instantiate the Service Platform environment. This process is based on the use of Ansible[4], and the details on the infrastructure requirements and on the configuration required are contained in D6.1[14]. The Service Platform operator can run the service platform modules inside a Linux virtual machine. After the platform instantiation, the operator can connect to the GUI to configure the pool of resource to be used by the platform to run services. To do this the service platform must be connected to one or more Virtual Infrastructure Manager and WLAN Infrastructure managers, which will offer computing, network and storage resources to the platform. This is done by adding one or more VIM or WIM configurations to the platform, using the SONATA GUI. This configuration includes, among others, the endpoint of the VIM and the credentials for authentication. After these initial steps the platform is ready to be used to run services.

2.2 Network Service Development

Network services are developed and provided to the platform operator by another actor, the service developer. It uses the SDK to control the different phases of the service development, using the SDK modules to develop VNFs, produce descriptors, package the produced software and validate the outputs of the development phase. The SDK offers the developer a Command Line Interface (CLI), through which it can access the SDK functions. Figure B.2 Figure B.3 and Figure B.4 show the interaction between the service developer and the SDK modules for the different phases of the service creation. The developer sets up its development environment and creates a new project (Section 1 and Section 2 in Figure B.2). The developer can combine existing VNFs to build and package his service and add its own configuration through the SONATA Network Service Descriptor (NSD) and VNF Descriptor (VNFD). When the needed software is ready to be packaged, the developer uses the CLI to create a SONATA Package (Figure B.3). This triggers the SDK to validate the produced descriptors for the network service, network functions and the package, using the SONATA descriptors schemas and the service provided by the son-schema modules. Descriptors for reused VNFs are retrieved from the local catalogue and the package is built (Steps 4 to 6 in Figure B.3), and the service is ready to be published in the local SDK catalogue (Step 12 to 19 in Figure B.3). Finally, the SDK offers also an emulated SONATA environment that can be used by the developer to test its services locally. The details of the interaction between the developer and the emulator are shown in Figure B.4.

2.3 Network Service On-boarding

When the network service is ready to be released, the service developer/provider decides to publish it into the catalogue of the platform operator. Again the SDK toolkit allows him to push the created package from its local catalogue to the service platform catalogue. On the service platform side, the *Gatekeeper* module is responsible for supervising the service on-boarding, validating the packages and the descriptors through the service platform schema repository and storing it in the catalogue. Once the service is on-boarded, the service platform operator can retrieve information on it from the catalogue using one of the gatekeeper sub-module, the BSS. This offers a web-based GUI that can be used to managed stored services and services instance. The involved interaction are depicted in Figure B.5.

2.4 Network Service Instantiation

The platform operator or the service provider can request the instantiation of a network service using again the BSS web interface. This triggers a complex set of action depicted in Figure B.6. The gatekeeper sub-modules are called in sequence to retrieve the descriptors for the requested service and the involved VNFs from the catalogue. These descriptors will represent the template against which the service instance is built. This information is sent to the MANO framework through the platform message broker. The service deployment request is consumed by the Service Lifecycle Manager (SLM) plugin of the MANO framework, which in turn uses the functionality offered by the infrastructure abstraction layer to retrieve information on the amount of available virtual resource. This information is used to select the best VIM among the ones registered to the abstraction layer.

When the service placement is computed, the deploy request is sent to the infrastructure abstraction layer, via the message broker, along with the output of the placement process. The abstraction layer will use its VIM Adaptor sub-module to translate the service and function descriptors in the

vendor-specific language. For the first prototype OpenStack[25] is used as a VIM, and its orchestration service Heat[9] is used to deploy the needed virtual infrastructure facilities. Figure B.7 shows the information flowing from the infrastructure abstraction to the VIM and eventually resulting in the VNF instantiation. The WIM adaptor is similarly used to manage the WAN network resources, available to steer the service user traffic through the network service. For the first prototype we used the so called Virtual Tenant Network (VTN) technology to setup network rules on a SDN-based network (controlled by ODL) to steer service related user traffic flows towards the NFVI-PoP, through the VNF, and out (See deliverable D4.1[13] for more details).

After the deployment, the infrastructure abstraction layer builds the response for the SLM, including IP addresses and instances identifiers. This information is consumed by the SLM to build a so called Network Service Record (NSR), that represents the service instance in the platform repositories. Another task of the SLM is to push the monitoring configuration encoded in the service and VNF descriptors to the monitoring manager. This trigger the monitoring manager to deploy and configure the monitoring facilities for the new service instance, creating the relevant entries in the monitoring repository and configuring the alert system Figure B.8). After this procedure the deployment status is updated and sent back through the gatekeeper to the repositories. The operator can now control the status of the service through the BSS web interface.

2.5 Network Service Operation

Finally, during the service operation, both the service developer/provider and the platform operator are interested to access monitoring data coming from the service instance. They will have different purposes, and depending on the role of the developer in the service provisioning and on the SLA between developer and platform operator, different views on the complete set of monitoring data will be granted to them. For this purposes they are supposed to use a different interface to access this data, filtered by the *Gatekeeper* and the monitoring manager. On one hand, the SDK offers two dedicated modules to access (son-monitor) and to analyse (son-analyze) the monitoring data coming from instances of the services developed by the specific developer. On the other hand, the service platform GUI offers to the platform operator a view on the overall resource consumption caused by instances running on his platform. The interaction of these scenarios are depicted in Figure B.9 Section 1 and 2, respectively. Moreover the son-analyze and son-monitor toolset offer also a tool for the developer to collect and analyse the data coming from service instances running in the son-emu emulator.

3 The CI/CD Approach

In this section we give a description of the software development methodologies used to built the SONATA ecosystem elements. These are based on the so called Continuous Integration/Continuous Delivery approach. This approach is designed to decrease the time-to-market of software development process. In fact, it is based on the assumption that developers can contribute to the development of software by publishing new features and functions multiple times a day, without the risk of breaking the code integrity. This is assured through a set of test steps that will validate the developers' contributions and include them in the master version of the software, as they meet the validation requirements.

3.1 Technologies involved in CI/CD Ecosystem

The SONATA CI/CD Ecosystem is composed by a group of tools. They support the developers through the process of creating software, following the DevOps philosophy. Each of these tools collaborate to create a complete development environment for the SONATA project. The most important are: Docker Engine, Github, Jenkins, Openstack and Graylogs.

3.1.1 Docker Engine

The Docker Engine [19] is used to package the SONATA code in all the stages of the development, from the developer local environment, to the Qualification server, including the way SONATA stores module images and performs tests. In the development phase, Docker Engine is used to build the container (docker), perform the unit tests (docker), create the local development environment (docker-compose), store the containers in the SONATA local docker registry (docker registry). In the integration phase, the Docker Engine improves the speed of deploying the integration tests (docker and docker registry). In the qualification phase, it offers a quick and efficient way to deploy a fresh installation of the SONATA platform (docker and docker registry), ready to be used for qualification tests.

3.1.2 Github

Github [5] has been selected as the *Source Code Management* tool for SONATA, and it is at the core of the development phase. It allows developers to upload their code, manage code versions and track issues. Moreover, when the SONATA platform codebase will be released as open source, GitHub will easy allow sharing the codebase with the open source community. Finally, the main advantage of using Github for the SONATA development phase is its integration with the Continuous Integration tool Jenkins. In fact we used the so-called *webhook* mechanism to automatically trigger tests in Jenkins when new code is submitted to the SONATA repositories.

3.1.3 Jenkins

Jenkins [21] is the core of the *Continuous Integration* environment of SONATA, in all its phases. It acts as the key piece of the SONATA development puzzle. The SONATA's Jenkins uses a large set

of plugins, which are installed and used to improve the productivity of developers and the quality of the code. It also represents the nearest front-end tool for the developer, providing him with quick feedback about new features of his code. In the Development phase, each request issued by a developer to merge its local version of the code with the version on the SONATA repository (called pull-request in the GitHub vocabulary) triggers Jenkins to build the containers affected by the code update. This building process performs the unit tests and pushes *only successful builds* to the container registry, so that they can be used in the integration phase. Jenkins also uses plugins and tools to ensure the code quality, and these will be discussed later. In the integration phase, Jenkins deploys the containers and performs an instantiation of the service platform, putting all the containers together. Afterwards, it performs the Integration tests. At the end, in the qualification phase, Jenkins creates a VM in Openstack and installs the service platform from scratch. After the installation and instantiation of the service platform, Jenkins will be able to perform the qualification tests.

3.1.4 Openstack

Openstack [25] is used by SONATA as IaaS Cloud manager, in order to instantiate and run the virtual machines needed by the project CI/CD environment. These VMs are used to run different services: Jenkins, the integration and qualification servers, the docker local registry, etc. SONATA also uses Openstack to deploy VMs from scratch, triggered by Jenkins, and then to perform specific tests in the development and integration phases. Additionally, Openstack is used as a VIM for the Infrastructure Abstraction, therefore it is also used to deploy services during the tests of all the phases.

3.1.5 Graylog

The Graylog [20] tool is a centralised logging system. It is used by SONATA in the integration environment, so to give access to the developers to real time information on the status inside each container. As for the integration environment, the containers are configured to send their logs directly to the Graylog server. When the developer performs an integration test, he can access the trace logs for his containers and the other SONATA's containers from the Graylog web GUI. The information asynchronously arrives from the container and is indexed to trace easily the actions of each micro-service, involved in one integration test. The final goal of Graylog is to allow the developer to inspect the produced logs in a quick and efficient way, speeding up the debugging process.

3.1.6 Development phase toolset

The key of the continuous integration is to provide an early feedback to the developer about the status of his code. To guarantee the Quality of the Code in SONATA, the continuous integration system uses a set of developer tools, which are described in the following paragraphs.

3.1.6.1 Code Quality

Code Quality is controlled throughout the SONATA CI/CD loop. This is enforced by using different tools embedded in the Jenkins Continuous Integration environment. Here we summarise the most important ones with their functionality and the repositories on which they are used.

Rubocop

RuboCop [2] is a Ruby static code analyser. Out of the box it will enforce many of the guidelines outlined in the community Ruby Style Guide. SONATA use this ruby gem for all its ruby repositories.

This tool is used by:

- sonata-nfv/son-catalogue
- sonata-nfv/son-catalogue-repos
- sonata-nfv/son-gkeeper

pep8

Pep8 (Python Enhancement Proposal) [18] is a tool for python to ensure that the code follows a predefined style guide. It's very useful to find the code style errors. In SONATA, this tools is automatically used in the pipeline of Python based repositories. The created report is consumed by Jenkins, and it is accessible from the Jenkins dashboard.

This tool is used by:

- sonata-nfv/son-cli
- sonata-nfv/son-emu
- sonata-nfv/son-monitor
- sonata-nfv/son-analyze
- sonata-nfv/son-sp-infrabstract
- sonata-nfv/son-mano-framework

Maven Checkstyle

The maven checkstyle [16] is a tool that generates reports regarding the code style used by the developer. Jenkins checkstyle plugins is designed to consume the output of this tool and to create graph and show the report on the dashboard.

This tool is used by:

- sonata-nfv/son-sp-infrabstract

Maven Cobertura

The maven cobertura [3] is a tool that generates reports on the test coverage of the project, that is the portion of the code which is actually tested by unit tests. It can be used to identify parts Java program which are never executed during tests, and gives a feedback on the unit test design. Jenkins Cobertura plugin parses the output of maven cobertura and creates a trend graph on the dashboard.

This tool is used by:

- sonata-nfv/son-sp-infrabstract

Maven Findbugs

The maven findbugs [22] is a tool that looks for bugs in Java programs. This is based on common best practices and known coding critical issues, highlighting potential mal-functioning code section even before the compilation. Jenkins FindBug plugin parses the output of maven findbugs and creates a trend graph on the dashboard.

This tool is used by:

- sonata-nfv/son-sp-infrabstract

3.1.6.2 Unit Tests

Rspec

RSpec [8] is a behaviour-driven development (BDD) framework for the Ruby programming language. SONATA uses it to perform the unit test for Ruby's code. Jenkins has a plugin to parse the reports from RSpec and to show them on the Jenkins dashboard.

This tool is used by:

- sonata-nfv/son-catalogue
- sonata-nfv/son-catalogue-repos
- sonata-nfv/son-gkeeper

Junit

Junit [7] is a unit test framework used to write repeatable unit tests for Java code. SONATA uses it to perform unit test on the Java-based repositories. Jenkins has a plugin to parse the report output of Junit and show it on the dashboard.

This tool is used by:

- sonata-nfv/son-sp-infrabstract

Unittest

Unittest [15] is a unit test framework for python. It is the correspondant of Junit for Python code. Jenkins has a plugin integration with its output report format, so it is able to parse it and show it on the dashboard.

This tool is used by:

- sonata-nfv/son-cli
- sonata-nfv/son-emu
- sonata-nfv/son-monitor
- sonata-nfv/son-analyze
- sonata-nfv/son-sp-infrabstract
- sonata-nfv/son-mano-framework

Selenium

Selenium [24] is a tool used to reproduce navigation on a web browser. It's used to test the GUIs of SONATA platform. It generates a report that Jenkins can read and show in the dashboard.

This tool is used by:

- sonata-nfv/son-gui
- sonata-nfv/son-bss

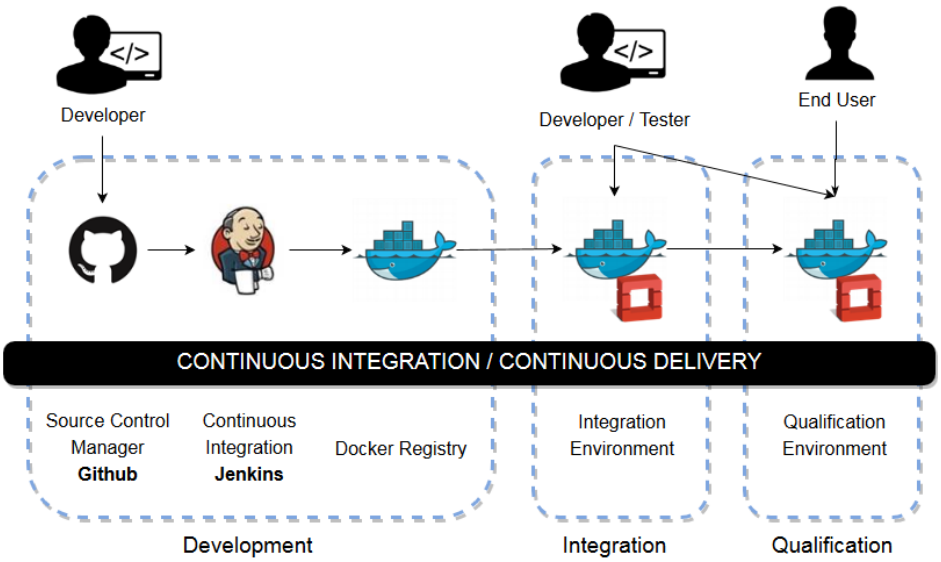

Figure 3.1: The SONATA CI/CD Pipeline

3.2 SONATA Pipeline

The pipeline, proposed in SONATA, helps the developer to create his piece of code and integrate it with other pieces of code taking the most valuable points of continuous integration and continuous delivery. Figure 3.1 describes the pipeline used by SONATA.

The SONATA pipeline consists of three phases:

- Development phase
- Integration phase
- Qualification phase

3.2.1 Development phase

The development phase is the key phase of SONATA pipeline. This phase accomplishes very important tasks: the source code generation, the Dockerfile definition where the microservice will be run, the build of the developer local environment with docker-compose and the unit tests to guaranty the quality of the code.

Figure 3.2 resumes the main interaction envisioned for this phase. Once the developer has created a piece of code working correctly and he has tested it in his local environment, he will do a pull request to the source code management tool: Github.

Github is connected to the continuous integration / continuous delivery tool (Jenkins) through a webhook. When Jenkins receives this pull request, it triggers the following predefined tasks as a standard workflow for all repositories:

1. Jenkins downloads the new code of the pull request from the source code management.
2. Jenkins builds the docker container using the Dockerfile generated by the developer.
3. Jenkins runs the unit tests using the latest version of the container.

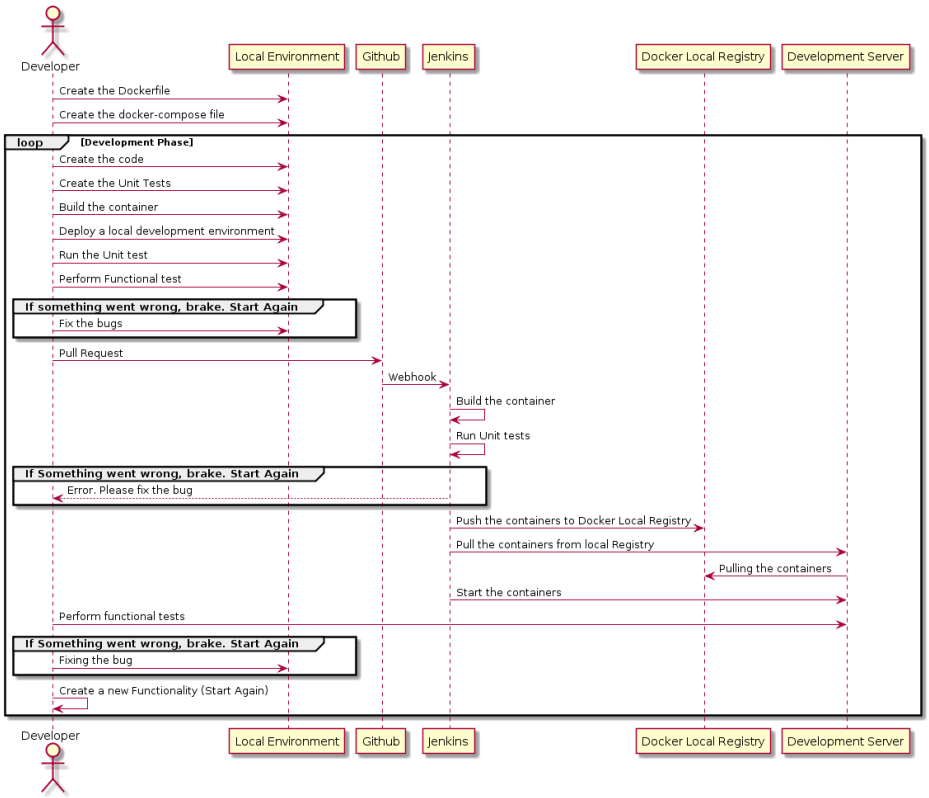

Figure 3.2: The SONATA pipeline development phase

In some cases it is necessary to use a complementary container with a service like postgres or rabbitmq to run the unit tests. In this way it is very easy to implement it. If the unit tests are all successful, then Jenkins will push the container to a local docker registry for future use.

Additionally, in some cases the continuous integration system deploys the new containers using the docker-compose tool in an development server. This allows to share the recently created code with other partners. With this development server, the developer will be able to do functional tests.

3.2.2 Integration Phase

The integration phase is composed by a battery of tests with the main goal of testing the interaction between the different modules. For this purpose, a series of tests was created starting with the deployment of all the containers in the integration server. After this deployment, Jenkins checks if all the containers are running and the databases are initialised, as shown in Figure 3.3. Then, Jenkins runs the integration tests one by one.

There are three ways of performing the integration tests:

- **Triggered by schedule:** Every night the continuous integration system performs all the integration tests.
- **Triggered by other integration test:** There is the possibility to create a chain of integration tests. When a build finishes or other integration test finish, the continuous integration system can be configured to perform another specific integration test.
- **Triggered manually:** An integration test can be triggered manually.

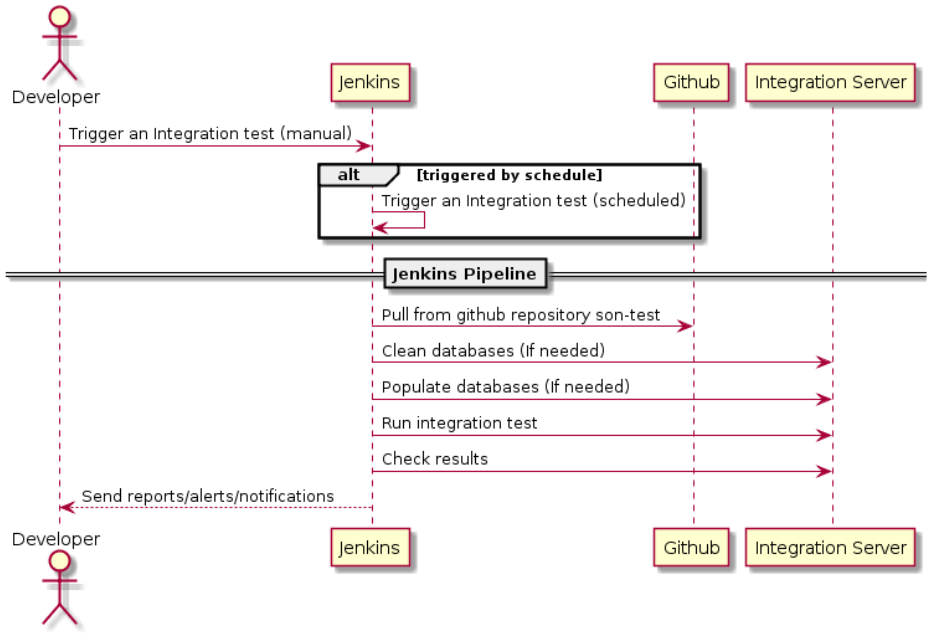

Figure 3.3: The SONATA pipeline integration phase

In case of failures in the testing procedures of the aforementioned cases, the continuous integration system will send a notification the lead developer.

The main advantage of having an integration environment is that it will be always accessible by all the partners and it can be used to debug their developments, since it provides also debugging tools to get the logs for all the components and find the errors very easily.

3.2.3 Qualification Phase

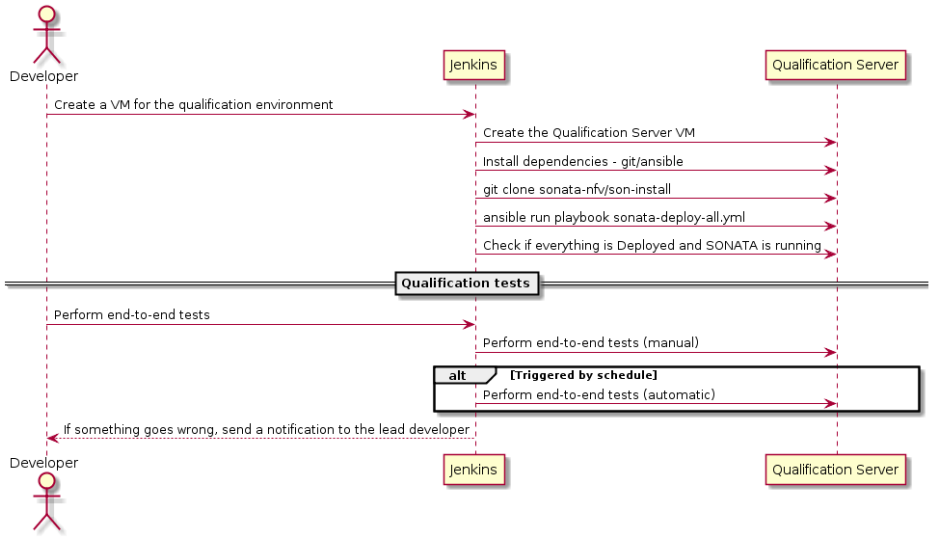

Figure 3.4: The SONATA pipeline qualification phase

The Qualification phase in SONATA is oriented to perform end-to-end tests. In order to perform the qualification tests, the service platform and the SDK are installed from scratch in a VM. This

VM for the service platform just needs to satisfy two dependencies:

1. Have the ansible package version 2.0
2. have installed the latest version of git package

The github's repository son-install is used to install the service platform by cloning the repository, and running the ansible playbooks. For the SDK, SONATA provide a .deb package to install easily the SDK.

