D3.1 Basic SDK Prototype

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	D3.1 Basic SDK Prototype
Workpackage	WP3 Service Programmability and Toolset
Due Date	April, 2016
Submission Date	May, 2016
Version	1.0
Status	Final
Editor	Wouter Tavernier (iMinds)
Contributors	Wouter Tavernier, Steven Van Rossem (iMinds), Tiago Batista (UBI), Michael Bredel (NEC), Geoffroy Chollon (TCS), Muhammad Shuaib Siddiqui (i2CAT), Manuel Peuster, Sevil Dräxler (UPB)
Reviewer(s)	Tiago Batista (UBI)

Keywords:

SDK, DevOps, Software Development Kit

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

Executive Summary:

The next generation of mobile networks and telecommunication standards is referred as 5G. It offers speeds which are higher than current 4G mobile networks and has a higher spectral efficiency. However, the true potential of 5G technology is not only to be found in higher access speeds, but also in the broad spectrum of services it could bring to the customer. These services should be able to optimally use the available network and cloud infrastructure, beyond the pure access network. This will enable increased **flexibility, rapid deployment** and **elasticity** depending on the context in which services are deployed. Building further on the novel paradigms of Network Function Virtualization (NFV) and Software-Defined Networking (SDN), SONATA aims to contribute to this trend by providing an appropriate Service Platform and corresponding Software Development Kit. This deliverable documents the architecture, design, programming model and usage of the SONATA Software Development Kit (SDK) after the first phase of the project. The SDK is the principle set of development tools to design and test services to be deployed on the SONATA Service Platform as documented in D4.1 [11].

The SDK is built up as a **set of small independent tools** which can be combined in one or **multiple workows** to develop a SONATA service. This design enables **agile development**, involving quick and iterative cycles of development, with the possibility of rapidly transitioning between development and operations, which is one of the key characteristics of the SONATA approach. The software design of the SDK tries to re-use existing workflows and concepts in software development such as the use of workspaces, project folders and packaging techniques. This will enable new developers to get acquainted with the SONATA development philosophy and tool set in a short time. Where possible, existing software or libraries have been re-used, increasing the robustness and overall feature set of the SONATA SDK.

The SONATA **programming model** focuses on the common concepts of **service chains and service graphs** comprised by individual network functions. Each of these components are defined by their corresponding **descriptors** following a particular data model. The used data models are built upon ongoing standards or outcomes of research projects, extending them when appropriate. A SONATA service is characterized by a network service descriptor (NSD), a set of virtual network function descriptors (VNFD), and a package descriptor for the overall service. Each of these descriptors define artifacts containing data and information such as images, files and/or configuration parameters or scripts. The descriptors follow a YAML or JSON schema language format. A set of validation tools have been made in order to check if given descriptors are compliant with the SONATA format. These can be used across the project in different modules and tools of both the SDK and the Service Platform (SP).

In this phase of the project, the development tools have been focused on the support of a basic process which can be accommodated by both Service Platform and SDK. This workflow is documented as the phase 1 storyboard of the project, and ensures consistency in the global design of SP and SDK components. The SONATA SDK itself consist of a set of tools. Manuals for each of these tools including concrete usage examples have been added to the appendix of this document. The main workflow and functionality of these tools is as follows. `son-workspace` and `son-project` enable a developer to prepare a filesystem structure for a **SONATA workspace** and SONATA service under development. This involves the creation of templates for file descriptors, as well as the creation of appropriate subfolders for individual project components such as images. The developer can further work on this project by editing descriptors or referencing appropriate descriptors for functions or function interconnection from the SONATA catalogue component. Once the project is ready, it can be packaged using the `son-package` tool, which compiles all necessary information into a single file which might be uploaded to a Service Platform using the `son-push` tool. Because

a full Service Platform might require too many (or external) resources or setup effort, the SDK provides an **emulator** based on Docker technology, enabling to locally deploy a SONATA service package on the developer's computer. Initial versions of the `son-monitor` tool enable **monitoring** of basic characteristics of the deployed service such as cpu and/or bandwidth usage, or to enable a simple profile of the service or involved network functions. The resulting profile might be used in a next phase to fine-tune service scaling logic on a range of physical platforms. The `son-analyze` tool enables to **visualize** monitoring data and perform statistical **analysis** of this.

The SDK functionality for the next phase of the SONATA project will build further on the existing tools, by improving their robustness, security as well as their initial feature set. In addition, the SDK will be extended with a set of new development tools, focusing on the development of individual Virtual Network Functions, as well as on the development of Service Specific Managers and Function Specific Managers. The latter will enable SONATA services to define their custom scaling and/or placement logic.

Contents

List of Figures

List of Tables

1 Introduction

This deliverable documents the development that has been done for the basic SONATA Software Development Kit (SDK), after 10 months since the start of the project. Because this is one of the first documents providing further details on the SONATA programming process, below we briefly recapitulate the SONATA architecture from D2.2 [8].

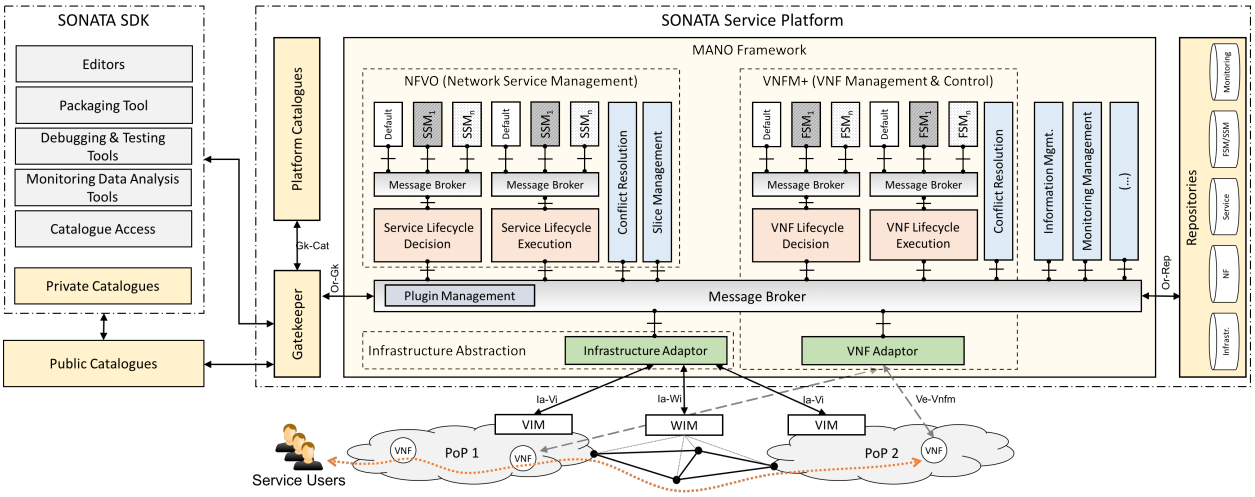

Figure 1.1: Detailed architecture of SONATA service platform

The **SONATA architecture** is roughly divided into two parts: the **Software Development Kit** which is depicted on the left side of Figure 1.1, and the **Service platform** on the right side of the figure. The Service Platform involves: i) a set management and orchestration (MANO) framework for actually steering the deployment of virtualized services interacting with different Virtual Infrastructure Managers (VIM) such as OpenStack, ii) catalogues storing service and network function descriptions and associated attributes, ii) a set of repositories providing storage of the instances of services and Network Functions, and at last the iv) the gatekeeper which is the single point of interaction with the outside world including the SONATA SDK. The Software Development Kit provides tools supporting the service and function development process.

1.1 An SDK for NFV-based 5G services

As indicated in the 5G PPP vision paper [3], **5G** will not only be an evolution in (faster) mobile broadband networks. It will bring **new unique network and service capabilities**. It will integrate networking, computing and storage resources into one programmable and unified infrastructure. This will enable for a more flexible, dynamic and optimal usage of all distributed resources as well as the convergence of fixed, mobile and broadcast services. Through the design and development of a Service Platform and an appropriate Software Development Kit building upon the principles of SDN and NFV, SONATA will enable faster development and thus innovation in networked services.

This deliverable focuses on the SONATA Software Development Kit. Following the Wikipedia definition, a **software development kit** (SDK or "devkit") is typically a set of **software development tools that allow the creation of applications for a certain hardware platform or software framework**. It may be something as simple as the implementation of one or more application programming interfaces (APIs) in the form of some libraries to interface to a particular programming language or to include sophisticated hardware that can communicate with a particular embedded system.

The SONATA SDK focuses on the **programmability of services which combine network, cloud components and configuration**. This requires an adequate programming model and the set of the actual tools.

As documented in D2.1 [7] and D2.2 [8], the SONATA **programming model** builds further on the concept of service function chaining and composition. The central programming concept here is the **Network Function Forwarding Graph** as defined in ETSI NFV and further extended in projects such as UNIFY [37] and T-NOVA [33] in the form of service templates, or adding service scaling options. SONATA builds further on these concepts by extending and/or redefining appropriate schema and descriptors for services and functions (documented in Section 3.1 and Appendix A).

The basic version of the SONATA SDK, which is the focus of this deliverable, provides SDK components and tools for setting up a SONATA programming environment, a SONATA platform **emulator** and an initial set of tools for **monitoring and analysing services** developed using the SDK. The current SDK interacts with the initial version of the Service Platform documented in D4.1 [11] via the gatekeeper of the Service Platform. The developed SDK tools provide one of the first integrated set of tools which enable to deploy services combining both network and cloud resources in a controlled and uniform way. This version of the SDK prototype focuses on the basic functionality in the sense of a Minimum Viable Product. Additional features related to scaling, VNF development and editors will be added in later phases of the project.

1.2 Structure of the deliverable

The rest of the documents is structured as follows: Section 2 introduces the different components of the SDK, and presents the overall interactions and workflow in between them. Next, the main design aspects of each of the individual SDK components is described in Section 3. This involves characterization of adequate schema and descriptors, tools for setting up the development work- and project space, catalogues, a packaging tool, an emulator, a monitoring and analysis tool. Section 4 describes the common storyboard which captures the main functionality, features and interactions between the SONATA SDK and the Service Platform. Section 5 concludes the main document, capturing the most important aspects of the SDK design, and sketches some future plans on enhancing the basic SONATA Software Development Kit for phase 2 of the project. In order to increase the accessibility and usability of the developed tools, manuals and instructions of each of the components have been included in the appendices of the deliverable.

2 Global SDK design and component interactions

The ultimate goal of the SONATA SDK is to **assist the developer in designing services** which can be pushed and deployed to the SONATA Service Platform. As indicated earlier in this document, the SDK is built as a **set of individual small tools** (similar to the different git tools or unix shell tools) which in the first phase are mainly intended to be run from the command line. SONATA services are built using a sequence or combination of execution of these tools. The goal of this section, is to give a global, conceptual introduction of these tools, before we go into the detailed workflow and interactions between these tools.

The main product in SONATA is the **service**. In its basic form, a SONATA service is conceived as **combination of Network Functions** (NFs) interconnected through a forwarding graph. In addition to NFs, a service can also consist of Service and Function Specific Managers (SSM and FSMs). As indicated in D2.2 [8], NFs provide the actual execution functionality of the service, while SSMs provide meta-logic for scaling and/or placing the service. The details of each of these components are captured in **descriptors** (see Section 3.1). Descriptors might refer to information available in a public or private catalogue (`son-catalogue`). The transferable unit of a service is a **service package**. The latter is built through a set of environment and packaging tools. The SONATA development environment requires a workspace to be set up (`son-workspace`) in which different SONATA projects can be developed. A SONATA project is the construction environment (`son-project`) for a particular SONATA service within a workspace. Particular SDK tools will support the development of individual VNFs and SSMs (`son-vnf` and `son-ssm`, planned for phase 2. When a project is ready to be deployed, it will be packaged into a service package (`son-package`). The latter can be (and ultimately will be) deployed on an actual Service Platform. However, in order to enable quick iterations in development and testing without requiring the setup of, or interaction with a real service platform, an SP emulator (`son-emu`) is part of the SDK and enables to locally (i.e., on the computer of the developer) deploy a service on an environment which is highly similar to the SP. Within the emulation environment (but also on the SP), monitoring tools (`son-monitor`) can be used to generate data related to functional or performance-related behaviour. Subsequently, the resulting data might be analysed using analysis tools (`son-analyze`) in order to improve or update the resulting descriptors and ultimately the service package.

2.1 SDK interfaces

In order to support multiple workflows, the SDK is mainly built of individual tools which interact with the files on the project and workspace filesystem of the SONATA developer. Most of these tools currently have a CLI interface which manipulates or acts upon this. The Figure 2.1 illustrates the interfaces between the main SDK components (as well as the Gatekeeper of the SP). Every interface corresponds to a reference point with a particular label and will be documented further in the document:

- The development on SONATA services occurs on a **project folder** (PRJ) in a **workspace**

(WKSP) on the **filesystem**, build by `son-workspace`. The structure of this system is documented in Section 3.2.

- The **interface between the SDK and the SP** is mediated via the **Gatekeeper**, following the **REST API** documented in D4.1 [11], where the SDK side is implemented by `son-push`.
- The interface between the `son-monitor` component and the (emulated) platform `son-emu` is using a **ZeroRPC interface** [1]. This is documented in Section 3.4.
- The interface between the `son-monitor` component and the `son-analyze` tools is either the **REST API of the Prometheus platform**, or **file-based**.
- The interface to `son-catalogue` is a **REST API** which is documented in Section 3.3.

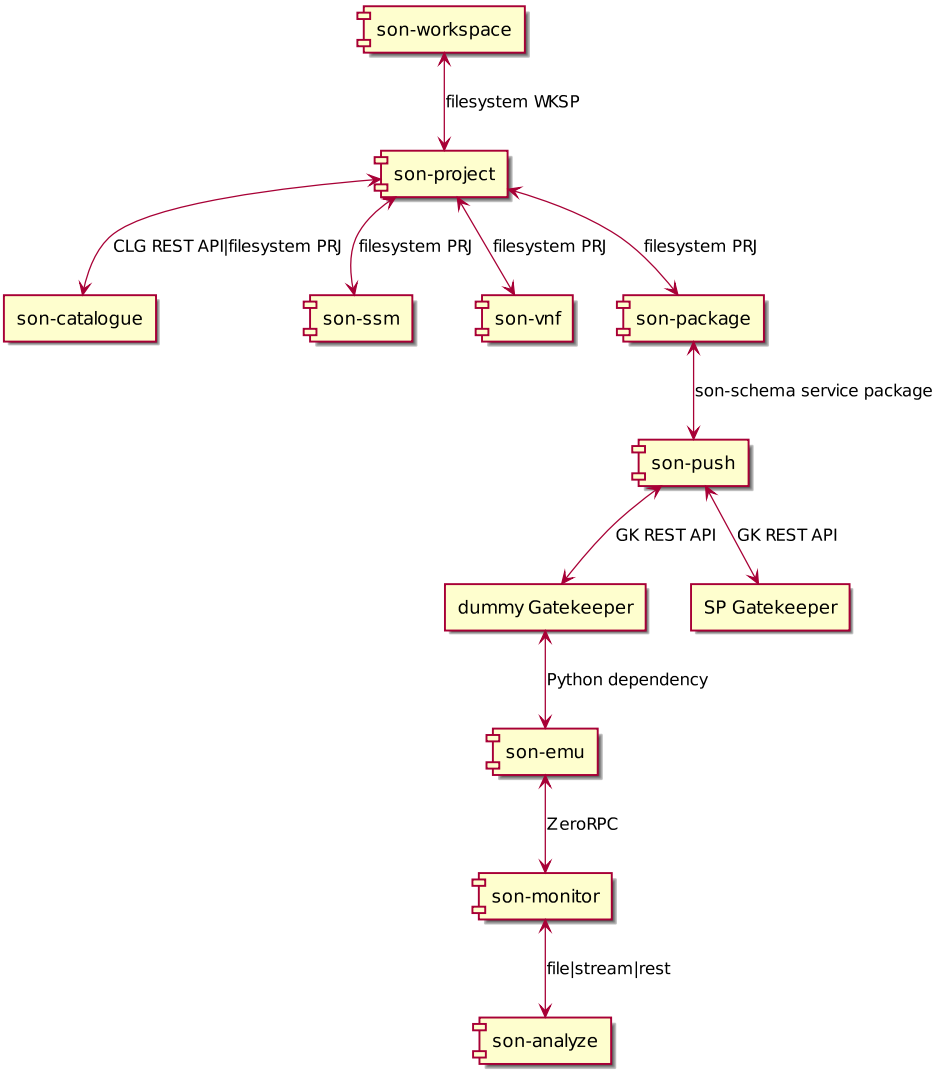

Figure 2.1: Interfaces between the main SDK components

Note that the SDK tools `son-editor`, `son-ssm` and `son-vnf` are not part of the SONATA SDK for phase 1. However, for completeness, they have already been included in the figure.

2.2 SDK workflow

The above section has conceptually introduced the individual SDK tools, as well as the way in which they operate. The developed SDK functionality needs to be understood in the context of the overall SONATA phase 1 storyboard. The storyboard provides the process-related reference framework of the first phase of the SONATA project, ensuring that the workflow and functionality of both the Service Platform and the Software Development Kit are well-aligned and that a basic set of functionality can be supported by both of them. The entire storyboard is documented in Section 4. The goal of this section is to clarify the **sequence of the canonical SONATA development process**, although the component-based design enables other workflows. This refines the SDK workflow documented in Deliverable D2.2 [8]. The main process is depicted in Figure 2.2, which follows the following steps:

1. In order to be able to deploy a developed service, either a **Service Platform**, or a Service Platform emulator (`son-emu` as indicated in the figure) must be **initialized** by the operator of the platform. Here it is started before the service is deployed. However, this could be done just before step 7 as well.

2. Next, a SONATA development **workspace must be created** via `son-workspace` before a project can be created (this will be done automatically if this step is omitted) by the developer in the SDK.

3. In order to prepare the development workspace for the development of an individual SONATA project, a **project space is created** via `son-project` by the developer in the SDK.

4. A service can be built using different pre-existing VNFs, SSMs and other **artifacts which might be fetched from the catalogue** (`son-catalogue`) in order to include them in the project space from the SDK.

5. **Project development** using a range of manual actions and/or tools (which might be provided in the second phase of the project, such as `son-vnf` and `son-ssm`) by the developer in the SDK.

6. The necessary descriptor files of the service are **bundled into a package** which can be deployed using `son-package` by the developer in the SDK.

7. The resulting package is **uploaded to the gatekeeper** of the (emulated) Service Platform via the `son-push` tool by the developer in the SDK. As a result, an identifier is received from this interaction.

8. Once uploaded, the received identifier can be used by the developer to actually trigger the Service Platform to **deploy** it, again using the `son-push` tool from the SDK.

9. In order to monitor particular functional or performance-related service, VNF or SSM parameters, a **deployed service can be monitored** using `son-monitor` from the SDK.

10. The SDK translates the requested monitoring actions into **platform specific instructions** (e.g., using the ZeroRPC interface of the emulator).

11. The Service Platform actually starts the requested **monitoring actions** on the infrastructure and/or on the VNFs.

12. **Monitoring data is generated** within the infrastructure or from the VNFs and is sent back to the Service Platform

13. The Service Platform returns the resulting **monitoring data to the SDK** in the form of a stream, file or other format.

14. The SDK tool `son-analyze` can be used to **analyse and visualize** the resulting data, helping the developer to improve the service design and re-start the process at an earlier step of this process.

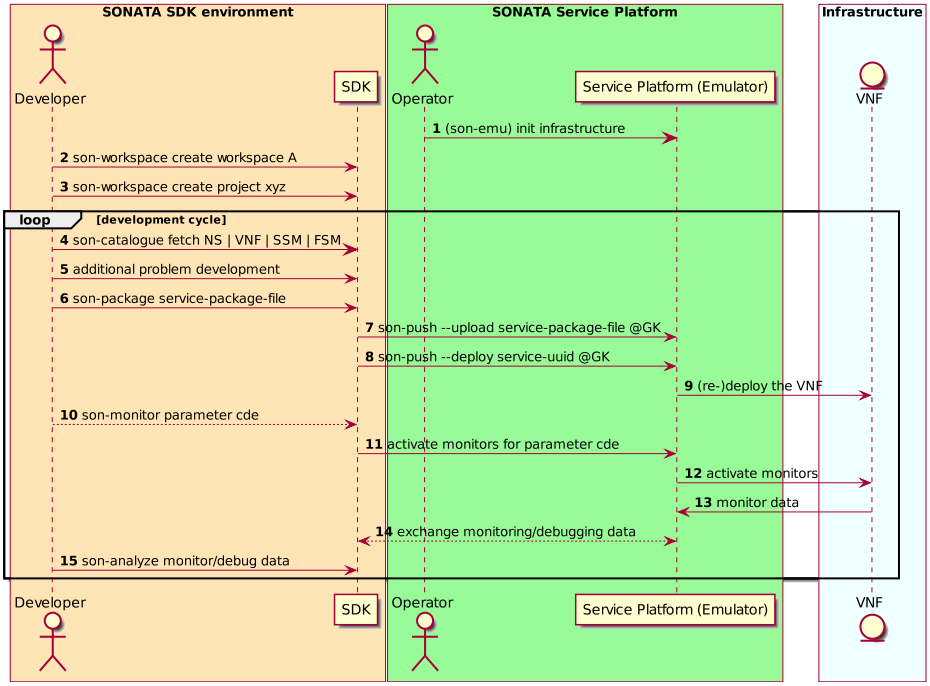

Figure 2.2: SDK development workflow

The following sections will go in deeper detail on each of the described tools and how the above steps are actually conceived in the design of the tools.

