D3.3 SONATA SDK final release

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	D3.3 SONATA SDK final release
Workpackage	WP3 Service Programmability and Toolset
Due Date	June 30th, 2017
Submission Date	June 29th, 2017
Version	1.0
Status	To be approved by EC
Editor	Wouter Tavernier (imec)
Contributors	Wouter Tavernier, Steven Van Rossem (imec), Luis Conceiçao, Tiago Batista (UBI), Michael Bredel (NEC), Geoffroy Chollon (TCS), Daniel Guija, Muhammad Shuaib Siddiqui (i2CAT), Manuel Peuster, Hadi Razzaghi Kouchaksaraei (UPB)
Reviewer(s)	Phil Eardley (BT)

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)	

Disclaimer:

This document has been produced in the context of the SONATA Project. The research leading to these results has received funding from the European Community's 5G-PPP under grant agreement n° 671517.
All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.
For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors' view.

Executive Summary:

This deliverable documents the final release of the SONATA SDK. The core philosophy has remained the same: the **SDK as a set of loosely coupled, light-weight tools** helping the NFV developer in developing Virtual Network Functions (VNFs) and network services composed of VNFs. The developed service target the SONATA Service Platform as developed in WP4. Following a **DevOps philosophy**, the development and deployment process are well-integrated and enable multiple cycles alternating the development (SDK) and operational (SP) environment in a seamless manner.

The foundations of the SONATA SDK have been laid out in prior phases of the project, and still following the same workflow involving workspace and project creation, service composition, package construction, service onboarding and deployment, service monitoring and profiling, and/or statistical analysis, possibly leading to another cycle. In this release, the SDK has been extended with **new components for service validation** (syntactical as well as structural/semantic validation) and associated visualization (`son-validate`), as well as an **integrated graphical user interface for composing NFV services** (`son-editor`). Besides these new components, the SDK has incorporate a significant number of additional features and improvements to the existing tools. These improvements largely focus on **extensive security support**, as well as increasing the **usability** and **extensibility** of the global SDK solution.

Increased security refers to the ability of the SDK (and in particular of the `son-access` component) to authenticate and authorize users, to sign service packages, and to validate the resulting signature.

The **emulator** (`son-emu`), unique in the NFV landscape, is now not only capable of supporting SONATA's own Service Platform, but also **supports the MANO solution provided by the OSM project**. In addition, the emulator can now be used for a range of important development tasks for the NFV developer: modifying and evaluating the placement process, and/or assessing potential service scaling alternatives. The latter is strongly supported by the i**mproved monitoring and profiling components**, enabling to test both functional as well as performance-related characteristics of services and VNFs.

Furthermore, all SDK tools have been improved and revised in order to increase the usability and extensibility of the global solution. This is important in order to maximally attract external users, enabling the SDK software to remain actively used and extended beyond the duration of the project. This begins by reducing the installation barrier as low as possible, and is continued by adding graphical user interfaces where possible, or providing base examples for each component feature.

Although this document refers to the final release of the SONATA SDK, this does not imply that the SDK will stop to be further refined, extended and improved. However, this deliverable is the last formal documentation point in the context of the project. Future SDK updates will be documented as part of the corresponding GitHub code repositories or in external documents.

Contents

List of Figures

List of Tables

1 Introduction

This deliverable documents the final release of the SONATA SDK. It builds further on the SDK foundations as documented in D3.1 [3] and refined in D3.2 [4]. The core philosophy has remained the same: the SDK as a set of loosely coupled, light-weight tools helping the NFV developer in developing Virtual Network Functions (VNFs) and network services composed of VNFs. The SDK is mainly focused as a set of tools targeting the releases of the SONATA Service Platform as documented in D4.1 [5], D4.2 [6] and D4.3 [7]. The core interface between the SDK and the SP remains to be the interface to the Gatekeeper.

The release of the SDK mainly focuses on improvements of the existing tools, including extensive security support, and increasing the usability and extensibility of the global solution. This is important in order to maximally attract external users, enabling the SDK software to remain actively used and extended beyond the duration of the project. The focus on security, usability and extensibility is reflected in the addition of a user-friendly and attractive editor environment, support for authentication and signatures in the project and packaging tools, as well as increased interfaces for the emulator part of the SDK.

Although this document refers to the final release of the SONATA SDK, this does not imply that the SDK will stop to be further refined, extended and improved. However, this deliverable is the last formal documentation point in the context of the project. Future SDK updates will be documented as part of the corresponding GitHub code repositories or in external documents.

1.1 Structure of the deliverable

In order to keep consistency with the deliverables documenting earlier SDK release, we have kept the document structure very similar to D3.1 and D3.2. The Section 2 recapitulates the main SDK workflow and identifies the most important changes to the SDK toolset. The next section clarifies the SDK installation process, to make the barrier as low as possible for new users. Next, the main design aspects of new components and updates to existing SDK components are described in Section 4. This involves characterization of adequate schema and descriptors, tools for setting up the development work space and project space, catalogues, the packaging tool, the emulator, the monitoring, analysis tool, as well as the novel editor tool. In order to maximize the sustainability of the SDK toolset, Section 5 documents how future SDK updates will be documented, and how external developers can contribute to the software. Finally, Section 6 concludes the document, capturing the most important updates of the SDK design. In order to increase the accessibility and usability of the developed tools, updates of manuals and instructions of each of the components also have been included in the appendices of the deliverable.

2 Workflow and characteristics of the revised SDK

The latest SDK release has kept the main SDK structure as a set of light-weight (CLI-focused) tools assisting the NFV developer. These SDK design choices are documented in D3.1 [3] and updated in D3.2 [4]. This release keeps the same workflow as previously, although it has been extended with some additional SDK tools. The global workflow is depicted in Figure 2.1. On the upper part we see the SONATA Service Platform, which is the ultimate target platform for the developed services with the SDK. On the lower part we see the SONATA SDK, and its associated toolset organised from left to right. This workflow has been focusing on a DevOps process from the very start, and therefore allows to seamlessly transition from one environment to another in an iterative way. The canonical workflow starts on the local developer's computer by creating a workspace/project using `son-workspace` and `son-project`. Next, the developer can fetch service components such as VM or Docker images, as well as associated descriptors from external sources, which can be adapted and composed using the (newly designed) graphical user interface `son-editor`. The resulting service or network function subsequently can be packaged using `son-package` such that it can be deployed. Using the updated `son-access` tool, the package can be onboarded on either the local SDK emulator (the development environment, using `son-emu`) or the Service Platform (which ultimately will be the operational environment). In the local deployment, the SDK provides two tools: `son-monitor` and `son-profile` which enrich the SDK with powerful features in order to test the composed service and its components in a quantitative manner, storing indicated metrics in the appropriate database, and providing invaluable performance data guiding the developer to further improvements. The data in the database can even be further analyzed using advanced statistical tools such as R statistical environment or Python Jupyter scientific libraries using the `son-analyze` tool. The DevOps circle can now be closed be modifying/editing the service using the SDK tools in order to trigger another cycle.

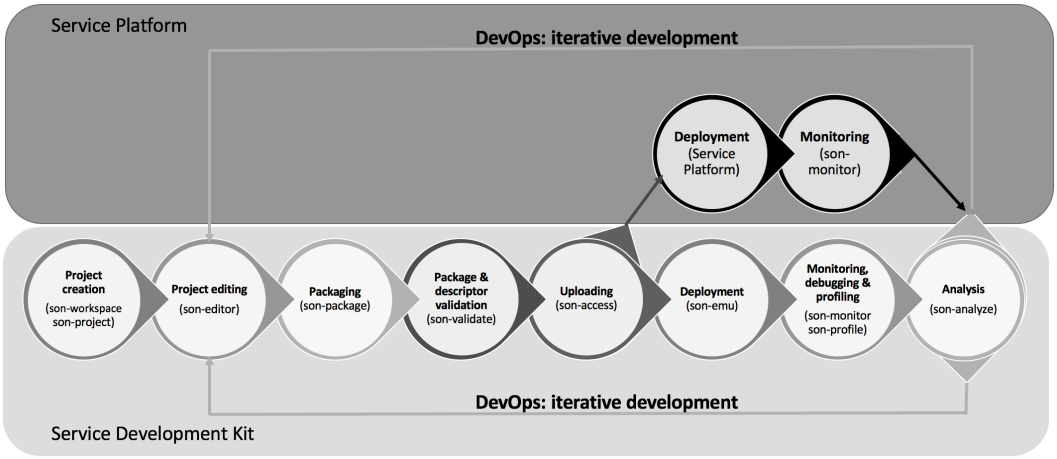

Figure 2.1: SDK workflow

For the final release of the SONATA SDK, particular attention has been paid to three aspects:

usability, security and extensibility. The main philosophy behind this focus is to maximise the sustainability of the developed toolset beyond the duration of the SONATA project. Below, we briefly clarify how we interpret these aspects and clarifies how we have incorporated them into the SDK.