The qualification phase uses a stable environment called qualification environment, to show the functionalities of SONATA and perform end-to-end tests:

- Functional
- Performance
- Security
- Conformance

4 Module Integration

This section gives an overview on the different software modules developed for the first SONATA prototype. A complete overview on the modules of the prototype is give in Appendix A. The details on the internal architecture of the modules of SONATA SDK and SONATA SP is given in deliverable D3.1 [12] and D4.1 [13], respectively. Here we recapitulate their main functions and we further extend their description with respect to previous deliverable documents, including information on their inter-working functionalities and with some consideration on how the CI/CD approach helped and accelerated the development process.

4.1 Service Platform

4.1.1 Gatekeeper

The Gatekeeper is the entry point of the Service Platform, or its northbound interface. It provides the needed services to the SDK, the BSS and the GUI. It validates submitted packages and service instantiation requests and passes them to the remaining Service Platform components. It collects data generated from these other Service Platform components, such as the IP addresses of the instantiated VMs, and passes them to the relevant external systems requesting it.

As described in D4.1[13], the Gatekeeper has adopted a **micro-services** approach, isolating the API itself from the services that implement it. This separation of concerns allows a greater modularity and flexibility, namely if and when we need to scale any of those modules. This separation is reflected in the Gatekeeper's repository, `son-gkeeper`, with the remaining modules as folders of the main project.

4.1.1.1 Gatekeeper API

The Gatekeeper API is comprised of the following entry points:

- `/packages`: accepts and makes a first round of validations of packages, submitted by the SDK. Also answers queries on existing packages, generating a package file when the result of these queries is one query;
- `/services`: accepts queries for existing services;
- `/functions`: accepts queries for existing functions;
- `/requests`: accepts requests for instantiating a service and answers queries about the status of these requests;

Environment and Inter-working

As previously stated, the Gatekeeper's API module/repository (son-gtkapi) serves as the front-end for the remaining services, which have been developed in the following repositories

- `son-gtkpkg`: supports the `/packages` endpoint, interacting with the Catalogues (son-catalogue-repos);

- `son-gtksrv`: supports the `/services` and `/requests` endpoints, interacting with the Catalogues (son-catalogue-repos) and the SLM (son-mano-framework);
- `son-gtkfnct`: supports the `/functions` endpoint, interacting with the Catalogues (son-catalogue-repos);

Note: Services and functions are created by creating packages, and not by them self.

4.1.1.2 Gatekeeper Package Management

The Package Management is the Gatekeeper's micro-service that is in charge of on-boarding new packages and providing already existing packages.

On-boarding a package includes the following sub-steps:

1. **Validate:** the package has to follow the `son-schema` defined format, and must be rejected if not;
2. **Decompose:** a package describes several kinds of assets, like a service, functions, managers, etc.; these have to be gathered and validated, before being submitted to the relevant catalogues;
3. **Catalogue submission:** assets are submitted to their catalogues; duplication is handled in this part (e.g., a function might already be part of the catalogue, because another package has brought it in);

Queries about packages are also handled by this micro-service.

Environment and Inter-working

As previously stated, the Gatekeeper's Package Management module/repository (son-gtkpkg) has the Gatekeeper's API (son-gtkapi) as its front-end and interacts with the Catalogues (son-catalogue-repos).

4.1.1.3 Gatekeeper Service Management

The Service Management is the Gatekeeper's micro-service that is in charge of interacting with the Catalogues (son-catalogue-repos), supporting queries about the stored services. The main purpose of this feature is to serve the GUI (son-gui), where the registered services can be listed.

This micro-service also supports requests for instantiating a service. This feature serves the BSS module (son-bss).

Environment and Inter-working

As previously stated, the Gatekeeper's Service Management module/repository (son-gtksrv) has the Gatekeeper's API (son-gtkapi) as its front-end and interacts with the Catalogues (son-catalogue-repos) for reading meta-data about services.

For the service instantiation functionality, the Service Management also interacts with the MANO Framework (son-mano-framework), to which it sends service instantiation requests and from which it receives results of these requests. This interface is asynchronous, through the exchange of a simple set of messages.

4.1.1.4 Gatekeeper Function Management

The Function Management is the Gatekeeper's micro-service that is in charge of interacting with the Catalogues (son-catalogue-repos), supporting queries about the stored functions. The main purpose of this feature is to serve the GUI (son-gui), where the registered functions can be listed.

Environment and Inter-working

As previously stated, the Gatekeeper's Function Management module/repository (son-gtkfnct) has the Gatekeeper's API (son-gtkapi) as its front-end and interacts with the Catalogues (son-catalogue-repos) for reading meta-data about functions.

4.1.1.5 Gatekeeper BSS

The Business Support System (BSS) is the graphical user interface who allows to instantiate a new network service in the platform, obtaining information about the available services that can be deployed and the requests instantiated by the system.

Environment and Inter-working

As a point of entry on the Sonata's Service Platform, the BSS interacts with the Gatekeeper:

- `son-gkeeper/son-gtkapi` (standalone)
- `son-monitor` (standalone)

4.1.1.6 Gatekeeper GUI

Gatekeeper GUI is an API management and visualisation tool that enables SONATA developers to manage their services throughout their whole lifecycle and enables the Service Platform administrator to provision and monitor platform resources easily and securely. In this perspective, the Gatekeeper GUI has been designed, developed and implemented to cover the needs of the two aforementioned groups, namely service developers and platform administrators, in supporting the process of DevOps in SONATA.

Environment and Inter-working

Due to its position in the SONATA architecture, Gatekeeper GUI communicates with several components, especially those offering APIs that are visualised through GUI. In particular, Gatekeeper GUI interacts with the following repositories:

- `son-gkeeper/son-gtkapi`

4.1.1.7 CI/CD added value

The micro-service (in a container) approach adds value to the CI/CD in the sense that changes are restricted to the changed container. In the future, if scaling of a specific micro-service is needed, more instances of this micro-service can be added.

New micro-services are very easily added and connected with the existing ones. Unit tests are also ran in this restricted environment, and integration tests imply deploying the to be tested containers (and their dependencies), which is simple and fast.

4.1.2 Message Broker

The message broker is the core of the MANO Framework provided by the Service Platform.

The message broker is responsible for inter-component communication between plugins who communicate with it asynchronously.

The message broker is based on a RabbitMQ system [26].

4.1.2.1 Inter-working

The message broker is the core of the MANO Framework and so interacts with all its components and also it is the gate for communication between the MANO Framework and components that are external to it:

- `son-mano-framework`
- `son-sp-infrabstract`
- `son-gkeeper`

4.1.2.2 CI/CD added value

As mentioned before the message broker is the core of the MANO Framework as it is responsible for the inter-component communication. It is based on a widely used technology so there wasn't much uncertainty about its capabilities. Nevertheless, due to its importance the CI/CD approach helped to identify issues related to its integration in the SONATA code and to identifies missing functionality needed to be developed on it.

4.1.3 MANO Plugins

The core of SONATA's service platform is the highly flexible management and orchestration framework. Figure 4.1 shows this extensible framework consisting of a set of loosely coupled components, called MANO plugins. Each of these plugins implements a limited, well-defined part of the overall management and orchestration functionality.

In our design, all components are connected to an asynchronous message broker used for inter-component communication. This makes the implementation and integration of new orchestration plugins much simpler than it is in architectures with classical plugin APIs. The only requirement a MANO plugin has to fulfil is the ability to communicate with the messaging system, such as RabbitMQ and ActiveMQ. We do not enforce a programming language for implementing plugins. Similarly, we do not prescribe how the plugins are executed.

To keep track of the connected MANO plugins, the *plugin manager* component is used, which is an plugin-like component that communicates with other plugins over the broker, offers a management interface, and interacts with the configuration interface of the message broker.

4.1.3.1 The Service Lifecycle Plugin

The Service Lifecycle Plugin, or Service Lifecycle Manager(SLM), is the plugin that handles all decisions concerning the lifecycle of the services. Through the message broker, it receives requests to deploy a service. Based on the Network Service Descriptor (NSD) and the virtual network function descriptors (VNFDs), which are part of the request, the SLM decides where to place it and how to scale the service. The SLM translates the request into a message for the Infrastructure Adaptor, the plugin that is responsible for the actual deployment. When the Infrastructure Adaptor reports

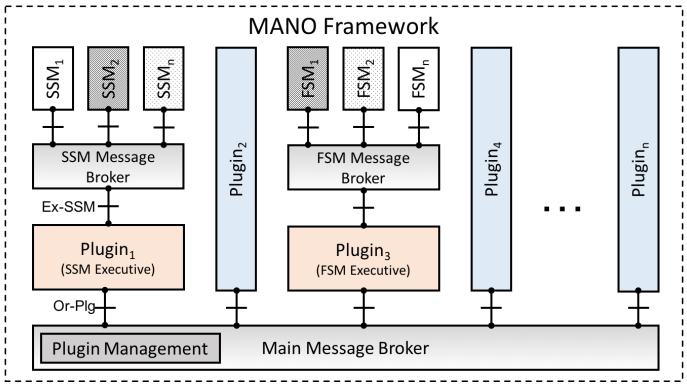

Figure 4.1: MANO framework with MANO plugins and service-/function-specific management components

back the status and the specifics of the deployment, the SLM uses this data to form a Network Service Record (NSR), and a Virtual Network Function Record (VNFR) for each VNF involved and stores these records in the repositories through a REST API. Based on the information in the descriptors, the SLM informs the Monitoring Manager which metrics concerning the service need to be monitored and which thresholds of these metrics are to be reported to the SLM. When one of these metrics reaches its threshold, the SLM adapts the lifecycle of the service, based on the information in the descriptor.

4.1.3.2 Service Specific Manager Plugin

Service Specific Manager Plugins (SSMs) are responsible for managing one or more services belonging to exactly one Service Platform customer. If no custom event handlers are needed, the service could rely on the default SSMs that will be part of the SONATA platform.

4.1.3.3 Inter-working

All loosely coupled components of the MANO framework interact with each other and they interact with other components of the service platform. The involved components are:

- `son-plugin-manager` (MANO framework plugin)
- `son-service-lifecycle-manager` (MANO framework plugin)
- `son-infrastructure-adaptor` (standalone)
- `son-gatekeeper` (standalone)
- `son-repositories` (standalone)

4.1.3.4 CI/CD added value

SONATA's Service Platform functionality is based on the MANO plugins, moreover in the development of the various plugins many developers from different partners took part. Using CI/CD was the only way to achieve the required functionality within the relatively short time frame.

4.1.4 Catalogues

SP Catalogues provide management and storage of Package Descriptors (PD), Network Service Descriptors (NSD), VNF Descriptors (VNFD), and SONATA packages (SON packages). In fact, the SP Catalogues are collections of PD catalogue, NSD Catalogue, VNFD catalogue and SON packages binary-data catalogue. Gatekeeper is the only other module that interacts with the SP Catalogues i.e., the SP Catalogues get populated with valid descriptors only via the Gatekeeper. The stored descriptors can be used to instantiate new services, list available network services to BSS, or to provide developer community access to these descriptors.

In terms of implementation, a mongoDB acts as the main storage and each catalogue, i.e., NSD, VNFD maps to a separate collection within the database. Hence, the schema of each collection in a database is adaptable to the structure of the descriptor stored. A REST API, as front-end to the database, enables access for Gatekeeper to retrieve or store descriptors in the SP Catalogues. The SP catalogue accepts NSDs, VNFDs, etc. in both YAML and JSON format.

4.1.4.1 Inter-working

Within the SP, the son-gtkpkg, son-gtksrv and son-gtkfnct modules need son-catalogues-repositories module for storing and retrieving descriptors, NSD, VNFD and PD respectively.

- `son-gtkpkg` executes a GET or POST operation, according to the API defined above, to retrieve or publish a package descriptor, respectively.
- `son-gtksrv` executes a GET or POST operation, according to the API defined above, to retrieve or publish a service descriptor, respectively.
- `son-gtkfnct` executes a GET or POST operation, according to the API defined above, to retrieve or publish a function descriptor, respectively.

4.1.4.2 CI/CD added value

CI/CD approach ensured smooth integration of Gatekeeper module with the SP Catalogues. Furthermore, it enabled easy and robust testing of different features of SP Catalogues as they were being developed.

4.1.5 Repositories

This section contains information about module integration related with Network Service (NS), Virtual Network Function (VNF) and Monitoring repositories.

4.1.5.1 NS Instance Repository

The NS Instance Repository is the component that acts like a data container of the Network Service (NS) instances, implementing the create, retrieve, update and delete functionalities.

Inter-working

The NS Instance Repository interacts with:

- `son-service-lifecycle-manager` (MANO framework plugin)

CI/CD added value

The CI/CD approach helped to identify issues related to its integration in the rest of SONATA code, making easier the task of integrating all the contributions from the different SONATA's developers. Moreover, it allows to build, test, and release the code in a faster and more agile way.

4.1.5.2 VNF Instance Repository

The VNF Instance Repository is the component that acts like data container of Virtual Network Functions (VNF) instances. This module implements the creation/retrieve/update/deletion functionalities for VNF instances.

Inter-working

The VNF Instance Repository interacts with:

- `son-service-lifecycle-manager` (MANO framework plugin)

CI/CD added value

The CI/CD approach helped to identify issues related to its integration in the SONATA code, helping the integration of the contributions from the various SONATA's developers. On the other hand, it allows to building, testing, and release the code in a faster and agile way.

4.1.5.3 Monitoring Repository

The SONATA Monitoring Repository collects and processes monitoring data from the VNFs and NSs. The developer has the ability to activate predefined metrics (RAM and CPU usage, hard disk usage, etc.) or develop and implement his own monitoring metric in order to capture service-specific behaviour. Moreover, the developer can define rules based on metrics gathered from one or more VNFs in order to receive notifications in real time. In general, the developer will be able to subscribe to a message queue or he can get the alert notifications by email and/or SMS on his smartphone. Most importantly, monitoring data and alerts are accessible through a RESTful API or directly accessing the Gatekeeper GUI. The internal architecture of Monitoring System, including Monitoring Repository, is depicted in Figure 4.2.

Inter-working

The monitoring manager offers a RESTful API for communicating with other SONATA components, while it sends alerts and notifications through the message broker of the service platform. In particular, the components consisting the Monitoring framework interact with the following repositories:

- `son-service-lifecycle-manager` (MANO framework plugin)
- `son-monitor-probe` (standalone)
- `son-broker` (standalone)
- `son-gui` (standalone)

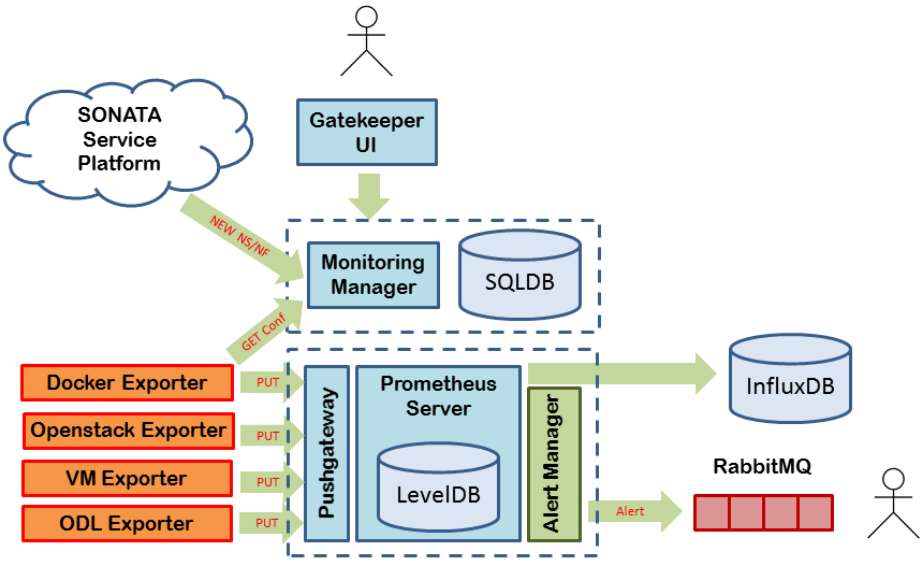

Figure 4.2: SONATA Monitoring Repository.

CI/CD added value

The development of the components comprising the monitoring solution requires synergies among several components of the SONATA architecture. The same requirement is valid for the purposes of planning and executing integration tests. By adopting the CI/CD workflow and utilizing specific development tools, we allowed developers to keep in synchronization and we increased efficiency both in terms of development and integration testing by minimizing overlaps and replications that allowed for having results in a relatively short time frame.

4.1.6 Instrastructure Abstraction

The Infrastructure Abstraction offers the functionalities for interactions between the underlying virtual resource layer and the service platform MANO Framework. It exposes an interface to interact with infrastructure managers, managing a distributed set of computational, storage and networking resources, making the service platform vendor-agnostic with respect to the infrastructure management layer. Figure 4.3 shows an high level view of the Infrastructure Abstraction layer. It is composed by two main modules, which are briefly described in the following paragraphs.

4.1.6.1 VIM Adaptor

The VIM Adaptor is the component of the Infrastructure Abstraction Layer responsible to offer the functionalities of the underlying Virtual Infrastructure Managers to the SONATA Service Platform (SP) modules in a vendor-independent way. The adaptor is used by the Service Platform operator through the Gatekeeper GUI to attach virtual infrastructure points of presence (PoP), managed by a VIM. Once the VIM is attached to the SP, it can be used to deploy network services. VIM from different vendors are connected to the VIM Adaptor using specific VIM Wrapper entities, which are in charge of executing VIM specific tasks that map to the generic functionalities exported to the SP. The integration between southbound interface of the adaptor and the VIM is therefore a responsibility of the Wrapper developer, when it comes to specify the medium and technologies used by adaptor and the VIM to interact, and of the SP operator, when it comes to configure credentials and endpoints to establish the connection. A repository is used to store the registered

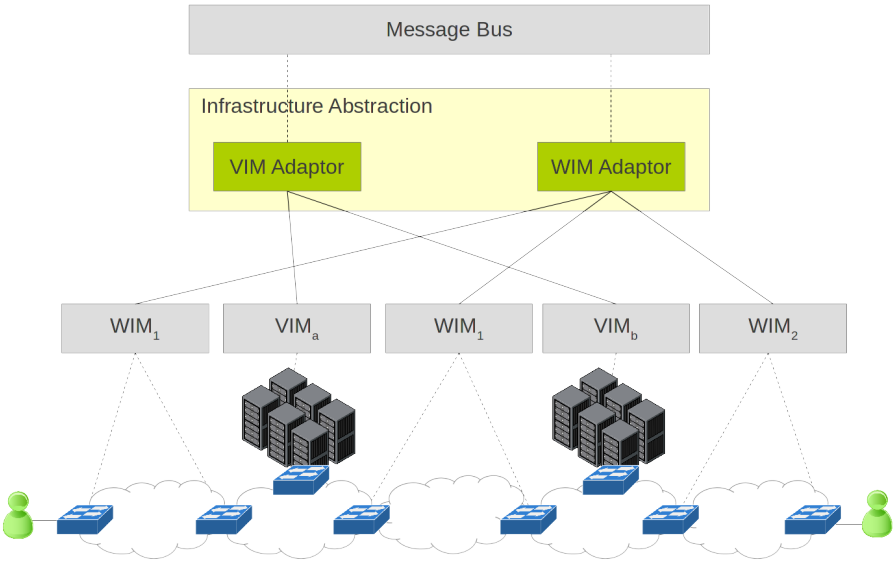

Figure 4.3: Infrastructure Abstraction Layer composition and its relation with the underlying infrastructure

VIMs configuration and the VIM-dependant information on the deployed and running network services.

4.1.6.2 WIM Adaptor

The WIM Adaptor component part of the Infrastructure Abstraction Layer is the one responsible to connect the WAN Infrastructure Manager to the SONATA SP. The WIM adaptor reuses the general architecture of the VIM adaptor as described above, adjusted to the WIM needs and functionalities. The adaptor provides a northbound API system in order to be controlled and use WIM functionalities. Then, the call is pass to the specific WAN Infrastructure Manager detected to be used by the SP, with a wrapper designed for that particular manager.

4.1.6.3 Inter-working

The VIM and WIM Adaptors offer their API through the message broker of the service platform and interact mainly with the following repositories:

- `son-plugin-manager` (MANO framework plugin)
- `son-service-lifecycle-manager` (MANO framework plugin)
- `son-gatekeeper` (standalone)

Although both the Adaptors are stand-alone modules, for internal monitoring purposes they register and send status information to the MANO Plugin Manager (see Section 4.1.3).

4.1.6.4 CI/CD added value

The development of an efficient abstraction layer for the functionalities offered by modern VIMs is a task which requires different kind of expertise, from pure software engineering to networking

and cloud computing. This expertise came from different partners in the consortium, and the CI/CD workflow and development instruments allowed these different actors to interact and focus on different aspects of the same module without overlaps, using a tested and controlled environment. The CI/CD workflow and roles ensured the integrity of the module code throughout the whole development, and facilitated the integration of the contributions coming from the various developers.

4.2 SDK

4.2.1 son-schema

Today's MANO systems use templates, also known as descriptors, to describe virtual network functions and services and to provide the necessary information in order to on-board and execute the functions. To describe these descriptors in a formal way, such that they can be verified automatically, SONATA leverages the JSON Schema language [17] and provides schemata for the VNF Descriptor (VNFD), the NS Descriptor (NSD), as well as the corresponding function and service records (VNFR and NSR). These schemata have been publicly available on GitHub from the beginning. Thus, they act as ground truth for any other SONATA components that consume SONATA descriptors, like the SONATA catalogs, e.g. for verification.

4.2.1.1 Inter-working

The son-schema can be directly used by any other component. Usually, there is a language-specific library for parsing JSON Schema and applying it, e.g., for verification purposes. However, since son-schema does not offer any consumable service itself, there is no inter-working with other components as such.

4.2.1.2 CI/CD added value

Since son-schema does not offer any consumable service, there is no integration nor an integration test with other components. The schema files themselves, however, are tested automatically.

4.2.2 CLI Tools

To support the development of network services and functions, a set of SDK tools was developed, providing the creation and management of user-specific work spaces and projects. Additionally, when the project reaches its maturity, it can be validated, published to private repositories (catalogues) and finally packaged into a bundle, ready to be instantiated.

4.2.2.1 son-workspace

The `son-workspace` tool plays two major roles, the creation and management of a development workspace/environment and the creation of projects. A workspace contains a user-specific configuration, such as user credentials and addresses to private catalogues, which can be used for the creation and maintenance of multiple projects. As such, it is recommended to instantiate the workspace at a common location (default is the user home folder), instead of at a directory associated with a specific project. While, a workspace belongs to a specific user/developer, a project may be composed of multiple shared components between developers, e.g. network functions.