3 SDK component design

This section will document the SONATA SDK design for phase 1 in deeper detail. Each of the individual components will be documented in larger detail with respect to its architecture, the design choices that have been made, the general workflow in using the component, what tests have been implemented and how it compares to related work.

3.1 son-schema

In the NVF-world, descriptors are used to describe entities, such as Virtual Network Functions and Network Service in terms of deployment and operational behaviour requirements. Descriptors contain connectivity, interfaces, and KPI requirements that can be used by an NVF-MANO system to create virtual links between VNF Components to form a VNF and between VNFs to form a network service, respectively. Moreover, VNF Descriptors are used by the VNFM to perform lifecycle management. Similarly, Network Service Descriptors are used by the NFVO to orchestrate network services.

The descriptors files are created by a network service architect or a VNF developer. They capture the information require at each level of the orchestration process. For example, the NSD identifies that a firewall needs to be instantiated to create a given network service. The VNFD associated with the firewall captures the internals of the VNFs within the VNF.

3.1.1 Related Work and State of the Art

Today's cloud services, like OpenStack, use cloud service descriptors and template files to describe even complex cloud services, which may contain several components, such as load balancers, web and application servers, and databases. The Heat Orchestration Template (HOT) [6] is descriptor that can be parameterized at instantiation time. It is meant to replace the Heat CloudFormation-compatible format (CFN), which was based on the AWS CloudFormation template, as the native format supported by the Heat component of OpenStack. HOT is still under development and exists in various versions; each OpenStack release has its own specific version supporting the current Heat features. Within the HOT template, service developers can specify the service inputs, like the VM instance type, that have to be provided at instantiation time, the actual service specification, i.e. the used resource, the VM image, and the service outputs, like the public IP address of deployed instance. The HOT template focuses on OpenStack Heat only but can easily be used to describe VNFs that should be deployed on OpenStack.

To overcome the limitation of VIM-specific descriptors, the Organization for the Advancement of Structured Information Standards (OASIS) created the Topology and Orchestration Specification for Cloud Applications (TOSCA) language [38] to describe a topology of cloud based web services, their components, relationships in a uniform and vendor-independent way. Moreover, the TOSCA standard includes specifications to describe processes that create or modify web services. With respect to OpenStack there exists a TOSCA-to-HOT translator that is developed and maintained by the OpenStack community.

While most of the template languages can be used to describe web service and compute requirements, and are therefore candidates to describe VNFs, they usually lack a detailed description of

network functionality and requirements, and therefore fail to describe complete network service. To overcome these requirements, ETIS is in the process of specifying descriptors that cover virtual network functions as well as network services [21]. This standardization process, however, is still in an early stage and the descriptor cannot be considered as final. So far we consider the ETSI standard as a minimum set of VNFD and NSD elements considered necessary to on-board network function and services.

3.1.2 VNF and NS Descriptors

The SONATA descriptors, also referred to as SONATA programming model [9], are loosely based on the ETSI descriptors. However, since the ETSI information model and descriptors are still in draft stage, we adapted them to our needs whenever needed. In addition, we adopted ideas from the OpenStack Heat Orchestration Template and the TOSCA Simple Profile for Network Functions Virtualization [39].

High-level VNFD informational element categories include:

- Virtual Deployment Units
- Connection Points
- Virtual Links
- Lifecycle Events
- Deployment Flavours
- Monitoring Parameters

High-level NSD informational element categories include:

- Network Functions
- Connection Points
- Virtual Links
- Forwarding Graphs
- Lifecycle Events

A more detailed description of the various categories can be found in the Appendix A.

3.1.3 Package Descriptors

In order to on-board artifacts, such as descriptors, images, and files, and make them available to the NFV service platform, we foresee packages similar to the TOSCA Cloud Service ARchive (CSAR) [18]. These packages are ZIP files that may contain one or more files or directories that may have been compressed. The internal directory structure is described by a manifest file, also known as package descriptor, that contains various meta-data including name, vendor, version number, and the constituting files.

3.1.4 Identifying Artifacts

Within the NVF-space, we find various artifacts which contain different data and information:

- **Images**, such as Virtual Machine images and Container images
- **Files**, such as configuration files and executable scripts
- **Descriptors**, such as VNFDs and NSDs
- **Packages**

These artifacts are related to each other and might be referenced in the various descriptors. The VNFD, for example, might contain references to images, where the NSD holds references to VNFDs. The package again can have references to all other artifacts. Evidently, this imposes the necessity to identify an artifact in a unique and distinct way. As an additional requirement, we want the identifiers to be human readable.

A well-known method to create human readable identifies is to use a primary key that is composed of multiple parameters. Often these parameters are the name of the artifact, a vendor or maintainer, and - to support operation processes, like updates - a version number. Within the SONATA descriptors, we exactly use a tuple, i.e. vendor, name, and version, to identify an artifact. This also is in line with the current ETSI specification.

3.1.5 JSON Schemata for Descriptors

To support the creation and handling of various descriptors, we created schemata that act as ground truth for the whole SONATA project. Thus, every SONATA sub-projects obeys the schema description. The schemata are based on the JSON schema specification [17] and can be used to verify formal correctness of descriptors, as a basis for databases, and as a blueprint for code implementations.

3.1.6 Schema Validator

In order to validate descriptor files against schemata, `son-validate` was created as a validation library. The library, which is written in Java, allows to validate YAML or JSON files against a given JSON schema, that again can be provided in either JSON or YAML. To this end, it uses existing Java libraries, such as SnakeYAML [35] and Everit JSON Schema Validator [14], to perform the validation. Using the Google Guice Dependency Injection Framework [40], the library can be easily integrated with other tools, such as a holistic NFV editor, CLI tools, and Web services.

son-validate-cli

The SONATA validator tool, `son-validate-cli`, is a command line tool that uses the son-validate library to validate YAML or JSON files against a given schema. By default, `son-validate-cli` checks against the standard JSON-Schema Draft-04. If the YAML/JSON file to check has provides a $schema-key, the file is checked against that schema. You may also provide your own schema, which overrides the previous actions. The provided schema file must meet the JSON-Schema Draft-04 standard.

son-validate-web

The SONATA validator web service, `son-validate-web`, leverages the `son-validate` library to validate YAML or JSON files against given schemata by using a REST interface. Similar to the `son-validate-cli` tool it takes YAML or JSON files as input and returns the validation result. The validation web service can be used by various components of the SONATA system, such as the Gatekeeper and the Catalogues.

3.2 son-workspace and son-package

3.2.1 Architecture and design choices

The layout for the project and workspace on the developer disk was specified on D2.2 [8], and it is repeated here on Figure 3.1. D2.2 then specifies a set of tools that are meant to interact with the developer workspace and project, aiding in the development and deployment of new services and their components.

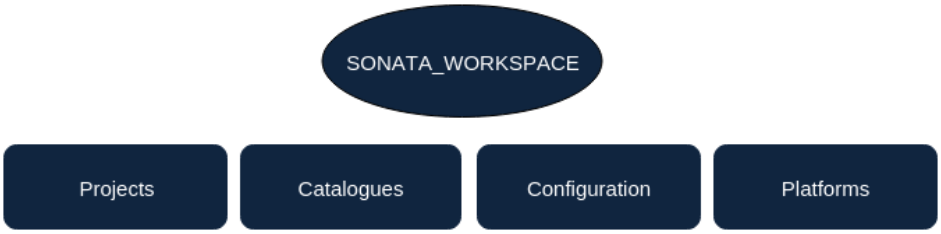

Figure 3.1: SONATA Workspace as initially proposed

For the first prototype, the development focused on a subset of the tools that manipulate the workspace. Those that have a wide impact were prioritized over those that manipulate sub components. For the first development iteration, the tools will allow the creation of a new workspace, a new project, and the packaging of a project. For subsequent iterations, the tools that aid in the creation and manipulation of some of the project sub components will be developed, and the existing tools will be improved.

While developing the initial tools, a shortcoming of the layout proposed for D2.2 [8] became apparent. Parts of the workspace for a project need to be shared among developers cooperating on a project, and other parts should be kept private to each team member. This lead to the separation of the initially proposed workspace into two directories. The first directory keeps the name workspace, and contains all the personal information about the developer. On this directory, information such as the identification of the developer and the credentials for the service platforms or other private information will be stored and managed by the developer. The second directory is the project directory, that contains all the shared information usually associated with a project. The project directory can and should be kept under source control and will be shared among the team of developers that are creating and maintaining a service. This split does not imply a change in the way the information is arranged logically, as the creation of a package still combines information from the workspace and information from the project to create the final package, but significantly eases the information sharing that should take place among developers of the same project while discouraging the use of shared credentials, something that severely harms the capacity to audit the actions of a developer in the event of a security breach.

3.2.2 Workflow

The current usage of the tools follows a simple workflow that start with the creation of a workspace. In order to do that, the developer should invoke the `son-workspace` tool with the `--init` option. This creates a new workspace. The developer should then check the created files and edit them where needed, as the workspace contains personal information. The next step is the creation of a project. In order to do so the developer should invoke the `son-workspace` tool with the `--project` option. The new project can be committed to source control and shared with other developers. A basic workspace workflow is shown in Figure 3.2.

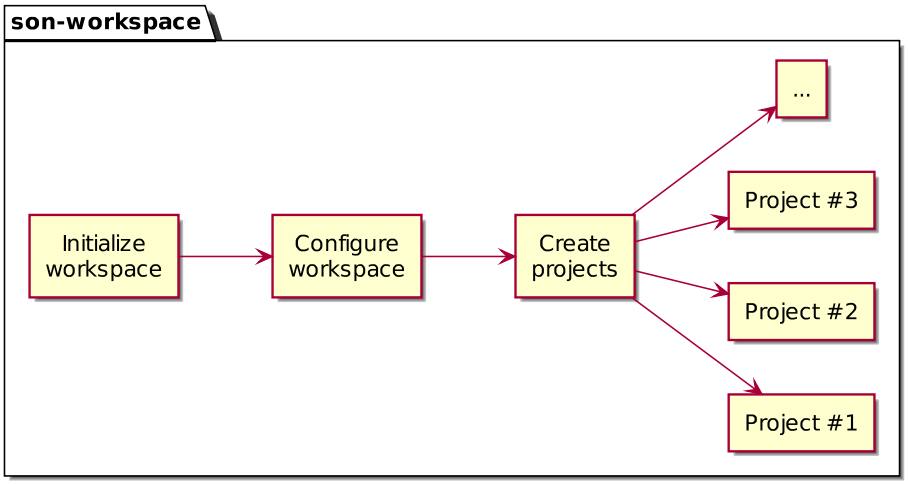

Figure 3.2: son-workspace tool

After a development cycle, the developer then has to package the project for submission to the gatekeeper. The packaging is done using the `son-package` tool with the `--project` argument. Figure 3.3 illustrates the packaging process of a project. During the packaging process, the project must be verified regarding external dependencies, which may not be present in the developer file system. External dependencies, such as VNF descriptors, should be retrieved from the private catalogues to which the developer has access. The service and function descriptors of the project are syntactically validated against the schema templates, available at the son-schema repository. When all the required project components are ready to be packaged, a syntax validation is performed and the package file is created on the file system.

This toolset should eventually be distributed as a wheel over the standard PyPI and installed via pip, but at the moment the binaries are only available on the project's Continuous Integration server. When installing, the developer should be able to just invoke a command like `sudo pip install son-cli` after installing the system requirements, but for the moment the user has to manually download the binary and invoke pip with the appropriate file.

3.2.2.1 Example workflow

When starting the development for the first time, the user has to initialize the workspace. This step only needs to be performed once as the resulting workspace can be shared among several projects. To initialize the workspace at the default location (`$HOME/.son-workspace`) the developer runs the following command:

```
son-workspace --init
```

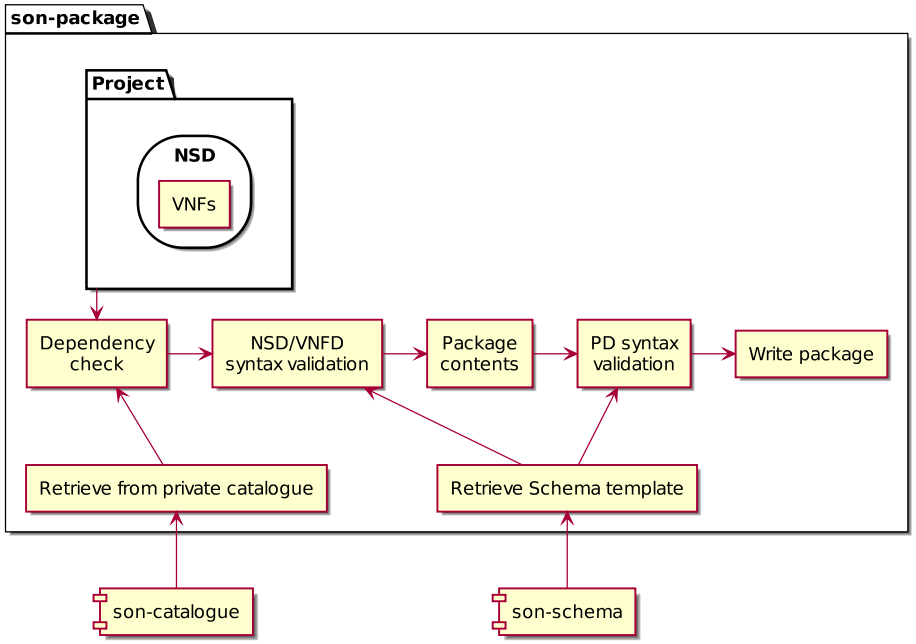

Figure 3.3: son-package tool

If a different workspace location is required, the argument `--workspace` should be used. The user should then edit the created files, namely the workspace configuration, to make sure that all the information is correct.

The following step is the configuration of the workspace. The schema template server must be configured correctly to ensure a successful validation of the project descriptors. Without schema templates, the packaging of the project will fail. The private catalogue server addresses must also be configured correctly, particularly if the project has third-party dependencies. For example, the developer can configure the following parameters:

```
catalogue_servers:
- id: son-cat
  publish: 'yes'
  url: http://son-catalogue.sonata-nfv.com:4011
schemas_local_master: ~/.son-schema
schemas_remote_master: https://raw.githubusercontent.com/sonata-nfv/son-schema/master/
```

The next step involves the creation of a project. The developer invokes the following command to create a project, located at `$HOME/projects/proj1`

```
son-workspace --project ~/projects/proj1
```

At this stage the development of the project occurs. Service and function descriptors may be developed or referenced as external dependencies. After the first cycle of development, the project must be sent to the Gatekeeper. The project is then packaged into a single container file using the `son-package` tool. The packaging process will resolve external dependencies and validate all descriptors, including the descriptor of the package itself. After a correct packaging, `son-push` is used to send the package to the Gatekeeper through its API.

3.2.3 Tests

The son-workspace and son-package tools implement several unit tests to ensure their functionalities. The unit tests are implemented using `unittest` python framework and `unittest.mock` to provide full isolation to the module that is being tested.

3.2.3.1 Unit tests for son-workspace

- **Initialization of workspace** - Tests the creation of a workspace. It ensures that the init parameters are well configured, such as the root directory of the workspace, workspace name, etc.
- **Directory structure** - Determines if the workspace directory structure contains all the required components and are all located within the workspace root directory.
- **Instantiate workspace from configuration** - External modules, such as son-package and son-publish use a given workspace to operate. This workspace object is instantiated based on the workspace configuration file. This test ensures that a workspace is instantiated from the configuration file with the correct properties. More importantly, it ensures that a workspace is **not** instantiated when an invalid or incomplete configuration is provided.
- **Create configuration** - Assures the correct writing of the workspace configuration.

3.2.3.2 Unit tests for son-package

- **Package generation** - Tests that the package file is created with the correct name at the correct location
- **Project configuration** - Ensures that an incomplete or incorrect project configuration is detected and the packaging process is aborted
- **Load remote schema** - Checks if schema templates are being loaded from the correct URLs and ensures they are returned with the appropriate format.
- **Load local schema** - For the case when the son-schema repository is not available, a local copy of schemas is kept at the local file system. This test ensures that schemas are loaded from the correct paths with the appropriate format.

3.2.4 Technologies used

The `son-workspace` and `son-package` tools were developed in python 3.4. The library dependencies are described in Table 3.1.

Table 3.1: Libraries used in son-workspace and son-package tools

Name	Purpose
PyYAML (http://pyyaml.org/)	YAML parser for python
validators (https://validators.readthedocs.io/)	validation of multiple expression types. Here, mostly used for URL validation
pytest (http://pytest.org/)	program testing framework to execute unit tests
coloredlogs (https://coloredlogs.readthedocs.io)	provides a colored and visual enhancement of the output messages
jsonschema (https://pypi.python.org/pypi/jsonschema)	responsible for schema validation, namely the package, service and function descriptors
requests (http://docs.python-requests.org)	high-level HTTP library, mostly used for the interaction with son-schema repositories and son-catalogue servers

3.2.5 Related work

The son-workspace and son-package are inspired by project management tools such as Maven [16]. The son-workspace tool provides a configuration environment suitable for the creation and maintenance of multiple projects. This configuration comprises user credentials, service addresses and several custom parameters. Based on the provided workspace, the son-package tool gathers all the project components, resolves external component dependencies, validates the components, and provides a bundle ready for deployment. Using the SDK, the user is able to efficiently develop and maintain network service functions from an abstract platform, which can be shared among partners. As a result, the provided SDK enables a novel approach for NFV service provisioning, empowering the collaboration of multiple partners.

3.3 son-catalogue

`son-catalogue` refers to the SDK catalogues that are locally available to the developer within the SONATA SDK. The SDK catalogues contain the package descriptors (PD), network service descriptors (NSD), VNF descriptors (VNFD), etc. that are either imported from other third party service 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.

3.3.1 Architecture and Component Design

The SDK catalogue is a collection of NSD, VNFD, and SSMD/FSMD catalogues, as shown in Figure 3.4. The SDK catalogue serves as the local storage point for all the NSDs, VNFDS, SSMDs/FSMDs, etc. for the developer or team of developers. In terms of implementation, a mongoDB acts as the main storage and each catalogue, i.e., NSD, VNFD, & SSMD/FSMD maps to a separate collection within the mongoDB. Hence, the schema of each collection in the mongoDB is adaptable to the structure of the descriptor stored.

A REST API, as front-end to the mongoDB, enables access for other SDK modules to retrieve or store descriptors in the SDK catalogue. The SDK catalogue accepts NSDs, VNFDs, SSMDs/FSMDs, etc. in both YAML and JSON format.

3.3.1.1 RESTful API for communication with other modules of the SDK

This sub-section describes the RESTful API of SDK catalogues. Table 3.2 shows which are the available interfaces currently working:

Table 3.2: SDK Catalogues REST management API

Uri	Method	Description	Returned code(s)
/	GET	REST API Structure and Capability Discovery	Ok (200)
/network-services	GET, POST	List all NSDs or store an NSD.	Ok (200), Not Found (404), Created (201)
/network-services/log	GET	List stored log entries	Ok (200), Not Found (404)
/network-services/id/{id}	GET, PUT, DELETE	List, update or delete a specific NSD	Ok (200), Not Found (404), Unsupported Media Type (415)
/network-services/name/{name}	GET	List a specific NSD or specifics NSD with common name	Ok (200), Not Found (404)

Uri	Method	Description	Returned code(s)
/network-services/name/{name}/version/{version}	GET	List a specific NSD by name and version	Ok (200), Not Found (404)
/network-services/vendor/{vendor}/name/{name}/version/{version}	GET	List a specific NSD by vendor, name and version	Ok (200), Not Found (404)
/network-services/name/{name}/last	GET	List last version of specific NSD by name	Ok (200), Not Found (404)
/vnfs	GET, POST	List all VNFs	Ok (200), Not Found (404)
/vnfs/id/{id}	GET, PUT, DELETE	List, update or delete a specific VNF.	Ok (200), Not Found (404), Unsupported Media Type (415)
/vnfs/vendor/{vendor}	GET	List a specific VNF or specifics VNF with common vendor	Ok (200), Not Found (404)
/vnfs/vendor/{vendor} /name/{name}	GET	List a specific VNF or specifics VNF with common vendor and name	Ok (200), Not Found (404)
/vnfs/vendor/{vendor}/name/{name}/version/{version}	GET, PUT, DELETE	List, update or delete a specific VNF	Ok (200), Not Found (404), Unsupported Media Type (415)
/vnfs/vendor/{vendor}/last	GET	List the latest VNFD from a vendor	Ok (200), Not Found (404)
/vnfs/name/{name}/version/{version}	GET	List last version of specifics VNF by name	Ok (200), Not Found (404)

3.3.2 Functional Workflows

In this first year prototype, we considered two main features of the SDK catalogues including storing and retrieving descriptors. Within the SDK, the `son-publish` and `son-package` modules need SDK catalogues for storing and retrieving descriptors, NSD and VNFD, respectively. The following workflows, Figure 3.5 Figure 3.6, illustrate the main features of SDK catalogues.

3.3.2.1 Retrieve a Service Descriptor

As mentioned above, the SDK catalogue exposes a REST API for interacting with other modules. As shown in Figure 3.5, `son-package` executes a GET operation, according to the API defined above, to retrieve the required descriptor.

3.3.2.2 Store a Service Descriptor

The `son-publish` module performs a POST operation, according to the defined API, in order to store a descriptor in the SDK catalogues, as shown in Figure 3.6.

3.3.3 Tests

The following tests are implemented to ensure that other SDK modules are able to retrieve, store and delete descriptors from the SDK Catalogues. In order to carry out the unit tests easily, an external mongoDB is required. In this subsection, unit tests for NSD are defined however, similar test are performed for VNFDs The existence of NSD and VNFD is prerequisite to the following tests.