2.1 Usability

The usability of software has a high impact on the adoption of it. New users therefore must be maximally encouraged and likewise minimally discouraged to try the software. This release therefore has included a low-barrier installation process, as well as a graphical editor tool, and a GUI for the son-validate tool.

As the software installation defines the first interaction with the SDK software, the installation process has been streamlined among all components, documented in a dedicated section in this document, and a full all-in virtual machine setup has been prepared. New users/developers can easily download and deploy the virtual machine on their local laptop, and follow the accompanying documentation and step-by-step guide to get started with the SDK. This process has been successfully demonstrated at a tutorial given at the SOFTNETWORKING 2017 conference.

Once the SDK has been successfully installed, a developer has to become acquainted with the descriptors, following predefined SONATA schema. To ease the roll-in into this process, this SDK release now includes a graphical user-interface enabling to load existing VNF- and service examples. These examples are pre-loaded in the aforementioned SDK VM installation, or can be downloaded from the SONATA son-examples repository. GitHub interaction has been maximally integrated into the son-editor tool, enabling to re-use tools and environments that software developers are used to. Once this first stage has been crossed, advanced SONATA developers may fall back on the well-tested process following the powerful SDK workflow recapitulated above.

Next, the `son-validate` tool has been extended significantly in order to detect a range of potential syntactical or semantic service/VNF errors, but also with a graphical user interface, enabling to indicate in a visually appealing and multi-level manner where issues occur in the service description.

2.2 Security

Telecom services are used in a range of situations (e.g., hospitals) where reliability and security are of utmost importance. Whereas previous SDK releases largely focused on core functionality, and maximizing the flexibility and power of the existing SDK tools, this release includes core security features, focusing on user management, authentication, authorization, verification and package signing.

Most of the security features are provided directly or indirectly in the update of the `son-access` component. This release has now built-in support for keypair generation, platform authentication, package signing and package verification. The authentication features of son-access are well-integrated with the other components of the SDK. User credentials can be stored in the workspace configuration of son-workspace. This enables authentication of the SDK within the Service Platform via son-access, generating a token stored with limited lifespan in the local workspace. Other SDK tools, such as `son-monitor` can now use the token within the authentication header of the calls to the Service Platform.

2.3 Extensibility

In order to remain usable, software needs to evolve with respect to the changing environment. One way to enable future software evolution is to adequately prepare the software architecture. The SDK has enabled this in multiple ways: i) by following an approach of loosely coupled tools which can interact on the same project, ii) by preparing SDK tools with adequate interfaces, enabling them to support alternate platforms. The latter is illustrated in particular for the SDK emulator, as it now not only supports the SONATA Service Platform, but also the OSM MANO framework. The emulator architecture is structured in such a way that in can be further extended towards other MANO platforms in the future.

Next to an adequate software architecture, adequate information and processes must be available to inform potential contributors/developers on how to extend the software, and how to contribute to the project. As SONATA has been following a strong CI/CD approach with clear rules on how to contribute, this process is already documented in the context of WP5, but SDK-specific aspects have been recapitulated in section Section 5 of this document. This should minimize the barrier for potential contributors to adapt or extend the available SDK software.

3 SDK Installation

The SONATA SDK is envisioned as a light-weight set of software tools. They can be installed on a single laptop, allowing a service developer to locally create and debug network services. The table below gives an overview of the different SDK components, with their GitHub repository where detailed installation instructions are provided. According to the developer's needs, all or a limited number of components can be installed.

Table 3.1: SONATA SDK components

SDK component	GitHub repository	Description
son-cli	`https://github.com/sonata-nfv/son-cli`	Extensive toolset that creates a workspace including functions for validation, monitoring, profiling and packaging of NFV-based services.
son-emu	`https://github.com/sonata-nfv/son-emu`	Emulation environment to deploy services on an emulated infrastructure topology.
son-editor	`https://github.com/sonata-nfv/son-editor-backend`	Graphical web-based tool to create and edit service packages.
son-analyze	`https://github.com/sonata-nfv/son-analyze`	Analysis tools to process the monitored service data.
son-examples	`https://github.com/sonata-nfv/son-examples`	Set of example services that can be used to demo or try-out the SONATA environment.
son-sm	`https://github.com/sonata-nfv/son-sm`	Tools to support FSM/SSM development

Given the modular approach of the SDK architecture and the clear interfaces between the different components, a distributed SDK deployment is also supported. In this sense, the `son-emu` SONATA emulator might be a good candidate to install on a separate server or even implement as a dedicated Infrastructure Node under a MANO platform. The implemented `son-emu` REST API and monitoring system certainly support such a deployment.

3.1 SDK installation script

An automated SDK installation script is provided in the form of a Vagrant script [11]. This script builds and configures a VM where all necessary SDK tools and some examples are pre-installed. This VM offers an isolated sandbox environment , creating a low entry-level to try and use the SONATA SDK. All details are available in our public GitHub repository [1].

[1] `https://github.com/sonata-nfv/son-tutorials/tree/master/demo_vm_SDK`

4 SDK Component Design

This release of the SDK consists of updates to existing schema and components, as well as the introduction of some new components. The main new features are: the introduction of authentication features into son-access, the integration of these features into the other son-cli tools, the extensive validation tool son-validate, a graphical GUI and service editing tool son-editor, and a range of new monitoring, debugging and profiling features. Each of these novelties are documented below.

4.1 son-schema

The SONATA schemata are used to specify the various descriptors used by the SONATA system. They are based on the ETSI NFV descriptor specification.

4.1.1 Interfaces

In the latest version of son-schema, the interfaces have been slightly modified to support even more types of interfaces. To this end, it is possible to specify *Ethernet*, *IPv4*, and *IPv6* interfaces. Based on the interface type, the Service Platform configures the interface accordingly.

4.1.2 Connections Points

Closely related to Interface specification described above, is the description of Connection Points. Connections Points can now be of a specific type, that is *internal*, *external*, or *management*. Based on that type, the Service Platform configures the Connection Point accordingly. To this end, Connections Points of type *management* are used for management and configuration connections to a Network Service or VNF. Thus, it usually connects the SSM or the FSM to the service or function respectively. Internal Connection Points interconnect either VDUs inside a VNF or various VNFs inside a Network Service. Likewise external connection points connect to the outside world. In case of a Network Service external Connection Point, the CP can be accessed from other network services or used, e.g. by using a floating IP (accessible from the public Internet).

4.1.3 Licences

The latest version of the descriptors support a preliminary licence management with public and private licences. To this end, the descriptors contain a new licence field that is interpreted by the Service Platform, namely the Gatekeeper. The field states whether the descriptor is published under a private or public licence. Moreover, it contains an URI to the actual licence file. If the licence is marked a public, the descriptor may be used publicly by any party. However, if the descriptor is marked as private, any use must hold the correct licence key. While this implementation of licensing in the descriptors is very generic, the actual - and possibly complex - licence handling is left to the service platform.