The `son-workspace` tool itself does not interact with external components, it merely maintains the file structure and configuration of workspaces and projects which will be the base of operation for the remaining CLI tools.

4.2.2.2 son-publish

The `son-publish` tool is intended to provide a way of storing the components of a project to private catalogues. A developer typically implements a project that depends on third-party components, usually residing in private catalogues. This tool provides an easy way of publishing individual or full project components to a catalogue.

The publishing of a component first involves a validation process of the components before the actual upload to the catalogue. As such, the `son-publish` tool interacts with the `son-schema` repository to retrieve the appropriate template in order to validate the components.

4.2.2.3 son-package

The `son-package` tool is responsible for the packaging of a project, making it ready and available for instantiation in the Service Platform. The packaging process typically involves several interactions with `son-catalogue` for the verification and retrieval of dependencies and with `son-schema` for the syntax validation of project components and the final validation of the package itself.

4.2.2.4 son-push

While the tools son-project and son-package (amongst others) focus on the creation of workspaces and services, and are pure development-related tools, son-push forms the bridge with the service platform, which actually executes the created service package. As such, it forms the end of the pure development/SDK workflow. The son-push tool basically wraps the requests towards REST API of the (emulated) GK API, enabling to interact with the platform using a CLI interface. Currently, son-push focuses on basic functionality such as: i) uploading a package to the GK, ii) listing uploaded/available packages at the GK, iii) (on the emulated GK) deploying a service, and iv) listing deployed services (instances). A more in-depth description of son-push and its usage is described in D3.1 and its annexes.

4.2.2.5 Inter-working

As previously described, this set of tools directly interacts with the following modules:

- `son-schema`
- `son-catalogue`

As for the son-push module, it interacts with either the GK of SDK emulator (son-emu) or the service platform using a REST API. The interaction with other SDK tools is limited to the production of the file(s) which will be used in the communication process towards the GK.

4.2.2.6 CI/CD added value

The CLI toolset is the main support framework for the creation, development and sharing of network services. As such, the integration of these tools with its dependable modules is of crucial matter. Following a CI/CD test workflow promotes a constant verification of interoperability, to make sure that a working version is always available. Moreover, since son-push forms the bridge between

the SDK and the SP, the CI/CD approach enables to automatically validate this relationship and quickly identify in case mismatches occur during the parallel development of both. Issues related to the implemented REST API or supported package/schema formats can therefore be easily identified.

4.2.3 son-catalogue

SDK catalogues are developer's local or private catalogue within the SONATA SDK. The SDK catalogues contain the network service descriptors (NSD) and VNF descriptors (VNFD) that are either imported from the Service Platform catalogues or produced by the developer him/herself. The main objective of the SDK catalogues is to support the development of new NSDs, VNFDs, etc. by making existing NSDs, VNFDs, etc., developed by other developers, available locally. The SDK Catalogues component enables an API between SDK components to the Catalogues Database for the management of CRUD operations for the different descriptors.

In terms of implementation, a mongoDB acts as the main storage and each catalogue maps to a separate collection within the mongoDB. A REST API, as front-end to the mongoDB, enables access for other SDK modules to retrieve or store descriptors in the SDK catalogue.

4.2.3.1 Inter-working

Within the SDK, the son-publish and son-package modules need SDK catalogues for storing and retrieving descriptors, NSD and VNFD, respectively.

- `son-package` executes a GET operation, according to the API defined above, to retrieve the required descriptor.
- `son-publish` module performs a POST operation, according to the defined API, in order to store a descriptor in the SDK catalogues.

4.2.3.2 CI/CD added value

The CI/CD approach facilitated the integration of son-catalogues with son-package and son-publish to verify correct creation and storage of different descriptors in the SDK Catalogues, respectively.

4.2.4 son-emu

Son-emu was created to support network service developers to locally prototype and test complete network services in realistic end-to-end multi-PoP scenarios. This emulation platform allows the direct execution of real network functions, packaged as Docker containers, in emulated network topologies running locally on the network service developer's machine. son-emu integrates with management and orchestration (MANO) system, like the SONATA service platform, which are responsible to deploy and manage the network services tested in the emulated environment. This is not possible with existing approaches, that either rely on local cloud testbeds that lack multi-PoP support, simulations that only execute simplified versions of network functions, or network emulation tools that do not offer cloud-like interfaces to interact with MANO systems.

Figure 4.4 shows the scope of our solution and its mapping to the ETSI NFV reference architecture in which it replaces the network function virtualisation infrastructure (NFVI) and the virtualised infrastructure manager (VIM). The design of `son-emu` is based on a tool called Containernet which extends the well-known Mininet emulation framework and allows us to use standard Docker containers as VNFs within the emulated network. It also allows adding and removing

containers from the emulated network at runtime which is not possible in Mininet. This concept allows us to use the emulator like a cloud infrastructure in which we can start and stop compute resources (in the form of containers) at any point in time. Containernet is developed by Paderborn University and is publicly available on GitHub [23].

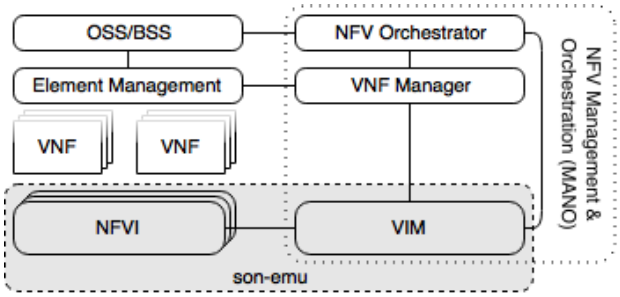

Figure 4.4: SONATA emulation tool mapped to ETSI reference architecture

4.2.4.1 Inter-working

Son-emu emulates a service platform together with a multi-PoP cloud environment on a local machine. As a result, it interacts with the same SDK components like the service platform does. These are in particular:

- `son-push`
- `son-monitor`

These tools are used by network service developers to deploy and monitor their networks services on the emulation platform, like described in D3.1 Section 3.4.3. Based on this, integration tests are needed that validate the inter-working between these components and the emulator.

4.2.4.2 CI/CD added value

Son-emu is currently the most complex tool in SONATA's SDK an relies on several other open source projects, like Containernet [23] and complex software libraries like docker-py which are all still under development. This often causes changes in the underlying code bases which might cause bugs in son-emu which were not introduced by implementation actions on the emulator itself. The used CI/CD approach helped to identify these issues and to ensures that a running version of the emulator is always available.

Another great benefit of the used CI/CD approach is that new package (schema) formats are directly tested on the emulator to ensure that both, schema processing components in the emulator's dummy gatekeeper and the schemas as such are still aligned.

4.2.5 son-monitor

The SDK component son-monitor is a tool to facilitate monitoring of network and compute resources of deployed services in the SDK emulator, and in a second stage also monitoring in the real SONATA SP. The toolset aims to contribute to a more efficient development and deployment cycle by providing monitor functionalities that are useful in the context of network service creation and debugging. Son-monitor relies on interaction with two infrastructure management components to obtain this information:

1. the network control entity (e.g., the SDN controller used by son-emu)

2. the SONATA monitoring framework which is built upon the combination of Prometheus and PushGateway functionality as documented in D4.1, section 5.2.4. Both interact with the son-monitor tool using a REST api, enabling the developer to inspect ongoing network (e.g., number of packets exchanged) or compute statistics (e.g., cpu usage) of instantiations of service components. The architecture and provided functionality of the SDK son-monitor tool can be found in D3.1, section 3.6.

The general monitoring approach is shown in fig:son-mon-general. The monitoring framework can connect to VNFs, monitoring agents, the infrastructure, network or the SP itself. Metrics to be monitored are either defined in the initial service/VNF descriptor, or started/stopped manually.

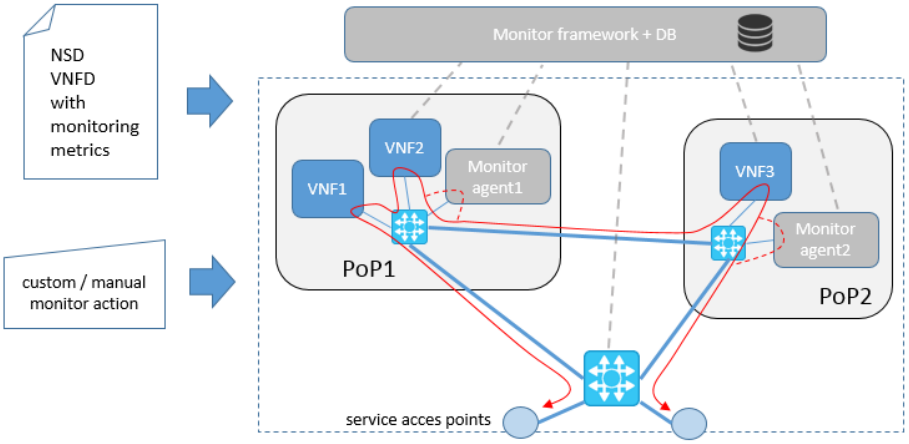

Figure 4.5: SONATA monitoring framework general approach

4.2.5.1 Inter-working

Son-monitor interacts with son-emu, as well as the monitoring framework via REST APIs. In addition, the son-cli tool provides a command-line interface to son-monitor functionality:

- `son-emu` : The emulator where a VNF is deployed and whose metrics are exported.
- `son-monitor` : The Prometheus monitoring framework which gathers all exported metrics (in the SDK and in the SP).
- `son-cli` : The SDK workspace from where a developer gives the instruction to start and monitor certain metrics of a VNF.

4.2.5.2 CI/CD added value

In order to ensure that interfacing between son-monitor and son-emu, as well as between son-monitor and the monitoring platform is adequately functioning, a CI/CD approach has been set up. This will check that, despite changes in individual components, the defined REST APIs are still operational, and enable the son-monitor tool to fetch monitoring data from either platforms. Besides the interfacing between the different modules, the son-monitor functionality also builds

upon the ability of setting up a service correctly. The CI/CD test will ensure that this basic functionality is working in the available versions of son-emu and son-cli.

4.2.6 son-analyze

The `son-analyze` tool aims to bridge network services with an analytic framework. It completes the `son-monitor` tool to offer a data oriented view. The end-user has the possibility to quickly implement his custom analysis. This approach can also embed SONATA's algorithms. To achieve this goal, `son-analyze` creates a Docker container on the end-user's laptop. This container ships with an analytic framework and a SONATA library to handle the communication with the SONATA Service Platform. It manages, for example, the download of metrics corresponding to a desired time span and creates a high level structure to work with.

4.2.6.1 Inter-working

`Son-analyze` interacts with the SONATA Service Platform's Gatekeeper through a REST API. It can also communicate with `son-emu`, by using a dummy gatekeeper. For now, as the SONATA Service Platform is developed in an agile way, the `son-analyze` retrieves metrics directly from `prometheus`. The integration tests validate the correct retrieval of metrics.

4.2.6.2 CI/CD added value

`Son-analyze` requires metrics to work in its basic form. This data must come from external sources by relying on several REST apis. A CI/CD loop is important to check the compatibility across all components.

5 Integration Tests

In this section we report the Integration tests designed to guarantee the integrity of the interfaces between the SONATA modules described in details in deliverables D3.1 and D4.1 [12][13], and which are resumed in Appendix A.

In the following Table 5.1 we summarize the designed integration tests, and for each test we explicit the involvement of the software modules and repositories (see Table 5.2), described in Section 4.

Table 5.1: Integration tests table

No.	TEST	1	2	3	4	5	6	7	8	9	10	11	12
1	Catalogue - Gatekeeper							X		X			
2	CLI - Development Workflow	X	X	X									
3	Upload/Retrive a package from SDK to Gatekeeper		X					X		X			
4	Deploy a New Service							X	X	X	X		X
5	MANO Framework Plugin Management										X		
6	Deploy a Service Package on the Emulator by using Son-Push	X	X		X								
7	Service Lifecycle Manager - Repositories									X	X		
8	Monitor service instance running on Emulator by using Son-Monitor				X	X	X						
9	Platform instantiation							X	X		X		
10	BSS - Gatekeeper							X		X			X
11	Service Lifecycle Management and Infrastructure Abstraction								X		X		
12	Monitoring Server Integration						X				X		
13	Monitoring Server, GK API, GK GUI connectivity						X	X				X	
14	Gatekeeper - Son-monitor - Son-analyze					X	X	X					

Table 5.2: Integration tests legend

Module name	Numbers in Table 5.1
SCHEMA	1
CLI	2
CATALOGUE	3
EMU	4
ANALYZE	5
MONITOR	6
GKEEPER	7
INFRABSTRACT	8

Module name	Numbers in Table 5.1
CATALOGUE-REPOS	9
MANO-FRAMEWORK	10
GUI	11
BSS	12

5.1 Catalogue - Gatekeeper

5.1.1 Test Description

This integration test focuses on the interactions between the "**Gatekeeper**" and the "**SP Catalogues**". More specifically, the test verifies the storage of a package, along with its constituent elements e.g., NSD, VNFD, etc., in the SP-Catalogues. Afterwards the Gatekeeper retrieves them for a possible service deployment or for network service listing purpose in the BSS. Figure 5.1 and Figure 5.2 illustrate the storage and retrieval phase of the test, respectively.

5.1.1.1 GK stores SON Package in SP Catalogues

This test should be performed for the following scenarios:

- Storing a SON Package including required NSD, VNFDs and PD

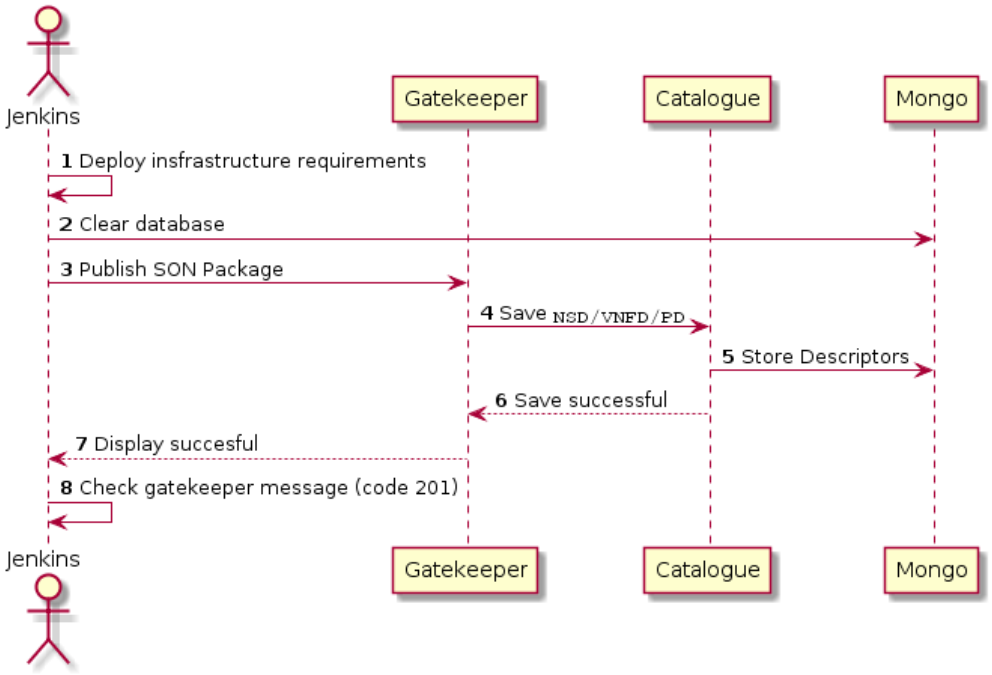

Figure 5.1: Store a SON package with NSD, VNFDs and PD

5.1.1.2 GK retrieves NSD and VNFD from SP Catalogues

This test should be performed for the following scenarios:

- Retrieving NSD (using UUID)

- Retrieving VNFDs (using UUID)

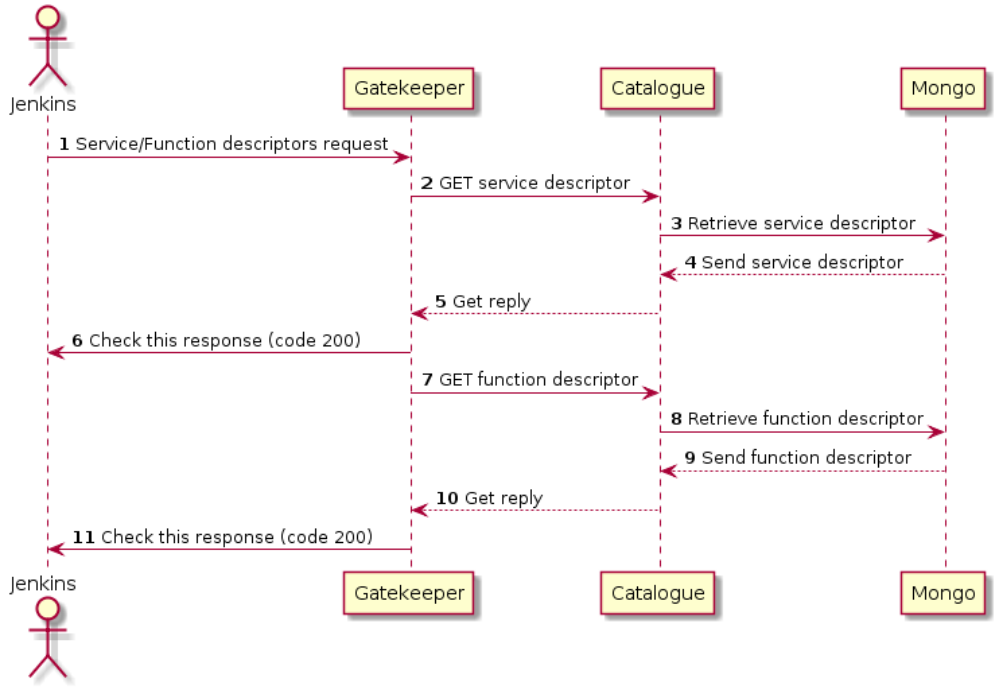

Figure 5.2: Retrieve NSD, VNFDs from SP Catalogues PD

5.1.2 Infrastructure Requirements

- Gatekeeper API (son-gkeeper/son-gtkapi)
- Gatekeeper API (son-gkeeper/son-gtkpkg)
- Catalogue (son-catalogue-repos)
  - Catalogue API
  - Mongo Database

5.1.3 Tests Requirements

- **From test**
  - A SON Package which includes services and functions descriptors and a package meta-inf
  - A packaged NSD
  - Packaged VNFDs
  - Packaged PD (meta-inf)
- **From gatekeeper**
  - API Method to publish a SON Package
  - API Method to retrieve descriptors

- **From Catalogue**
  - API Method to store descriptors
  - API Method to retrieve descriptors

5.1.3.1 Initial configuration of the environment

- **From gatekeeper**
  - Gatekeeper API is running
- **From Catalogue**
  - Catalogue API is running
  - Catalogue Mongo Database is empty

5.1.3.2 Expected results from modules:

- **From gatekeeper**
  - Expected message code 201 (Save successful)
  - Expected message code 200 (Retrive data from Catalogue)
- **From Catalogue**
  - Expected message code 201 (Save successful)
  - Expected raw data stored in Mongo DB, code 200

5.1.4 Triggers

- Who will launch this test?
  - Jenkins
- When?
  - Daily
  - Scheduled on need basis

5.1.5 Post actions

If the test fail, send email to *lead developers*.

5.2 CLI - from zero to package and beyond

This set of integration tests start with an empty filesystem, creates workspaces, several projects, publishes and packages them. This integration workflow is used to test the CLI components with different workspace configurations and projects with different sets of components. Additionally, it assures that packages are generated correctly by instantiating them in not only in the SDK emulator but also in the Service Platform.

The CLI is composed by several independent tools, described as follows:

- `son-workspace` is responsible for the initialisation of the working environment for development of projects. This tool is also used to create and instantiate projects.
- `son-package` gathers all components from a project and packages them in a zipped container in order to be pushed to the service platform Gatekeeper. During the packaging process, external project components are retrieved from the son-catalogue repository and all components are syntactically validated against latest schema templates (provided by son-schema repository).
- `son-publish` can be used to publish project components or entire projects to son-catalogue repositories. Before publishing, this tool validates the components against the latest schema template, also retrieved from the son-schema repository.

5.2.1 Infrastructure Requirements

List of modules affected and corresponding repository:

- **son-workspace**: son-cli
- **son-package**: son-cli
- **son-publish**: son-cli
- **son-catalogue**: son-catalogue
- **son-schema**: son-schema
- **son-emu**: son-emu
- **son-gtkapi**: son-gkeeper
- **son-gtkpkg**: son-gkeeper

5.2.2 Test Story

The CLI tools are independent, functionality and communication wise. However they dependent on the available file structure. In other words, the `son-package` and `son-publish` tools must have a workspace and a project in order to be executed, thus an invocation of `son-workspace` must be preceded. To be able to test the integration between CLI tools, a test story was defined following two major sets of tests:

- Test CLI tools and test the integration with the Emulator Gatekeeper (`son-emu`)
- Test CLI tools and test the integration with the Service Platform Gatekeeper (`son-gkeeper`)

In each set of tests, the following procedure is followed:

5.2.2.1 Workspace configuration

1. Create a new workspace: a workspace environment is created to support the development of projects
2. Configure workspace: the workspace is configured with the developer credentials, son-schema repository address, son-catalogue repository address and several customized parameters, such as level of debugging, name of workspace, etc.

3. Create a new project: a project is created. A service descriptor (NSD) is added to the project, as well as all its required function descriptors (VNFDs). This results in a self-contained project, i.e. it does not depend on external components.

The sequence diagram for the above steps is shown in Figure 5.3

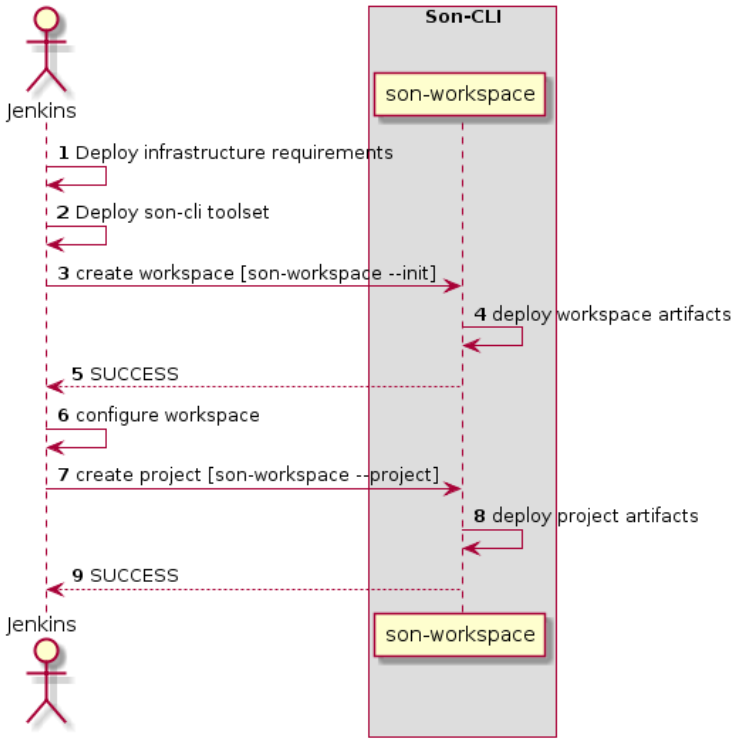

Figure 5.3: Workspace configuration

5.2.2.2 Project Publication in SDK catalogue

1. Publish project: using the `son-publish` tool, the components of the project are published to the son-catalogue repository. This will ensure that all the components used in this project are available at the son-catalogue repository.