3.3.3.1 Retrieval Test

Tests the retrieval of a single NSD or VNFD as well as bulk retrieval of NSD or VNFD by performing a GET using the API defined above.

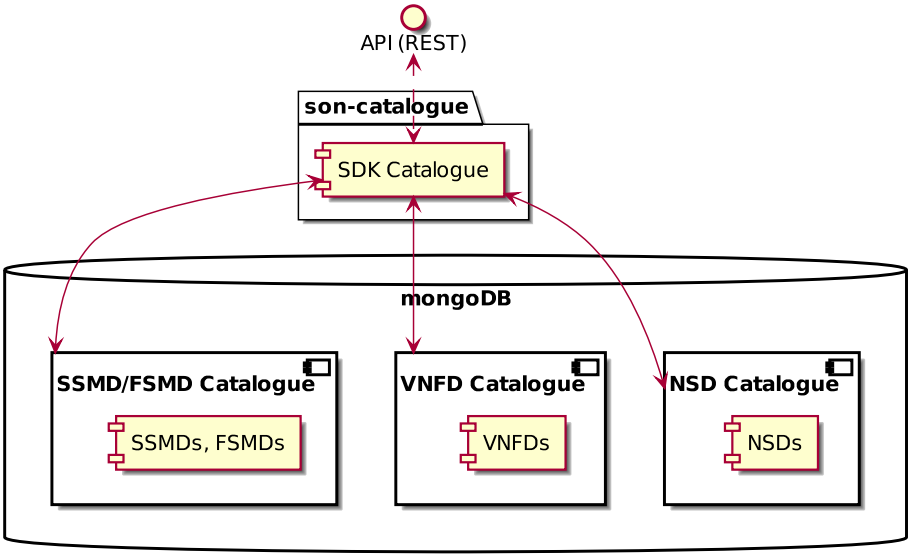

Figure 3.4: SDK Catalogues components and interfaces

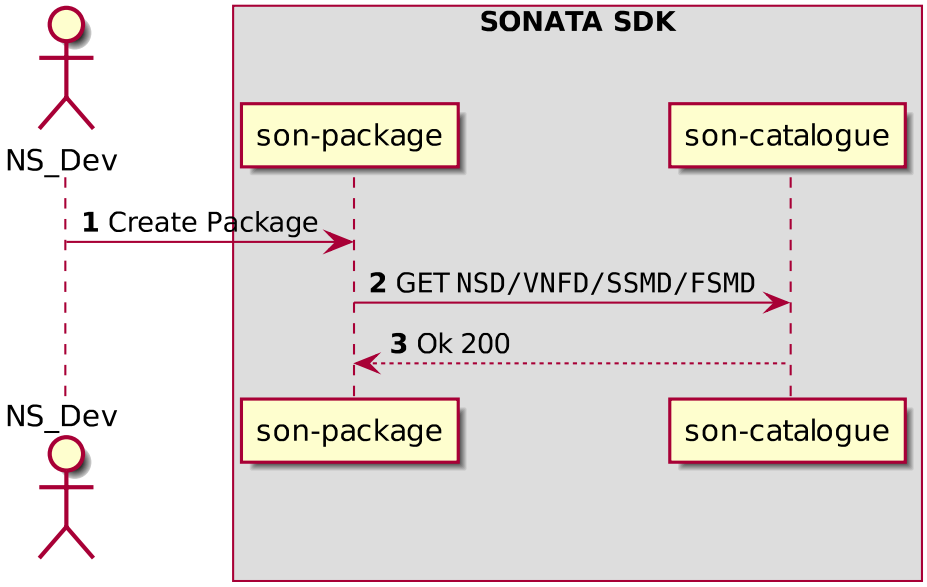

Figure 3.5: Retrieve Descriptor from SDK Catalogue

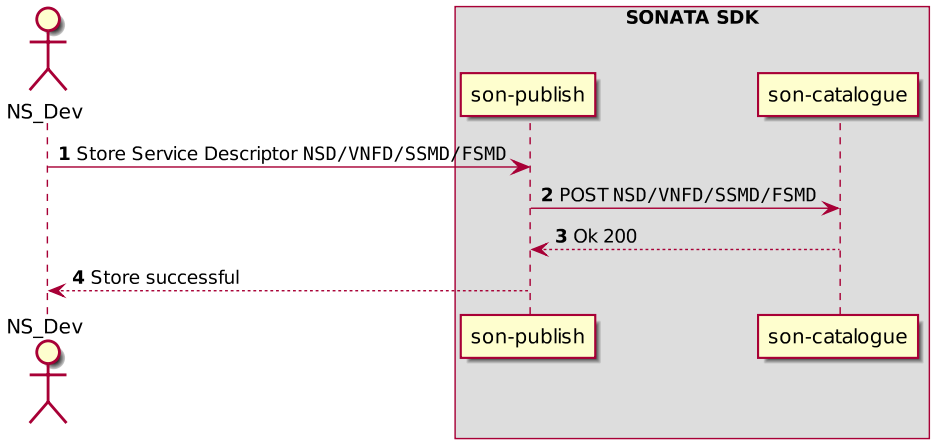

Figure 3.6: Store Descriptor in SDK Catalogue

1. List API methods by accessing the root:
   - **Set-up:** NA
   - **Execution:** curl -X GET //localhost:4012/
   - **Expectation:** HTTP code 200 (OK) is returned, with the list of methods available
   - **Tear-down:** not applicable
2. List all the available NSDs:
   - **Set-up:** NA
   - **Execution:** curl -X GET localhost:4012/network-services
   - **Expectation:** HTTP code 200 (OK) is returned, with the list of NSD
   - **Tear-down:** not applicable
3. Retrive a NSD with naming trio:
   - **Set-up:** NA
   - **Execution:** curl -X GET localhost:4012/network-services?name=eu.sonata-nfv.service-descriptor&vendor=sonata-demo&version=0.2
   - **Expectation:** HTTP code 200 (OK) is returned, together with the JSON representation of the NSD limited by 10 per page
   - **Tear-down:** not applicable
4. Retrive a NSD with UUID:
   - **Set-up:** NA
   - **Execution:** curl -X GET localhost:4012/network-services/id/ 32adeb1e-d981-16ec-dc44-e288e80067a1
   - **Expectation:** HTTP code 200 (OK) is returned, together with the JSON representation of the NSD with UUID 32adeb1e-d981-16ec-dc44-e288e80067a1
   - **Tear-down:** not applicable

3.3.3.2 Store Test

Tests the storage of a single NSD or VNFD by performing a POST using the API defined above and verifies the response.

1. Submit a new nsd:
   - **Set-up:** have a valid nsd in a json format (nsr-example.json, in the folder ./spec/fixtures)
   - **Execution:** curl -X POST -d @nsd-example.json `http://localhost:4012/network-services` --header "Content-Type:application/json"
   - **Expectation:** HTTP code 200 (OK) is returned
   - **Tear-down:** not applicable

3.3.3.3 Delete Test

Tests the removal of a NSD or a VNFD by performing a DELETE operation as defined in the API.

1. Delete an nsd with UUID given:
   - **Set-up:** NA
   - **Execution:** curl -X DELETE `http://localhost:4012/network-services/id/` 32adeb1e-d981-16ec-dc44-e288e80067a1
   - **Expectation:** HTTP code 200 (OK) is returned
   - **Tear-down:** not applicable

3.3.4 Technologies Used

Table 3.3 lists the technologies used in the implementation of the SDK Catalogues.

Table 3.3: Technologies used in the SDK Catalogues API

Name	Type	Purpose
ci_reporter_rspec	library	Connects CI::Reporter to RSpec
json	library	JSON implementation as a Ruby extension in C.
json_schema	library	A JSON Schema V4 and Hyperschema V4 parser and validator.
thin	library	Application server
rack-test	library	Rack::Test is a small, simple testing API for Rack apps
rake	library	Dependency manager
rest-client	library	REST client
rspec	framework	Tests framework
rspec-mocks	library	To be used by **rspec**
rspec-its	library	To be used by **rspec**
rubocop	library	For styling checking
rubocop-checkstyle_formatter	library	To be used by **rubocop**
ruby	programming language	Programming language
mongoid	framework	ODM (Object Document Mapper) Framework for MongoDB
mongoid-pagination	library	A simple pagination module for Mongoid
sinatra	framework	Web application framework
sinatra-contrib	library	Add-ons for the **sinatra** web-app framework

Name	Type	Purpose
web mock	library	For mocking external services
yard	tool	Documentation generation tool for the Ruby programming language

3.3.5 Related Work

In the context of NFV, the evolution of network services as chains of VNFs is still under discussions and development. Thus, currently, to the best of our knowledge, there are no existing SDK specific NSD or VNFD catalogues solution available.

3.4 son-emu

An emulation platform called `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 3.7 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 [29].

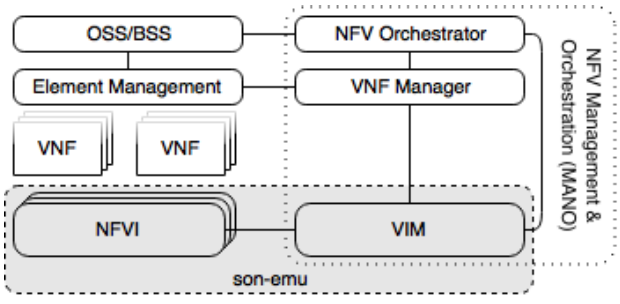

Figure 3.7: SONATA emulation tool mapped to ETSI reference architecture

`son-emu` maps to step 3 point 4 of the storyboard presented in Section 4. It is used by network service developers during design and implementation of network services, like mobile phone emulators are used for mobile application development. A developer runs its service locally and checks if it behaves like intended (functionality, not performance). If this is not the case, the service can be changed and again be tested. When the service operates as expected, it cab be deployed on a

real service platform (either for qualification or production) so that it can again be tested to check its performance in a real cloud environment.

3.4.1 Workflow and Usage

Figure 3.8 shows the general architecture of `son-emu` platform and depicts the high-level workflow of a developer using it. First, the service developer defines a network service package, consisting of service (NSD) and function (VNFD) descriptors as well as Docker files or pre-build images that contain the network functions to be tested (1). Second, the developer defines a multi-PoP topology on which he wants to test the service (2) and starts the emulator with this topology definition (3). After the emulator has been started, the developer connects the MANO system of his choice to the emulated PoPs in the platform by using standard cloud interfaces (4). Now, the network service can be deployed on the platform by pushing it to the MANO system (5) which starts each network function as a Docker container, connects it to the emulated network, and sets up its forwarding chain. Finally, the service is deployed and runs inside the platform (6). In this stage, a developer can directly interact with each running container through Containernet's interactive command line interface (CLI), e.g., to view log files, change configurations, or run arbitrary commands, while the service processes traffic generated by additional containers running in the emulation using, e.g., iperf. Furthermore, a developer and the MANO system can access arbitrary monitoring data generated by the SDN switches or the network functions.

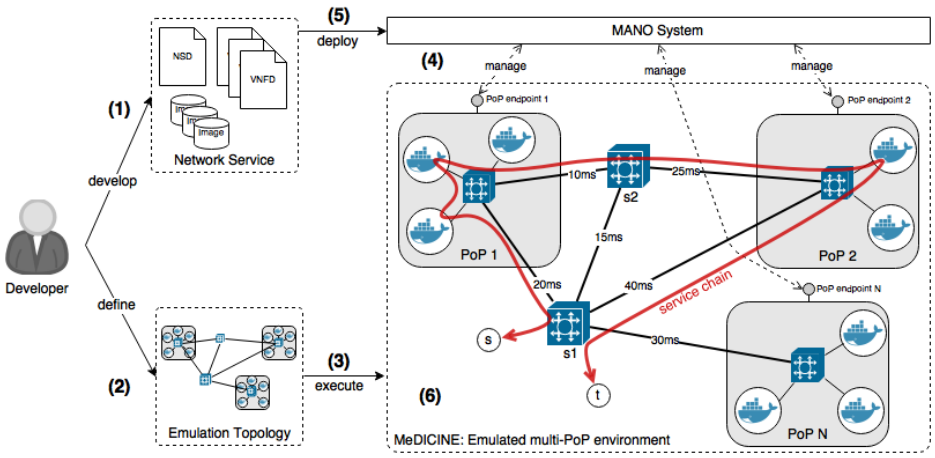

Figure 3.8: SONATA emulation tool general approach and high-level network service developer workflow

3.4.2 Architecture and Component Design

The `son-emu` system design follows a highly customizable approach that offers plugin interfaces for most of its components, like cloud API endpoints, container resource limitation models, or topology generators.

Figure 3.9 shows the main components of our system and how they interact with each other. The `emulator core` component implements the emulation environment, e.g., the emulated PoPs. It is the core of the system and interacts with the `topology API` to load test topology definitions. The `resource management API` allows to connect resource limitation models that define how much resources, like CPU time and memory, are available in each PoP. The flexible cloud `endpoint API` allows our system to be extended with different interfaces that can be used by MANO systems

to manage and orchestrate emulated services. Finally, we provide an easy-to-use CLI that allows developers to manually perform MANO tasks, like starting/stopping VNFs while our platform is running.

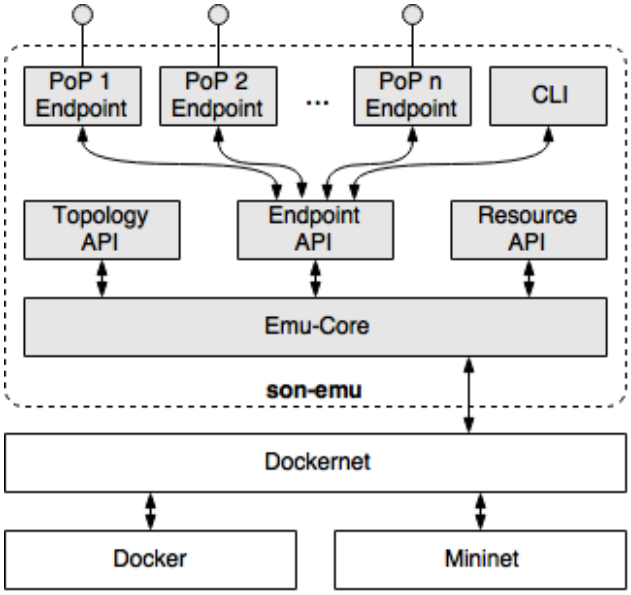

Figure 3.9: SONATA emulation tool system architecture and component design

In contrast to classical Mininet topologies, `son-emu` topologies do not describe single network hosts connected to the emulated network. Instead, they define available PoPs which are logical cloud data centers in which compute resources can be started at emulation time. In the most simplified version, the internal network of each PoP is represented by a single SDN switch to which compute resources can be connected, like shown in Figure 3.8. This can be done as the focus is on emulating multi-PoP environments in which a MANO system has full control over the placement of VNFs on different PoPs but limited insights about PoP internals. We extended Mininet's Python-based topology API with methods to describe and add PoPs. The use of a Python-based API has the benefit that developers can use scripts to define or algorithmically generate topologies.

Besides an API to define emulation topologies, an API to start and stop compute resources within the emulated PoPs is needed. To do so, `son-emu` uses the concept of flexible cloud API endpoints. A cloud API endpoint is an interface to one or multiple PoPs that provides typical infrastructure-as-a-service (IaaS) semantics to manage compute resources. Such an endpoint could be an OpenStack Nova or HEAT like interface, a specific interface for SONATA's SP, or a simplified interface to which SONATA service packages can be uploaded (dummy gatekeeper). These endpoints can be easily implemented by writing small, Python-based modules that translate incoming requests (e.g., an OpenStack Nova start compute) to emulator specific requests (e.g., start Docker container in PoP1).

Realistic PoPs offer limited compute, memory, and storage resources which have to be considered by a MANO system when placement and scaling decisions are taken. To simulate this resource limited behaviour, `son-emu` provides the concept of `flexible resource models` that can be optionally assigned to each PoP in the topology description. These models are called whenever compute resources are allocated or released and they compute CPU time, memory, and storage limits for each started container. They are also able to reject allocation requests, to indicate that there are no free resources on a given PoP. The provided resource API of our systems allows developers to easily create their own resource models. The usage of these resource models is optional and they

are needed to test MANO systems in more realistic environments but not for simple functional tests of manually deployed service chains.

To emulate a fully working network service, we need to deploy its VNFs but also set up the forwarding paths between them, as shown in the service chain of Figure 3.8. A networking API is provided for this purpose. Since the emulated topology with PoPs consist out of SDN switches, a service developer can have full control over the forwarding paths of the service network traffic. Setting up a chain where traffic is steered along a defined path is now a matter of setting the correct forwarding entries in the SDN switches, enhanced by an SDN controller. This brings Service Function Chaining (SFC) functionality into `son-emu`. VLAN tags are currently used as identifier to differentiate chains in multiple emulated services, similar to ongoing research regarding the use of Network Service Headers (NSH) [20]. An internal graph representation of the topology and its attached VNFs is kept in the emulator. The forwarding path between the VNF interfaces can be found by using this graph, using different algorithms (eg, shortest path) and custom weights (eg. link delay). Every SDN switch on the chain path is set with the correct OpenFlow forwarding entries.

3.4.3 MANO System Integration

The emulation platform offers a cloud-like interface to control the deployment of VNFs inside the emulated multi-PoP environment. These interfaces will be used by a locally running version of SONATA's SP to which a network service developer pushes the service package that should be locally tested. Additionally, a simple module that behaves like SONATA's gatekeeper was implemented as the final service platform prototype provided by WP4 was not available when the first prototype of the emulator was implemented. This module offers a REST interface to which service packages can be uploaded with `son-push` and deploys the uploaded packages on the emulator. We call this module `dummy gatekeeper`.

3.4.3.1 Dummy Gatekeeper

The dummy gatekeeper module tries to provide the basic functionality of SONATA's service platform needed to initially deploy a service package created by the SDK. It is a placeholder solution to demonstrate the emulator and to support the development of SDK tools, like `son-push` until the emulator is integrated with a functional version of SONATA's service platform. The dummy gatekeeper uses the same API as the real gatekeeper implementation and provides the following (limited) set of features:

- REST endpoint to upload a *.son service package (onboard)
- Extract service package and parse relevant parts of the contained descriptors
- Download and optionally build referenced Docker images
- REST endpoint to trigger instantiations of an uploaded service (instantiate)
- Simple placement logics (all VNFs on a single PoP or round robin)
- Instantiation of each VNF described in the service
- SDN-based chain setup between instantiated VNFs
- REST endpoints to list uploaded and instantiated services

The dummy gatekeeper does in particular *not implement* functionalities, like scaling, advanced placement, FSM/SSM support, further lifecycle management actions, platform recursiveness, or service package updates. These features are beyond the scope of a placeholder solution and would require a lot of duplicated implementation effort with WP4.

3.4.3.2 SONATA Service Platform Integration

The emulator will be able to be integrated with SONATA's SP that will then be responsible to manage and control the deployment of network services on the emulation platform. We foresee the following options to do this integration when the needed SP components are available:

- **Option A: `son-emu` specific infrastructure wrapper:** For this option, a wrapper has to be implemented that connects to the *wrapper bay* of SONATA's infrastructure adaptor which is a component of SONATA's SP. Using this, the emulator can be tightly integrated with the SP since we can define arbitrary interfaces between these major components.
- **Option B: HEAT-like transparent interface:** In this option, the emulator implements a cloud-like interface that behaves like a standard HEAT interface and can read HEAT templates given as HOT descriptors. This option has the benefit that it is completely transparent for the service platform which can use its default HEAT adaptor. The downside of this approach is the limited functionality offered by a HEAT-like interface.

A final decision on the chosen option will be done during the second major iteration of the project.

3.4.3.3 SONATA Monitoring Integration

It is the intention that the emulator can integrate with SONATA's SP. The SP can then deploy, manage and control network services on the emulation platform like any on other infrastructure domain. In this case it would make sense to also let the SP's monitoring framework work with `son-emu`. To achieve this, the monitored metrics gathered inside the emulator, should be exported in a compatible way so they can be read and stored by the standard SONATA SP monitoring framework. The use of containers and the SDN-based networking in the emulator allow a wide range of metrics that can be captured relatively easy and provide useful information to a service developer:

service-generic metrics For each deployed VNF or forwarding chain, some default metrics such as container compute/memory/storage parameters or network traffic rate can be monitored.

service-specific metrics By exploiting the NFV/SDN based features of the emulator, more detailed parameters such as end-to-end delay, jitter or specific network flows can be monitored.

To prepare the emulator as much as possible for integration with the SP, the same metrics gathering system is used, which is based on Prometheus [31]. Further details on this monitoring framework are given in Section 3.6.

3.4.4 Tests

Son-emu implements several unit tests to check its functionalities. These test implementations are based on Python's default test module `unittest` and have to be executed with `root` privileges, just like normal emulations because they instantiate small emulation topologies and deploy test containers. The available test cases are described in the following.

3.4.4.1 Tests: Emulator topology

Single data center Executes a topology with a single PoP and checks if the PoP's network is reachable.

Multi data center direct Executes a topology with multiple PoPs and checks if each PoP is reachable. Also tests if the PoPs can reach each other.

Multi data center with intermediate switches Executes a topology with multiple PoPs and switches in between them. Checks if each PoP is reachable. Also tests if the PoPs can reach each other.

3.4.4.2 Tests: Emulator networking

Single service Executes a topology with two PoPs connected with different switches in between. The default learning switch behaviour of the SDN controller is disabled. One Docker compute instance is started in each PoP and a forwarding chain is set up. Connectivity between the compute instances is verified.

Multi service A first service consists out of two compute instances and a forwarding chain. A second service is started with a different forwarding chain. It is verified that instances of the same service can communicate, but instances of a different service cannot. When a forwarding chain of the first service is removed, the one of the second service should still work.