4.1.4 Compatibility with ETSI and OSM

In the following, we analyse the SONATA and the OSM descriptors, i.e. VNFD and NSD, and identify the similarities and differences. This analysis provides the bases for a translation tool (son-translate) that aims at translating the SONATA descriptors into OSM descriptors. Using such a translator tool, the SONATA SDK can be of use for OSM as well. As mentioned earlier already, the SONATA descriptors are specified in a machine-readable format using JSON schema and can be found in the SONATA GitHub repositories. The OSM descriptors are defined in the related OSM information model for Release 2 document.

VNFD Comparison - First Level

In Table 4.1 we compare the first level of the two VNFDs. We present the key names, the key types, a brief description, and state whether the key is mandatory or not.

Table 4.1: A comparison of the first level keys of the VNFDs.

SONATA Key Name	SONATA Type	OSM Key Name	OSM Type	Description
-	-	id	string	Identifier for the VNFD. SONATA identifies the VNFD by vendor-name-version tuple. Thus, there is no ID. The VNFR however, creates an SP-internal ID.
name	string	name	string	The name of the VNF.
-	-	short-name	string	Short name to use as a label in the UI.
vendor	string	vendor	string	The name of the provider of the VNFD.
-	-	logo	string	File path to the VNF-specific logo.
description	string	description	string	A human readable description of the VNF.
version	string	version	string	The Version of the VNF.
-	-	vnf-configuration	object	Information about the VNF configuration for the management interface.
-	-	mgmt-interface	object	Interface over which the VNF is managed.
virtual_links	array	internal-vld	array	A list of internal virtual link descriptors
connections_points	array	connection-point	array	A list of external connection points.
virtual_deployment_units	array	vdu	array	A list of virtual deployment units.
-	-	vdu-dependency	array	List of VDU dependencies.
-	-	service-function-chain	enum	Type of node in service function chaining architecture.
-	-	service-function-type	string	Type of service function.
monitoring_rules	array	monitoring_param	array	List of monitoring parameters.
-	-	placement_groups	array	Placement group construct to define the compute resource placement strategy in cloud environment.

VNFD Comparison - Second Level

In the following we present the comparisons of the various second level objects of the VNFDs, such as virtual-link objects, connection-point objects, virtual-deployment-unit objects, etc.

Table 4.2: A comparison of the virtual-link object keys.

SONATA Key Name	SONATA Type	OSM Key Name	OSM Type	Description
id	string	id	string	VNFD-unique identifier for the internal VLD.
-	-	name	string	The name of the internal VLD.
-	-	short-name	string	A short name to use as a label in the UI.
-	-	description	string	Description of the internal VLD.
connectivity_type	enum	type	enum	The type of the virtual link.

SONATA Key Name	SONATA Type	OSM Key Name	OSM Type	Description
-	-	root-bandwidth	integer	The aggregated bandwidth for ELANs.
-	-	leaf-bandwidth	integer	The bandwidth of the branches in ELANs.
connection_points_reference	reference	internal-connection-point-ref	reference	Reference to one or more internal connection points.
-	-	provider-network	object	Information about the provider network.

Table 4.3: A comparison of the connection-point object keys.

SONATA Key Name	SONATA Type	OSM Key Name	OSM Type	Description
id	string	id	string	Unique identifier of the connection point.
-	-	name	string	Name of the connection point.
-	-	short-name	string	Short name to use as a label in the UI.
type	enum	type	enum	The type of the connection point.

Table 4.4: A comparison of the virtual-deployment-units object keys.

SONATA Key Name	SONATA Type	OSM Key Name	OSM Type	Description
id	string	id	string	A unique identifier for the VDU.
-	-	name	string	A unique name for the VDU.
-	-	description	string	A description of the VDU.
-	-	count	integer	The number of VDU instances.
-	-	mgmt-vpci	string	The address of the virtual PCI bus.
resource_requirements	object	-	-	Additional information that identifies the flavor of the VM instance.
vm_image	reference	image	string	Image name of the software image.
vm_image_md5	string	image-checksum	string	The MD5 checksum of the VM image.
connection_points	array	internal-connection-point	array	List of connection points.

Table 4.5: A comparison of the monitoring-rules object keys.

SONATA Key Name	SONATA Type	OSM Key Name	OSM Type	Description
-	-	http-endpoint	array	List of HTTP endpoints to be used by monitoring parameters.
-	-	monitoring-param	array	List of monitoring parameters.
-	-	monitoring-param-ui-data	array	Description of monitoring parameters for UI data.
-	-	monitoring-param-value	array	values for the monitoring parameters.

NSD Comparison - First Level

In Table 4.6 we compare the first level of the two NSDs. We present they key names, the key types, a brief description, and state whether the key is mandatory or not.

Table 4.6: A comparison of the first level keys of the VNFDs.

SONATA Key Name	SONATA Type	OSM Key Name	OSM Type	Description
-	-	id	string	The identifier for the NSD.
name	string	name	string	The name of the network service.

SONATA Key Name	SONATA Type	OSM Key Name	OSM Type	Description
-	-	short-name	string	A short name to use as a label in the UI.
vendor	string	vendor	string	The vendor of the network service.
-	-	logo	string	File path to a network service specific logo.
description	string	description	string	A description of the network service.
version	string	version	string	The version of the VNFD.
connection_points	array	connection-point	array	A list of all external connection points.
virtual_links	array	vld	array	A list of all NS-internal virtual link interconnects.
-	-	placement-groups	array	A list of placement groups.
monitoring_parameters	array	monitoring-param	array	A list of monitoring parameters for the network service.
network_functions	array	constituent_vnfd	array	A list of VNFDs that are part of this service.
-	-	scaling-group-descriptor	array	Scaling group descriptor within this network service.
-	-	ip-profiles-list	array	A list of IP profiles.
-	-	vnf-dependency	array	A list of VNF dependencies.
forwarding_graphs	array	vnffgd	array	A list of forwarding graph descriptors.
-	-	input-parameter-xpath	array	A list of xpath parameters inside the NSD that can be customized during instantiation.
-	-	parameter-pool	array	Parameters that can be used for configuration.
-	-	service-primitive	array	Network service level configuration primitives.
-	-	initial-config-primitive	array	Initial set of configuration primitives for the NSD.

4.2 son-cli

This section describes the updates, enhancements and incremental work for the SONATA `son-cli` SDK command line interface tools, a SDK component meant to assist the SONATA service developers on their tasks. It includes a series of tools to cover critical points of service development within SONATA SDK. This section also presents implementation details for new features and improvements of `son-cli` tools for the SONATA final release version. Overall information on the component can be found in deliverables D3.1 [3], D3.2 [4] and D2.2 [1].

4.2.1 son-access

The `son-access` was component introduced to SONATA Year 2 plan for the SDK side which was responsible of replacing and merging some key functionalities of the SDK command line tools. `son-access` has become a key component inside the `son-cli` as the SDK interface that enables not only communication to the SONATA Service Platform (SP), but also to other components such `son-emu`. Latest updates focus on new functionalities for authorization and authentication from SDK side to the SP Gatekeeper.

The `son-access` component provides a secured connection based on authentication and authorization processes between SDK end-users and the Service Platform, which offers possibilities to use end-user credentials and JSON Web Tokens to access the Service Platform features such the unified SP Catalogue and enable end-users to submit and request package files and descriptors from the SP Catalogue.

4.2.1.1 Improvements

The `son-access` component changes include improvements, integration changes and implementation of some new features listed in the section below. The component main architecture remains the same, however it has been integrated with other components from `son-cli` and the SP.