The sequence diagram for this step is shown in Figure 5.4

5.2.2.3 Package upload with catalogue dependencies

1. Package project: the `son-package` tool will create a bundled file containing the necessary components that define the service of the project. The packaging of the project validates all the components against the schema templates retrieved from the son-schema repository. Since this project does not depend on external dependencies, in this case the son-catalogue will not be contacted.

2. Remove project components: this task will remove two VNF descriptor files from the project. This will enforce the retrieval of such descriptors from son-catalogue, the next time the project is packaged.

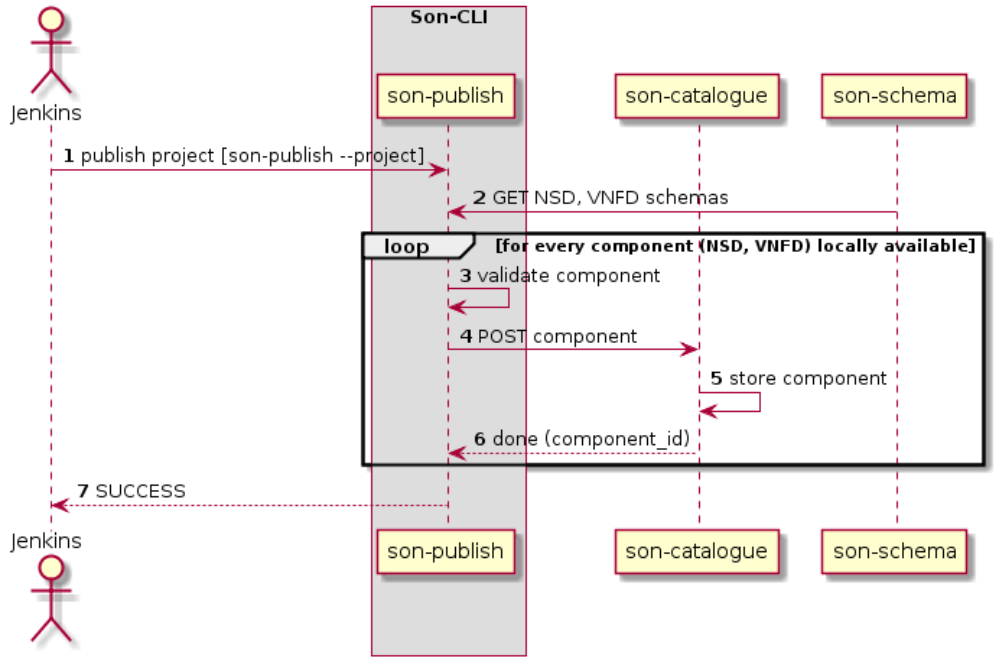

Figure 5.4: Project Publication in SDK catalogue

3. Package project: similar behaviour of step 5 will occur, with the difference that now the project depends on components that are not present in the local filesystem. Thus, the son-catalogue is contacted in order to retrieve the missing components. They will of course be available because the project components were published in previous test story.

The sequence diagram for these steps is shown in Figure 5.5

5.2.2.4 Package instantiation

1. Instantiate package: the `son-push` tool is used to upload the generated package to the Gatekeeper API.

The sequence diagram for this step is shown in Figure 5.6

5.3 Upload/Retrieve a Package from SDK to Gatekeeper

5.3.1 Test Description

This integration test is intended to validate the interaction of the Service Platform Gatekeeper. The `curl` tool is used to upload a valid and pre-stored service package to the Gatekeeper. Therefore, this test is not aimed to validate the package itself, as it does not contemplate its creation by the SDK tools, only the Gatekeeper API.

5.3.2 Infrastructure Requirements

List of modules and corresponding repository:

- **son-gtkapi**: son-gkeeper

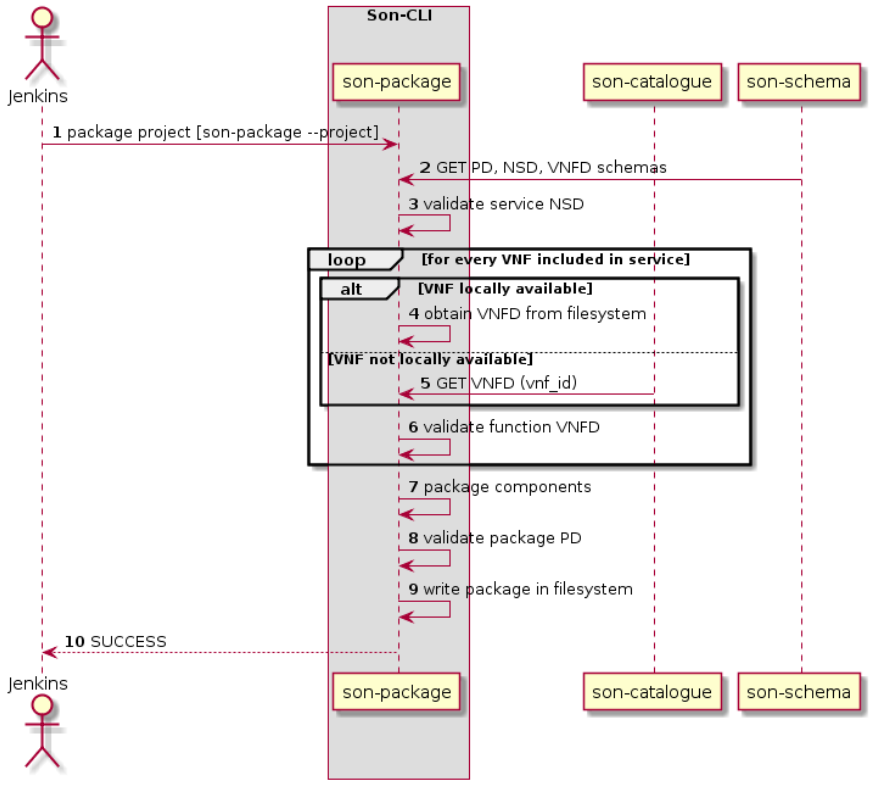

Figure 5.5: Packages upload with catalogue dependencies

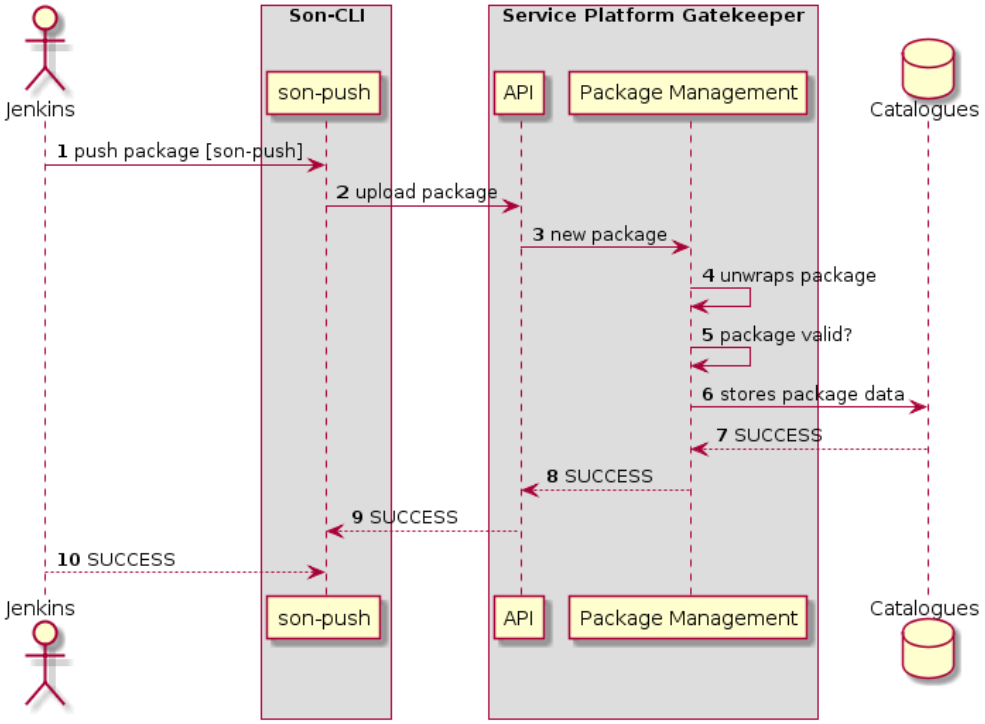

Figure 5.6: Package instantiation

- **son-gtkpkg**: son-gkeeper
- **son-catalogue-repos**: son-catalogue-repos

5.3.3 Test Story

To test the Gatekeeper API, a valid package available at the son-schema repository is used, as detailed in Section 4.2.1. Tests are executed by sending HTTP requests with curl to the GK front-end, using the targets specified below:

1. submit a valid package:
   - **Set-up:** have the package (`sonata-demo.son`, in the current folder)
   - **Execution:** curl -X POST -F "package=@sonata-demo.son" at `/packages`
   - **Expectation:** HTTP code 201 (Created) is returned, together with the JSON representation of the created package;
   - **Tear-down:** remove the package from the current folder;
2. get a specific package by its UUID:
   - **Set-up:** have the `:uuid` of a package successfully created in the Catalogue;
   - **Execution:** curl at `/packages/:uuid`
   - **Expectation:** HTTP code 200 (OK) is returned, together with the package file;
   - **Tear-down:** remove the package from the folder where it has been downloaded;
3. get a specific package by its vendor `:vn`, name `:n` and version `:vr`:
   - **Set-up:** have the `:vendor`, `:name` and `:version` of a package successfully created in the Catalogue;
   - **Execution:** curl at `/packages?vendor=:vn&name=:n&version=:vr`
   - **Expectation:** HTTP code 200 (OK) is returned, together with the package file;
   - **Tear-down:** remove the package from the folder where it has been downloaded;
4. get a list of packages:
   - **Set-up:** have at least two packages successfully created in the Catalogue;
   - **Execution:** curl at `/packages`
   - **Expectation:** HTTP code 200 (OK) is returned, together with the list of packages in the Catalogue;
   - **Tear-down:** remove the packages from the Catalogue;

The test could be divided into two sub-stories, shown in Figure 5.7 and Figure 5.8, respectively.

5.4 Deploy a New Service

This chapter contains information about the end to end test for deploying a new service from the SONATA's Business Support System.

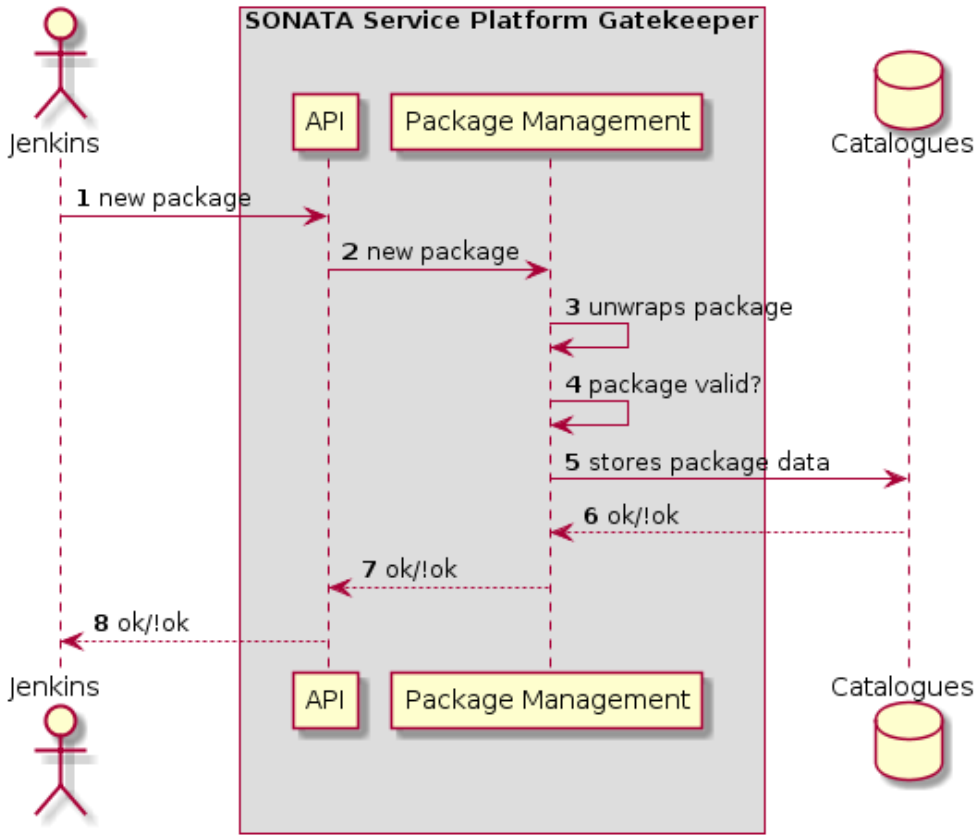

Figure 5.7: The Jenkins sends a package to the Gatekeeper

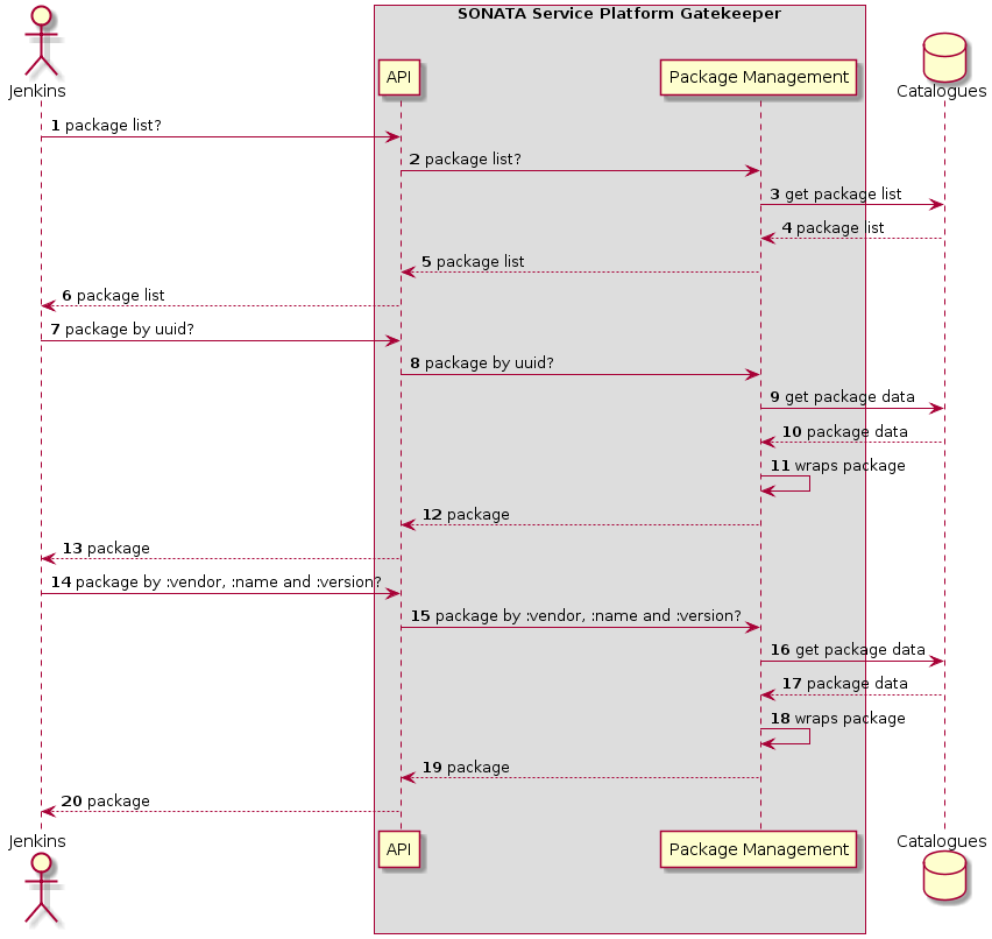

Figure 5.8: The SDK requests a package to the Gatekeeper

5.4.1 Test Description

This test uses the BSS system to deploy a new service in the SONATA service platform, and retrieves the functional information of all involved components to check the proper functioning of the system: logs traces, database records, etc. The test flow is shown in Figure 5.9.

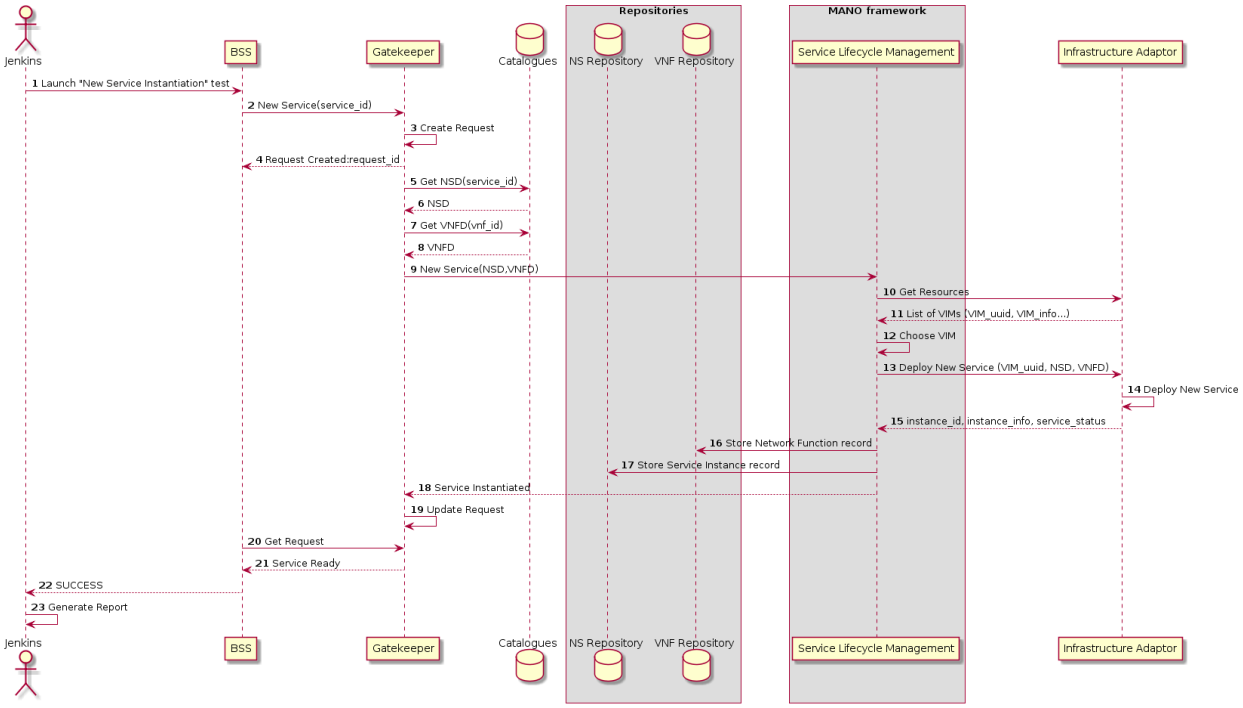

Figure 5.9: Deploy a new service

5.4.2 Infrastructure Requirements

This test makes use of the following components:

- Gatekeeper:
  - BSS: son-bss
  - API: son-gkeeper/son-gtkapi
  - Service Management: son-gkeeper/son-gtksrv
- Catalogues: son-catalogue-repos
  - Catalogues API
  - Mongo Database
- Mano-Framework: son-mano-framework
  - RabbitMQ
  - Service Lifecycle Manager
- Infrastructure Adaptor: son-sp-infrabstract
- OpenStack instance with Heat

5.4.3 Tests Requirements

This sections reflects the initial configuration needed to perform the test and the expected results to achieve.

5.4.3.1 Initial configuration of the environment

Installed and running BSS, Gatekeeper API, Gatekeeper Service Management Service platform catalogues, Repositories, MANO framework and Infrastructure Adaptor.

From Gatekeeper

- API Method to retrieve the list of available services
- API Method to retrieve the stored requests
- API Method to create an instantiation request

From Catalogues

- API Method to retrieve the network service descriptors (NSD) and the virtual network function descriptors (VNFD)
- Valid NSDs and VNFDs stored in mongo database

From Mano-framework

- RabbitMQ
- MongoDB (for plugin manager)
- Plugin manager
- Plugin manager control CLI

From Infrastructure Abstraction:

- VIM-Adaptor should be up and running
- The OpenStack VIM is registered to the Service platform

5.4.3.2 Expected results from modules

From BSS:

- List of available services for instantiation
- New Service Instantiation launched
- List of Instantiation Requests created by Gatekeeper

From Gatekeeper:

- Creation of Service Instantiation Request
- Status Update of Service request when service is ready

From Mano-Framework:

- Communication with Gatekeeper and Infrastructure Abstraction through the rabbitmq
- Creation of new services and functions instances in repositories

From Infrastructure Adaptor:

- List of VIMs
- New Service Deployed

5.4.4 Test Triggers

- Jenkins (daily)
- Scheduled on need basis

5.4.5 Post Actions

- Send an email to *lead developers*

5.5 MANO Framework Plugin Management

5.5.1 Test Description

This test targets the integration between different components of the MANO framework. Specifically it tests the functionalities needed to manage and monitor MANO plugins connected to the system. The test runs an instance of the MANO framework in which only the plugin manager component together with its database and message broker is running. In this environment, a test MANO plugin is started and connected to the system to perform the tests. There are three main test cases that are performed during this integration test. The first case tests the plugin registration mechanism and is shown in Figure 5.10. The second test case checks if a platform operator can stop a running plugin using the plugin manager's control interface. This test scenario is shown in Figure 5.11. The third case validates that the plugin monitoring functionalities, in particular the heartbeat mechanism, are working by fetching plugin status information from the plugin manager (Figure 5.12).