3.4.4.3 Tests: Emulator compute API

Start compute instance Starts a couple of compute instances in a simple test topology and validates that the instances (Docker containers) are created correctly and are accessible from the outside over the network.

Stop compute instance Stops a couple of compute instances and tests if the Docker containers are removed correctly.

Fetch status of compute instance Requests status information of a previously started compute instance. Validates the response and its contents, such that container id, assigned PoP, and IP addresses.

Check compute instance connectivity Starts a couple of compute instances and check if there is network connectivity between them, i.e., execute ping commands in the containers.

Multiple interleaved compute API requests Starts and stops multiple containers directly after each other to stress the emulator and the underlying Docker service. Checks if always the correct set of containers is running.

3.4.5 Technologies used

Table 3.4 lists the technologies used in the implementation of `son-emu`.

Table 3.4: Technologies used in the son-emu project

Name	Type	Purpose
argparse	library	parse command line arguments, CLI

Name	Type	Purpose
docker-py	library	Docker client library, control Docker service
Containernet	external tool	Mininet-based emulation platform using Docker containers, controlled by son-emu
flask	library	lightweight web framework for fake gatekeeper REST interface
flask-restful	library	REST extension for flask
networkx	library	graph library, chaining support
oslo.config	library	support library for ryu
paramiko	library	SSH client
pyml	library	YAML parser
pytest	library	unittest framework
pytest-runner	library	support for pytest
Python 2.7	programming language	programming language
requests	library	HTTP client library, test fake gatekeeper
ryu	library	SDN controller framework
six	library	support library for ryu
tabulate	library	pretty print tables on terminal, CLI
zerorpc	library	RPC library that uses ZeroMQ messages to do RPC calls, communication CLI to son-emu

3.4.6 Related work

NFV development support is still a novel research direction with a limited amount of existing solutions, most of them focusing on SDN debugging not on prototyping of complex network services [28] [15] [13]. Other approaches are based on simulations to test and validate management solutions, e.g., placement algorithms, but they only provide a very limited amount of realism since the simulated network functions are only proxies and not real implementations used in production [5] [43]. The VLSP [27] offers more realism but the tested network functions are still limited to be simple Java programs not real NF implementations.

Emulation tools, like Mininet, are able to execute any network function implementation in its own virtualised network namespace [24] [41] [2]. However, moving these network functions into a productive cloud is still a manual task and these tools lack the possibility to emulate cloud sites, e.g., they have no functionality to stop and remove hosts at runtime. The ESCAPE platform overcomes some of these limitations by combining a MANO system with Mininet but it does not target development support or prototyping tasks and it is limited to Click-based NFs [36].

Real cloud testbeds, that might be installed on a single physical machine [30], are typically not able to run services in arbitrary network topologies [19]. And even if they do, they come with a management overhead and a limited number of PoPs that can be used for tests [23].

3.5 son-push

3.5.1 Architecture and design choices

The main goal of the `son-push` tool is to interface with the gatekeeper module of the SONATA service platform or with the dummy gatekeeper of the `son-emu` part of the SDK (cfr. Figure 3.10). The `son-push` tool is part of the `son-cli` suite of tools. The tool `son-push` is used by the developer to send a package to the Service Platform. Section 2 of this deliverable indicates where this step fits in the overall workflow.

The interface to the Gatekeeper is a REST API, which is documented extensively in deliverable D4.1 [11]. Because this REST API might change often over time, it is desirable to have a `son-push` tool design which can be easily adapted. Therefore, for simplicity and to be compliant with other SONATA SDK tool development, an easily updatable Python-based implementation has been made.

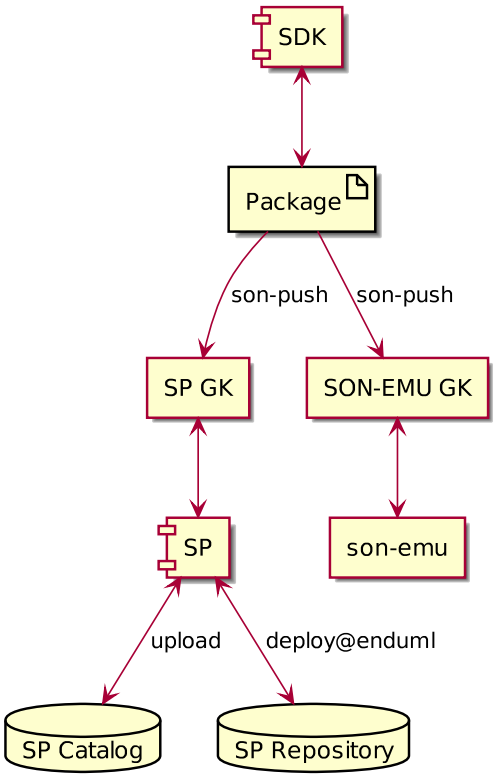

Figure 3.10: son-push interactions

Table 3.5 lists the technologies used in the implementation of the `son-push` tool. A more detailed manual of this tool can be found in the appendix of this document.

The features of the provided `son-push` tool are:

- List available packages at gatekeeper of the SP (the gatekeeper is identified by a URL)
- Upload a package to the gatekeeper of the SP (and receive UUID of the uploaded package)
- Deploy an already uploaded package via the gatekeeper on the SP
- List available service instances of the SP

3.5.2 Workflow

Once project/service has been packaged using the `son-package` tool, it can be uploaded using `son-push` to the gatekeeper of a (emulated) Service Platform. This step is single-step process as is illustrated in the manual of `son-push`, which is part of `son-cli` Appendix B.

3.5.3 Tests

A small set of unit tests are implemented to check functionality. These are following the Python default `unittest` module. The resulting test cases are the following:

Gatekeeper reachability Test if a particular gatekeeper mockup URL can be reached

Package check Test if the available packages can be retrieved from a given mockup URL

Upload package Test if a package can be uploaded to a given mockup URL

Instantiate service Test if a service can be instantiated from a give mockup URL

3.5.4 Technologies used

Table 3.5: Technologies used in son-push

Name	Type	Purpose
requests	Python library	HTTP library
validators	Python library	data and url validation tool

3.5.5 Related work

Apart from existing command-line tools such as CURL, there is currently not relevant state of the art.

3.6 son-monitor

The monitoring framework developed in SONATA will leverage existing tools to make them easier to use in a network-service context. This means that network, compute and storage metrics are not only measured and stored in an isolated way, but can also be aggregated and linked to a higher-level service-oriented viewpoint. In this context, monitoring is a crucial part to get a picture of the QoS offered by a service. Monitoring is also useful during the development of a service. In the SONATA SDK, the monitoring tools are envisioned to support a wide range of debugging and profiling features at the disposal of a service developer. This toolset aims to help in creating a more efficient development and deployment cycle for a network service.

3.6.1 Workflow and Usage

The metrics to be monitored on a service can be configured in two ways:

1. As specified in the service descriptors (NSD, VNFD defined by `son-schema` in Section 3.1). These metrics are configured in the monitoring framework at deployment of the service and gathered during its whole lifetime.

2. Metrics can also be gathered at a later point in time and can only be needed temporarily. The features implemented in the `son-monitor` commands, setup new metrics to be gathered or deploy dedicated monitoring agents. These agents can execute dedicated monitoring functions which would imply otherwise too many modifications to an existing VNF or load it too much.

This general monitoring approach is shown in Figure 3.11. The monitoring framework can connect to VNFs, monitoring agents, the infrastructure, network or the SP itself. Upon the start of a dedicated monitoring feature, the network chaining of the service could be altered so traffic is passing (transparently) through the monitoring agent. Network services are likely to contain third-party, closed-source VNFs. In this context, a service developer is unable to modify the VNF itself in order to gather any customized metrics. To deal with this kind of situation, the SONATA SDK could allow the creation and deployment of customized monitor agents to help a developer in monitoring the wanted parameters, with the least possible dependency on the service VNFs.

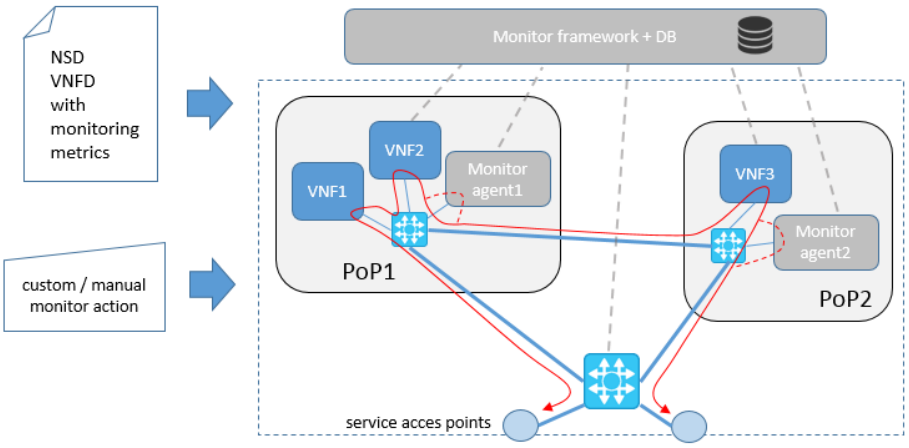

Figure 3.11: SONATA monitoring framework general approach

3.6.2 Architecture and Component Design

The total architecture of the monitoring tools as used in the SDK and emulator are depicted in Figure 3.12. The grey colored blocks depict where specific monitoring functionality is implemented.

- In `son-emu`, cAdvisor is used to gather and export metrics related to resource usage of deployed containers.
- In `son-emu`, the REST API of the Ryu SDN controller is addressed to gather network related metrics of the deployed service.
- The monitoring framework outside of the emulator (Prometheus + Pushgateway) gathers the exported metrics from the emulator, the same it would gather metrics in the SP from another infrastructure domain.
- The yellow line depicts the interface to Prometheus where monitored metrics by monitor agents, son-emu and cAdvisor are pushed to.
- On the highest level, the SDK user (the service developer) can use the monitoring features implemented in `son-cli` tool set. From there the son-emu API and the Prometheus REST API can be addressed to test a service in the emulator and get the wanted monitoring data.

The supported metrics when the service is deployed on the actual Infrastructure, depend on the specific Infrastructure Manager. For `son-emu` currently the following are supported:

- cAdvisor assists in exporting various parameters of the deployed containers related to compute and memory metrics.
- The SDN controller (Ryu) can export specific networking related metrics in a finer grained way using packet counters per flow or other Openflow-specific counters.
- Custom-made metrics can be gathered and exported by the VNF itself. Either custom-built monitor agents can be deployed as part of the service graph or the SDK can assist in adding a metric export function to an existing VNF.

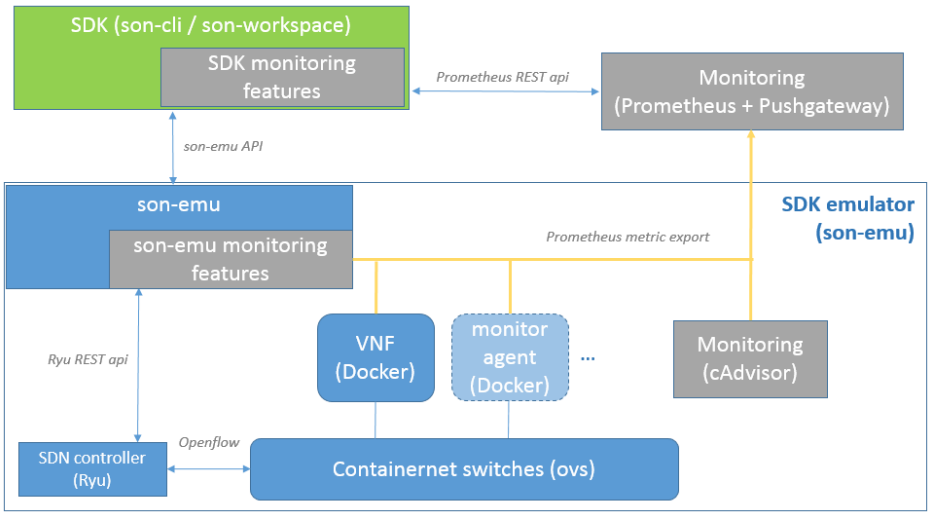

Figure 3.12: Architecture and components in son-monitor

- Prometheus is used as metric gathering framework where all measured samples are stored as a time series and can be queried.

Further details and examples are given in Appendix E

3.6.2.1 Son-monitor as Profiling Tool

As illustration of how an SDK environment can provide useful tools for a network service developer, a first feature is implemented that allows the automated performance profiling of a VNF. This function allows a developer to retrieve with a simple command a performance table, giving an indication of the load the VNF under test is able to process. The output of this profile command can later be used as input for an automated scaling algorithm (eg. developed in an SSM).

The execution flow of the profiling function can be summarized like this:

- A VNF is started and awaits traffic to process.
- Additional monitoring agents are started and linked to the VNF:
  - A **Traffic Generator**: can generate a packet stream with a fixed rate, remotely configurable (SSH).
  - A **Traffic Sink** : receives the generated traffic and calculates specific metrics (eg. packet loss, jitter).
- Metrics are gathered (CPU load, traffic rate, packet loss).
- The traffic rate is gradually increased until the cpu is fully loaded.
- A table is returned with the profiling results. Listing: Packet rate | CPU load | packet loss
- The deployed VNFs, agents and links are removed.

Figure 3.13 shows the different steps in deploying the profiling function from the SDK, it uses these available commands:

1. Commands to start/stop VNFs in the emulator (`son-emu` API, Section D.2.1)
2. Commands to link VNFs in the emulator (`son-emu API`, Section D.2.2)
3. Commands to export metrics from the emulator (`son-emu API`, Section E.1)
4. Commands to query metrics from the monitoring framework (Prometheus Rest API, more info in D4.1 [11])

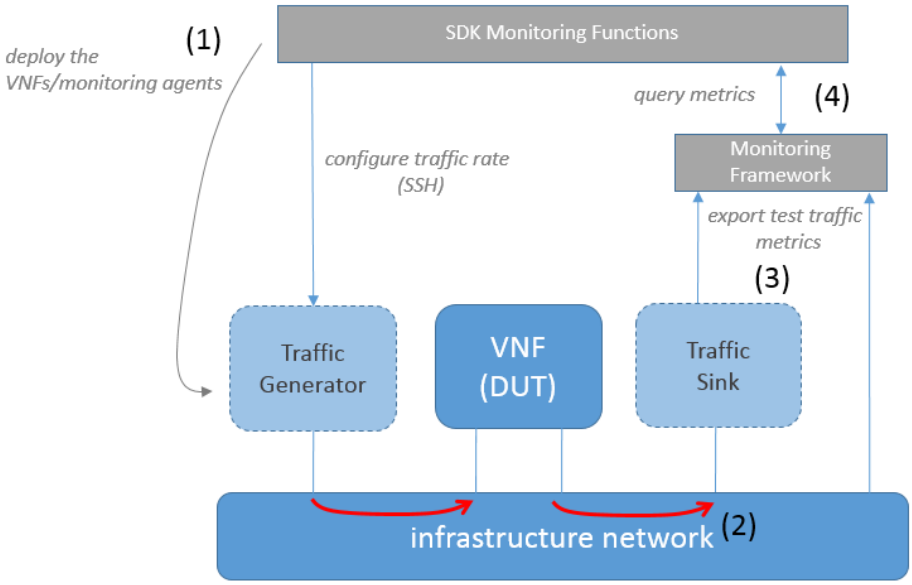

Figure 3.13: Profiling function of a VNF as SONATA SDK feature example

Further integration will implement this and more useful features so they can be executed from the `son-cli` tool set. Additional arguments to the ones detailed in Section E.2 are planned. This in order to deploy the VNFs/agents in this setup as a Service Descriptor with VNF Descriptors. This generic approach will make it possible to do the profiling action on different VIMs connected to the SONATA platform and not only the emulator. Next to this, also different flavours could be selected to define which resources to use for testing the VNF. This will allow a developer to create, test and debug the service from the same workspace and will further enhance the SONATA SDK environment.

3.6.2.2 Technologies used

This software builds further upon the technologies used in `son-emu` Section 3.7.4 and `son-monitor` in the SP (as explained in D4.1 [11]). As a summary, Table 3.6 lists the most important technologies used in the implementation of monitor features in the SDK.

Table 3.6: Technologies used in the son-monitor features of the SDK

Name	Type	Purpose
Python 2.7	Programming Language	Preferred Python version to be used with son-emu

Name	Type	Purpose
OpenvSwitch	SDN switch	Software SDN switch, used by son-emu and supported by other platforms such as Openstack
Ryu	SDN controller	Controller to do the SDN chaining in the emulator and gather network metrics. Ryu can also integrates with other platforms such as Openstack [32]
Prometheus	Docker container	Monitoring system and time series database [31]
Prometheus Pushgateway	Docker container	push acceptor for ephemeral and batch monitoring jobs. Holds the metrics while they are scraped by Prometheus
prometheus_client	Python library	export metrics to the Prometheus Pushgateway
cAdvisor	Docker container	Analyzes resource usage and performance characteristics of running containers [4]

3.6.3 MANO system integration

The current integration status allows a service to be deployed either in the SDK emulator (`son-emu`) or via the SP itself. A developer should be able to access monitored metrics of the service, no matter where it is deployed:

- In `son-emu`, monitored metrics are gathered and stored using a framework based on Prometheus.
- An extended version of Prometheus is also deployed on the SP to store all gathered metrics there.
- The same API can be used for querying monitored data both from the SP or SDK.

This allows a more optimized integration of monitoring tools where a developer has access to metrics of a deployed service in the SDK emulator and the SP. These data sets can serve as further input for other tools such as `son-analyze` or SSM/FSM development.

At a later stage, the SDK emulator will be integrated with SONATA's SP that will then be responsible to manage and control the deployment of network services on the emulation platform. Since support for the Prometheus monitoring framework is present in both the SDK and the SP, the monitored metrics in the emulator can also be managed by the SP's monitoring framework.

The integration of the `son-monitor` commands in the general MANO workflow is shown in Figure 3.14. The SP can deploy the service either on the emulator or on the actual infrastructure, and monitored metrics will be exposed to the same monitoring framework (based on Prometheus). It can be noted that monitoring is valid for all modules in the SP, whether it is an SSM, FSM, VNF, the network controller or the SP itself.

3.6.3.1 SONATA Monitoring Framework

For full compatibility, we envision that any VNF or service deployed in the emulator can be monitored in the same way as when it would run in any other supported infrastructure domain. For this reason, any metric measured in the emulator is exported as supported by the Prometheus monitoring framework [31]. An RESTful API is available which enables correct configuration and querying of monitored metrics.

The **drawbacks** of this approach are that there are lower limits to the interval to sample the metrics. Prometheus can scrape the available metrics in the order of max. 1Hz. Also the accuracy of this sampling interval might not be perfect. Prometheus attaches a timestamp to each scraped

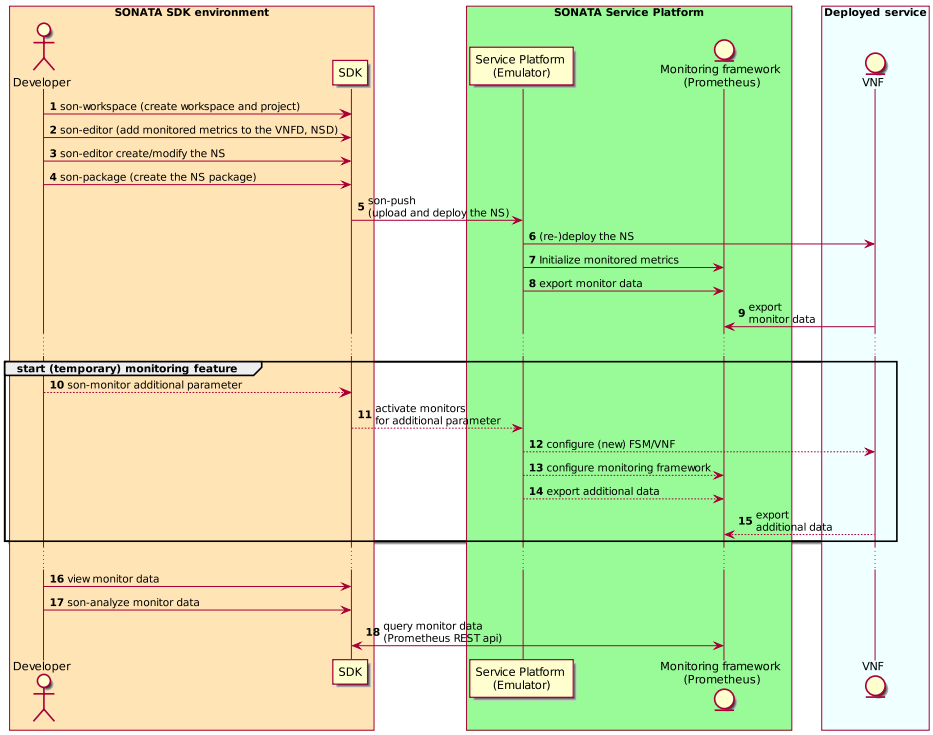

Figure 3.14: SONATA monitoring workflow

metric. Since there is jitter on the scraping interval, the attached timestamp by Prometheus is not exactly the same as the time when the metric was actually measured. This can be overcome by exporting the correct timestamp also as a metric, together when the parameter was sampled. Prometheus is however good in calculating longtime averages and statistics of longer time-series, the above described errors are then averaged out.

The **advantages** of this framework are however that monitored metrics can be instantiated and removed 'on-the-go', fitting into the context of typical SDK usage, where VNFs are often only temporarily deployed for the duration of a test. This architecture furthermore allows easy integration into any cloud-like platform, as it provides a fixed, well-documented and easy to use interface to export custom metrics from any new VNF or network service.