Current architecture is shown in Figure 4.1 which presents `son-access` internal modules with the complete workflow between each component:

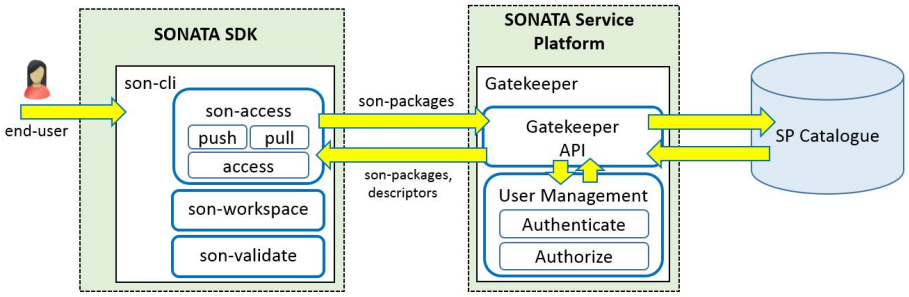

Figure 4.1: SDK and SP Catalogues components interaction workflows

Access This sub-component implements the security layer (authentication and authorization of end-users) to the communications with the Service Platform and performs automatically required authentication processes for each interaction. The following improvements have been made in this sub-component:

- Integration with other `son-cli` components, specially with `son-workspace`, which has become a dependency to this sub-component as it includes all the end-user and configuration related settings, such user credentials, workspace directories and key locations. It currently supports shared use of the access token between different SDK tools for the same end-user.
- Improvements for Access Token management, automatically including Access Token in each message header to authenticate and authorize developers.
- Fully integrated with the Gatekeeper API to perform authentication and authorization processes, but can also work without authentication and authorization processes to keep compatibility with platforms that do not enables a authentication and authorization layer.
- Completed implementation to perform the following main functionalities: List available resources in the SP, submit package files, request resources from the SP, configure the access parameters and authenticate the end-user.

Push This sub-component is fully integrated with the other `son-access` sub-components and presents some changes in order to implement new features:

- It enables submission of Package files `son-package` including Service and Function descriptors to the SP Catalogue. The Gatekeeper API requires package files as input, however the `son-access` is still able to submit descriptors. This is useful in order to be able to connect the `son-access` to other platforms that accepting this input element or `son-emu`.

- Fully implements and enhances former `son-push` functionalities, automatically adding authorization access token if authentication is enabled.
- Supports new Package signing feature, including the submission of end-user Public Key and optional certificate.

Pull This sub-component is fully integrated with the other `son-access` sub-components, and it also works along with `son-workspace` SDK tool in order to retrieve package files (`son-packages`), service and functions descriptors, from the SP and store them in the end-user configured file system or storage system according to their preferences set in the workspace configuration file.

This component is now able to downstream SP Catalogue contents using `access` sub-component as interface, authenticating the developer and providing the required access token to the requests. No more improvements have been added to the sub-component.

4.2.1.2 New features

The `son-access` component present new features described in this section

Keypair Generation

This feature enables the possibility to create a keypair to the end-user. The new Package Signing feature requires a Private Key owned by the SDK end-user in order to perform the signature process, however there is the possibility that the end-user does not have any keypair available. The Keypair Generation features also leverages from the integration with `son-workspace`, using a configuration file where the end-user can set the directory where the provided keypair is located or where the generated keypair will be stored. With public-key algorithms, a keypair (two different keys) is required: where one is used to encrypt and one to decrypt. Generate a private/public keypair is a simple process where the size of the key in bits is specified: it can be of 1024 bits, or larger to be more secure. It also needs a random number generator function. An example is shown below in Python programming language

```
key = RSA.generate(2048)
public = key.publickey().exportKey('PEM').decode('ascii')
private = key.exportKey('PEM').decode('ascii')
```

This feature is not always applied. It requires the end-user to specify the command to apply Package signing in a submission request. When the end-user requires a signature, this feature work in a simple way. It tries to find the keypair specified in the `son-workspace` configuration file. If the two key files do not exist, assumes they have to be generated and saved in the specified location. Once generated, they are available for the signature process. The private/public keypair is stored in PEM format.

Package Signing

This new feature enables Package signing mechanism in the `son-access` component. It allows to generate a digital signature from the Package file, which consists in a mathematical scheme for demonstrating the authenticity of digital messages or documents. When sending a valid digital signature along with the package file to the Service Platform, it gives to the Gatekeeper API reason to trust that the message was created by a known user (enhances authentication) plus that the

sender cannot deny having sent the message (non-repudiation), and ensures integrity, providing a method to check if the package file was altered in transit.

This feature is optional, meaning that it is up to the end-user to generate a digital signature when submitting a package file to the Service Platform. When the signing feature is enabled, the `son-access` component reads end-user's settings from the workspace configuration file to load the user's Private Key, Public Key, and an optional certificate. If the keypair and certificate are not provided by the user, then `son-access` will generate a keypair and save it in the location configured in by the user in workspace settings.

Before the `POST` of the package, `son-access` will sign the package using the user's Private Key (Kpr). Then, the `POST` request will include the original package, signed package hash (digital singture), user's Public Key (Kpb) and the certificate (optional):

```
Payload : son-package + signed son-package + Kpb + certificate (optional)
```

You only need to share the public encryption key and only you can decrypt the message with your private decryption key.

For Python programming language, the requirements to implement this functionality are:

```
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
```

This is the method definition for the Package signature process:

```
def sign_package(self, package_path, private_key):
      # Private key used to test
      private_key_obj = RSA.importKey(private_key)
      try:
          with open(package_path, 'rb') as fhandle:
              package_content = fhandle.read()
      except IOError as err:
          print("I/O error: {0}".format(err))
      # File read as binary, it's not necessary to encode 'utf-8' to hash
      package_hash = SHA256.new(package_content).digest()
      # Signature is a tuple containing an integer as first entry
      signature = private_key_obj.sign(package_hash, '')
      return str(signature[0])
```

A signature is a `string` that looks like:

```
SIGNATURE =  95619130475652336905177552262412838552850...
```

Package Verification

When a package is submitted to the SP, the package signing process is performed. When a signed package is requested by the end-user through the `son-access`, it needs to be verified before delivering the package contents to the end-user. In order to verify a package's signature, the next parameters are required:

- `signature`
- `package file` OR package file's `hash`

- owner's `public key`

This process provides integrity of the package as it was submitted by the owner or author, and verifies that it has not been modified by a third party. The verification process is performed by the `son-validate` component, which is responsible of all kind of validations in the SDK side.

The `verify` method receives the `package file` which is sent along with the `signature` and the owner's `public key`, in order to verify the package. Then, `son-validate` component calculates the hash value of the package file (if package file was sent) and then uses the public key's own `verify()` method to validate its origin.

4.2.1.3 Access configuration and usage

The `son-access` needs to be configured in along with `son-workspace` in order to enable the SDK to authenticate any end-user to the SONATA Service Platform to obtain access. This process grants authenticated access to the platform by a limited period time using JSON Web Tokens (JWT). All SONATA components authenticating to the platform will use the Access Token provided inside the JSON Web Token.

The next section describes how the SDK can get access to the SP by authenticating end-users through the `son-access`, and then performing requests to the Service Platform using the granted Access token.

Configuration

The required configuration for son-access is found in the `son-workspace` generated configuration file. The `son-workspace` tool is responsible for creating workspaces and generating project layouts. By default, it generates a new "sonata" workspace and project layout in `$HOME/.son-workspace` directory. For further information about `son-workspace`, check `son-workspace` section.

To enable "access" capacities, a workspace configuration file is required. It is possible to set access parameters to multiple Service Platforms using different credentials inside this same YAML file. This configuration section keeps the following parameters:

```
service_platforms:
  sp1:
    url: http://sp.int3.sonata-nfv.eu:32001
    credentials: {password: '1234', token_file: token.txt, username: user01}
    signature: {cert: null, prv_key: prv_key.pem, pub_key: pub_key.pem}

url = 'URL address to the platform'
```

This setting must contain the protocol, address and port number of the platform, e.g.:

```
'http://sp.int3.sonata-nfv.eu:30021'

username = "user01" (optional)
password = "1234" (optional)
token_file = "token.txt"
```

This setting stores the user's credentials and the temporary file name of the access token. User's credentials are optional allowing users to type them when signing in to the Platform, or let son-access to automatically read the them from the configuration file. Token files are stored in the

workspace folder 'platforms_dir' using the 'token_file' parameter as file's name. This file will save the current access token data in order to be used by different SDK tools, such monitoring component.

```
pub_key = "pub_key.pem" (optional)
prv_key = "prv_key.pem" (optional)
cert = "trust.crt" (optional)
```

This signature settings indicates the files names for the users public key, private key and certificate. These files are optional, as in case of signing needs, son-access will generate a private and public key for the user. Generated public and private keys will be stored in the users workspace directory, and the public key will be sent to the Platform User Management module.