5.5.2 Infrastructure Requirements

This test makes use of the following repositories:

- son-mano-framework

5.5.3 Tests Requirements

Initial configuration of the environment

- Installed and running MANO framework (each component is oner Docker container)
  - RabbitMQ
  - MongoDB (for plugin manager)
  - Plugin manager
  - Plugin manager control CLI

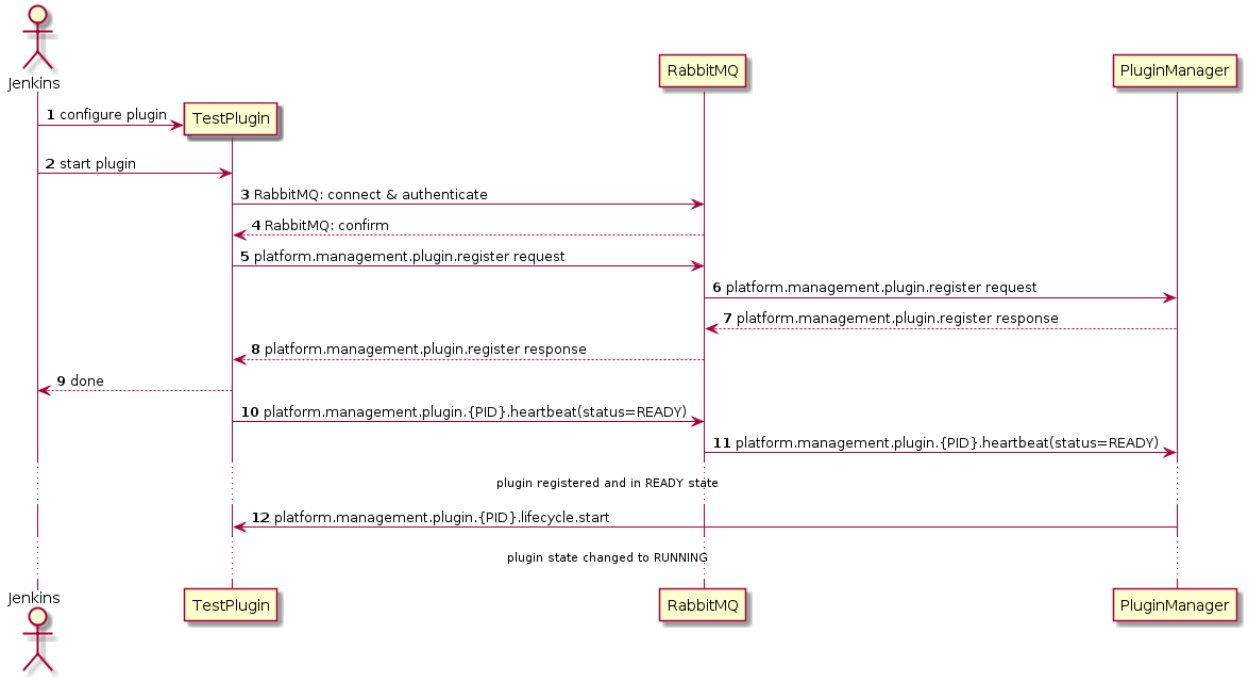

Figure 5.10: Test case 1: Plugin registration

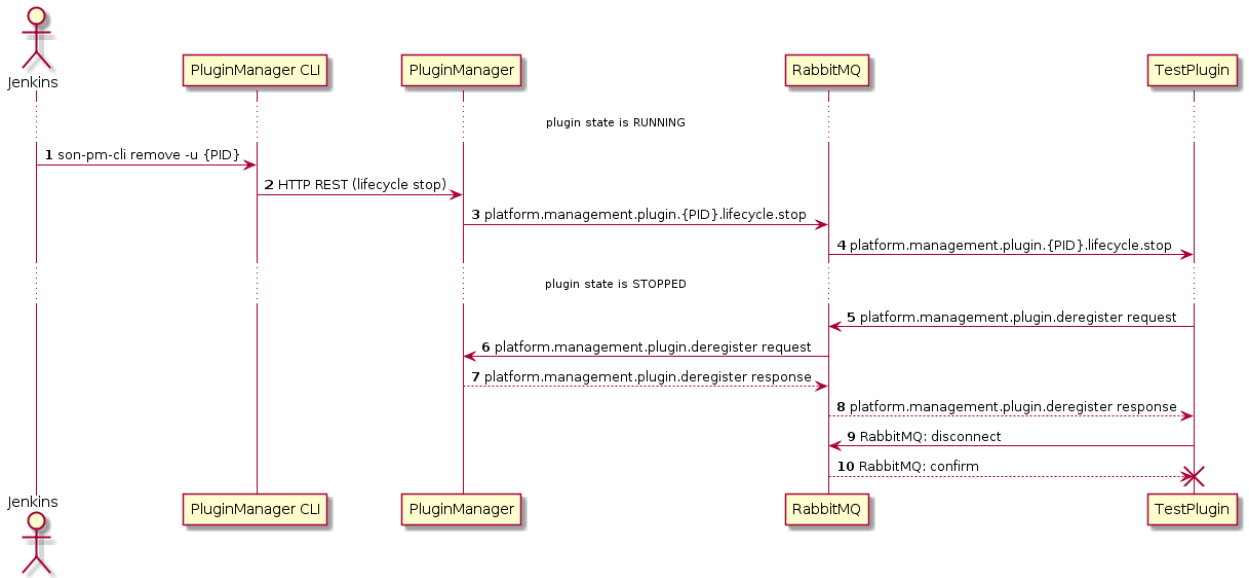

Figure 5.11: Test case 2: Plugin de-registration

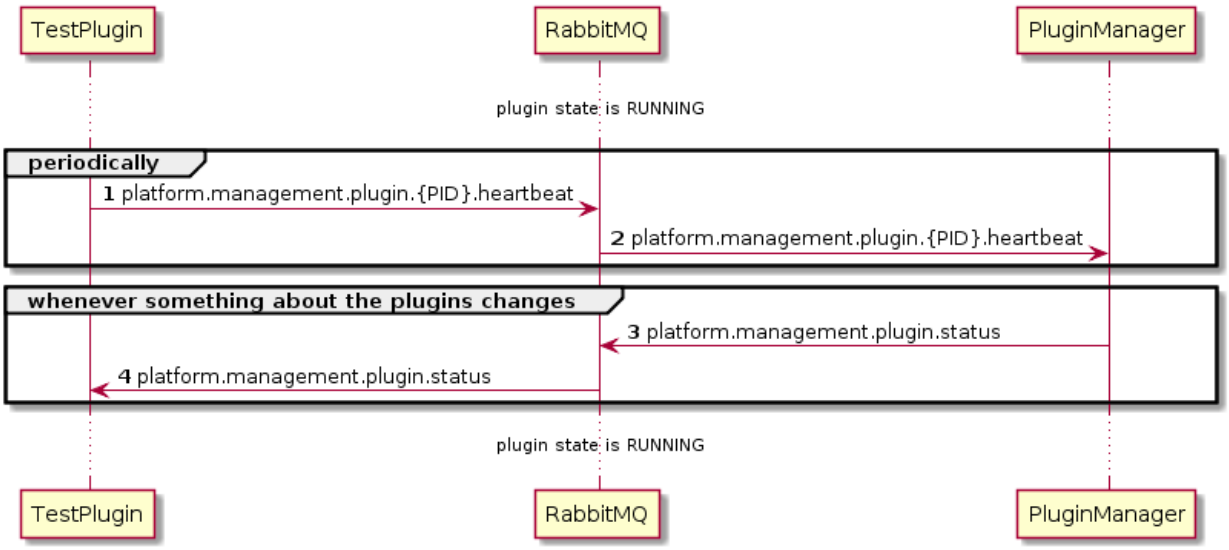

Figure 5.12: Test case 3: Plugin status collection

Expected results from modules:

- There is a new test plugin started and connected to the system. The CLI tool is used to verify that the plugin is correctly registered and running. After this, the plugin is instructed to be disconnected and stopped. This is again verified by the CLI. Additionally the CLI and the plugin status functionality is used to verify that the periodic health status updates reported by each plugin are working.

API Method tested

- Plugin manager (RabbitMQ):
  - Plugin register
  - Plugin heartbeat
  - Plugin lifecycle start
  - Plugin lifecycle stop
  - Plugin status
- Plugin manager (REST control interface):
  - List plugins
  - Get plugin status details
  - Stop plugin remotely

5.5.4 Test Triggers

- Jenkins (daily)
- Scheduled on need basis

5.5.5 Actions after the test

- Send an email to *lead developers*

5.6 Deploy a Service Package on the Emulator by using Son-Push

5.6.1 Test Description

This test targets the integration between SDK components and the emulation platform *son-emu*. It uses the latest predefined example service package available in the *son-schema* repository and uploads it to the emulator by using the *son-push* tool of the SDK. After this the example service is instantiated using the son-push tool. The test uses the dummy gatekeeper component of the emulator that provides the same interface like the real service platform towards the SDK tool, like son-push. It deploys the example service on a small emulated topology consisting of two emulated PoPs on which the VNFs of the service are placed with a round robin scheme. The test validates that the emulator is able to deploy the latest example service package, including the latest descriptor formats (NSD, VNFD) and that the interfaces offered by the dummy gatekeeper are compatible with the son-push tool. The interaction described above is depicted in Figure 5.13.

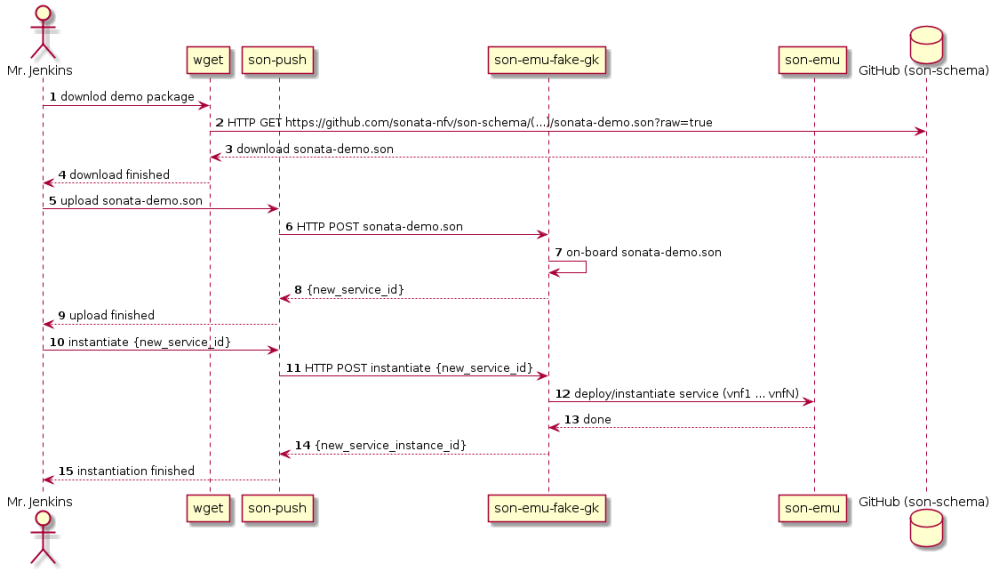

Figure 5.13: Deploy service package on son-emu using son-push

5.6.2 Infrastructure Requirements

This test makes use of the following repositories:

- son-schema
- son-cli
- son-emu

5.6.3 Tests Requirements

Initial configuration of the environment

- A newly provisioned VM in which the emulator and the son-cli tools are installed.
- The emulator is started with the test topology.
- The latest example service package is downloaded from the son-schema repository.

Expected results from modules:

- The example service is deployed and running on the emulator platform. This is validated through the emulator CLI by listing all executed containers and verifying their status.

API Method tested

- Dummy Gatekeeper:
  - Upload a service package
  - Get UUID of new service
  - Instantiate the service
  - Get UUID of new service instance
- Son-emu CLI:
  - List running containers

5.6.4 Test Triggers

- Jenkins (daily)
- Scheduled on need basis

5.6.5 Actions after the test

- Send an email to *lead developers*

5.7 Service Lifecycle Manager - Repositories

5.7.1 Test Description

This integration test targets the interactions between the service lifecycle manager (son-service-lifecycle-manager) and NF, NS and monitoring repositories (son-catalogues-repositories and son-monitor). More specifically, it verifies the storage of NSR and VNFRs in the NS and NF repositories along with triggering the monitoring for the respective NSR and VNFRs. The following figure Figure 5.14 illustrates the three verifications including storage of NSR in NS repository, VNFR in the NF repository and triggering of monitoring for the particular NSR and VNFR.

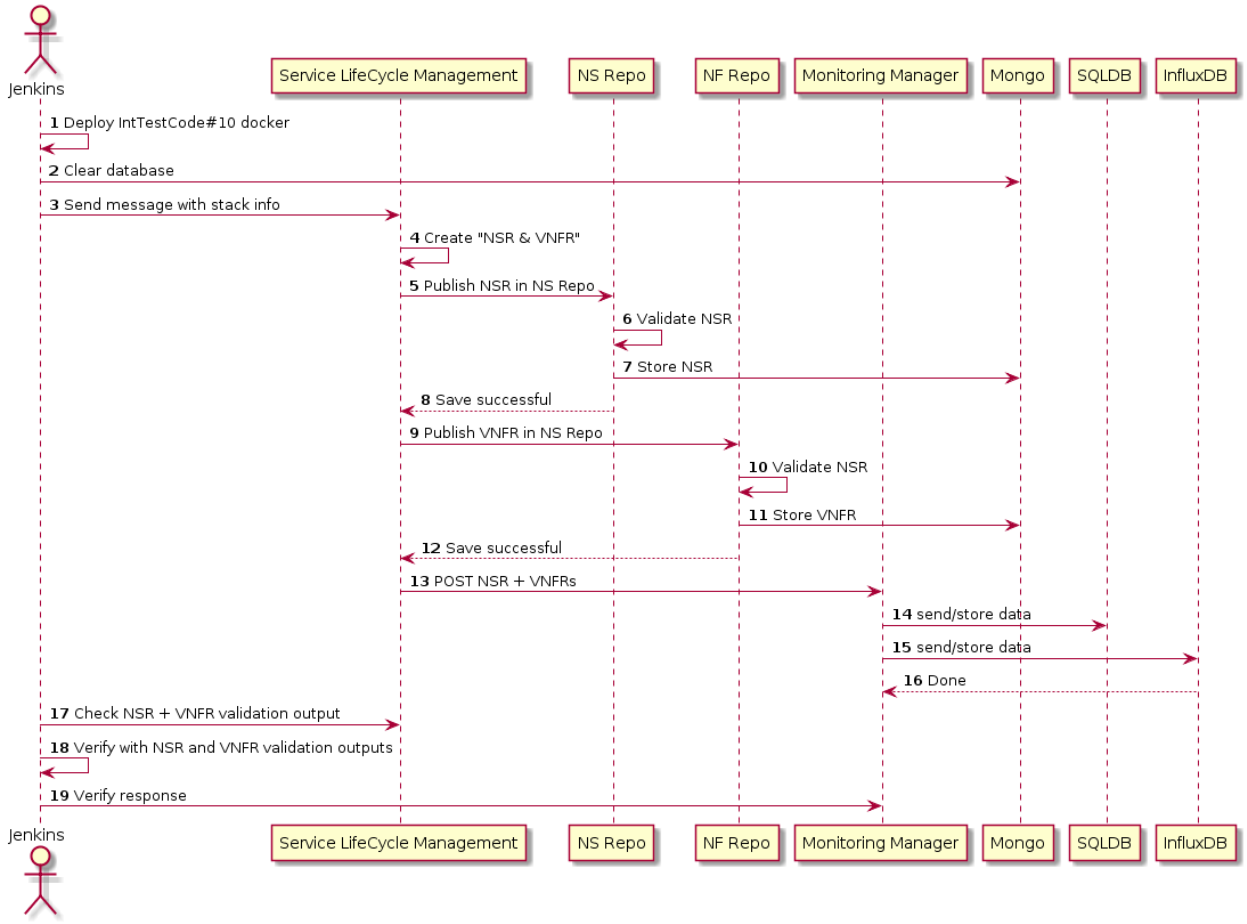

Figure 5.14: SLM NS + NF + Monitoring Repos

5.7.1.1 Infrastructure Requirements

List of modules that will be used for the test and the respective repositories are:

- son-mano-framework
  - SLM
  - RabbitMQ
- son-catalogue-repos
  - NS Repo
  - NF Repo
    - MongoDB
- son-monitor
  - Monitoring Repo
    - SQLDB
    - InfluxDB

5.7.1.2 Tests Requirements

- From Test
  - Information of the deployed stack is available to SLM for creating NSR and VNFR.
  - Integration test Section 5.11 is a pre-requisite.
- From SLM
  - API method to publish NSR and VNFR
    - Expected message (Save Successful)
- From NS Repo
  - Expected message (Save Successful)
  - NSR validated and stored.
- From NF Repo
  - Expected message (Save Successful)
  - VNFR validated and stored.
- From Monitoring Repo
  - Monitoring initiated for the particular NSR.
- Overall Expected results from modules
  - Creation of NSR and VNFR in SLM which are validated and stored in NS repo and NF repo, respectively and triggering up monitoring for NSR and VNFRs.

5.7.1.3 Triggers

In this section you should define:

- Who will launch this test?
  - Jenkins
- Scheduled
  - Need basis

5.7.1.4 Post actions

- Send an email to *lead developers*

5.8 Monitor service instance running on Emulator by using Son-Monitor

5.8.1 Test Description

This test targets the integration between SDK monitoring components and the emulation platform *son-emu*. For emulated compute and network infrastructure, it relies on a demo topology provided by the emulator, which consists of two interconnected data centers. Once this infrastructure is loaded in the emulator, a dummy VNF (i.e., a default Ubuntu Docker container) with two interfaces is deployed on one of the datacenters. The test subsequently validates that network, as well as node monitoring queries are correctly executed.

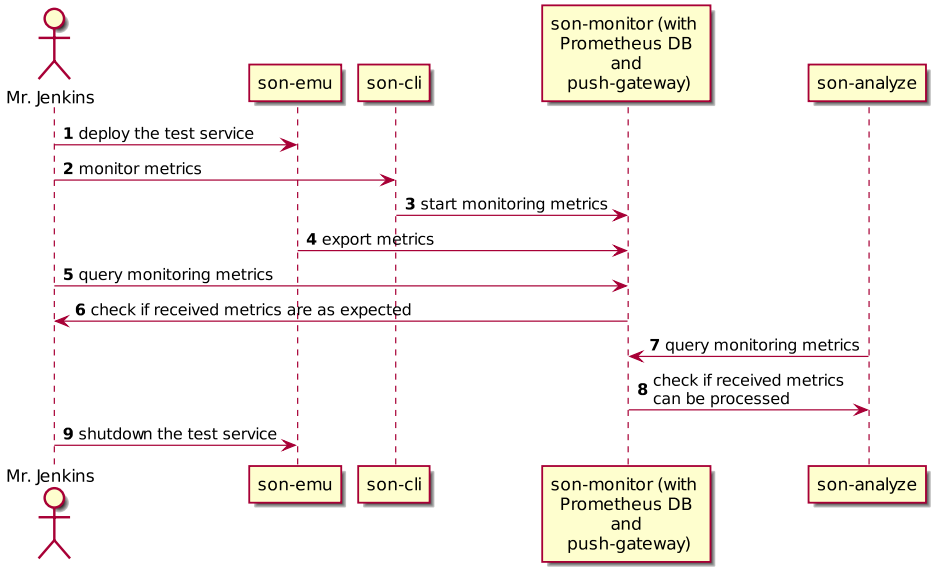

Figure 5.15: Integration test: SDK son-monitor

5.8.2 Infrastructure Requirements

For using the latest build of the dependent modules, the test needs access to the Sonata docker registry and publicly available docker images . This test makes use of the following repositories:

- son-emu: SDK emulator where a test VNF is deployed. A REST api and a ZeroRPC interfaces are available to deploy the test VNF and monitor several metrics. son-emu contains cAdvisor (from the public repository google/cadvisor) to export metrics. Son-emu can be deployed as a docker container.
- son-cli: Environment where specific son-monitor commands are implemented and can be executed. These are additional features which are not available in son-emu and are part of future SONATA implementations.
- son-monitor: Monitoring platform, i.e. Push Gateway and Prometheus (available from the public repository prom/prometheus or as build by the son-monitor SP repository) together with a set of commands available from son-cli.
- son-analyze: Queries the monitoring platform and analyzes the monitored metrics of the test VNF (eg. cpu load, tx packet rate). This can be deployed as a docker container which will use the interface to son-emu or son-monitor to gather metrics.

The total setup and the interfaces between the modules is shown in Figure 5.16.

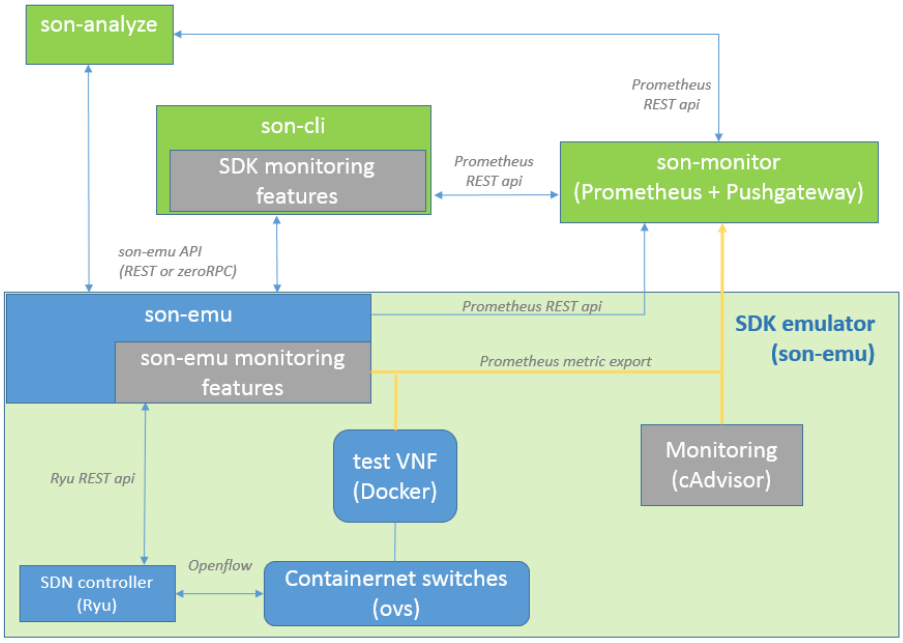

Figure 5.16: Architecture and components in son-monitor

5.8.3 Tests Requirements

Initial configuration of the environment

- The emulator is started with the test topology.
- cAdvisor is deployed inside the emulator to gather standard compute and network metrics
- Prometheus and Push Gateway are correctly deployed
- son-cli is correctly deployed
- son-analyze is correctly deployed

Expected results from modules:

- A test VNF or service is correctly deployed and returned successful by son-emu.
- Monitoring data is correctly fetched from son-monitor and son-analyze. The test displays that the test was OK or not OK.

API Method tested

- Son-emu:
  - Start VNF
- Son-monitor:
  - Trigger network monitoring, i.e. fetch packet counters
  - Start node monitoring, i.e. fetch aggregated cpu load
- Son-analyze:
  - Query the VNF cpu load from son-analyze and check if data can be processed.
- Son-emu:
  - Stop node monitoring
  - Stop VNF

5.8.4 Test Triggers

- The build is triggered after a successful build of any of these three other dependent projects:

son-emu, son-monitor, son-cli, son-analyze.

- Scheduled on a need basis

5.8.5 Actions after the test

- Send an email to *lead developers*

5.9 Platform Instantiation

5.9.1 Test Description

This test tries to deploy all major components of the service platform. The goal is to verify that all components are in a deployable state and can directly interact with each other after they have been deployed on a single system. Based on this integration test we can define the automated installation/instantiation procedure for service platform operators. Scripts used in this test might also be useful when the platform should be deployed in the qualification infrastructure.