The further details of the SONATA SP monitoring framework are given in D4.1, Task 4.4 [11].

3.6.4 Tests

The testing for the monitor functionality depends on these other projects:

- `son-emu` : The emulator where a VNF is deployed and whose metrics are exported. This is deployed as a Docker container.
- `son-monitor` : The Prometheus monitoring framework which gather all exported metrics. This is deployed as a Docker container.
- `son-cli` : The SDK workspace from where a developer gives the instruction to start and monitor certain metrics of a vnf.

The SDK son-monitor test is triggered when any of the above projects has been rebuild. The test deploys the above three projects as given in Figure 3.12. The execution flow of the test is as

follows:

1. Deploy a VNF or service in `son-emu`
2. Export metrics monitored in the emulator to the Prometheus monitoring framework
3. Check if the monitored metrics can be accessed from the SDK workspace

3.6.5 Related Work

Following the trend of cloud-like deployments of network services, the field of network monitoring is yet to experience the same revolutionary transformation that software-defined networking (SDN) and network function virtualization (NFV) brought to the telecom world. The relevance of dynamic monitor probes in cloud environments is illustrated in [25] and [34]. This relates to the SONATA SDK and Service Platform framework where monitoring probes can be developed and deployed as VNFs in a network service. The monitoring features that are implemented on the SDK side of SONATA are dynamic in a sense that they can (intermediately) deploy monitoring agents into a running network service.

Technologies with faster boot-time and better memory isolation than containers exist in the form of uni-kernels, as explored in [26]. In a monitoring context, a unikernel vnf developed as monitoring agent enables more light-weight and flexible monitoring tools, which can be a better fit to deploy also outside of the cloud datacenter, like local micro-datacenters or even embedded devices at the customer side.

It is clear that modern service platforms need to support many flexible ways of monitoring deployed services, with metrics that are standard available, but also likely to be custom build by service developers. The monitoring framework proposed and implemented by SONATA is capable of monitoring metrics which are standard exposed by the infrastructure provider. In addition, it is also fully supporting the dynamic deployment of (custom-build) monitoring agents, which can be seen as (temporary) VNFs inside of the network service.

The SONATA monitoring framework is a push-based model, where metrics are measured inside the service or infrastructure and pushed to a central monitoring server. The monitoring server stores and exposes the metrics to higher-level features such as service profiling or analysis tools. Other popular monitoring frameworks such as Nagios are by default operating in a pull-based way. A monitoring server is pulling metrics from the monitored clients where dedicated monitoring plugins are installed as services or daemons. These plugins can also be customized by a developer [22]. The SONATA push-based model is a better fit for a flexible SDN/NFV context. Monitored metrics are more likely to exist only temporarily (eg. during profiling or intermediate scaling actions). The push-based model simplifies the configuration in this case because it does not need additional changes at the monitor server side.

3.7 son-analyze

The son-analyze tool bridges instantiated services with a high level analysis/statistical framework. It goes beyond the son-monitor tool to further scrutinize running services. Data analyst experts are the main target of this utility.

3.7.1 Workflow and Usage

The `son-analyze` tool's focus is to provide analysis sessions to the end-user. It integrates the SONATA Service Platform (SP) and SDK with a main-stream analysis language/framework. The

main feature is to give total control to the end-user over his analysis of services.

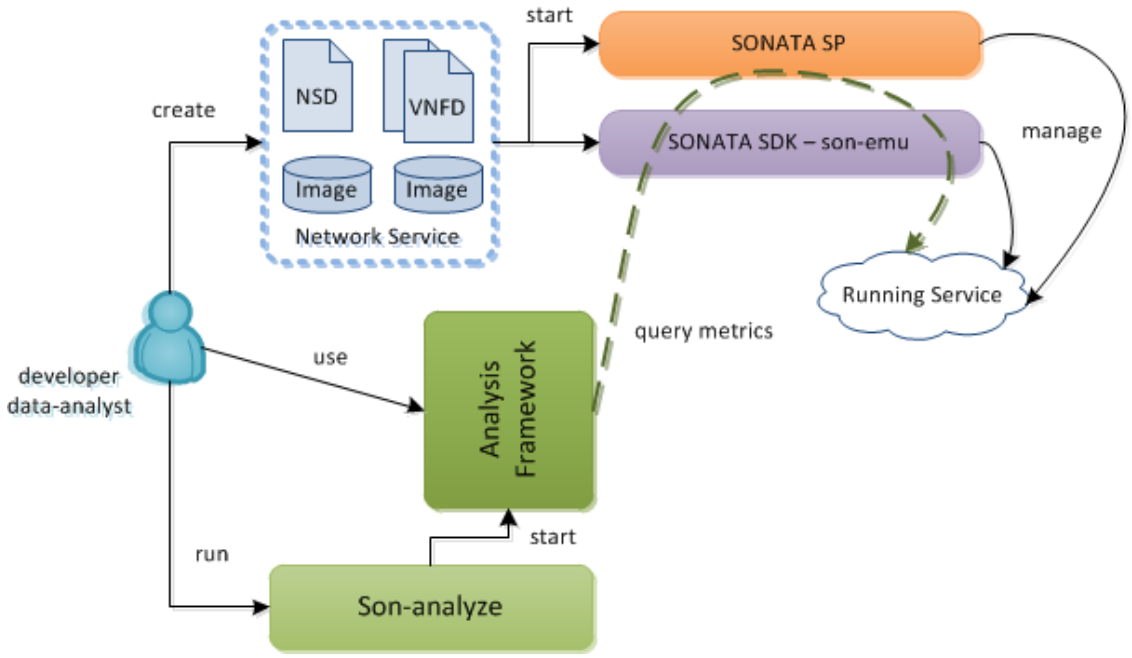

Figure 3.15: son-analyze overall view

Figure 3.15 shows a high level view of the underlying process. First the service developer creates a network service. This service is shipped as a package. It contains the service's description (NSD), the used functions' descriptors (VNFDs), and the required software images. The package is then pushed to SONATA and started. From this point, a service instance is running. A remote IaaS hosts this instance if the SONATA SP was targeted. Otherwise `son-emu` runs the instance on the local host. In both cases, the running service is monitored. The generated metrics are also stored. Those metrics are query-able through a REST API.

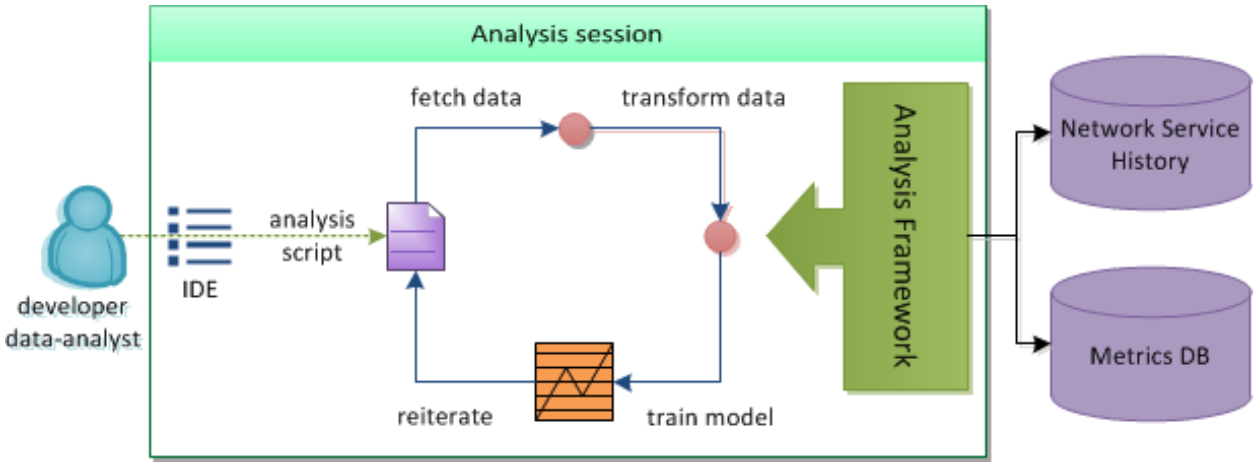

Figure 3.16: son-analyze main steps

With this context, the Figure 3.16 displays the main usage of the `son-analyze` tool. Some runtime information is stored in the SONATA SP. To compute meaningful results, it needs to be processed. This processing changes depending on the end-user or the targeted service. For

example, a system administrator may inquire how a service's internal congestion impacts the global throughput. But an application owner may try to infer new service graph definitions (to compromise between the price and the quality of service). To achieve this goal, the analysis session needs to be fully controllable. With manual but configurable custom analyses, it differs to alternatives armed with automatic but static inferences.

Back to the Figure 3.16, the end-user accesses an analysis session thought an IDE. In it, he can develop analysis scripts achieving some expected computation. They generally follow these steps: first the raw data is queried. Then some transformations are applied to clean and better shape the information. Finally, a model is elaborated and trained with the transformed values. An analysis framework executes those steps. It also provides to the end-user a high level computational language with a set of specialized libraries.

3.7.2 Architecture and Component Design

The `son-analyze` is a command line tool. It is coded in Python to follow the main programming language used across the SONATA SDK.

The R language was selected for the underlying analysis framework. It is one of the mainstream languages in the statistical and machine learning area. In particular, developers use the R language for fast prototyping and the vast collection of third-party libraries. Additionally, the RStudio Server was chosen as an IDE. It has the main advantage to be a web application. The developers only require a ubiquitous web browser to use RStudio Server. The Figure 3.17 shows the RStudio interface. It contains a small R script emulating a CPU. After executing this script, a graph is generated in the bottom right corner. It displays the raw CPU state and a simple model fitting. The magenta line is a prediction on the upcoming CPU consumption.

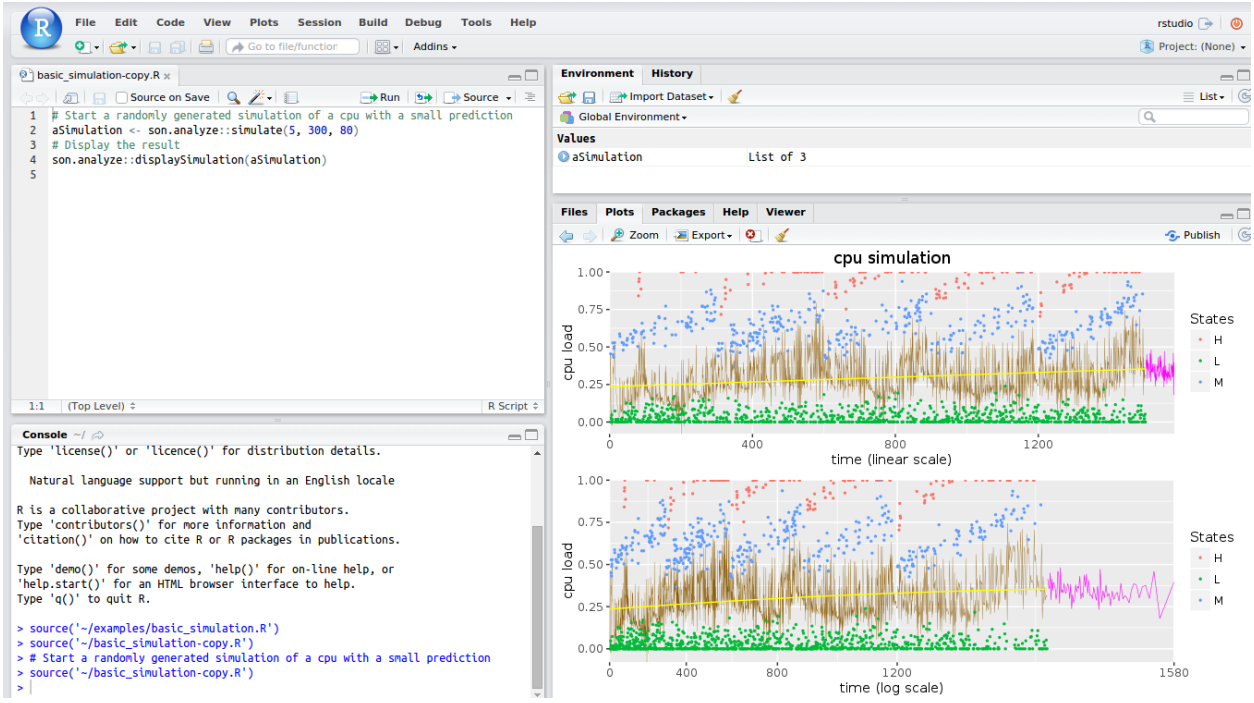

Figure 3.17: RStudio interface

The Figure 3.18 shows the current low level architecture. The `son-analyze` command line is the first entry point. It uses the Docker container technology to harness the RStudio Server.

The `son-analyze bootstrap` call builds the base `son-analyze` docker image. It extends the official RStudio image maintained by the community. To provide an immediate ease-of-use, this call also downloads and compiles various third-parties R libraries (for example `lubridate` for date manipulation). Finally, the `son-analyze bootstrap` call injects and compiles a SONATA R library in the Docker image. After having successfully built the `son-analyze:latest` image, the current host is initialized. The end-user doesn't have any reason to use this command again, apart for fetching a potential update.

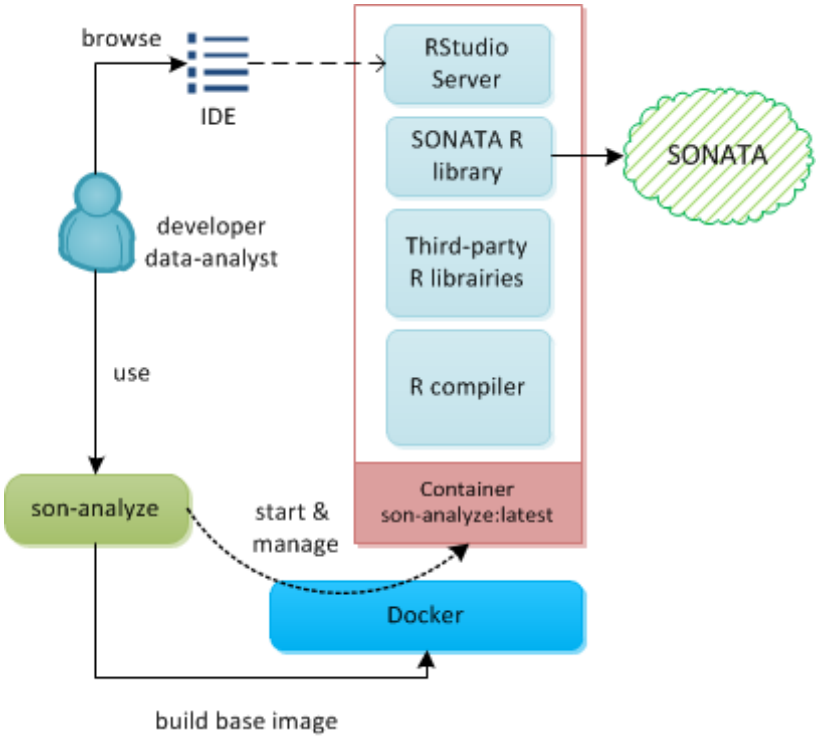

Figure 3.18: son-analyze components

The SONATA R library has been developed for this project. It ties the R analysis framework to the rest of SONATA ecosystem. It mainly revolves around two key points: fetch data from SONATA and shape it to a higher-level structure. With the result, the end-user can exploit the native R language's features and treat SONATA as a data source.

The end-user must call the `son-analyze run` command to start an analysis session. The `son-analyze` will start a Docker container with the `son-analyze:latest` image. From there, the end-user accesses the IDE through his web browser. He is able to immediately write analysis script calling the SONATA R library and querying the SONATA SP or the `son-emu` emulator. The code snippet joined in the Annex Section F.2.1, is an example of an end-user R script. It queries the SONATA integration platform to retrieve the CPU load of a specific container for the past hour. Then the script ends by plotting the raw values. The Figure 3.19 is the resulting output.

3.7.2.1 Supported Commands

See the Appendix F section.

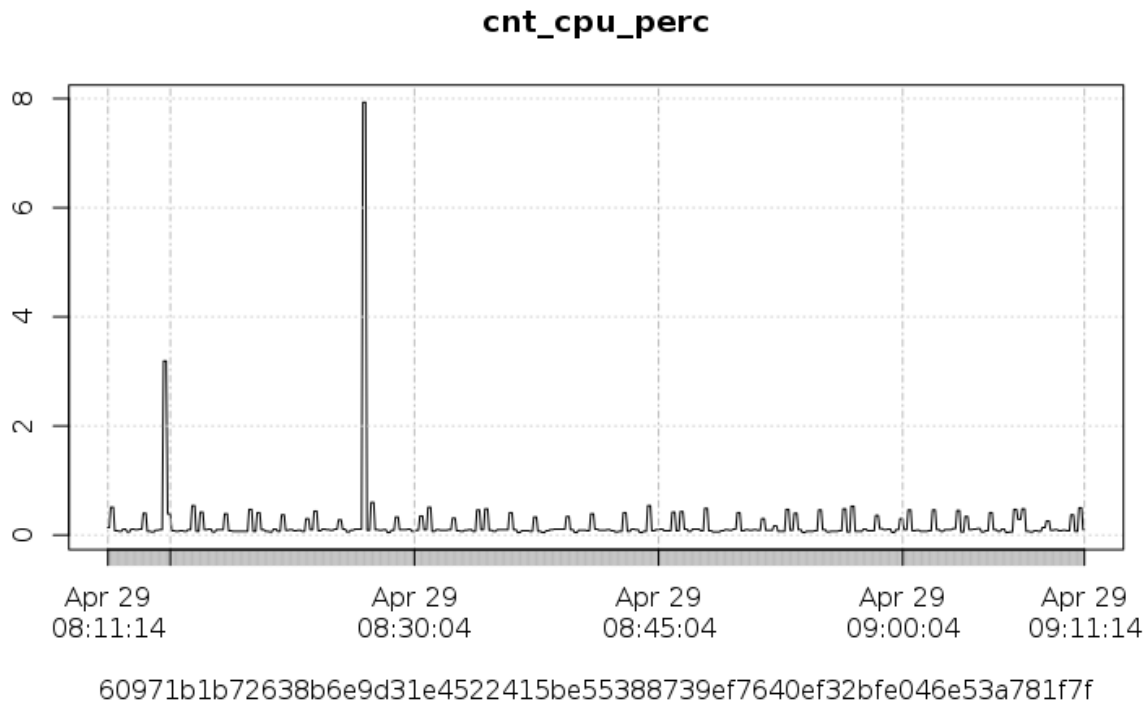

Figure 3.19: Output plot of a query to SONATA

3.7.3 Tests

A minimal amount of unit tests check the core features of `son-analyze`.

Environment creation Test if the `son-analyze` bootstrap created a valid environment. In particular if the underlying container and IDE are responsive and that they contain the required libraries.

Data binding Test if data can be retrieved from a Prometheus by loading static json fixtures

3.7.4 Technologies used

The Table 3.7 lists the technologies for: running, developing and testing `son-analyze`.

Table 3.7: Technologies used in the `son-analyze` features of the SDK

Name	Type	Purpose
Python 3.4	Programming Language	Language used for the `son-analyze`'s cli
R 3.2.3	Programming Language	Language used for the SONATA analysis library for statistical data analysis
docker-py	Python library	Docker client library
pyaml	Python library	YAML parser
typing	Python library	Type Hints for Python (PEP 484)
ggplot2	R library	An Implementation of the Grammar of Graphics
markovchain	R library	Easy Handling Discrete Time Markov Chains
forecast	R library	Forecasting Functions for Time Series and Linear Models
purrr	R library	Functional Programming Tools
httr	R library	Tools for Working with URLs and HTTP

Name	Type	Purpose
lubridate	R library	Make Dealing with Dates a Little Easier
rjson	R library	JSON for R
colorama	Python library	Cross-platform colored terminal text
flake8	Python library	Modular source code checker: pep8, pyflakes & co
mypy-lang	Python library	Optional static typing for Python
pip-tools	Python library	Pinned dependencies
pylint	Python library	Python code static checker
pytest	Python library	Python testing framework
requests	Python library	HTTP client library
testthat	R library	R testing framework

3.7.5 Related work

Identical approaches have been identified. AppDynamics [1] proposes the Application Intelligence Platform. It determines the normal performance of an application. It learns the behavior of the application to setup dynamic base-lining. With this result, AppDynamics can automatically detect anomalies without requiring the user to setup thresholds. Prelert [2] adopts the same idea. It implements advanced predictive analytics for accurate alerting without any user's thresholds. It also deploys machine learning techniques to discover root causes when anomalies happen.

Compare to SONATA, these tools are led and supported by strong data scientist / statistical experts. Those solutions are very advanced but, at the same time, very static. The end-user has little customization available. Whereas SONATA aims for a lower goal: less autonomous processes but a total control for the end-user. This is achieved by providing a full-fledged statistical framework with a good integration to the SONATA ecosystem.

[1] `https://www.appdynamics.com`
[2] `http://info.prelert.com`

4 Phase 1 storyboard

This Annex documents the storyboard of the first year of the SONATA project. The goal of this script is to provide a reference framework for both the SONATA Service Platform as well as for the Software Development Kit on how the combined installation, development and deployment process is expected to work for the first phase of the project.The idea behind the storyboard is to highlight features of interest. Its main goal is to ensure consistency among SP and SDK, and to provide a guideline and roadmap for the implementation and demonstration process planned at this stage of the project.