Usage

The next lines are a short guide that briefly describes the main commands to properly use son-access component

```
usage: son-access [optional] command [<args>]
      The supported commands are:
         auth     Authenticate a user
         list     List available resources (service, functions, packages, ...)
         push     Submit a son-package or request a service instantiation
         pull     Request resources (services, functions, packages, ...)
         config   Configure access parameters
```

The son-access tool supports five different subcommands to deal with authentication, listing of resources, uploading of resources, requesting of resources and configuration of access parameters. The son-access Appendix section includes further information about usage.

4.2.1.4 SDK Authentication and Authorization

Other tools included in the SDK, such son-monitor, might require authentication in order to properly access to the Service Platform to perform their tasks. This requirement is met thanks to the token_file located in the user's workspace, set by the workspace configuration YAML file. Basically, any SDK tool needs to read the contents of this file (the encoded access token) once the user has been authenticated successfully using the son-access. Then, the SDK tool needs to add the token_file contents to the authorization header of the request. Listed is a short guideline for this process:

1. Authenticate the User:

- Save user credentials in the workspace configuration file located in $HOME/.son-workspace. The next template can be used as an example, while replacing username and password fields:

```
catalogues_dir: catalogues
configuration_dir: configuration
default_descriptor_extension: yml
default_service_platform: sp1
log_level: info
name: ws1
```

```
platforms_dir: platforms
projects_dir: projects
schemas_local_master: ~/.son-schema
schemas_remote_master:
https://raw.githubusercontent.com/sonata-nfv/son-schema/master/

service_platforms:
    sp1:
    credentials: {password: '1234', token_file: token.txt, username: user01}
    signature: {cert: null, prv_key: prv_key.pem, pub_key: pub_key.pem}
    url: http://sp.int3.sonata-nfv.eu:32001
validate_watch: ~/.son-workspace/projects
version: '0.03'
```

- Changes must be saved in workspace configuration file and then use the next command to "log-in the user" using the `son-access` tool:

```
sudo python3 src/son/access/access.py auth
```

- When the authentication is successful, a response message similar to the shown below (the example is truncated) will appear in the console terminal:

```
Authentication response: {
   "username":"user04",
   "session_began_at":"2017-05-29 14:26:11 UTC",
   "token":
   {
     "access_token":"eyJhbGciOiJSUzI1NiIsInR5cC....",
     "not-before-policy":0,
     "session_state":"73e1c7f0-4f96-4c2d-80c6-22414c944dc5"
   }
}
```

2. The access token must be obtained from the token file content:

- By default, the workspace location will have the token file in

```
~/.son-workspace/platforms/token.txt
```

- The content of the token file should look like:

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICIxWXZQcGwxWmSE...
```

- The access token has a limited lifespan, which by default will be of `300 seconds`, set by the Service Platform

- The SDK tool, such `son-monitor`, needs to read the token file content and save it in memory:

```
with open('/path/to/token.txt', 'r') as token:
    access_token= token
```

- It is very important that the content read from the token file is not modified during the process and keeps the same as it was saved in the file, or it will cause `unauthorized` errors.

3. The access token must be added to the Authorization header of the request:

- The last step is to add the access token previously saved to the request `Authorization` header as shown below:

```
headers = {'Authorization': "Bearer %s" % access_token}
```

- The access token is sent in the header as 'Bearer' to the Service Platform. The complete request should look like (using Python requests):

```
r = requests.post(url, headers=headers, files=payload)
```

4. When the access token lifespan is expired, it will no longer be valid and requests will result in code `401` error responses from the Service Platform. Then, SDK tool will have `login` again to the Service Platform and repeat the process. In order to avoid the use of expired access tokens, the SDK tool has an option:

- Implement its own method to evaluate the access token status. It will require the JWT library in order to decode the access token and the Service Platform's Public Key. This library is available for multiple programming languages
- The Service Platform's Public Key can be obtained with a

`GET` request to `'.../api/v2/public-key'` in the Service Platform URL

- The Service Platform's Public Key must be converted to PEM format
- The Service Platform's Public Key must be used along with JWT library to decode the access token. If the decoding fails, it will probably mean that it has expired (among other possible failures). An example of this method is shown below:

```
def check_token_status(access_token, platform_public_key):
        try:
            decoded = jwt.decode(access_token, platform_public_key,
                           True, algorithms='RS256', audience='adapter')
            try:
                username = decoded['preferred_username']
                return True
            except:
                return True
        except jwt.DecodeError:
            print('Token cannot be decoded because it failed validation')
            return False
        except jwt.ExpiredSignatureError:
            print('Signature has expired')
            return False
        except jwt.InvalidIssuerError:
            return False
        except jwt.InvalidIssuedAtError:
            return False
```

If the SDK tool uses Python language, it is very recommended to use the next libraries:

- `PyJWT`
- `pycrypto`
- `cryptography`

5. The access token can be invalidated due to the user `logout` to the Service Platform. In that case, the access token lifespan may still valid (did not expired yet) but the Service Platform has marked it with status `"active": "false"`. Further use of this access token will return code `401` error. The SDK user will need to `login` again to the platform using `son-access`.

4.2.2 son-package

The `son-package` tool has the main role of packaging a project, making it ready and available for instantiation in the Service Platform.

4.2.2.1 New features

The new functionalities of `son-package` are described as follows.

Integration with son-access

To be able to solve external dependencies, `son-package` now interacts with `son-access` which is responsible for retrieving service and function descriptors from the Service Platform Catalogues. The, previously implemented, cache system for external dependencies follows the same logic, i.e. if the required component is cached it will not invoke its retrieval.

Integration with son-validate

In previous versions, `son-package` was performing the syntax validation of descriptors. Since features a far more comprehensive validation functionality, is now invoking to perform the validation process, not only to the syntax, but also to integrity and topology. As a result, during the generation of a package, `son-package` will validate all the required components to make sure the package is valid.

Configuration of package information

Package information such as vendor, name, version, maintainer and description can now be configured in the SDK project configuration. This way, when the developer generates a package the only parameters required is the path to the project.

Packaging of individual descriptors

It was established that a package doesn't necessarily requires to have a service ready to be instantiated, it should also serve as a vehicle to share and publish descriptors for later collaboration. Following this paradigm, `son-package` now supports the packaging of standalone/individual descriptors, without the definition of an entry point in the package descriptor.

4.2.3 son-workspace

The `son-workspace` tool plays two major roles, the creation and management of a development workspace/environment and the creation of projects. A workspace contains a user-specific configuration which can be used for the creation and maintenance of multiple projects.

4.2.3.1 New features

New configuration parameters

Both `son-access` and `son-validate` required new workspace configuration parameters.

- the `son-access` tool holds a configuration structure of connection and authentication details to multiple service platforms. Example:

```
service_platforms:
  sp1:
    credentials:
      password: s0n@t@
      token_file: token.txt
      username: sonata
      signature:
        pub_key: ""
        prv_key: ""
        cert: ""
    url: http://sp.int3.sonata-nfv.eu:32001
```

- the `son-validate` service holds the configuration structure required for automatic monitoring of objects. Example:

```
  ~/projects/prj-sample:
    syntax: true
    integrity: true
    topology: true
    type: project
```

Backward compatibility

Both workspace and project configuration now support backwards compatibility. With the introduction of a substantial amount of configuration parameters, this was a necessary feature to support the execution of old integration tests and modules. In spite of allowing the execution, old versions of `son-workspace` do not support new features required by `son-cli` modules.

4.3 son-validate

`son-validate` is the SONATA tool responsible for validation of SDK projects, packages, services and functions. It was initially developed with the purpose of aiding the development of services and functions within a an SDK project, however it was also implemented as a service, enabling its usage by third-party entities. For instance, the Service Platform is using the son-validate service API for the validation of packages. `son-validata` also features a GUI tool enabling a easier way of validating and visualizing services.

4.3.1 Validation scopes

The `son-validate` tool is able to validate different components: an SDK project, a service, a function and a package. Each component can be validated following different scopes, as shown in Figure Figure 4.2. An SDK project or a package typically contains services, and each service usually contains functions. As a result, if a validation of integrity or topology is performed on an SDK project or package it will unfold in the validation of the containing services and consequently in the functions of each service. Each validation scope is described as follows.

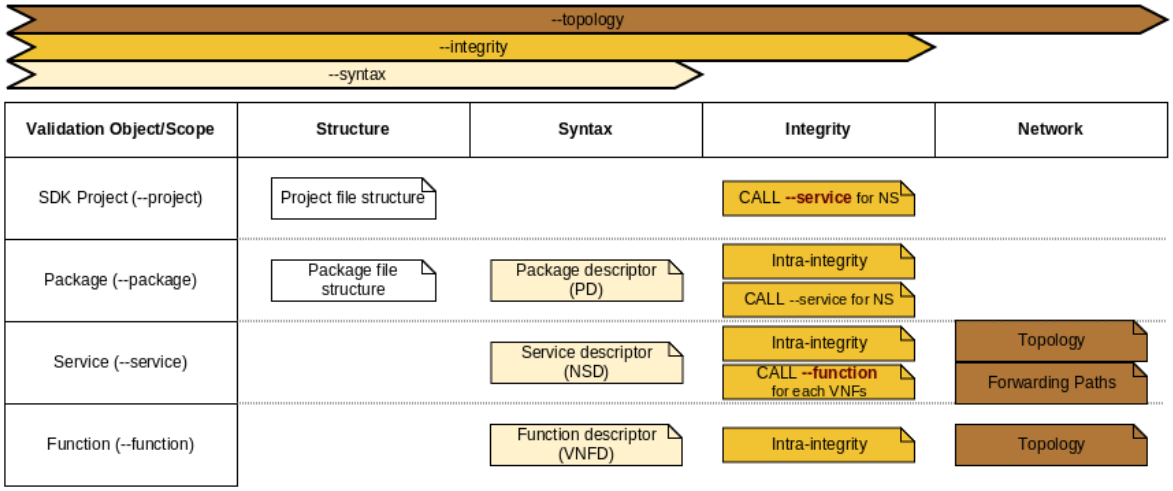

Figure 4.2: son-validate objects and scopes

4.3.1.1 Syntax

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

4.3.1.2 Integrity

The validation of integrity verifies the overall structure of descriptors by inspecting references and identifiers both within and outside individual descriptors. Because service and function descriptors have a different structure, it is differentiated in two components, the Service Integrity and the Function Integrity.

- **Service Integrity** - Service descriptors typically contain references to multiple VNFs, which are identified by a composition of the vendor, name and version of the VNF. The integrity validation ensures that the references are valid by checking the existence of the VNFs. Integrity validation also verifies the connection points of the service. This comprises the virtual interfaces of the service itself and the interfaces linked to the referenced VNFs. All connection points referenced in the virtual links of the service must be defined, whether in the service descriptor or in the its VNF descriptors.

- **Function Integrity** - Similarly to service descriptors, VNFs may also contain multiple subcomponents, namely the Virtual Deployment Units (VDUs). As a result, the integrity validation of a VNF follows a similar procedure of a service integrity validation, with the difference

of VDUs being defined inside the VNF descriptor itself. Again, all the connection points used in virtual links must exist and must belong to the VNF or its VDUs.

4.3.1.3 Network topology

The `son-validate` provides a set of mechanisms to validate and aid the development of the network connectivity logic. Typically, a service contains several inter-connected VNFs and each VNF may also contain several inter-connected VDUs. The connection topology between VNFs and VDUs (within VNFs) must be analysed to ensure a correct connectivity topology. The `son-validate` tool comprises the following validation mechanisms. Figure 4.3 shows a service example used to better illustrate validation issues.

- **unlinked VNFs, VDUs and connection points** - unconnected VNFs, VDUs and unreferenced connection points will trigger alerts to inform the developer of an incomplete service definition. For instance, *VNF#5* would trigger a message to inform that it is not being used.
- **network loops/cycles** - the existence of cycles in the network graph of the service may not be intentional, particularly in the case of self loops. For instance, *VNF#1* contains a self linking loop, which was probably not intended. Another example is the connection between *vdu#1* and *vdu#3* which may not be deliberate. The `son-validate` tool analyses the network graph and returns a list of existing cycles to help the developer in the topology design. In this example, `son-validate` would return the cycles:
  - [*VNF#1*, *VNF#1*]
  - [*vdu#1*, *vdu#2*, *vdu#3*, *vdu#1*]
- **node bottlenecks** - warnings about possible network congestions, associated with nodes, are provided. Taking into account the bandwidth specified for the interfaces, weights are assigned to the edges of the network graph in order to assess possible bottlenecks in the path. As specified in the example, the inter-connection between *vdu#2* and *vdu#3* represents a significant bandwidth loss when compared with the remaining links along the path.

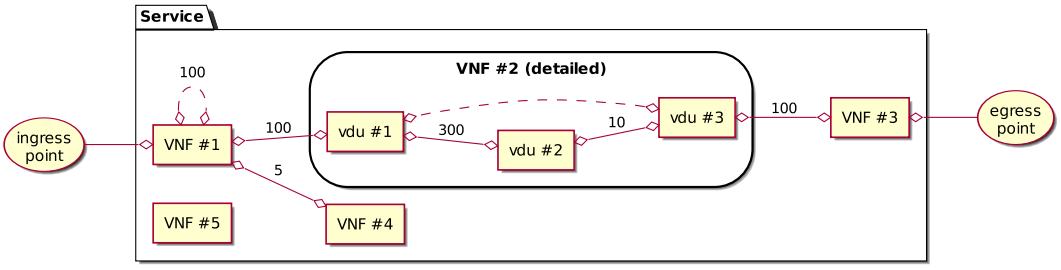

Figure 4.3: Example of a Service Network Topology

Functionalities and Requirements

The `son-validate` tool offers different validation levels, the syntax, integrity and topology. However, integrity validation must comprise a syntax validation and in turn topology must comprise an integrity validation.

As previously mentioned, `son-validate` can be used inside the SDK, under the developer environment, and as an external tool. In the first case, the workspace should be specified in order to read the environment configuration. Moreover, if an SDK project should be validated a workspace

must be specified. If `son-validate` is being used outside the SDK it is possible to validate a service or individual functions. The validation of a service comprises the validation of the service itself and, if at least integrity is specified, its referenced VNFs. On the other hand, the validation of functions only verifies each function individually.

4.3.2 Modes of operation

`son-validate` can be used locally via CLI or as a service via API.

4.3.2.1 CLI

The CLI interface is designed for developer usage, allowing to quickly validate SDK projects, package descriptors, service descriptors and function descriptors. The different levels of validation, namely syntax, integrity and topology can only be used in the following combinations:

- syntax `-s`
- syntax and integrity `-si`
- syntax, integrity and topology `-sit`

The son-validate CLI tool can be used to validate one of the following components:

- project - to validate an SDK project, the `--workspace` parameter must be specified, otherwise the default location $HOME/.son-workspace is assumed.
- package - to validate a package, only the `--package` should be specified indicating the path for the package file.
- service - in service validation, if the chosen level of validation comprises more than syntax (integrity or topology), the --dpath argument must be specified in order to indicate the location of the VNF descriptor files, referenced in the service. Has a standalone validation of a service, son-validate is not aware of a directory structure, unlike the project validation. Moreover, the --dext parameter should also be specified to indicate the extension of descriptor files.
- function - this specifies the validation of an individual VNF. It is also possible to validate multiple functions in bulk contained inside a directory. To if the --function is a directory, it will search for descriptor files with the extension specified by parameter --dext

4.3.2.2 Service API

son-validate can be executed as a service, providing a RESTful interface to validate objects and retrieve validation reports. son-validate API service can be executed in two distinct modes: `stateless` or `local`. Stateless mode will run as a stateless service only and can be instantiated at any remote location. Local mode is designed to run in the developer OS, providing additional functionalities. It aims to provide automatic monitoring and validation of local SDK projects, packages, services and functions. Automatic monitoring and validation can be enabled in workspace configuration, specifying the type of validation and which objects to validate. This functionality watches for changes in the specified objects automatically triggering the validation process as required.

Workspace configuration (local mode only)

When running in local mode, automatic monitoring and validation of objects may be set up in the workspace configuration, under the 'validate_watchers' key. Example for validating a project and a service:

```
validate_watchers:
    ~/sonata/sdk-projects/sample-project:
        type: project
        syntax: true
        integrity: true
        topology: true
    ~/sonata/sdk-projects/objects/nsds/sample-nsd.yml
        type: service
        syntax: true
```

API overview

The son-validate service API provides the following functionalities:

- validate an SDK project, a package, a service or a function (`/validate/<object_type> [POST]`)
- retrieve a list of available and validated objects (`/report [GET]`)
- retrieve the validation report for a specific object (`/report/result/<resource_id> [GET]`)
- retrieve the validated network topology graph (`/report/topology/<resource_id> [GET]`)
- retrieve the validated forwarding graphs structure (`/fwgraphs/<resource_id>) [GET]`)

4.3.3 Event configuration

son-validate enables the customization of validation issues to be reported by a user-defined level of importance. Each possible validation event can be configured to be reported as error, warning or none (to not report). For now, event configuration is configured statically but in the future we aim to support a dynamic configuration through the CLI and service API. A comprehensive list of the validation events and their description is available at the son-validate documentation.

4.3.4 GUI Visualization tool

The GUI enables the on-demand validation of SDK projects, packages, services and functions. It provides a powerful visualization engine of services and their functions, allowing a quick identification of the validation issues.

4.3.4.1 Layout

The GUI is composed by three main areas, the left panel, the central canvas and the right panel, as shown in Figure 4.4.

- Left panel - controls the visualization options, including the possibility to show and hide the management links. Moreover, it is also possible to only show a specific forwarding graph, in cases when multiple forwarding graphs are present.

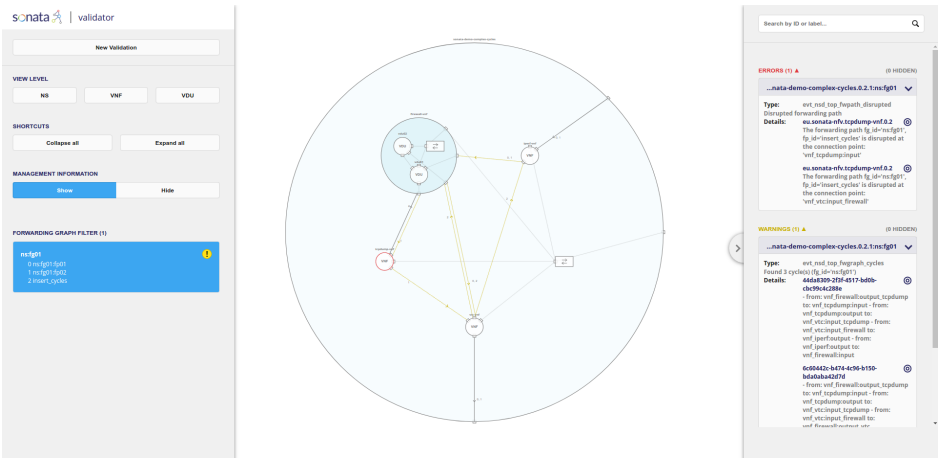

Figure 4.4: son-validate GUI

- Central canvas - dedicated to represent the network topology of the services, functions and VDUs. It allows the visualization of the virtual links of components, as well as the forwarding graphs and forwarding paths of each service.
- Right panel - contains the list of errors and warnings of the performed validation.

4.3.4.2 Features

The the main features of the son-validate GUI are described as follows.

Validation triggering

The GUI enables the possibility of viewing previous validations as well as trigger a new validation (Figure Figure 4.5). To perform a new validation, the object type must be chosen and its location provided, in addition to the validation scope (syntax, integrity, topology).

Issue source highlight

Whenever an issue is associated with a particular object, such as a specific service, function or VDU, the object is highlighted to a quick identification of the problem source (Figure 4.6).

Cycle identification and highlighting

Cycles are identified by coloured lines and can be highlighted in the right panel, passing the mouse over each cycle (Figure 4.7).

4.4 son-editor

The SONATA Editor (son-editor) is a web-based application whose frontend is running in the networks service developer's browser. It is compatible with the SONATA eco-system and makes use of tools from the SONATA SDK. Its main purpose is to assist in the creation and editing of SONATA network service projects and their contained descriptors. This is done by simplifying repetitive and complicated tasks e.g. the descriptors of network services (NS) are visualized as a graph of VNFs. In contrast to the textual representation of the graphs, users get a better understanding of the nodes and the connections between them. It allows users to take a quick

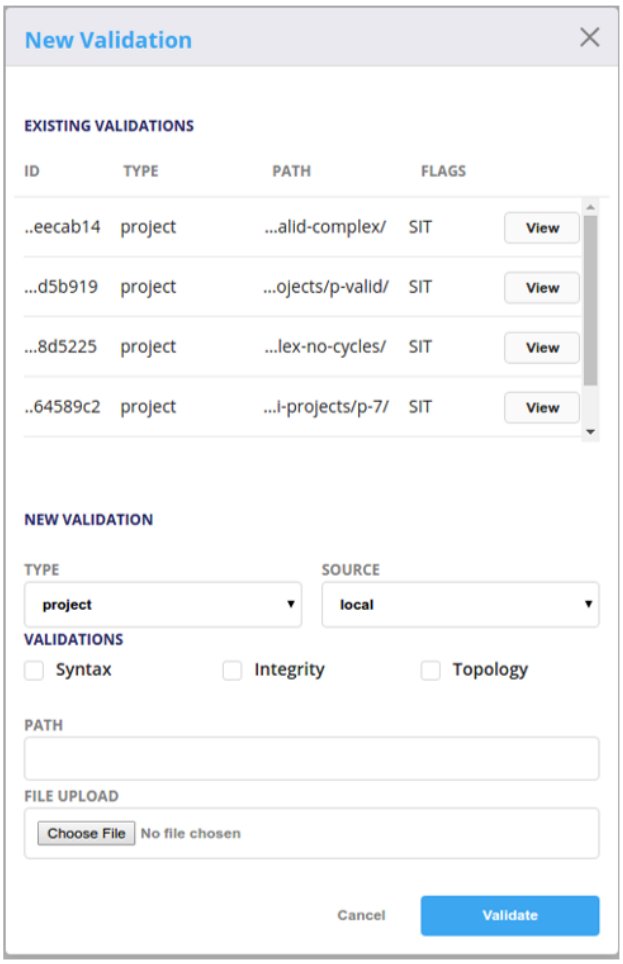

Figure 4.5: son-validate: new validation

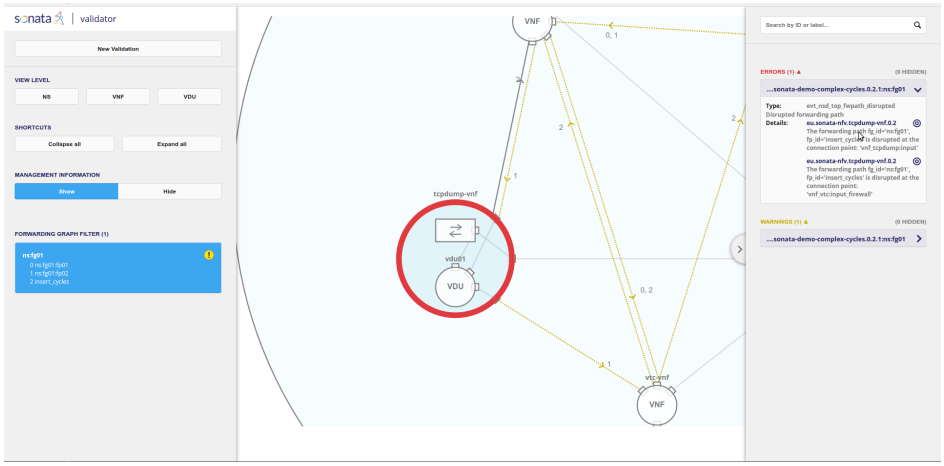

Figure 4.6: son-validate: issue source highlighting

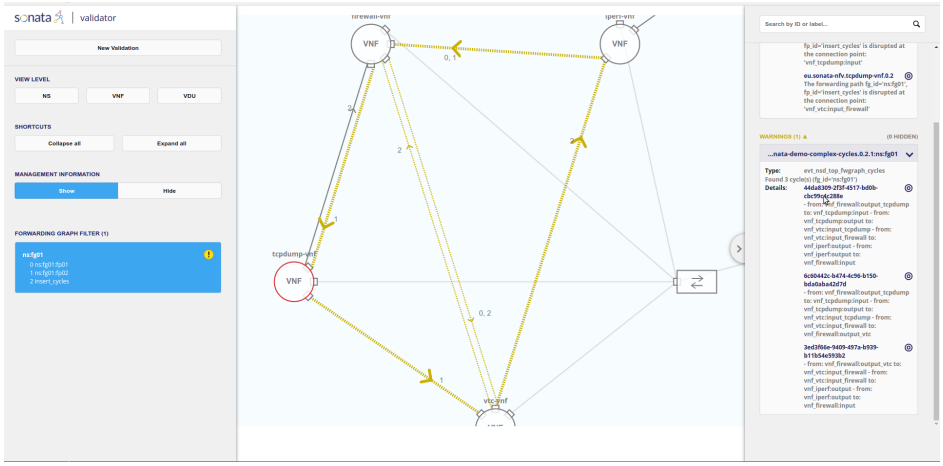

Figure 4.7: son-validate: cycle highlight

look at the graph layout and helps users to analyse the descriptor well. It also handles workspace and project creation and executes utilizes the son-cli tools (e.g. son-package, son-access) to handle descriptors and packages. The editor and its components component could either run on the developer's local computer and can be configured locally or it could be hosted by a central editor server as a part of the SONATA SDK e.g. for a company setup. One of its outstanding features is the integration with GitHub that is on the one hand used to authenticate users and on the other hand to share network service projects with other developers through versioned, public repositories.