5.9.2 Tests Requirements

5.9.2.1 Infrastructure Requirements

- Docker
- Integration server

5.9.2.2 Components Requirements

- pre-build Docker images for:
  - RabbitMQ
  - MongoDB
  - Postgres
  - GUI
  - BSS
  - Catalogues-repositories
  - PluginManager
  - Service Lifecycle Management
  - Infrastructure Abstraction
  - Gatekeeper API
  - Gatekeeper Packages
  - Gatekeeper Services
  - Gatekeeper Functions

5.9.2.3 Triggers

- Jenkins will trigger this test every night
- This test could be triggered by the developer

5.9.2.4 Post actions

- Send an email to *lead developers*

5.10 BSS - Gatekeeper

This chapter contains information about the integration test between the SONATA's Business Support System and the Gatekeeper.

5.10.1 Test Description

This test targets the integration between the next components of the Gatekeeper: BSS, Gatekeeper's API, Gatekeeper's Service Management and Gatekeeper's Request Management. Specifically it tests the functionalities needed to retrieve the list of available services that can be instantiated, deploy a new service in the Sonata service platform, and the information about the instantiation requests that were made to the system. There are three main test cases that are performed during this integration test:

- The first case tests the obtaining of the list of available services for instantiation and is shown in Figure 5.18.
- The second test case tests the execution of a new service instantiation order and is shown in Figure 5.19.

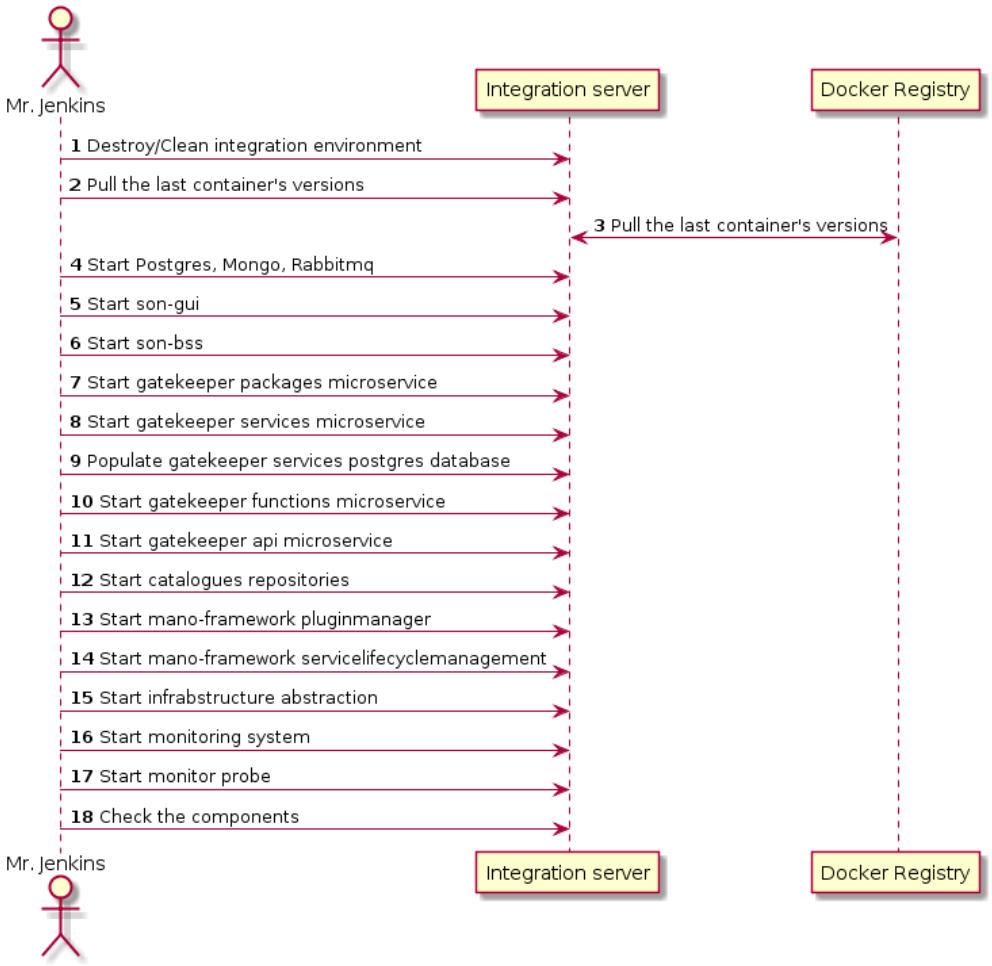

Figure 5.17: Platform Instantiation

- The third case checks the status of the service instantiation request (Figure 5.20).

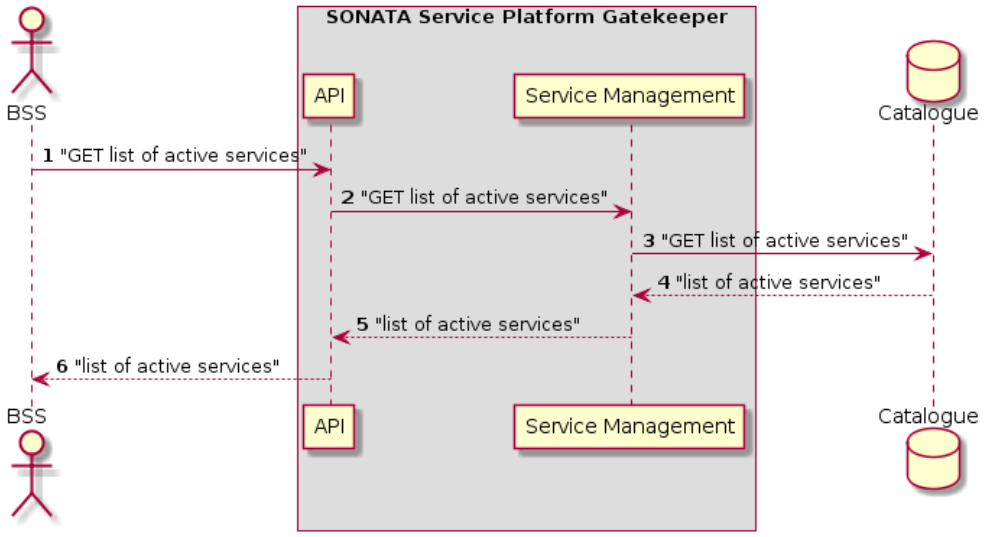

Figure 5.18: Services available for instantiation

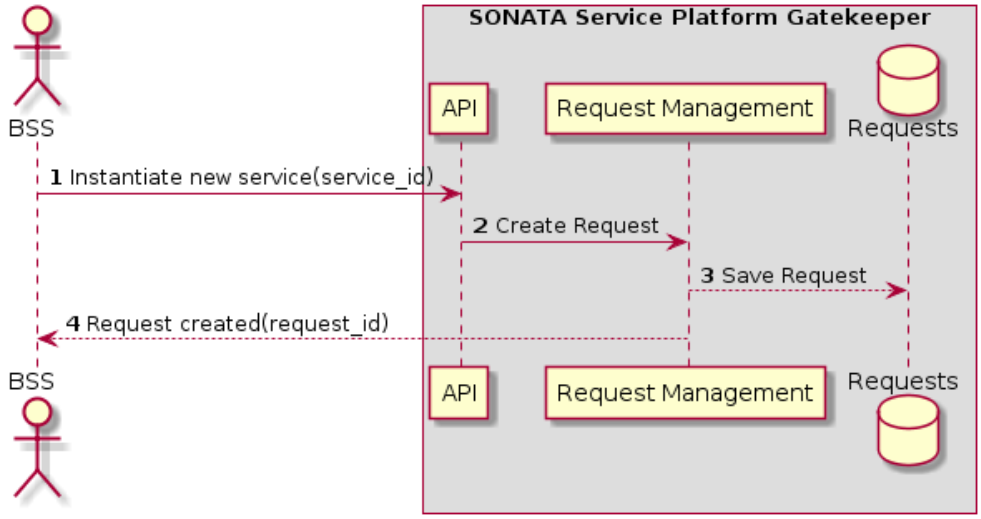

Figure 5.19: New Service Instantiation Request

5.10.2 Infrastructure Requirements

This test makes use of the following components:

- Gatekeeper:
  - BSS: son-bss
  - API: son-gkeeper/son-gtkapi
  - Service Management: son-gkeeper/son-gtksrv

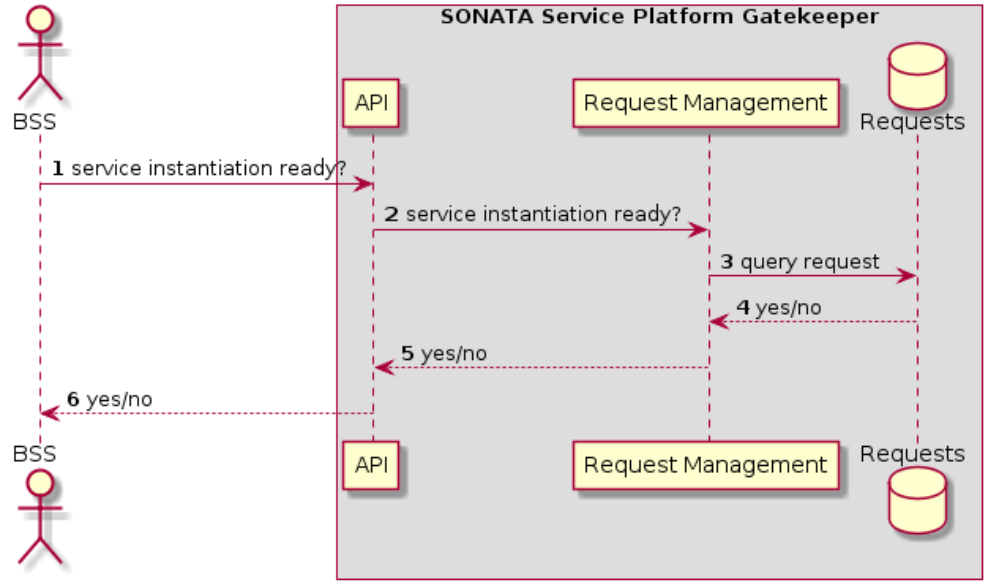

Figure 5.20: The BSS asks the Gatekeeper about a service instantiation

- Catalogues: son-catalogue-repos
  - Catalogues API
  - Mongo Database

5.10.3 Tests Requirements

This sections reflects the initial configuration needed to perform the test and the expected results to achieve.

5.10.3.1 Initial configuration of the environment

Installed and running BSS, Gatekeeper API, Gatekeeper Service Management and service platform catalogues.

From Gatekeeper

- API Method to retrieve the list of available services
- API Method to retrieve the stored requests
- API Method to create an instantiation request

From Catalogues

- API Method to retrieve the network service descriptors (NSD)
- Valid NSDs stored in mongo database

5.10.3.2 Expected results from modules

From BSS:

- List of available services for instantiation.
- New Service Instantiation launched.
- List of Instantiation Requests created by Gatekeeper.

From Gatekeeper:

- Creation of Service Instantiation Request

5.10.4 Test Triggers

- Jenkins (daily)
- Scheduled on need basis

5.10.5 Post Actions

- Send an email to lead developers

5.11 Service Lifecycle Management and Infrastructure Abstraction

5.11.1 Test Description

This test targets the integration of the MANO framework, in particular the Service Lifecycle Management (SLM) plugin, and Infrastructure Abstraction layer. The aim of the test is to trigger the SLM to instantiate a new service instance. In order to do so, a message is injected from the Jenkins Job into the Service Platform message bus. This triggers the SLM, which uses the message bus to publish a message to infrastructure.compute.list, in order to retrieve a list of the VIMs available to host the service, along with resource availability information, such as resource quota limits and utilisation. Using this information, a basic placement is executed. Finally, the SLM sends a deployment request message to the Infrastructure Abstraction, encapsulating the Network Service Descriptor (NSD), a list of the VNF Descriptors composing the service(VNFDs), the UUID of the VIM where the service should be deployed, as calculated by the placement, and other context information. The Infrastructure Abstraction layer uses the VIM Adaptor to translate the descriptors in the VIM specific language and connects to OpenStack Heat for the deployment. The VIM Adaptor waits for the service to be deployed or for an error message from the VIM. Once the deployment is completed, the VIM Adaptor collects the relevant instance information from OpenStack, forges the response message and sends it to the SLM. The interaction described above is depicted in Figure 5.22

5.11.2 Infrastructure Requirements

This test makes use of the following repositories:

- son-mano-framework
  - RabbitMQ

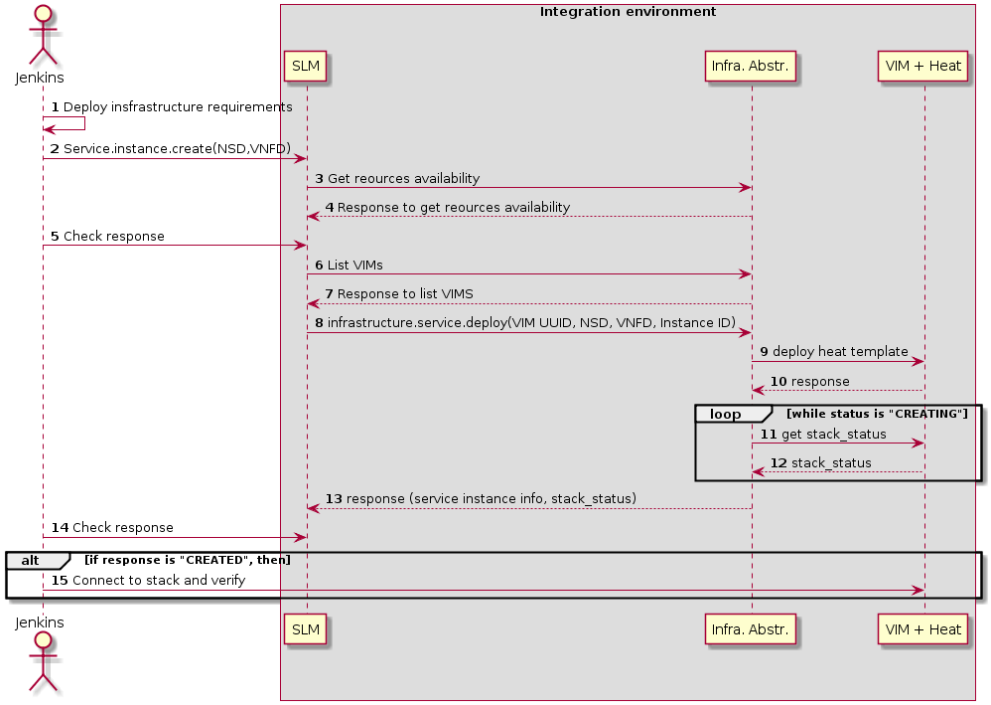

Figure 5.21: Message Sequence Chart of the interaction between the MANO framework and the Infrastructure Abstraction during service deployment

- son-sp-infrabstract
- OpenStack instance with Heat.

5.11.3 Tests Requirements

Initial configuration of the environment

- OpenStack VIM should be installed and running;
- The OpenStack VIM is registered to the Service platform;
- NSD & VNFD(s)

Expected results from modules:

- The Infrastructure Abstraction returns “done” or “failure” and the service instance information. If the process is successful, connect to VMs and verify.

API Method tested

- Infrastructure Abstraction:
  - List available VIMs with available resources
  - Deploy Service
- SLM:
  - Deploy a new service instance

5.11.4 Test Triggers

- Jenkins
- Scheduled on need basis

5.11.5 Actions after the test

- Send an email to *lead developers*
- Start other integration tests related to service deployment

5.12 Monitoring Server Integration

5.12.1 Test Description

This test procedure checks the connectivity between Monitoring server and Infrastructure components (SLM, Monitoring probes and Message Broker). The tests are based on a VNF with a monitoring rule, which is triggered in order to produce an alarm. In brief, the following tests check:

- The correct connectivity between the Service Platform monitoring probe and the Monitoring Server.
- The correct connectivity between a Virtual Machine monitoring probe and the Monitoring Server.
- The correct reconfiguration process triggered by the deployment of a new NS/VNF.
- The correct execution of the alerting mechanism triggered by a predefined rule and sent via Message Broker.

5.12.1.1 Infrastructure Requirements

- Gatekeeper API
- Service Lifecycle Manager
- Message Broker

5.12.1.2 Tests Requirements

- **From test**
  - A NSD which consists of one VNF with a test monitoring rule.
- **From Gatekeeper**
  - API Method to publish a Service Descriptor.
  - API Method to Service Deploy Request.
- **From Broker**
  - A “test” topic defined in order the test monitoring notification to be published.

5.12.1.3 Integration Tests Stories

In the first scenario the Monitoring Server collects monitoring data from a monitoring probe deployed in a VNF instance. The flow is shown in Figure 5.22.

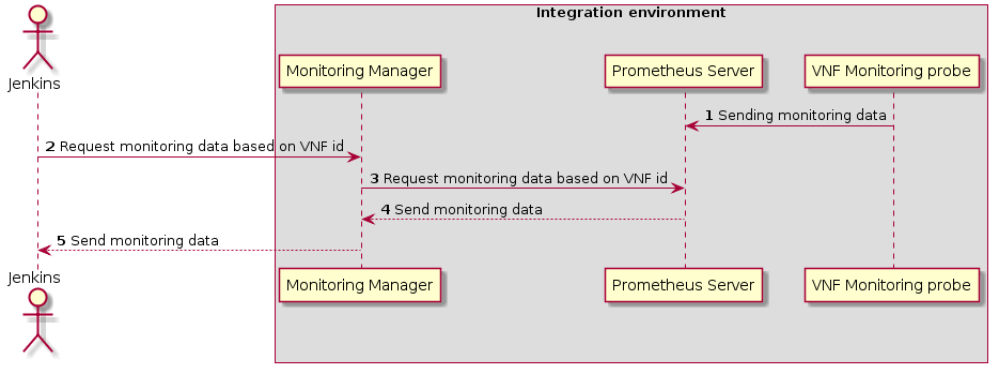

Figure 5.22: Monitoring Server VNF Monitoring Probe

In the second scenario the Monitoring Server collects monitoring data from the Service Platform monitoring probe. Figure 5.23 shows the flow for this scenario.

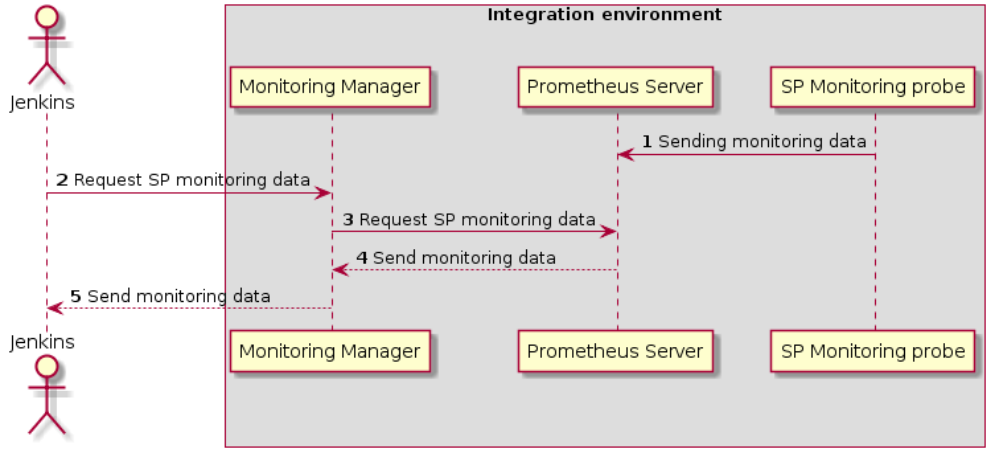

Figure 5.23: Monitoring Server SP Monitoring Probe

The third scenario is aimed at checking the Prometheus reconfiguration mechanism that are triggered when a new VNF is deployed. The associated flow is depicted in Figure 5.24.

Finally, the fourth scenario deals with the Prometheus Server sending alert notification to Message Broker. The associated flow is shown in Figure 5.25.

5.12.1.4 Test Triggers

- Jenkins (daily)
- Scheduled on need basis

5.12.1.5 Actions after the test

Send an email to *lead developers*.

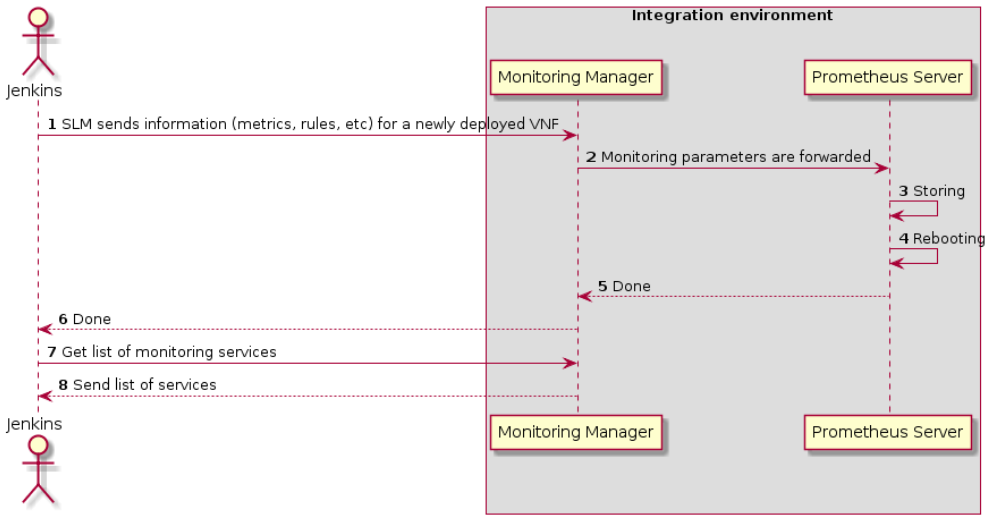

Figure 5.24: Monitoring Server Service Lifecycle Manager

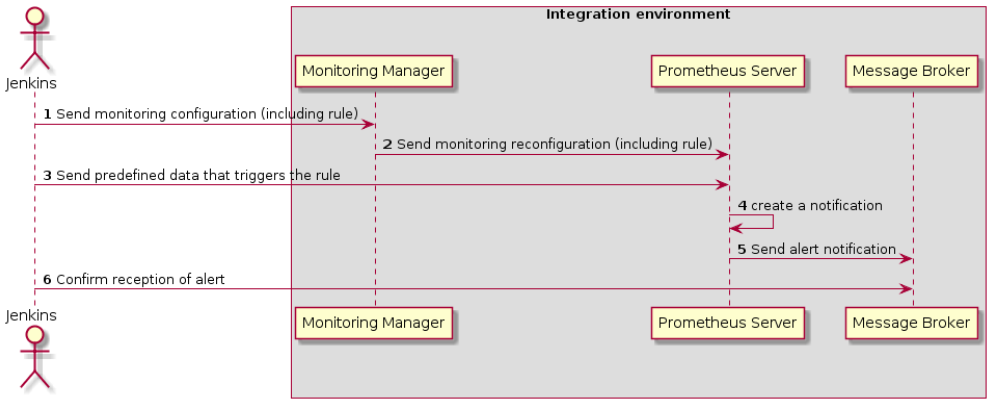

Figure 5.25: Monitoring Server Message Broker

5.13 Monitoring Server, GK API, GK GUI connectivity

This set of integration test procedures is based on the use of GruntJs[6], which is an Open Source tool for automating task in Javascript. Here we used Grunt to automate the execution of actions on the GUI. In fact, the test consists of several Grunt tasks which perform connectivity checks between **GK-GUI** implementation with **Gatekeeper** and **Monitoring server**, using the appropriate javascript code of the GUI.