4.1 Implanted Story Steps M10

1. Initial state:
   a) A number of servers are available.
   b) Some run the service plattform.
   c) Others will run the actual services and act as *two* VIMs. They will be running OpenStack.
2. Step: Initiate the service platform
   a) Operator Eve has heard great things about SONATA. She also has a couple of machines and two VIMs available and decides to give it a try.
   b) Introduce the VIMs to the SP, Operator Eve types:
      i. `son-platform add VIM IP-of-VIM-Access ssh-key`
3. Step: Get/create a service
   a) Eve talks to Developer Joe about SONATA. Joe loves it and decides to write a service. For that he creates a workspace in the SONATA SDK and starts a new project using the `son-package` and `son-project` from the SDK.
      i. `son-workspace --init --project prj1`
   b) (optional) Functionality to be included from the catalogue. The catalogue supports storage of NS and VNF descriptors in JSON and YAML format, which can be retrieved via a REST API. Validation of the schema (using `son-schema` validation is executed at this time:
      i. firewalls/iptables, iperf, ...
      ii. Three-tier web application, load balancer, nginx, django as application server, postgresql as database. E.g., in docker containers. Load balancing via DNS? Or nginx directly?
   c) Joe specifies this service using the SONATA schema of the network service and creates a package. At the time of packaging the project, the schema validators of `son-schema` are also executed.

i. `son-package --project prj1`

d) (Optional) Joe wants to use the SONATA emulator to locally test/verify the new service using `son-emu`. In the emulator, basic service chaining between NFs is supported using VLANs.

e) Joe uploads this package to the SP gatekeeper (having obtained a key for it from Eve,'using `son-push`.

i. `son-push gatekeeperId --upload_package packagename`

f) The service is then available to be deployed on demand by the BSS

i. `son-push gatekeeperId --deploy_package packagename`

g) Service triggers VIM to start it up.

h) User of the service, Adam does curls, and look, it works

i. `curl www.newservice.demo`

4. Step: Monitor the running service and the performance of the SP

a) Joe monitors the performance of the service by `son-monitor`. The tool already supports monitoring of basic node characteristics such as CPU, memory and storage, as well as monitoring network-related aspects such as bandwidth and delay.

b) (Optional)Joe can analyze the monitoring information using `son-analyze` to scrutinize the running service.

c) Eve monitors the performance of the SP through Gatekeeper GUI.

4.2 Future Story Steps to be Included in First Prototype

1. Step: Demonstrate ability to introduce a service which is managed by a SSMs

a) Joe writes in addition to the service a simple scaling SSM that will trigger creation of addition web servers

b) Service and its SSM are deployed (see same steps as above)

c) The SSM realizes overload situation

d) The SSM triggers installation of a new web server and reconfigures the chosen load balancing system.

1. Step: Demonstrate ability to introduce a service that its components are managed by FSMs

a) Joe writes in addition to the service a simple FSM that manages the lifecycle of the some of its network functions in s very specific manner

b) Service and its FSM are deployed (see same steps as above)

c) The FSM manages the lifecycle events as it should

4.3 Structure

Figure 4.1 shows a very high level representation of the main components of SONATA's SP architecture that enter the **Year One Storyboard**.

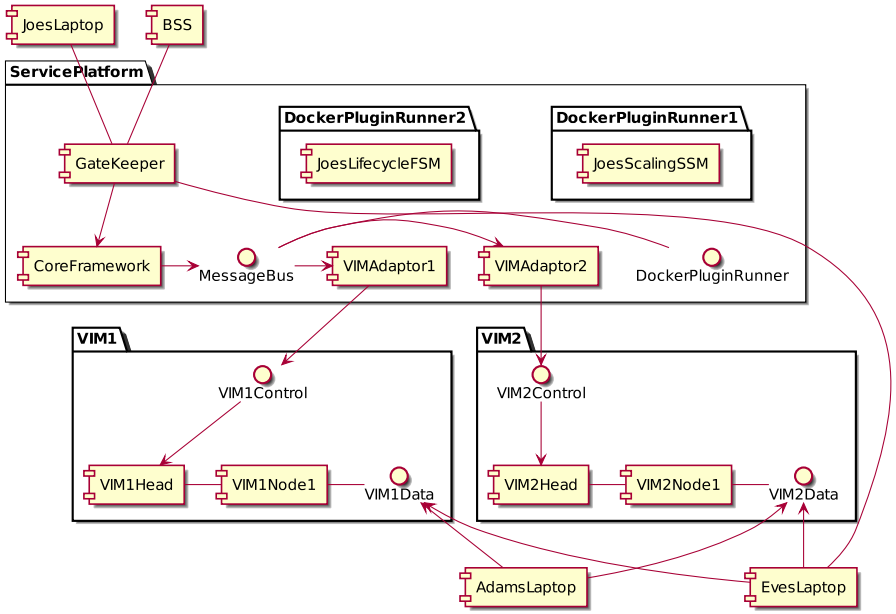

Figure 4.1: Overall structure of SONATA's main components

4.4 Message Sequence Charts

To help understand better the different interactions between all the above mentioned entities, we present in this section, the main message sequence chart.

Figure 4.2 shows an example of how a service could be installed.

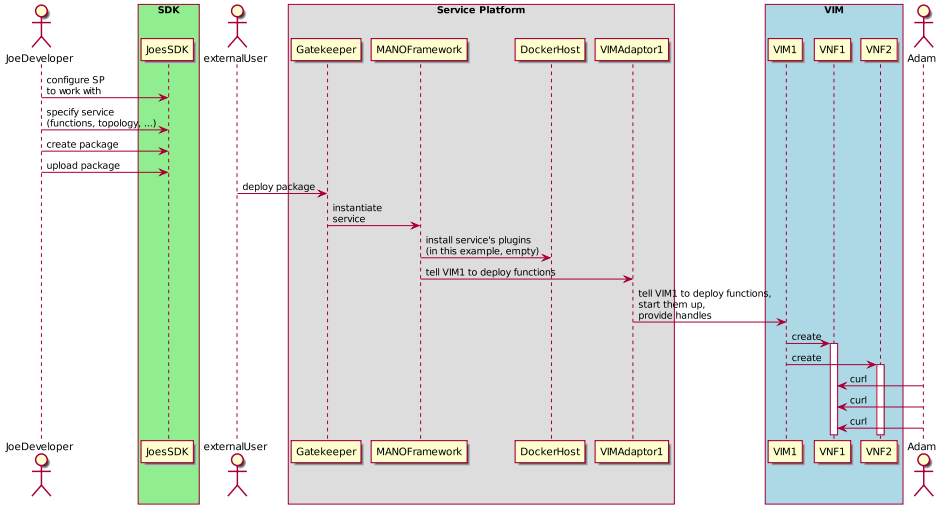

Figure 4.2: Install first service

5 Conclusion

This deliverable documents the architecture, design and usage of the SONATA Software Development Kit after the first phase of the project. The SDK is the **principle set of development tools** to design and test services to be deployed on the SONATA Service Platform as documented in [11]. The SDK is built up as a set of **small independent tools** which can be combined in one or **multiple workflows** to develop a SONATA service. In addition, the deliverable introduces the SONATA **programming model** in the form of appropriate service, network and function descriptors, building further on the outcomes of existing standardization groups such as ETSI NFV or research projects. The tools themselves have been designed in such a way to maximally re-use existing development philosophies, as well as existing technologies. The SDK has been designed in such a way to maximally support **agile software development**, which is currently the preferred and future-proof software development process, enabling rapid cycles of iterative software design, enabling quick transitions between Development and Operational phases (**DevOps**).

The current release of the SDK focuses on the basic tools to **build a workspace and project environment** to build a SONATA service. They can interact with the SONATA **catalogue** in order to fetch already existing services, network functions or network **descriptors**. **Packaging** functionality prepares the service to be **uploaded and deployed on a service platform**. The SDK provides an **emulator** enabling quick deployment in a Docker-based environment on the local developer machine. Initial versions of **monitoring, profiling tools** enable to generate useful data which the developer can use for **analysis** and improvement/debugging of service under development.

Although this release provides the basic toolset to develop and test SONATA services, multiple aspects are still planned for **future work**:

- **Extension of existing tools**: adding new features and supporting additional platforms, as well as improving robustness and security of the existing tools;
- **Development of additional tools**: i) editors and GUI tools to increase the usability of designing new service and network descriptors, ii) a tool to support improved development of individual network functions, and iii) a tool to support the development of scaling and placing algorithms in the form of SSM and FSMs.

A Details of son-schema

The SONATA schemata (son-schema) are used to describe and specify the various descriptors used in SONATA. The schemata that act as ground truth for the whole SONATA project. They are based on the JSON schema specification and can be used to verify formal correctness of descriptors, as a basis for databases, and as a blueprint for code implementations. In the following we describe the schemata for the VNFD, the NSD, and the package descriptor. Further information can be found at [10].

A.1 VNF Descriptor Schema

A VNF Descriptor (VNFD) is a deployment template which describes a VNF in terms of deployment and operational behaviour requirements. The VNFD also contains connectivity, interface and KPIs requirements that can be used by NFV-MANO functional blocks to establish appropriate Virtual Links within the NFVI between VNFC instances, or between a VNF instance and the endpoint interface to other Network Functions. Our function descriptor schema specifies the content of a VNFD. It is based on the T-NOVA [12] flavor of the ETSI VNFD that can be found at [42]. It is, however, adapted and extended to meet the SONATA specific needs.

A.1.1 Sections of the Function Descriptor

Below we discuss the various sections of a network function descriptor. The general descriptor section contains some of the mandatory fields that have to be present in each and every function descriptor. All other sections might be optional.

General Descriptor Section

At the root level, we first have the mandatory fields, that describe and identify the virtual network function in a unique way.

- **descriptor_version** identifies the version of the function descriptor schema that is used to describe the network function.
- **$schema** (optional) provides a link to the schema that is used to describe the network function and can be used to validate the VNF descriptor file. This is related to the original JSON schema specification.

Moreover, the VNF signature, i.e the vendor, the name, and the version, is of great importance as it identifies the VNF uniquely.

- **vendor** will identify the VNF uniquely across all VNF vendors. It should at least be comprised of the reverse domain name that is under your control. Moreover, it might have as many sub-groups as needed. For example: eu.sonata-nfv.nec.
- **name** is the name of the VNF without its version. It can be created with any name written in lower letters or alphanumeric symbols.

- **version** names the version of the VNF descriptor. Any typical version with numbers and dots, such as 1.0, 1.1, and 1.0.1 is allowed here. The VNF version must be increased with any new (changed) instance of the network function descriptor. Please note: The whole network function is composed of the descriptor and other artifacts, like virtual machine images. Thus, the network function may change, even if the description remains constant, just because another artifact changes. This might or might not be reflected in the version of the package descriptor.

The general descriptor section also contains some optional components as outlined below.

- **author** (optional) describes the author of the network function descriptor.
- **description** (optional) provides an arbitrary description of the VNF.

Virtual Deployment Units Section

The virtual deployment unit section contains all the information regarding the VDUs, such as virtual machines and containers, that constitute the virtual network functions. The section is mandatory and starts with:

- **virtual_deployment_units** contains all the virtual deployment units (VDUs) that are handled by the network function.

This section has to have at least one item with the following information:

- **id** represents a unique identifier within the scope of the VNF descriptor.
- **vm_image** (optional) specifies a reference to the virtual machine image (or container) that is used for the virtual network function. The image location can be a local file, a file within a package, a remove location, that might be accessed via HTTP, or a reference within the SONATA service platform.
- **vm_image_format** (optional) specifies the image format, such as raw, vmdk, iso, and docker.
- **vm_image_md5** (optional) represent an MD5 hash of the virtual machine image. It is highly recommended to provide an MD5 hash, not only to verify the image, but to also make versioning of the whole virtual network function easier.
- **resource_requirements** details the resources required by the VDU even further.
- **connection_points** (optional) names the connection points offered by the VDU. The connection points can be used to interconnect various VDUs or to connect the VDU to an VNF connection point and to the outside world.
- **monitoring_parameters** (optional) names the monitoring parameters that are collected for this specific VDU and used, e.g. to trigger scaling operations.
- **scale_in_out** (optional) specifies the minimum and maximum number if VDU instances.

Connection Points Section

- **connection_points** (optional)

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **id** A VNF-unique id of a connection point which can be used for references.
- **type** The type of connection point, such as a virtual port, a virtual NIC address, a physical port, a physical NIC address, or the endpoint of a VPN tunnel.
- **virtual_link_reference** (optional) (deprecated) A reference to a virtual link, i.e. the virtual_links:id.

Virtual Links Section

- **virtual_links** (optional) A VNF internal virtual link interconnects at least two connection points.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **id** A VNF-unique id of the virtual link which can be used for references.
- **connectivity_type** The connectivity type, such as point-to-point, point-to-multipoint, and multipoint-to-multipoint.
- **connection_points_reference** The references to the connection points connected to this virtual link.

VNF Lifecycle Events Section

- **lifecycle_events** (optional) An array that contains VNF workflows for specific lifecycle events such as start, stop, scale_out, update, etc.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **vnf_container** The VNF container that is associated with the lifecycle event.
- **events** The actual event such as start, stop, scale_out, update, etc.

Deployment Flavours Section

- **deployment_flavour** (optional) The flavours of the VNF that can be deployed.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **flavour_key** A VNF-unique id of the deployment flavour which can be used for references.
- **vdu_reference** A reference to the VDU, vdu:id.

Monitoring Rules

- **monitoring_rules** (optional) The rules used for monitoring.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **name** The name of the monitoring rule.
- **description** (optional) An arbitrary description of this monitoring rule.
- **duration** The duration the condition has to be met before an event is fired.
- **duration_unit** (optional) The unit of the duration, such as seconds, minutes, and hours.
- **condition** The condition, i.e. a Boolean expression, that must be met to fire the event.
- **notification** A list of notifications that are fired when the condition is met.

A.2 NS Descriptor Schema

A Network Service Descriptor (NSD) is a deployment template for a Network Service referencing all other descriptors which describe components that are part of that Network Service. Our network service descriptor schema specifies the content of a NSD. It is based on the T-NOVA [2] flavor of the ETSI NSD that can be found at [3]. It is, however, adapted and extended to meet the SONATA specific needs.

A.2.1 Sections of the Service Descriptor

Below we discuss the various section of a network service descriptor. The general descriptor section contains some of the mandatory fields that have to be present in each and every service descriptor. All other sections might be optional.

General Descriptor Section

At the root level, we first have the mandatory fields, that describe and identify the virtual network service in a unique way.

- **descriptor_version** identifies the version of the service descriptor schema that is used to describe the network service.
- **$schema** (optional) provides a link to the schema that is used to describe the network service and can be used to validate the VNF descriptor file. This is related to the original JSON schema specification.

Moreover, the service signature, i.e the vendor, the name, and the version, is of great importance as it identifies the network service uniquely.

- **vendor** will identify the NS uniquely across all NS vendors. It should at least be comprised of the reverse domain name that is under your control. Moreover, it might have as many sub-groups as needed. For example: eu.sonata-nfv.nec.
- **name** is the name of the NS without its version. It can be created with any name written in lower letters or alphanumeric characters.

- **version** names the version of the NS descriptor. Any typical version with numbers and dots, such as 1.0, 1.1, and 1.0.1 is allows here. The NS version must be increased with any new (changed) instance of the network function descriptor. Please note: The whole network service is composed of the descriptor and other artifacts, like VNFs. Thus, the network service may change, even if the description remains constant, just because another artifact changes. This might or might not be reflected in the version of the package descriptor.

The general descriptor section also contains some optional components as outlined below.

- **author** (optional) describes the author of the network service descriptor.
- **description** (optional) provides an arbitrary description of the network service.

Network Functions Section

The network functions section contains all the information regarding the VNFs that constitute the virtual network functions. The section is mandatory and starts with:

- **network_functions** contains all the VNFs that are handled by the network service.

This section has to have at least one item with the following information:

- **vnf_id** represents a unique identifier within the scope of the NSD.
- **vnf_vendor** as part of the primary key, the vendor parameter identifies the VNFD.
- **vnf_name** as part of the primary key, the name parameter identifies the VNFD.
- **vnf_version** as part of the primary key, the version parameter identifies the VNFD.
- **description** (optional) a human-readable description of the VNF.

Connection Points Section

- **connection_points** (optional) The connection points of the overall NS, that connects the NS to the external world.

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **id** An NSD-unique id of a connection point which can be used for references.
- **type** The type of connection point, such as a virtual port, a virtual NIC address, a physical port, a physical NIC address, or the endpoint of a VPN tunnel.
- **virtual_link_reference** (optional) (deprecated) A reference to a virtual link, i.e. the virtual_links:id.

Virtual Links Section

- **virtual_links** (optional) A NS internal virtual link interconnects at least two connection points.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **id** A NS-unique id of the virtual link which can be used for references.
- **connectivity_type** The connectivity type, such as point-to-point, point-to-multipoint, and multipoint-to-multipoint.
- **connection_points_reference** The references to the connection points connected to this virtual link.

Forwarding Graph Section

- **forwarding_graph** The forwarding graph describes the traffic steering through the network service. A network service might have more than one forwarding graph.

This section has to have some of the following information:

- **id** The NS-unique id of the forwarding graph.
- **number_of_endpoints** The number of endpoints of a graph.
- **number_of_virtual_links** The number of virtual links in a graph.
- **constituent_virtual**_links References to the virtual links that constitute the forwarding graph.
- **constituent_vnfs** References to the VNFs that constitute the forwarding graph.
- **network_forwarding_paths** The path, i.e. a concatenation of virtual links and VNFs, of the forwarding graph.

VNF Lifecycle Events Section

- **lifecycle_events** (optional) An array that contains VNF workflows for specific lifecycle events such as start, stop, Scale_out, update, etc.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **start** The start event, executed whenever the network service starts.
- **stop** The stop event, executed when the network service stops.
- **scale_out** The scale-out event, when the network service is scaled out.
- **scale_in** The scale-in event, when the network service is scaled in.

Auto-Scale Policy Section

- **auto_scale_policy** (optional) The auto-scale policy connects monitoring event with actions that are executed when some given criteria are met.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **criteria** The criteria that have to be met to execute the given action.
- **action** A list of actions that are execute when the criteria is met.

Monitoring Parameters Section

- **monitoring_parameters** (optional) The parameters used for monitoring.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **description** A human-readable description of the monitoring parameter.
- **metric** The metric to measure. The metric has to be supported by the service platform.
- **unit** The unit in which the metric is measured.

A.3 Package Descriptor Schema

The package descriptor file specifies the content, artifacts, and dependencies of a SONATA package. The corresponding schema file specifies the structure of the package descriptor. It makes sure the relevant information is provided to parse the package in a meaningful way. It can be used to validate the package descriptor file.

A.3.1 Sections of the Package Descriptor

Below we discuss the various section of a package descriptor. The general descriptor section contains some of the mandatory fields that have to be present in each and every package descriptor. All other sections are optional.

General Descriptor Section

On the root level, the general descriptor section contains the mandatory fields required in the package descriptor.

- **descriptor_version** identifies the version of the package descriptor schema that is used to specify the file structure.
- **$schema** (optional) provides a link to the schema that is used to specify the file structure and can be used to validate the package descriptor file. This is related to the original JSON schema specification.

Moreover, the package signature, i.e. the package_group, the package_name, and the package_version, is of great importance, as it identifies the package uniquely.

Best practices for creating the signature can be derived from the Java Maven naming conventions for groupId, artifactId, and version. To this end, the package_group, the package_name, and the package_version should be named as follows:

- **package_group** will identify the package uniquely across all packages. It should at least be comprised of the reverse domain name that is under your control. Moreover, it might have as many sub-groups as needed. For example: eu.sonata-nfv.nec.
- **package_name** is the name of the package without its version. It can be created with any name written in lower letters or alphanumeric characters.
- **package_version** names the version of the package. Any typical version with numbers and dots, such as 1.0, 1.1, and 1.0.1 is allows here. The package version must be increased with any new (changed) instance of the service.

The general descriptor section also contains some optional components as outlined below.

- **package_maintainer** (optional) describes the maintainer of the package, like John Doe, NEC.
- **package_description** (optional) provides an arbitrary description of the package.
- **package_md5** (optional) provides an MD5 hash over the package content, i.e. all files contained in the package EXCEPT the package descriptor, i.e. /META-INF/MANIFEST.MF, as this file contains this hash.
- **package_signature** (optional) provides a cryptographical signature over the package content, i.e. all files contained in the package EXCEPT the package descriptor, i.e. /META-INF/MANIFEST.MF. Thus, a package customer can verify the integrity and the origin of the package.
- **entry_service_template** (optional) specifies THE service template of this package. In general, the package can contain more than one network service descriptor as dependencies. In order to identify the descriptor that describes the service of this package, it has to be named here.
- **sealed** (optional) is a Boolean value that states whether this package is self-contained, i.e. it already contains all its relevant artifacts (true), or it has external dependencies that may have to be provided from somewhere else. Default is false.

Package Content Section

The package content section contains all the artifacts that are contained and shipped by the package. The section is optional and starts with:

- **package_content** (optional) holds an array of artifacts contained in the package

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **name** specifies the path to the resource in the package.
- **content_type** specifies the type of content, like application/vnd.sonata.service_template
- **md5** (optional) specifies an MD5 hash of the resource.
- **sealed** (optional) overrides the default sealed value specified in the general descriptor section on a per-artifact basis.

Package Resolver Section

The package resolver sections contain information about catalogues and repositories needed to resolve the dependencies specified in this package descriptor. This information might be used in addition to the default catalogues and repositories configured already on the service platform (or the SDK). The section is optional and starts with:

- **package_resolvers** (optional) holds an array of catalogues used to resolve dependencies and download additional packages.

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **name specifies** the path to the catalogue.
- **credentials** (optional) provides the credentials that might be needed to access the catalogue.