5.13.1 Infrastructure Requirements

- Gatekeeper API
- Monitoring Server
- Gatekeeper GUI

5.13.2 Tests Requirements

- **From test**:
  - Grunt task runner environment
  - Grunt test tasks configuration
- **From Gatekeeper**:
  - API Method to retrieve available descriptors
  - API Method to Service Deploy Request
- **From Monitoring Server**:
  - API Method to retrieve monitoring data

5.13.3 Integration Test Stories

- **Test 1: Request Service Package list from GK** is depicted in Figure 5.26
  - This test executes a grunt task which performs the following actions:
    - Triggers GUI's javascript code which retrieves the available packages from GK.
    - Checks if this file is available to download.
- **Test 2: Request VNFs list from GK** is depicted in Figure 5.27
  - This test executes a grunt task which performs the following actions:
    - Triggers GUI's javascript code which retrieves VNF list from GK.
    - Checks if the appropriate table in GUI is filled up.
- **Test 3: Request Network Services list from GK** is depicted in Figure 5.28
  - This test executes a grunt task which performs the following actions:
    - Triggers GUI's javascript code which retrieves service list from GK.
    - Checks if the appropriate table in GUI is filled up.

- **Test 4: Display Service Platform monitoring data on GK-GUI** is depicted in Figure 5.29
  - This test executes a grunt task which performs the following actions:
    * Triggers GUI's javascript code which retrieves monitoring metrics from Monitoring server.
    * Checks if metrics values are valid.

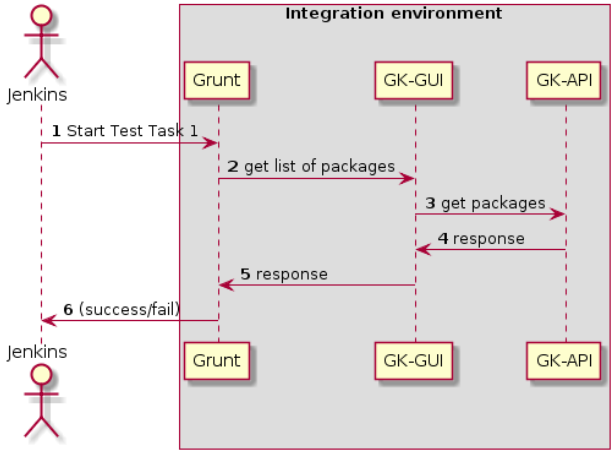

Figure 5.26: Request Service Package list from GK

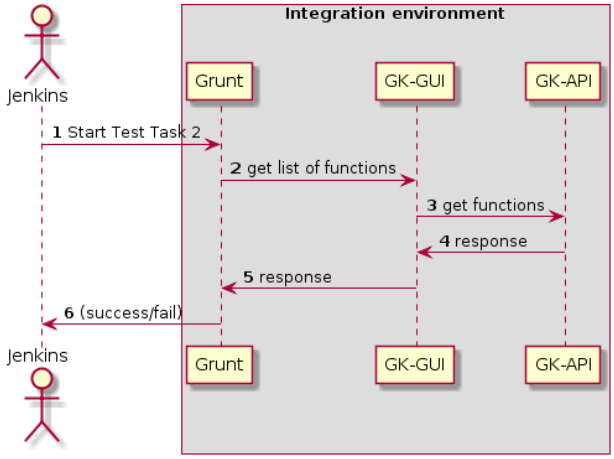

Figure 5.27: Request VNFs list from GK

5.13.4 Triggers

- Who will launch this test?.
  - Jenkins
- When?
  - After initialisation of the son-gui

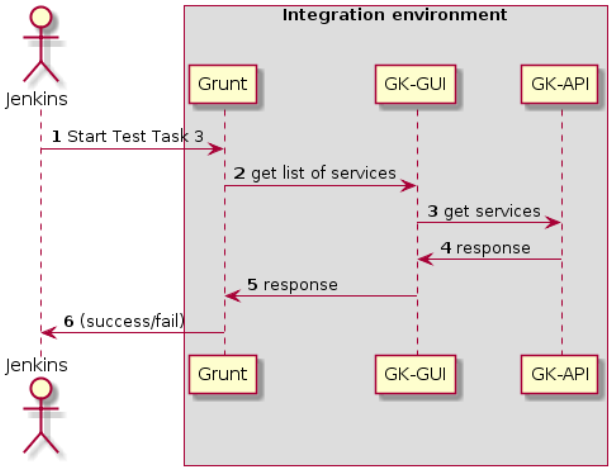

Figure 5.28: Request Network Services list from GK

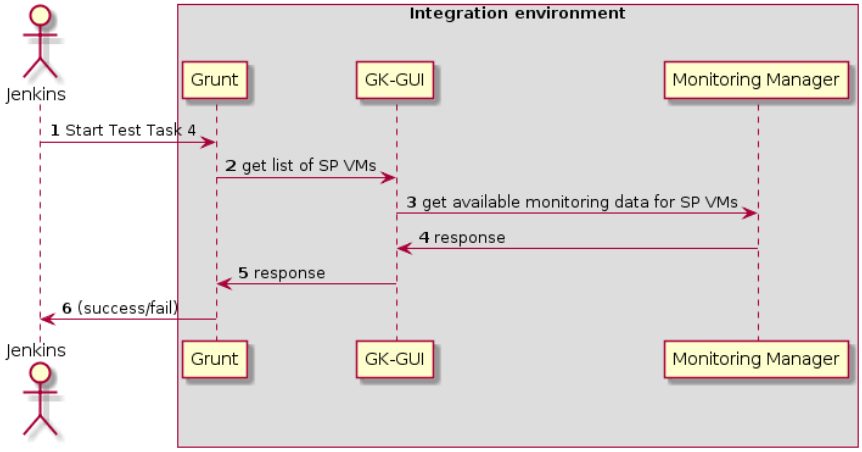

Figure 5.29: Display monitoring data

5.13.5 Post actions

In case of failure send emails to *lead developers*.

5.14 Gatekeeper - Son-monitor - Son-analyze

5.14.1 Test Description

This integration test verifies the interactions between the `gatekeeper` and the SDK tools: `son-monitor` and `son-analyze`. It checks the interoperability between those components and the expected results. The test starts by creating a new service using the `gatekeeper`'s REST api. The VNFs will start to generate metrics after their instantiations. The related metrics will end up in the `son-monitor-prometheus` database. The `son-monitor` and the `son-analyze` tools are used to query those metrics from the SP to the SDK.

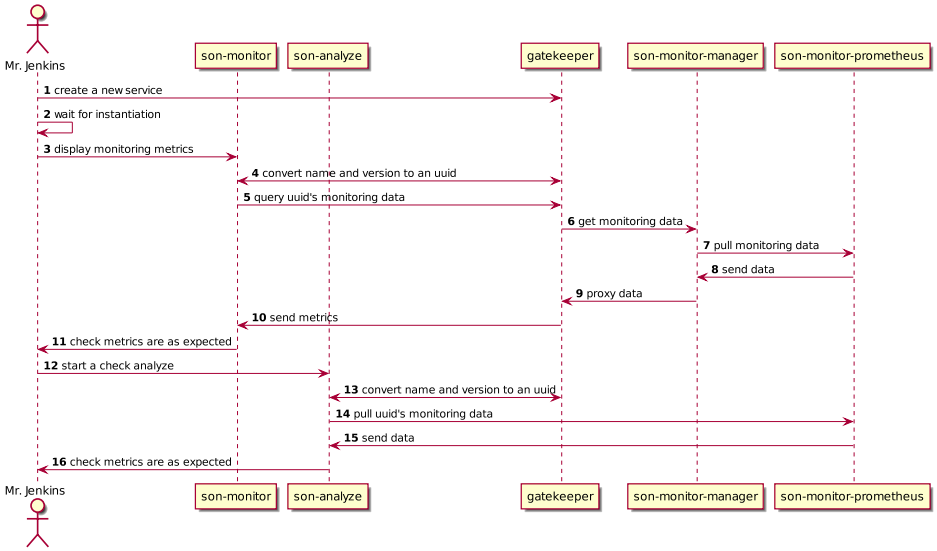

Figure 5.30: Integration test: GK, son-monitor, son-analyze

5.14.2 Infrastructure Requirements

The components used in this test are:

- Gatekeeper API (son-gkeeper/son-gtkapi)
- Son-monitor-manager
- Son-monitor-prometheus
- Son-monitor
- Son-analyze

5.14.3 Tests Requirements

5.14.3.1 Initial configuration of the environment

- **From gatekeeper**
  - A method to query service metrics
  - A method to find a component's uuid
- **From son-monitor-manager**
  - A method to query VNF metrics
- **From son-monitor-prometheus**
  - Valid metrics stored in the prometheus database
- **Son-monitor & Son-analyze**
  - A valid service containing one VNF
  - The gatekeeper endpoint

5.14.3.2 Expected results from modules:

- **From son-monitor**
  - Expected message code 200 (Retrive data from the Gatekeeper)
  - Expected raw data stored in the `son-monitor-prometheus` DB, code 200
- **From son-analyze**
  - Expected message code 200 (Retrieve data from the Gatekeeper)
  - Expected raw data stored in the `son-monitor-prometheus` DB, code 200

5.14.4 Triggers

- Who will launch this test?
  - Jenkins
- When?
  - Scheduled on need basis

5.14.5 Post actions

If the test fail, send email to *lead developers*.

6 Conclusions

In this deliverable we have described the first integrated, lab-based prototype of the SONATA environment.

The document illustrates the tools and methodologies used to drive the software development in a **DevOps** fashion, also giving an example of the strength of this approach, which is at the very core of the SONATA development cycle. This has been used to influence the development efforts in Work Package 4 and Work Package 3, giving the tools and methodologies to embrace this development philosophy. Moreover, the use of software management and continuous integration tools such as Jenkins and GitHub allowed the development team to integrate the outcome of the work in an **agile** way, continuously pushing improvements and integrating them progressively, throughout the first year of the project. This has allowed us to avoid the likely risk of needing a complex and long phase of integration at the end of the development process, too late to ensure the suitable level of **software quality**. This overall approach also has a strong impact on the release of the SONATA code-base as **open source**. The SONATA prototype makes use of other open source components, such as the RabbitMQ message system, the OpenStack IaaS, and also Jenkins, exploiting their maturity. We aim at creating an open source software product with the same level of quality, that can attract a large open source community around him, thanks to its native **agile** and **distributed** development life-cycle.

We also documented the **Prototype modules** that are part of this first integrated prototype, extending their characterisation with details on their interfaces and inter-working functionality.

Finally, we listed and detailed the **Integration tests**, which have been designed and developed to ensure the functionalities expected from the first prototype.

During the remaining lifetime of the project, we will focus on the qualification phase of the development, defining tools and tests for the SONATA prototype qualification. Moreover, we will extend the work on the SONATA development pipeline, to include a proper formalisation of the software **release** process, including release milestones and deadlines, documentation standards and improving the unit, module and integration tests set available, so to provide to the open-source community a robust code-base to work on.

A The SONATA prototype big picture

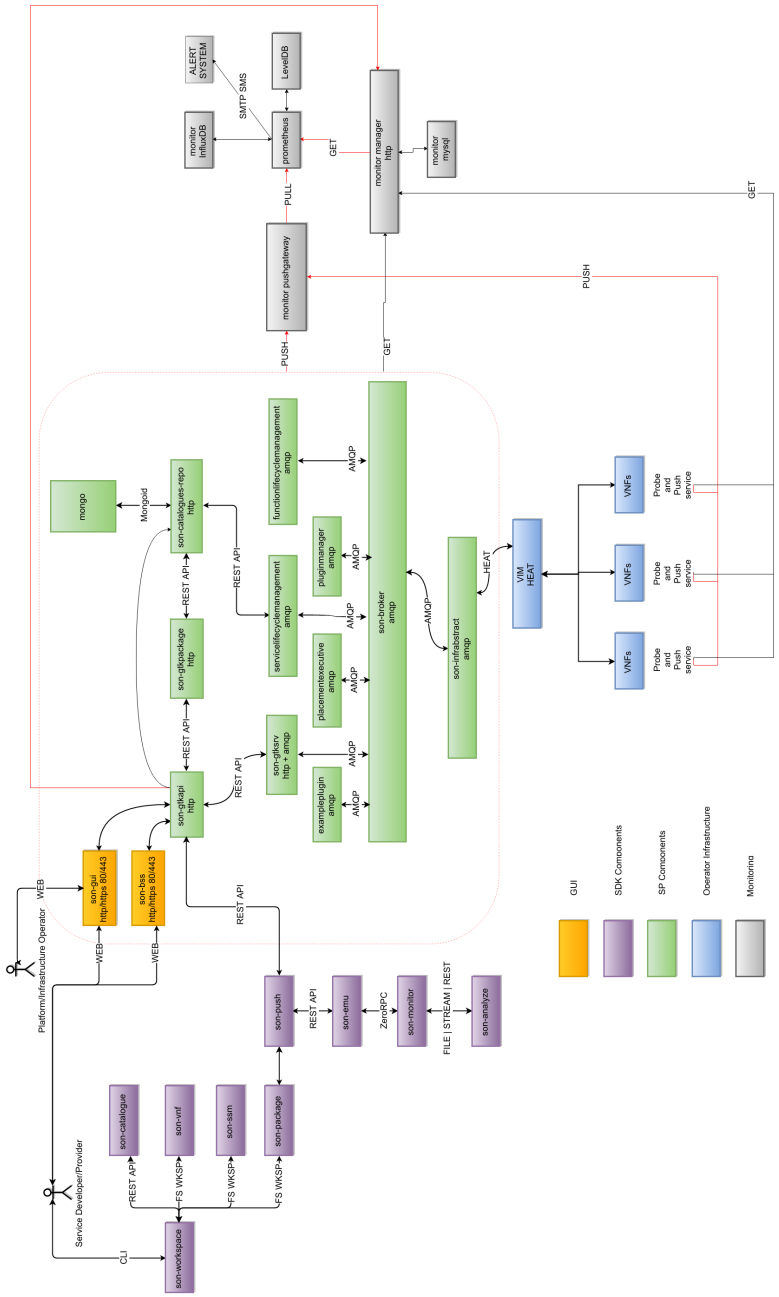

Figure A.1: The SONATA prototype big picture

B SONATA Prototype flows MSCs

In this appendix we collect the MSCs for the main interactions between actors and modules of the SONATA environment.

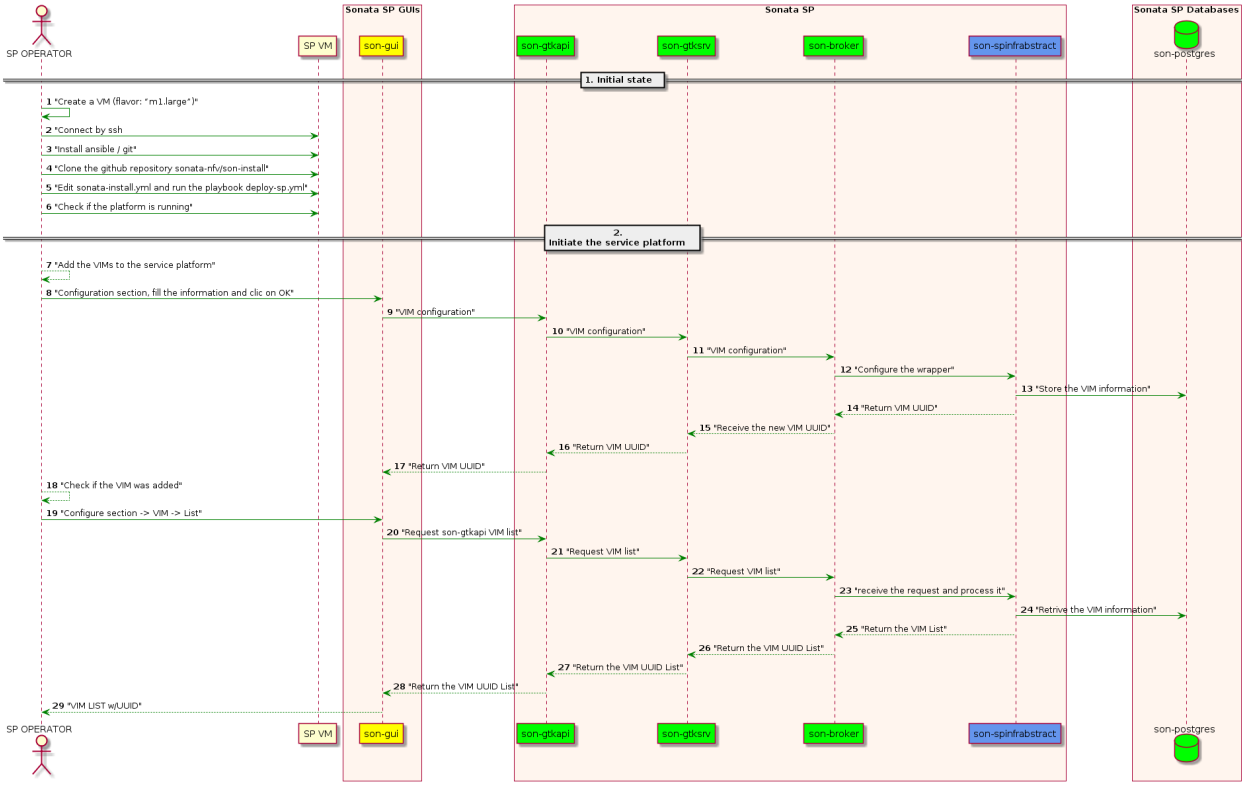

Figure B.1: Service Platform Instantiation

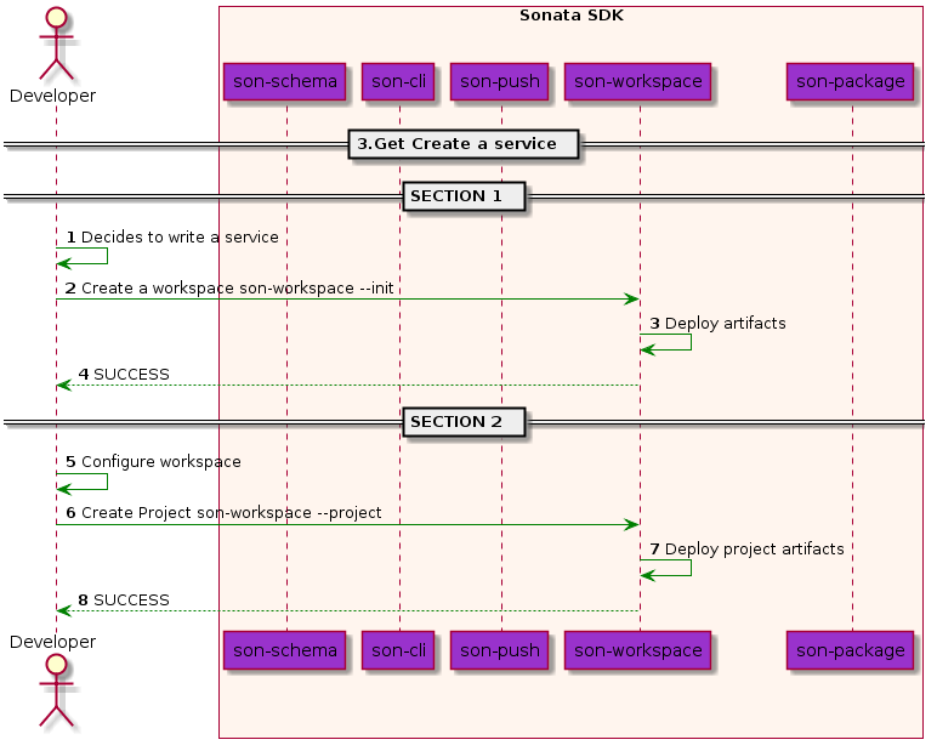

Figure B.2: Service Development A

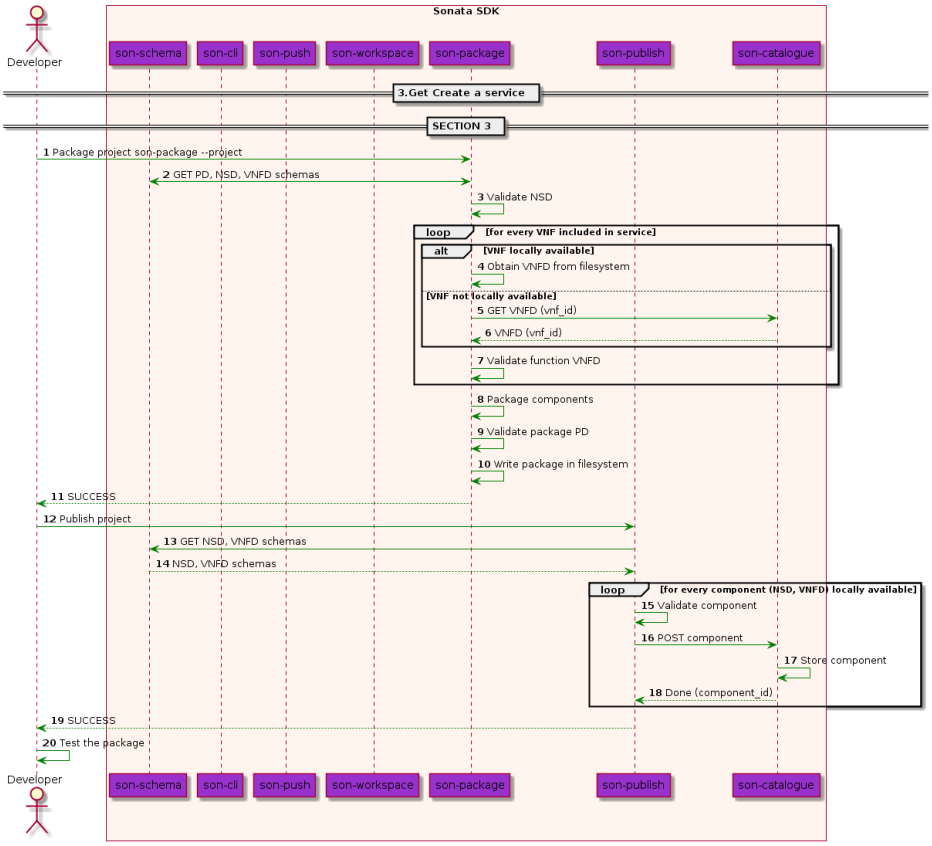

Figure B.3: Service Development B

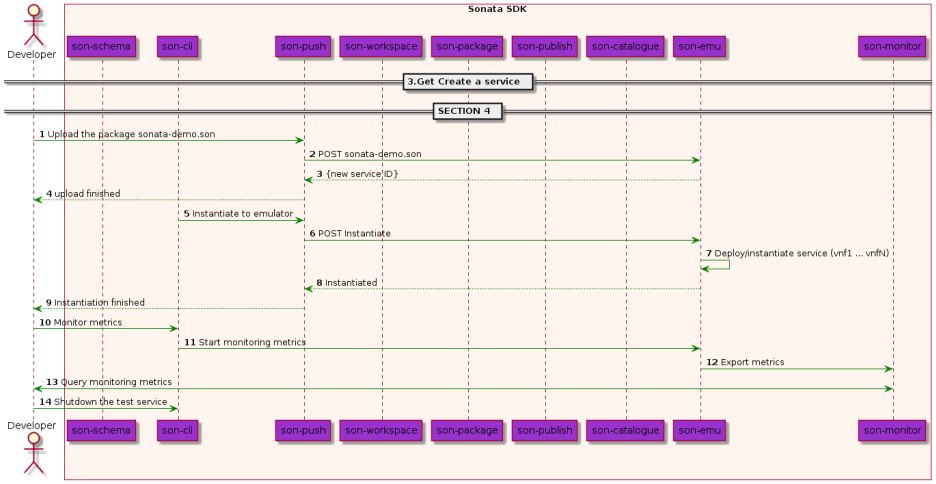

Figure B.4: Service Development C

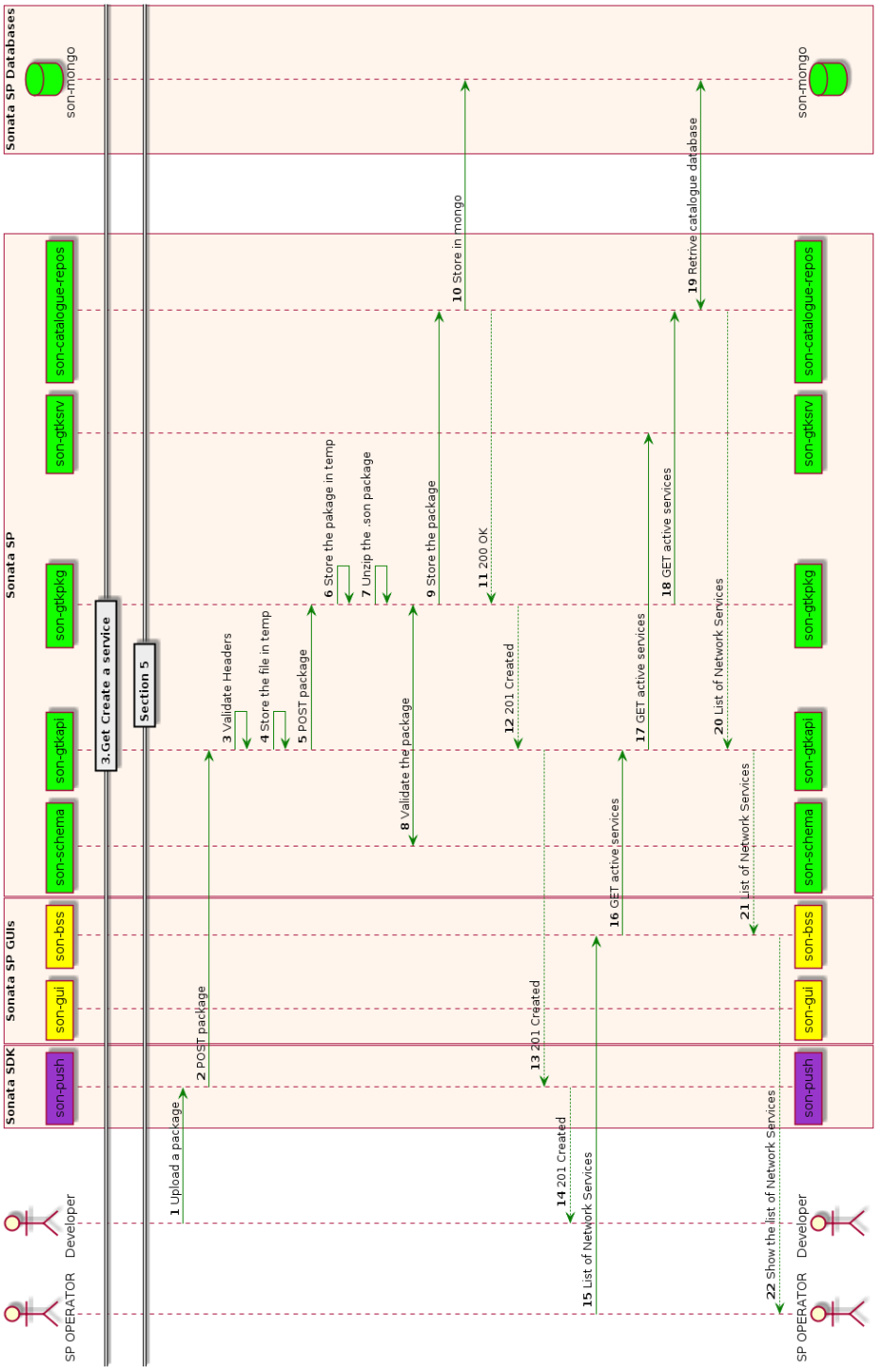

Figure B.5: Service on-boarding

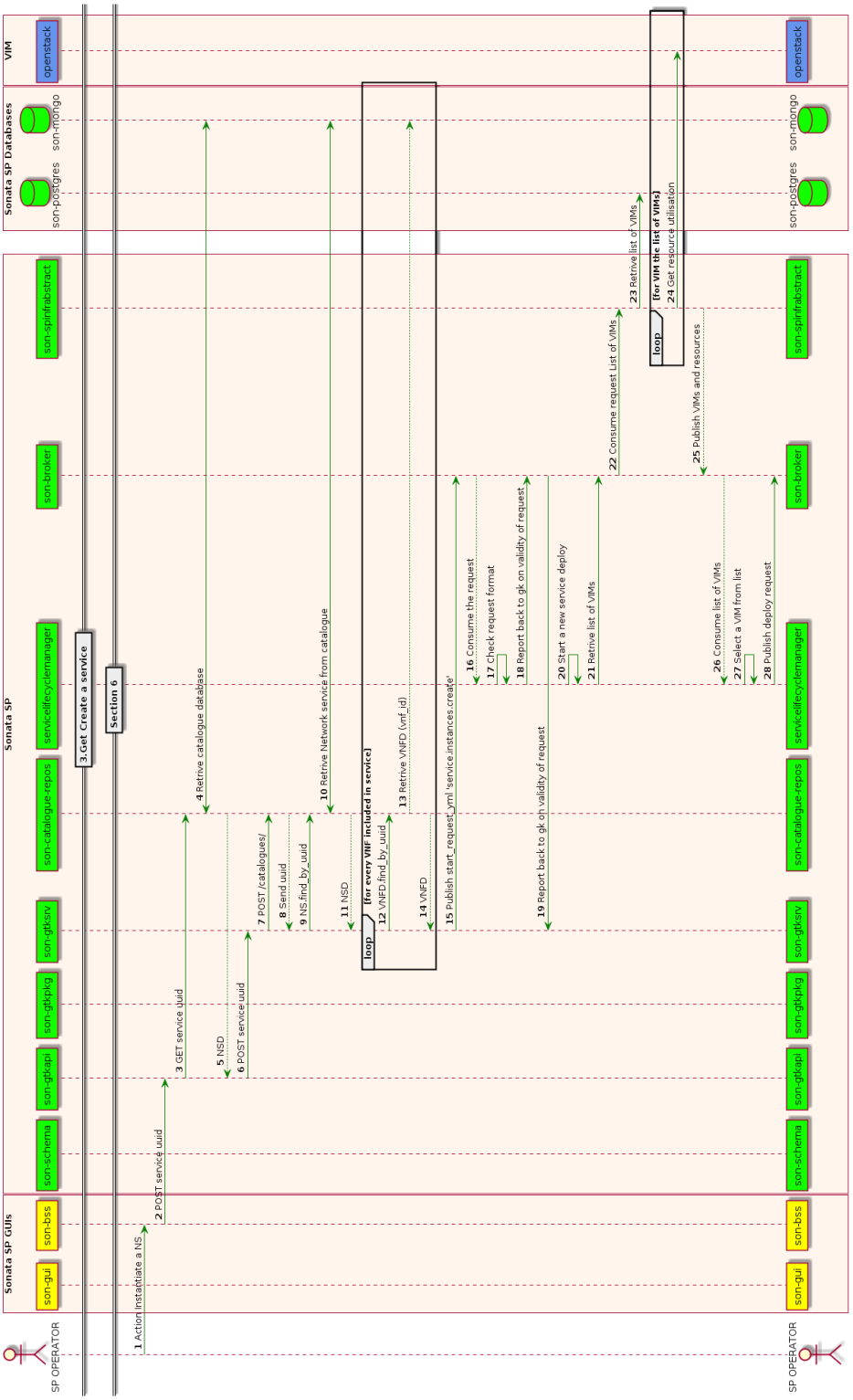

Figure B.6: Service instantiation A

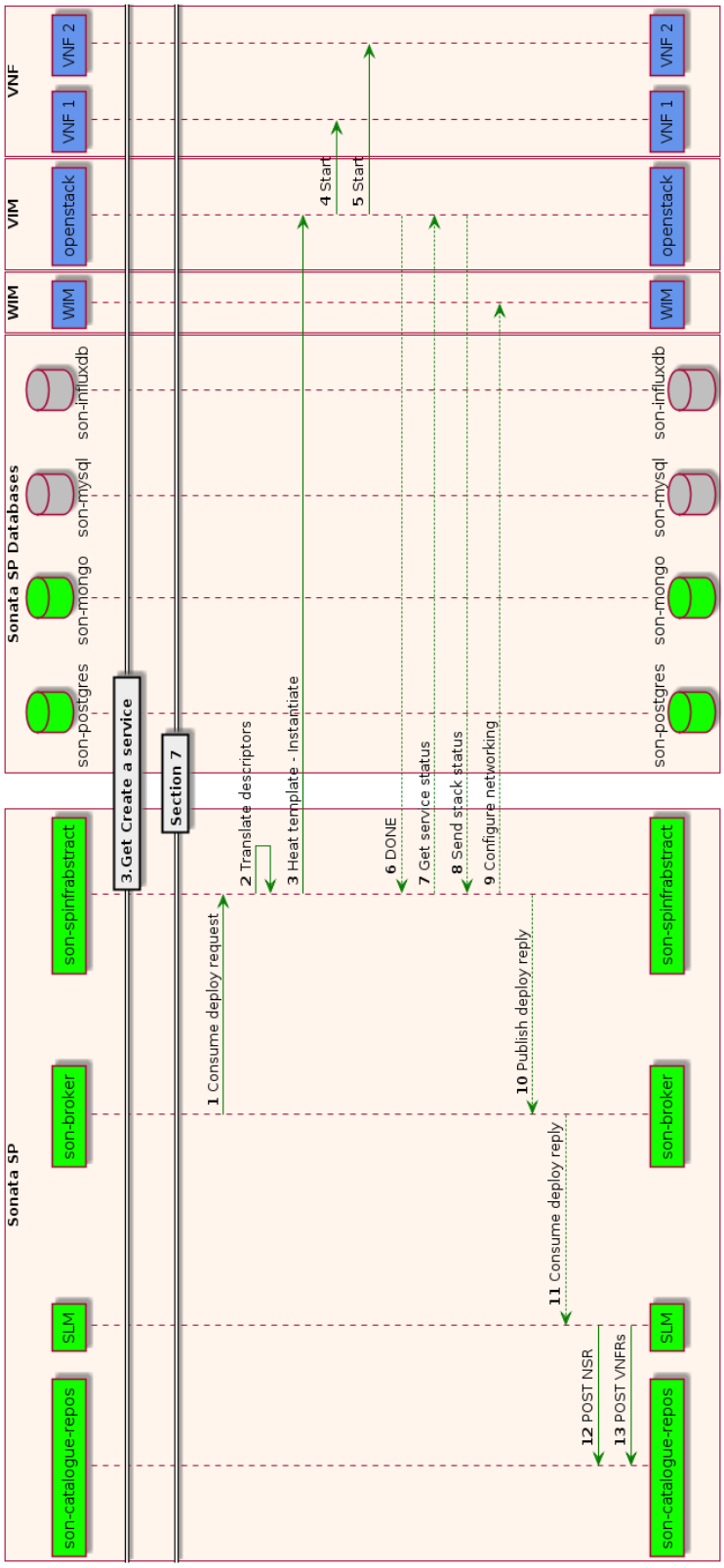

Figure B.7: Service instantiation B

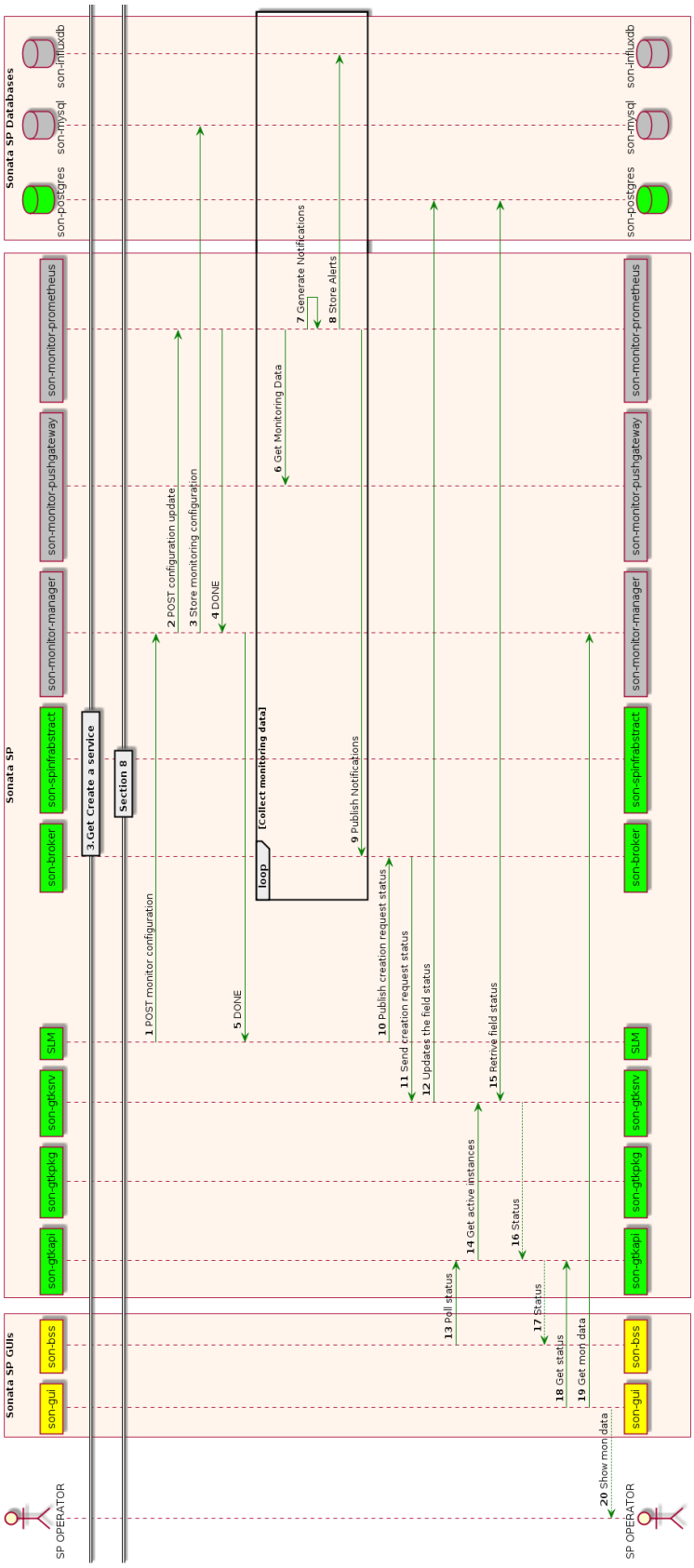

Figure B.8: Service instantiation C

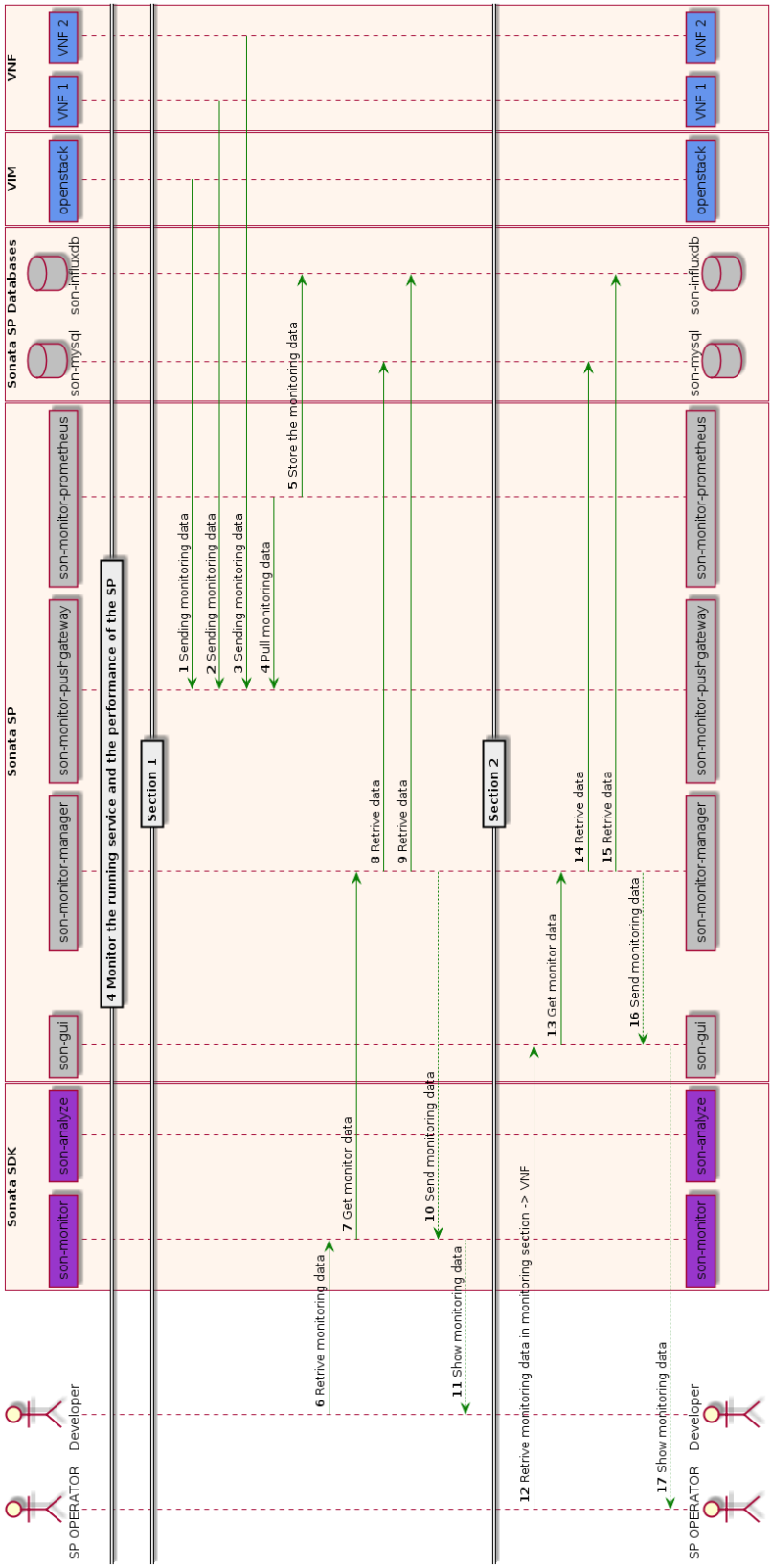

Figure B.9: Service Monitoring

C Glossary and acronyms

Table C.1: Acronyms table

API	Application Programming Interface
BSS	Business Support Systems
CAPEX	Capital Expenditure
CLI	Command Line Interface
CD	Continuous Delivery
CI	Continuous Integration
DevOps	Development and Operations
GK	Gatekeeper
GUI	Graphical User Interface
JSON	JavaScript Object Notation
MANO	Management and Operations
NSR	Network Service Record
NFV	Network Function Virtualisation
NSD	Network Service Descriptor
NSR	Network Service Record
OPEX	OPerating Expenditure
OSS	Operations Support System
PD	Package Descriptor
PoP	Points of presence
REST	Representational State Transfer
SDN	Software Defined Network
SDK	Service Development Kit
SP	Service Platform
SLM	Service Lifecycle Manager
SSM	Service Specific Manager Plugin
VNFD	VNF Descriptor
VNFR	VNF Record
VNF	Virtual Network Function
VIM	Virtual Infrastructure Manager
WIM	WAN Infrastructure Managers
YAML	A human-readable data serialization language

D Bibliography

[1] 5G-ppp. View on 5g architecture. Website, 2016. Online at https://5g-ppp.eu/wp-content/uploads/2014/02/5G-PPP-5G-Architecture-WP-For-public-consultation.pdf.

[2] Bozhidar Batsov and RuboCop contributors. Rubocop. Website, 2016. Online at http://rubocop.readthedocs.io.

[3] Codehaus. Cobertura maven plugin. Website, June 2016. Online at http://www.mojohaus.org/cobertura-maven-plugin/.

[4] Ansible Community. Ansible. Website, June 2016. Online at https://www.ansible.com/.

[5] Github Community. Github. Website, 2016. Online at https://help.github.com.

[6] GruntJs Community. Gruntjs. Website, June 2016. Online at (http://gruntjs.com).

[7] Junit Community. Junit - about. Website, June 2016. Online at http://junit.org/junit4/.

[8] RSpec Community. Rspec. Website, June 2016. Online at http://rspec.info/documentation/.

[9] The OpenStack Community. Openstack heat project. Website, May 2016. Online at https://wiki.openstack.org/wiki/Heat/.

[10] SONATA Consortium. Sonata deliverable 2.2: Architecture design.

[11] SONATA consortium. D5.1 continuous integration and testing approach. Website, December 2015.

[12] SONATA consortium. D3.1: Basic sdk prototype. Website, May 2016.

[13] SONATA consortium. D4.1: Orchestrator prototype. Website, May 2016.

[14] SONATA consortium. D6.1: Definition of the pilots, infrastructure setup and maintenance report. Website, June 2016.

[15] Python Software Foundation. Unit testing framework. Website, June 2016. Online at https://docs.python.org/2.7/library/unittest.html.

[16] The Apache Software Foundation. Apache maven checkstyle plugin. Website, June 2016. Online at https://maven.apache.org/plugins/maven-checkstyle-plugin/.

[17] F. Galiegue, K. Zyp, and G. Court. Json schema: core definitions and terminology - draft 4. Website, March 2013. Online at http://json-schema.org/.

[18] Nick Coghlan Guido van Rossum, Barry Warsaw. Pep 8 – style guide for python code. Website, June 2016. Online at https://www.python.org/dev/peps/pep-0008/.

[19] Docker Inc. Docker: An open platform for distributed applications, August 2013, howpublished=Website. Online at http://www.docker.com/.

[20] Graylog inc. Graylog documentation. Website, 2016. Online at http://docs.graylog.org.

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

[22] MojoHaus. Findbugs maven plugin. Website, June 2016. Online at http://gleclaire.github.io/findbugs-maven-plugin/.

[23] Manuel Peuster. Containernet. Website, 2016. Online at https://github.com/mpeuster/containernet.

[24] Selenium Project. Selenium documentation. Website, June 2016. Online at http://www.seleniumhq.org/docs/.

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

[26] Pivotal Software. Rabbitmq. Website. Online at https://www.rabbitmq.com/.