Package Dependencies Section

In the package dependencies section, one can specify additional packages this package depends up on. The packages are automatically downloaded from the various catalogues provided either by default from the service platform or as configured in the package resolver section. The section is optional and starts with:

- **package_dependencies** (optional) holds an array of packages this packages depends up on.

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **name** specifies the name of the package, similar to the name of this package.
- **group** specifies the name of the group, similar to the group name of this package.
- **version** specifies the version or version ranges of the package that is needed. For example one can specify the exact version like, 1.0.1-beta, but also ranges such as >= 1.0, = 1.0 && < 2.0, 1.1 || 1.2, etc.
- **credentials** (optional) provides the credentials that might be needed to use this package.
- **verification_key** provides the public key of the package maintainer to verify the package.

Artifact Dependencies Section

- **artifact_dependencies** (optional)

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **name specifies** the name of the artifact to download.
- **url** specifies the URL where to download the package from. Moreover, there needs to be a protocol handler that is able to download the artifact using the given protocol, like HTTP. For now, we only support HTTP and HTTPS. Thus, the URL has to start with one of these protocols.
- **credentials** (optional) provides the credentials that might be needed to download the artifact.

B Manual of son-cli tools

This manual provides a usage guide for the son-cli tools.

B.1 son-workspace

The son-workspace tool is responsible for creating a development environment, which can be shared to create and maintain multiple projects. For this reason, it is recommended to create the workspace at a neutral location, e.g. in user space. Typically, a workspace belongs to a specific developer, whereas a project may be shared by multiple developers.

The son-workspace tool receives the following arguments:

```
usage: son-workspace [-h] [--init] [--workspace WORKSPACE] [--project PROJECT]
                     [--debug]

  -h, --help            show this help message and exit
  --init                Create a new sonata workspace
  --workspace WORKSPACE location of existing (or new) workspace. If not
                        specified will assume '$HOME/.son-workspace'
  --project PROJECT     create a new project at the specified location
  --debug               increases logging level to debug
```

To create and initialize a new workspace execute the following command:

```
son-workspace --init --workspace /workspace/path
```

To create a new project, based on the created workspace, execute the following:

```
son-workspace --workspace /workspace/path --project /project/path
```

The `--workspace` argument can be omitted, in which case the workspace will be created at `.son-workspace` in the user home directory. Moreover, a workspace and project can be instantiated in one single command. For example, to create a new project `prj1` with a workspace at the default location, invoke:

```
son-workspace --init --project prj1
```

After the initialization of a workspace, a default workspace configuration file is provided, containing dummy parameters as examples. The workspace configuration file is `workspace.yml` located at the workspace root directory. The following paragraphs describe each configuration parameter in detail.

`name:` Representative name of the workspace.
`log_level:` Granularity of log messages during the execution. This affects all son-cli tools. The following levels are accepted: `DEBUG`; `INFO`; `WARNING`; `ERROR`; `CRITICAL`. (Default value: `INFO`)
`descriptor_extension:` Extension of descriptor files. (Default value: `yml`)

`catalogues_dir`: Location of descriptor's cache storage. Used to store temporary descriptors when retrieved from private catalogues. (Default value: `catalogues`)

`catalogue_servers`: Specifies catalogue servers from which the developer wants to retrieve external components. It also specifies the default catalogues used to publish components. Each catalogue server entry is configured with the following parameters:

`id`: Identification of catalogue server

> `url`: Address of the catalogue server, containing the port number of the service (e.g. `http://catalogueserver.com/srv1:4011`)
>
> `publish`: Boolean parameter which defines if, by default, the catalogue should be used for publishing project components. A value of `'yes'` or `'no'` must be declared.

`schemas_local_master`: The local directory to cache retrieved schema templates from the son-schema repository. (Default value: `$HOME/.son-schema`)

`schemas_remote_master`: The URL that specifies the son-schema repository address.

B.2 son-package

The son-package tool is used to create a container file of all project components, to be pushed to the SP Gatekeeper. The packaging process involves the verification and retrieval of dependencies, the syntax validation of descriptors and the validation of the package itself.

The son-package tool receives the following arguments:

```
usage: son-package [-h] [--workspace WORKSPACE] [--project PROJECT]
                   [-d DESTINATION] [-n NAME]

  -h, --help            show this help message and exit
  --workspace WORKSPACE Specify workspace to generate the package. If not
                        specified will assume '$HOME/.son-workspace'
  --project PROJECT     create a new package based on the project at the
                        specified location. If not specified will assume the
                        current directory.
  -d DESTINATION, --destination DESTINATION
                        create the package on the specified location
  -n NAME, --name NAME  create the package with the specific name
```

The catalogue servers from which `son-package` will retrieve external dependencies, as well as the schema templates server, are defined at the workspace configuration. Thus it is only necessary to indicate the workspace and the project to package.

For instance, in order to package a project named `prj1` based on the configuration of a workspace at the default location, simply run:

```
son-package --project prj1
```

Or to specify a workspace located at a different location, invoke:

```
son-package --workspace /workspace/path --project prj1
```

B.3 son-publish

To son-publish tool allows the developer to publish project components or even an entire project to private catalogues.

The son-publish tool receives the following arguments:

```
  -h, --help            show this help message and exit
  --workspace WORKSPACE Specify workspace. Default is located at
                        '$HOME/.son-workspace'
  --project PROJECT     Specify project to be published
  -d COMPONENT, --component COMPONENT
                        Project component to be published.
  -c CATALOGUE, --catalogue CATALOGUE
                        Catalogue ID where to publish. Overrides defaults in
                        workspace config.
```

For example, in order to publish a VNF descriptor named `firewall-vnfd.yml`, execute:

```
son-publish --component firewall-vnfd.yml --catalogue cat1
```

Please note that the catalogue ID `cat1` must be specified in the workspace configuration. On the other hand, if the argument `--catalogue` is not specified, the workspace configuration must have at least one catalogue server with the parameter `publish = 'yes'`.

B.4 son-push

The son-push tool enables the developer to interact with the gatekeeper of either the SONATA Service Platform or with the (emulated) gatekeeper of the SDK emulator. It is a command-line tool which can be used as follows:

```
$ son-push --help
usage: son-push [-h] [-P] [-I] [-U UPLOAD_PACKAGE] [-D DEPLOY_PACKAGE_UUID]
                platform_url

    Push packages to the SONATA service platform/emulator or list packages/instances
    available on the SONATA platform/emulator.


positional arguments:
  platform_url          url of the gatekeeper/platform/emulator

optional arguments:
  -h, --help            show this help message and exit
  -P, --list_packages   List packages uploaded to the platform
  -I, --list_instances  List deployed packages on the platform
  -U UPLOAD_PACKAGE, --upload_package UPLOAD_PACKAGE
                        Filename incl. path of package to be uploaded
  -D DEPLOY_PACKAGE_UUID, --deploy_package_uuid DEPLOY_PACKAGE_UUID
                        UUID of package to be deployed (must be available at
                        platform)
```

A couple of examples of the above functionality can be found below (when used in combination with the gatekeeper of the SDK emulator).

```
son-push http://127.0.0.1:5000 -U sonata-demo.son
son-push http://127.0.0.1:5000 --list-packages
son-push http://127.0.0.1:5000 --deploy_package <uuid>
son-push http://127.0.0.1:5000 -I
```

C Manual of son-catalog

The current version of the son-catalogue features multiple implemented methods in the API. These methods allow using some key object attributes (such ID, name, version...) for each core functionalities that enable testing the Catalogues as standalone or with other SONATA plugins with NS and VNF Descriptors only. SDK Catalogue allows to be used as a already installed remote solution on the NCSRD infrastructure (Integration infrastructure) which requires VPN connectivity, or as a local solution, which requires to be installed and configured.

These guidelines are intended to be used for LOCAL and STANDALONE testings. It is highly recommended to make use of the next tools:

- **Robomongo 0.9.0** - Application for MongoDB visual management
- **POSTMAN** - Chrome Plugin for HTTP communication

SDK Catalogue provides a installation script that will automatically install a fresh MongoDB instance with required fresh databases for each type of Descriptors. But first is critical to tweak a script component before running installation script, and then, once installation have finished, some configuration changes are required. Then, to test the SDK Catalogue, follow next steps:

1. Download source code from **son-catalogue code from Github repository**
2. Open and edit script file "dbs.js" found on root folder son-catalogue/.
3. On "dbs.js" file, comment and uncomment lines this way:

```
//db = connect("mongo:27017/ns_catalogue");
//db.createCollection("ns");
//db = connect("mongo:27017/vnf_catalogue");
//db.createCollection("vnfs");
//db = connect("mongo:27017/pd_catalogue");
//db.createCollection("pd");
/* Uncomment next lines if MongoDB installation will be done in localhost,
 * and comment lines above */
db = connect("127.0.0.1:27017/ns_catalogue");
db.createCollection("ns");
db = connect("127.0.0.1:27017/vnf_catalogue");
db.createCollection("vnfs");
db = connect("127.0.0.1:27017/pd_catalogue");
db.createCollection("pd");
```

4. Save changes and close "dbs.js" file.
5. Run "installation_mongodb.sh" script file found on root folder `son-catalogue/`.
6. When installation is completed, open and edit MongoDB configuration file "mongoid.yml" found on son-catalogue/son-sdk-catalogue/config/ folder.
7. Edit "mongoid.yml" config file, uncommenting localhost addresses and commenting the line above, twice for the whole file, this way:

```
#- mongo:27017
- 127.0.0.1:27017
```

8. Then start SDK Catalogue API server (use sudo in case it is required):

```
rake start
```

The workflow for testing the Catalogue is simple, once you have downloaded the source code and deployed a MongoDB (from fresh install or through the source code script), you can connect to the MongoDB using the Robomongo application. It enables the user to a have a graphic user interface to check and manage contents of the database while testing Catalogue API methods with different Descriptors. Then, using POSTMAN plugin, Descriptors can be sent using a POST request or retrieved sending a GET request to the `son-catalogue` API with the next contents:

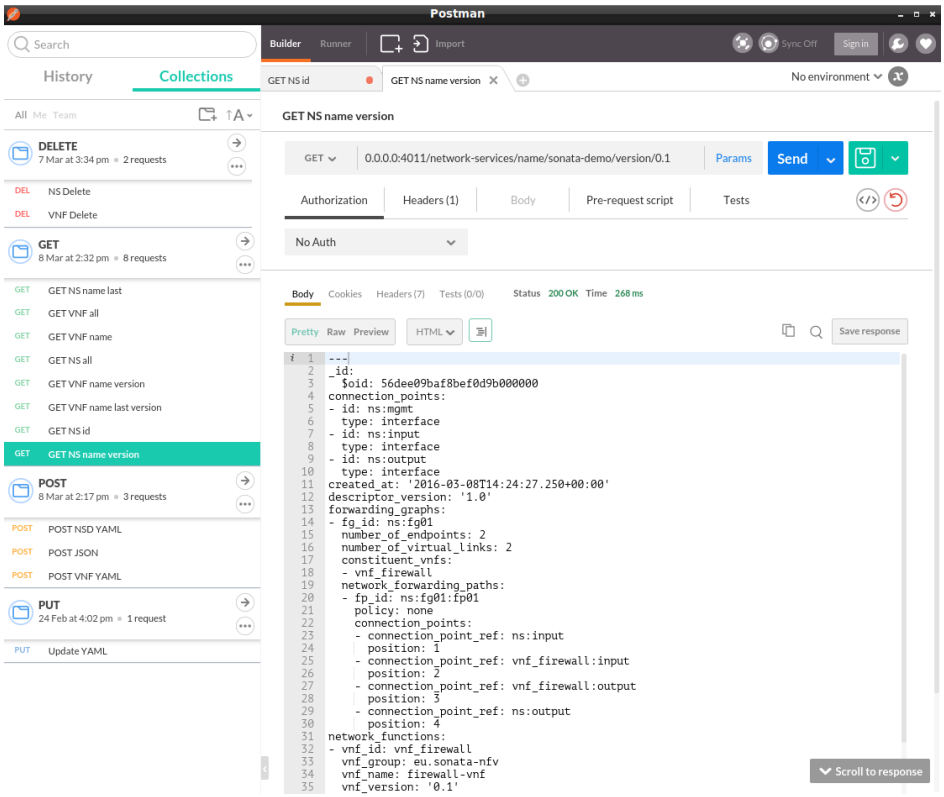

Figure C.1: Postman screenshot

If database is empty (it will be empty on a fresh install), nothing will be returned on a GET request. To fill the database with NS or VNF Descriptors and then get some descriptor back, the POST method must be run in first place. Descriptor data will be sent on the body of the request, in JSON or YAML format. The next steps can be applied for NS, VNF and Packages descriptors.

Set POSTMAN fields like:

- Method: POST
- Address: 0.0.0.0:4011/network-services

For a YAML format descriptor, set:

- Headers: Content-type application/x-yaml

For a JSON format descriptor, set:

- Headers: Content-type application/json
- Body (raw): Copy the contents of the YAML example file in source code folder 'samples'. Headers must be set to YAML
- Apply the Send button

If the request success, then a 200 OK code will be returned from the Catalogue, along with a copy of the Descriptor with an identifier assigned to it in a new field '_id' which will include a "vendor.name.version" identifier of the descriptor.

To receive a list of available NS Descriptors on the Catalogue, you can set POSTMAN fields:

- Method: GET
- Address: 0.0.0.0:4011/network-services (assuming IP address and port has not changed)
- Headers: Content-type application/x-yaml or application/json (set this field in the format you desire to receive the Descriptors)
- Apply the Send button

To update any of the available NS Descriptors on the Catalogue, similar to the POST, you can set POSTMAN fields:

- Method: PUT
- Address: 0.0.0.0:4011/network-services/id/:id where :id is the "vendor.name.version" identifier of a stored descriptor

For a YAML format descriptor, set:

- Headers: Content-type application/x-yaml

For a JSON format descriptor, set:

- Headers: Content-type application/json
- Body (raw): Copy the contents of the YAML example file in source code folder 'samples', and modify the Version key field increasing its value. Header must be set to YAML
- Apply the Send button

In case of need to delete some NS Descriptor entries from the database, you can use the Robomongo directly in the application and it will remove the selected document, or make a new operation such:

- Method: DELETE
- Address: 0.0.0.0:4011/network-services/id/:id (assuming IP address and port has not changed)(:id is the identifier attribute of the Descriptor)
- Apply the Send button

D Manual of son-emu

This sections briefly describes how `son-emu` can be used to prototype and test complex network services.

D.1 Starting the Emulator

First, a developer has to define (or generate) a multi-PoP topology in which network services should be tested. This is done by describing the topology in a Python script using an extended Mininet API. The following example shows a topology with two interconnected PoPs:

```
"""
Simple son-emu example topology definition.
"""
import logging
from dcemulator.net import DCNetwork
from api.zerorpcapi import ZeroRpcApiEndpoint

logging.basicConfig(level=logging.INFO)


def create_topology1():
    """
    1. Create a network object (distributed cloud network)
    """
    net = DCNetwork()

    """
    1b. Add endpoint APIs for the whole DCNetwork,
        to access and control the networking from outside.
        e.g., to setup forwarding paths between compute
        instances aka. VNFs (represented by Docker containers), passing through
        different switches and datacenters of the emulated topology
    """
    net_api = ZeroRpcApiEndpointDCNetwork("0.0.0.0", 5151)
    net_api.connectDCNetwork(net)
    net_api.start()

    """
    2. Add data centers (PoPs) to the topology
    """
    dc1 = net.addDatacenter("dc1")
    dc2 = net.addDatacenter("dc2")
```

```
33      dc3 = net.addDatacenter("dc3")
34      dc4 = net.addDatacenter("dc4")
35
36      """
37      3. (Optionally) Add additional SDN switches for data center
38         interconnections to the network.
39      """
40      s1 = net.addSwitch("s1")
41
42      """
43      4. Add links between your data centers and additional switches
44         to define you topology.
45         These links can use Mininet's features to limit bw, add delay or jitter.
46      """
47      net.addLink(dc1, dc2, delay="50ms")
48      net.addLink(dc1, s1)
49      net.addLink(s1, dc3, loss=0.01)
50      net.addLink(s1, dc4)
51
52      """
53      5. Add endpoint APIs, to access and control the data centers from the outside,
54         e.g., to connect an orchestrator to start/stop compute
55         instances aka. VNFs (represented by Docker containers)
56      """
57      # create a new instance of an endpoint implementation
58      zapi1 = ZeroRpcApiEndpoint("0.0.0.0", 4242)
59      # connect data centers to this endpoint
60      zapi1.connectDatacenter(dc1)
61      zapi1.connectDatacenter(dc2)
62      # run API endpoint server (in another thread, don't block)
63      zapi1.start()
64
65      """
66      5.1. For our example, we create a second endpoint to illustrate that
67           this is supported by son-emu. This feature allows us to have
68           one API endpoint for each data center. This makes the emulation
69           environment more realistic because you can easily create one
70           OpenStack-like REST API endpoint for *each* data center.
71           This will look like a real-world multi PoP/data center deployment
72           from the perspective of an orchestrator.
73      """
74      zapi2 = ZeroRpcApiEndpoint("0.0.0.0", 4343)
75      zapi2.connectDatacenter(dc3)
76      zapi2.connectDatacenter(dc4)
77      zapi2.start()
78
79      """
80      6. Start the defined network (the emulator) and enter its
```

```
81      interactive CLI. Stop the emulator when the user types 'exit'.
82      """
83      net.start()
84      net.CLI()
85      net.stop()
86
87  if __name__ == '__main__':
88      create_topology1()
```

After a developer has specified a topology script he can execute it to start the emulator:

```
sudo python <name-of-topology-file>.py
```

Executing the emulator requires root privileges since the system is based on Mininet/Containernet which manipulate the network configurations of the host machine to emulate the described network. Thus, it is recommended that the emulator is executed in a separated VM on the developer's machine. The emulator will start up and enter its interactive CLI, like shown in Figure D.1. After this, `son-emu` is ready to accept requests to instantiate single VNFs or entire services. This process is described in the next section.

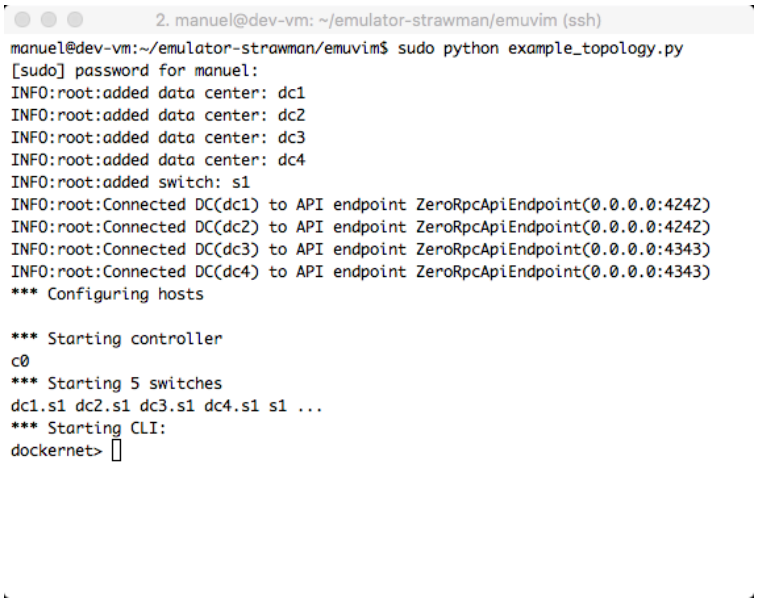

Figure D.1: Running emulator with interactive Containernet CLI

D.2 Controlling an Emulation

There are two possible ways to start VNFs inside the running emulator. The first is to start single VNFs manually and the second uses a module named *dummy gatekeeper* to which entire service packages can be pushed by using the `son-push` tool.

D.2.1 Manually Deploy VNFs

A developer can use the `son-emu-cli` tool to directly interact with a running emulator and start VNFs. For this, the used Docker images have to be pre-build and pre-registered to the Docker

service on the emulation host. The `son-emu-cli` tool is divided in several sub-modules and a developer has to use the `compute` module to manage single VNF instances as follows:

```
son-emu-cli compute -h
usage: son-emu-cli [-h] [--datacenter DATACENTER] [--name NAME]
                   [--image IMAGE] [--dcmd DOCKER_COMMAND] [--net NETWORK]
                   {start,stop,list,status}

son-emu compute

positional arguments:
  {start,stop,list,status}
                        Action to be executed.

optional arguments:
  -h, --help            show this help message and exit
  --datacenter DATACENTER, -d DATACENTER
                        Data center to in which the compute instance should be
                        executed
  --name NAME, -n NAME  Name of compute instance e.g. 'vnf1'
  --image IMAGE, -i IMAGE
                        Name of container image to be used e.g.
                        'ubuntu:trusty'
  --dcmd DOCKER_COMMAND, -c DOCKER_COMMAND
                        Startup command of the container e.g. './start.sh'
  --net NETWORK         Network properties of compute instance e.g.
                        '10.0.0.123/8' or '10.0.0.123/8,11.0.0.123/24' for
                        multiple interfaces.
```

D.2.1.1 Examples

- Run a empty Docker container using the default *ubuntu* image in PoP with name *dc1*. Give it the name *vnf1*:

```
son-emu-cli compute start -d dc1 -n vnf1
```

- Specify the image that should be executes:

```
son-emu-cli compute start -d dc1 -n vnf2 --image my_vnf:v1
```

- Add a custom startup command for the Docker container:

```
son-emu-cli compute start -d dc1 -n vnf3 --dcmd ./entrypoint.sh
```

- Add and specify the network interfaces for the Docker container:

```
son-emu-cli compute start -d dc1 -n vnf4 -i vnf4_image  \
--net '(id=input,ip=10.0.10.3/24),(id=output,ip=10.0.10.4/24)'
```

- List all running containers:

```
son-emu-cli compute list
```

- The list command will output a table with all running VNFs, like shown in Figure D.2.

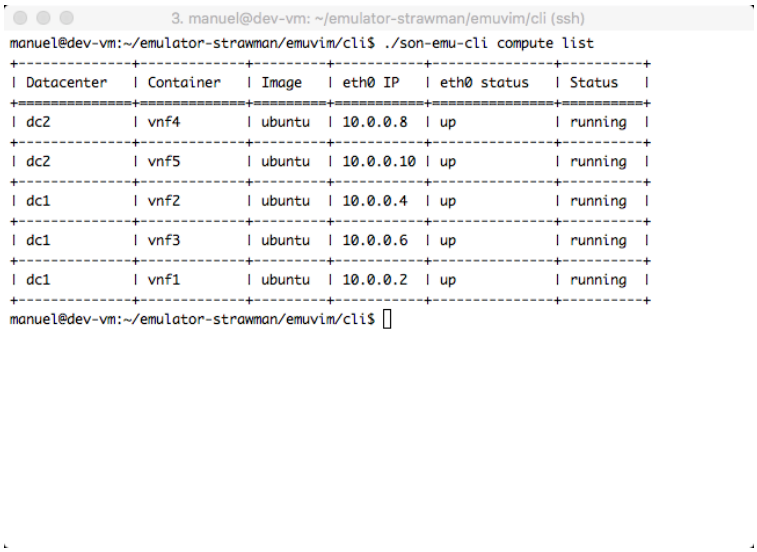

Figure D.2: Output of son-emu-cli comute list

D.2.2 Manually Deploy Forwarding Chains

A developer can use the `son-emu-cli` tool to directly interact with a running emulator and link any started VNFs. The VNFs/Docker images have to be up and running (started by the commands in the previous section). The `son-emu-cli` tool is divided in several sub-modules and a developer has to use the `network` module to manage forwarding chains as follows:

```
son-emu-cli network -h
usage: son-emu-cli [-h] [--datacenter DATACENTER] [--source SOURCE]
                   [--destination DESTINATION] [--weight WEIGHT]
                   [--match MATCH] [--bidirectional] [--cookie COOKIE]
                   {add,remove}

son-emu network

positional arguments:
  {add,remove}          Action to be executed.

optional arguments:
  -h, --help            show this help message and exit
  --datacenter DATACENTER, -d DATACENTER
                        Data center to in which the network action should be
                        initiated
  --source SOURCE, -src SOURCE
                        vnf name of the source of the chain
  --destination DESTINATION, -dst DESTINATION
                        vnf name of the destination of the chain
  --weight WEIGHT, -w WEIGHT
                        weight metric to calculate the path
  --match MATCH, -m MATCH
                        string holding extra matches for the flow entries
```

```
  --bidirectional, -b   add/remove the flow entries from src to dst and back
  --cookie COOKIE, -c COOKIE
                        cookie for this flow, as easy to use identifier (eg.
                        per tenant/service)
```

D.2.2.1 Examples

- Link *vnf1* to *vnf2*, setup the forwarding path in both directions (connects by default the first interface of the vnf) :

```
son-emu-cli network add -src vnf1 -dst vnf2 -b
```

- Link *vnf1* to *vnf2*, only connect interface *vnf1:output* to *vnf2:input* and assign this flow a cookie = 10 :

```
son-emu-cli network add -src vnf1:output -dst vnf2:input --cookie 10
```

- Link interface *vnf1:output* to *vnf2:input* and assign a cookie = 11 and extra match fields for the flow to only match udp traffic to port 5001:

```
son-emu-cli network add -src vnf1:output -dst vnf2:input --cookie 11 \
--match 'dl_type=0x0800,nw_proto=17,udp_dst=5001'
```

- Remove an earlier defined chain whose flows are identified by cookie = 11:

```
son-emu-cli network remove -src vnf1:output -dst vnf2:input --cookie 11
```

- The installed flows can be monitored by using the standard ovs commands:

Below listing shows the flows installed with cookie 10 and 11 by the commands above. Also a vlan id is added to the flow, as identifier for the chain. VLAN tags are used as identifier to differentiate chains in multiple emulated services. At each instantiation of a flow, a unique vlan number (in range 0:4096) is chosen for this particular chain of flows. The vlan tag is added at the first switch port where the chain source is connected to, and removed again when it outputs from the last switch at the chain destination.

```
ovs-ofctl dump-flows dc1.s1

NXST_FLOW reply (xid=0x4):
  cookie=0xa, duration=36.752s, table=0, n_packets=5, n_bytes=390, idle_age=36,
priority=0, in_port=2 actions=load:0->OXM_OF_VLAN_VID[],output:1
  cookie=0xb, duration=15.757s, table=0, n_packets=0, n_bytes=0, idle_age=15,
priority=0, udp,in_port=2,tp_dst=5001 actions=load:0x1->OXM_OF_VLAN_VID[],output:1
```

D.2.3 Automatically Deploy a SONATA Service Package

To be able to deploy SONATA service packages on son-emu, a developer has to specify a topology and assign an instance of the *SonataDummyGatekeeperEndpoint* to each PoP, like shown in the following listing.

```
# create topology
net = DCNetwork()
dc1 = net.addDatacenter("dc1")
dc2 = net.addDatacenter("dc2")
s1 = net.addSwitch("s1")
net.addLink(dc1, s1, delay="10ms")
net.addLink(dc2, s1, delay="20ms")

# add the SONATA dummy gatekeeper to each DC
sdkg1 = SonataDummyGatekeeperEndpoint("0.0.0.0", 5000)
sdkg1.connectDatacenter(dc1)
sdkg1.connectDatacenter(dc2)
# run the dummy gatekeeper (in another thread, don't block)
sdkg1.start()

# start the emulation platform
net.start()
net.CLI()
net.stop()
```

If this script is executed, the dummy gatekeeper module is instantiated and awaits HTTP REST requests on port 5000, just like the real gatekeeper of WP4's service platform does. This interface is compatible to the original gatekeeper's interfaces and offers the functionalities to upload (on-board) and instantiate (start) a predefined SONATA service package. When a package is uploaded, the dummy gatekeeper will extract the references from its VNFDs pointing to Docker images in a Docker repository, like `registry.sonata-nfv.eu:5000`. The dummy gatekeeper will then pull such images and register them in the local image store of the Docker installation from which they can be instantiated. To upload a service package, a developer can use the son-push tool as follows:

```
son-push http://127.0.0.1:5000 -U sonata-demo.son
```

This call will return an UUID as reference to the uploaded service which can then be used to instantiate the service, by executing:

```
son-push http://127.0.0.1:5000 -D <uuid>
```

Note: The dummy gatekeeper module is a temporary module that will be removed when son-emu is integrated with WP4's service platform in one of the next releases.

D.3 Interacting with single VNFs

A developer can interact with running VNFs (containers) at any time by using Containernet's interactive CLI, like shown in the following or by directly connecting to the Docker containers by using the Docker CLI `attach` command.

```
containernet> vnf1 ping -c 2 vnf2
containernet> vnf1 ifconfig
containernet> vnf1 cat /tmp/mylog.log
```

D.4 Screencast

There is a short screen cast available that showcases how an example SONATA service package can be deployed on son-emu by using the son-push tool. The video is publicly available: `https://youtu.be/BgWDp5CM0io`.

E Manual of son-monitor

This Annex describes in more detail how the `son-monitor` tools are used.

E.1 Export selected monitored metrics

A developer can use the `son-emu-cli` tool to directly interact with a running emulator and monitor any started VNFs. The commands to startup and link VNF are documented in Section 3.4. To be compatible with the monitoring framework of the SONATA SP, the metrics gathered in the emulator are exported to a Prometheus pushgateway. Metrics can be exported from dedicated monitoring agents or parameters by default available in the emulator's framework such as flow/port packet counters in SDN switches and compute/memory/storage metrics of the deployed containers. The VNFs/Docker images have to be up and running (started by the commands in the previous section). The `son-emu-cli` tool is divided in several sub-modules and a developer has to use the `monitor` module to export monitored metrics as follows:

```
son-emu-cli monitor -h
usage: son-emu-cli [-h] [--vnf_name VNF_NAME] [--metric METRIC]
                   [--cookie COOKIE] [--query QUERY] [--datacenter DATACENTER]
                   {setup_metric,stop_metric,setup_flow,stop_flow,prometheus}

son-emu monitor

positional arguments:
  {setup_metric,stop_metric,setup_flow,stop_flow,prometheus}
                        setup/stop a metric/flow to be monitored or query
                        Prometheus


optional arguments:
  -h, --help            show this help message and exit
  --vnf_name VNF_NAME, -vnf VNF_NAME
                        vnf name:interface to be monitored
  --metric METRIC, -m METRIC
                        tx_bytes, rx_bytes, tx_packets, rx_packets
  --cookie COOKIE, -c COOKIE
                        flow cookie to monitor
  --query QUERY, -q QUERY
                        prometheus query
  --datacenter DATACENTER, -d DATACENTER
                        Data center where the vnf is deployed
```

E.1.0.1 Examples

- Export the rx_packet count of interface 'input' of 'vnf1' :

```
son-emu-cli monitor setup_metric -vnf vnf1:input --metric rx_packets
```

- Stop exporting the rx_bytes count of interface 'input' of 'vnf1' :

```
son-emu-cli monitor stop_metric -vnf vnf1:input --metric rx_bytes
```

- Stop exporting all metrics of 'vnf1' :

```
son-emu-cli monitor stop_metric -vnf vnf1
```

- Export the transmitted packets count of the flow with cookie = 10, which inputs to the port where the interface 'input' of 'vnf1' is connected to:

```
son-emu-cli monitor setup_flow -vnf vnf1:input --metric tx_packets --cookie 10
```

- Stop exporting the transmitted packets count of the flow with cookie = 10, which inputs to the port where the interface 'input' of 'vnf1' is connected to:

```
son-emu-cli monitor stop_flow -vnf vnf1:input --metric tx_packets --cookie 10
```

- Query monitoring data from the prometheus server which is gathering metrics from the emulator.

the identifier in the command is replaced by the uuid which is assigned by the emulator when vnf1 is deployed.

```
son-emu-cli monitor prometheus -d datacenter1 -vnf vnf1 \
-q 'sum(rate(container_cpu_usage_seconds_total{id="/docker/<uuid>"}[10s]))'
```

E.2 Extract a Performance Profile of a VNF

- start a vnf called 'vnf1' with 2 network interfaces (called 'input' and 'output'):

The profiling command is currently implemented into the `son-emu` toolset, but will be moved into the `son-cli` repository, where it can be installed and executed from the SDK workspace.

-in / -out Specify the input and output interface of the vnf to connect respectively the traffic source and sink to.

Future integration will implement additional arguments to specify:

--vim Specify which VIM to use to test the vnf (currently only the emulator is possible).

--flavor specify which resources can be assigned to the vnf for this test.

```
son-emu-cli compute profile -d datacenter1 -n vnf1 -i vnf1_image \
--net '(id=input),(id=output)' -in input -out output
```

F Manual of son-analyze

F.1 Supported commands

At the current version, `son-analyze` supports the following commands:

```
user@foobar:~/workspace$ son-analyze --help
usage: son-analyze [-h] [-v VERBOSE] {version,bootstrap,run} ...

An analysis framework creation tool for Sonata

positional arguments:
  {version,bootstrap,run}
    version             Show the version
    bootstrap           Bootstrap son-analyze
    run                 Run an environment

optional arguments:
  -h, --help            show this help message and exit
  -v VERBOSE, --verbose VERBOSE
                        increase verbosity
```

F.1.1 `son-analyze bootstrap`

This command initializes the current host with the analysis framework's required components. It downloads the official RStudio Docker image. Then it extends this base image with the SONATA R library and some required external R libraries.

```
user@foobar:~/workspace$ son-analyze bootstrap
> b'{"stream":"Step 1 : FROM rocker/hadleyverse:latest\\n"}\r\n'
> b'{"stream":" ---\\u003e 2a039f703dad\\n"}\r\n'
> b'{"stream":"Step 2 : ENV TZ \'UTC\'\\n"}\r\n'
> b'{"stream":" ---\\u003e Using cache\\n"}\r\n'
> b'{"stream":" ---\\u003e 2795401f28c3\\n"}\r\n'
> b'{"stream":"Step 3 : RUN echo \\"TZ=\'UTC\'\\" \\u003e\\u003e
               /usr/lib/R/etc/Renviron
               \\u0026\\u0026 echo \'to_install \\u003c- c(\\"devtools\\", ...
               cat /tmp/cmds.R | r -     \\u0026\\u0026 echo \'Done\'\\n"}\r\n'
> b'{"stream":" ---\\u003e Using cache\\n"}\r\n'
> b'{"stream":" ---\\u003e a94fb1487229\\n"}\r\n'
> b'{"stream":"Step 4 : COPY ./examples /home/rstudio/examples\\n"}\r\n'
> b'{"stream":" ---\\u003e Using cache\\n"}\r\n'
> b'{"stream":" ---\\u003e 27ff498c01cc\\n"}\r\n'
> b'{"stream":"Step 5 : COPY ./son.analyze /var/tmp/son.analyze\\n"}\r\n'
```

```
> b'{"stream":" ---\\u003e Using cache\\n"}\r\n'
> b'{"stream":" ---\\u003e 319a4b052e86\\n"}\r\n'
> b'{"stream":"Step 6 : RUN echo \'setwd(\\"/var/tmp/son.analyze\\"); ... \\n"}\r\n'
> b'{"stream":" ---\\u003e Using cache\\n"}\r\n'
> b'{"stream":" ---\\u003e 03e2cf8dbf92\\n"}\r\n'
> b'{"stream":"Successfully built 03e2cf8dbf92\\n"}\r\n'
```

F.1.2 son-analyze run

This command starts an analysis session. The end-user can implement his own analysis scripts in the running session. As long as this command is running, the session is accessible in the end-user's browser.

```
user@foobar:~/workspace$ son-analyze run
Browse http://localhost:8787
The default username/password is: rstudio/rstudio
Type Ctrl-C to exit
```

F.2 Code snippets

F.2.1 Query to Sonata

```
library(son.analyze)

containerId <- "60971b1b72638b6e9d31e4522415be55388739ef7640ef32bfe046e53a781f7f"
step <- 5

r <- query(rangeQuery("http://sp.int2.sonata-nfv.eu:9090",
                      sprintf("cnt_cpu_perc{id='%s'}", containerId),
                      interval = lubridate::as.interval(lubridate::hours(-1),
                                                        lubridate::now()),
                      step = paste0(step, "s")))


with(r[[1]], {
  x <- xts::xts(df, order.by = df$Timestamp)
  plot(x$Value, t = "l", main = "")
  title(main = metricName, sub = id)
})
```

G Abbreviations

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

CM Configuration Management

CRUD Create, Read, Update, Delete

CSAR Cloud Service ARchive (i.e., TOSCA package)

DevOps Development and Operations

DSL Domain-Specific Language

ETSI European Telecommunications Standards Institute

FSM(D) Function-Specific Manager (Descriptor)

GUI Graphical User Interface

IDE Integrated Development Environment

JSON JavaScript Object Notation (i.e., data interchange format)

KPI Key Performance Indicator

MANO Management and Orchestration

NF Network Function

NFV Network Function Virtualization

NFVI-PoP Network Function Virtualisation Points of Presence

NFVO Network Function Virtualization Orchestrator

NFVRG Network Function Virtualization Research Group

NS(D) Network Service (Descriptor)

NSH Network Service Header

OASIS Organization for the Advancement of Structured Information Standards

OSS Operations Support System

REST Representational State Transfer

RPC Remote Procedure Call

SDK Software Development Kit

SDN Software-Defined Networking or Software-Defined Network

SLA Service Level Agreement

SNMP Simple Network Management Protocol

SP Service Platform

SSM(D) Service-Specific Manager (Descriptor)

TOSCA Topology and Orchestration Specification for Cloud Applications

UUID Universally Unique Identifier

VDU Virtual Deployment Unit

VIM Virtual Infrastructure Manager

VLAN Virtual Local Area Network

VLD Virtual Link Descriptor

VM Virtual Machine

VN Virtual Network

VNF Virtual Network Function (Desciptor)

VNFFGD VNF Forwarding Graph Descriptor

VNFM Virtual Network Function Manager

WAN Wide Area Network

WIM Wide area network Infrastructure Manager

XMPP Extensible Messaging and Presence Protocol

YAML YAML Ain't Markup Language (i.e., data interchange format)

H Bibliography

[1] ZeroRpc: An easy to use, intuitive, and cross-language Rpc. Website, 2016. Online at `http://www.zerorpc.io/`.

[2] J. Ahrenholz. Comparison of CorE network emulation platforms. In *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, pages 166–171, Oct 2010.

[3] 5G Infrastructure Association. 5G vision: The 5G infrastructure Public Private Partnership: the next generation of communication networks and services. `https://5g-ppp.eu/wp-content/uploads/2015/02/5G-Vision-Brochure-v1.pdf`, 2015. Online, accessed 13/05/2015.

[4] Google cadvisor. cAdvisor - Analyzes resource usage and performance characteristics of running containers. Website, 2016. Online at `https://github.com/google/cadvisor`.

[5] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

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

[7] SONATA Consortium. Sonata deliverable 2.1: Use cases and requirements.

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

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

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

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

[12] T-NOVA consortium. D3.1: Orchestrator interfaces. Website, September 2015. Online at `http://www.t-nova.eu/wp-content/uploads/2016/03/TNOVA_D3.1_Orchestrator_Interfaces_v1.0.pdf`.

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

[14] Bence Eros, Zsigmond Czine, and Andrej Kazakov. Json schema validator. Website, September 2015. Online at `http://www.everit.org`.

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

[16] Apache Software Foundation. Apache Maven. Website, 2016. Online at `https://maven.apache.org/`.

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

[19] Robert Grossman, Yunhong Gu, Michal Sabala, Collin Bennet, Jonathan Seidman, and Joe Mambratti. The open cloud testbed: A wide area testbed for cloud computing utilizing high performance network services. *arXiv preprint arXiv:0907.4810*, 2009.

[20] J Halpern and C Pignataro. Service Function Chaining (Sfc) Architecture. Technical report, 2015.

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

[22] Chavee Issariyapat, Panita Pongpaibool, Sophon Mongkolluksame, and Koonlachat Meesublak. Using Nagios as a groundwork for developing a better network monitoring system. In *Technology Management for Emerging Technologies (PICMET), 2012 Proceedings of PICMET'12:*, pages 2771–2777. IEEE, 2012.

[23] Matthias Keller, Christoph Robbert, and Manuel Peuster. An Evaluation Testbed for Adaptive, Topology-aware Deployment of Elastic Applications. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 469–470. ACM, 2013.

[24] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[25] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, 2015.

[26] Heikki Mahkonen, Ravi Manghirmalani, Meral Shirazipour, Ming Xia, and Attila Takacs. Elastic network monitoring with virtual probes. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 1–3. IEEE, 2015.

[27] Lefteris Mamatas, Stuart Clayman, and Alex Galis. A service-aware virtualized software-defined infrastructure. *Communications Magazine, IEEE*, 53(4):166–174, 2015.

[28] István Pelle, Tamás Lévai, Felicián Németh, and András Gulyás. One tool to rule them all: A modular troubleshooting framework for Sdn (and other) networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 24. ACM, 2015.

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

[30] OpenStack Project. OpenStack DevStack. Website, 2016. Online at `http://docs.openstack.org/developer/devstack/`.

[31] Prometheus Project. Prometheus - Monitoring system & time series database. Website, 2016. Online at `https://prometheus.io/`.

[32] Ryu project. Ryu integration with Openstack. Website, 2016. Online at `http://ryu.readthedocs.io/en/latest/using_with_openstack.html`.

[33] T-NOVA project. T-nOva- Network Functions as-a-Service over Virtualised Infrastructures. Website, 2015. Online at `http://www.t-nova.eu/`.

[34] Meral Shirazipour, Heikki Mahkonen, Ming Xia, Ravi Manghirmalani, Attila Takacs, and Veronica Sanchez Vega. A monitoring framework at layer4??? 7 granularity using network service headers. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 54–60. IEEE, 2015.

[35] Andrey Somov. Snakeyaml - a yaml processor for java. Website, March 2009. Online at `http://www.snakeyaml.org/`.

[36] Balázs Sonkoly, János Czentye, Robert Szabo, Dávid Jocha, János Elek, Sahel Sahhaf, Wouter Tavernier, and Fulvio Risso. Multi-domain service orchestration over networks and clouds: a unified approach. In *Proceedings of the 2015 ACM conference on SIGCOMM*. ACM, 2015.

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

[38] OASIS TOSCA. Topology and orchestration specification for cloud applications. Website, November 2013. Online at `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html`.

[39] OASIS TOSCA. Tosca simple profile for network functions virtualization. Website, March 2016. Online at `http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/csd03/tosca-nfv-v1.0-csd03.pdf`.

[40] Robbie Vanbrabant. Google guice: Agile lightweight dependency injection framework, 2008. Online at `https://github.com/google/guice`.

[41] P. Wette, M. Dräxler, and A. Schwabe. MaxiNet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP*, pages 1–9, June 2014.

[42] ETSI NFV WG. Network functions virtualisation (nfv); management and orchestration. Website, 12 2014. Online at `http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf`.

[43] W. Zhao, Y. Peng, F. Xie, and Z. Dai. Modeling and simulation of cloud computing: A review. In *Cloud Computing Congress (APCloudCC), 2012 IEEE Asia Pacific*, pages 20–24, 2012.