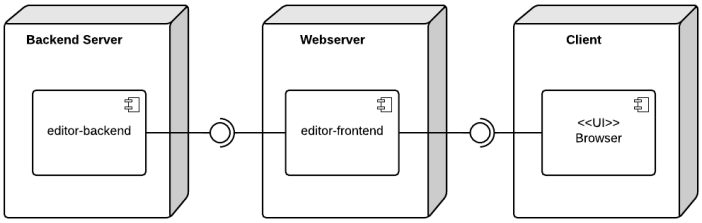

Figure 4.8: Son-editor high-level components

Like many modern web-applications, son-editor is split into a backend and a frontend part that can be executed by different serves to improve its scalability. Figure 4.8 shows this modularization and how the components can be split between backend server and frontend (webserver). In this setup the application logic is encapsulated in the backend server and exposed via a RESTful interface to the frontend parts. The frontend components are delivered by a standard webserver and are executed in the client's browser.

4.4.1 Features

- Graphical network service editor
  - Drag and Drop network components
  - Connect by dragging connections
  - Undo and redo, zoom and pan, multi-select
  - Automatic computation of the forwarding graph (optional)

- Immediate frontend side syntax validation
  - Visual feedback what needs to be filled and what is still missing
  - Easy to select constrained values if specified by schema
- Checking reference dependencies
  - Hinder deletion of referenced VNFs
  - Create new version or refactor references when renaming VNFs
- Direct upload to service platform
  - Package via son-package tool
  - Upload services directly from network service view
- GitHub Integration
  - GitHub OAuth login
  - Clone, share, pull and push projects from and to GitHub

4.4.2 Editor Backend

The editor backend was developed as a web-application based on Python using Flask and the corresponding extensions to create REST APIs. Figure 4.9 shows how the backend component integrates with the editor's frontend and other sonata components as well as GitHub to store and share network service projects. The backend utilizes in particular the tools provided by son-cli to do workspace, project and package handling during network service development. This avoids code replication and ensures that the editor stays compatible with the latest package formats supported by son-cli. The shown integration with son-catalogues is still available, however since the SDK catalogues where removed from the SDK during the project the integration became optional.

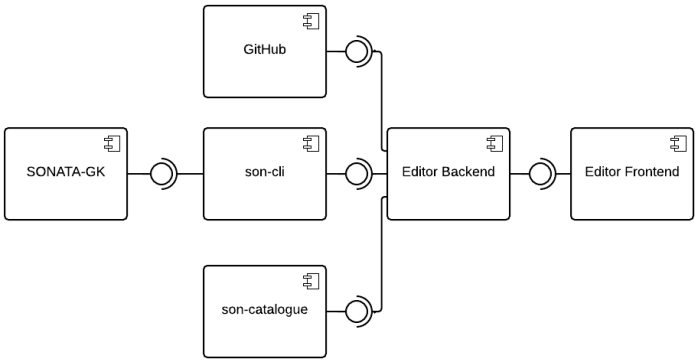

Figure 4.9: Son-editor detailed components and their integration with external tools and services

4.4.2.1 Project Structure

The root folder contains set-up scripts and docker file configurations. Furthermore, the functionalities are split in different components which are reflected by the hierarchy of the project's file structure as shown in Figure 4.10.

The source code is split into the following parts:

- **API Modules (apis):** The API modules, implemented as Flask RESTPlus resources that receive and interpret the requests from the frontend and relay those calls to the corresponding implementation modules. All API namespaces are registered on the main application from the module's initializer.

- **Implementation Modules (impl):** The implementation modules handle the actual execution of the demands from the frontend and will create, update or delete the corresponding descriptors, workspaces and projects.

- **App Module (app):** The App module contains the entry point of our application that handles the initial start and setup of the server. It also defines the connection and the setup of the database, manages the access permissions of requests and handles any exceptions that may be thrown by the implementation modules.

- **Database Models (models):** The database models create a mapping between the structures saved to the file system and their corresponding entries in the database. They define the database schema depicted in Figure 4.11.

- **Utility Functions (util):** The utility functions are a collection of small tools that are used to load and write descriptors, communicate to the son-cli tools and formatting the responses that are returned to the frontend.

- **Test Cases (tests):** This module defines the unit tests.

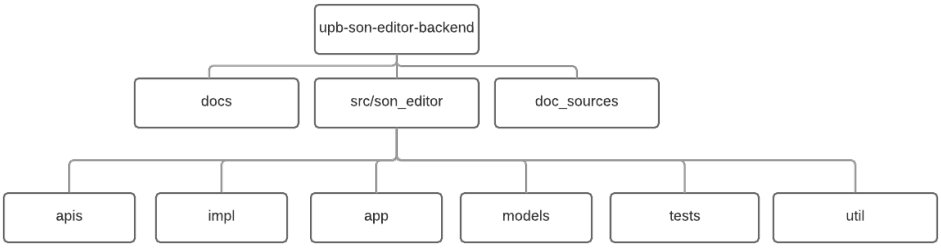

Figure 4.10: Project structure

4.4.2.2 Data Management

The most important task of the backend server is storing and serving the VNF and NS descriptors on demand of the frontend. In this section we will explain how a database and the file system is used to manage and serve the descriptors.

Database

To make the descriptor accessible as fast as possible and to tie the data to a particular user, workspace and project we used a relational database represented by the ORM-Models of SQL-Alchemy. This way we could quickly find and serve the descriptors requested by the frontend. Figure 4.11 shows the ER model of the used database schema.

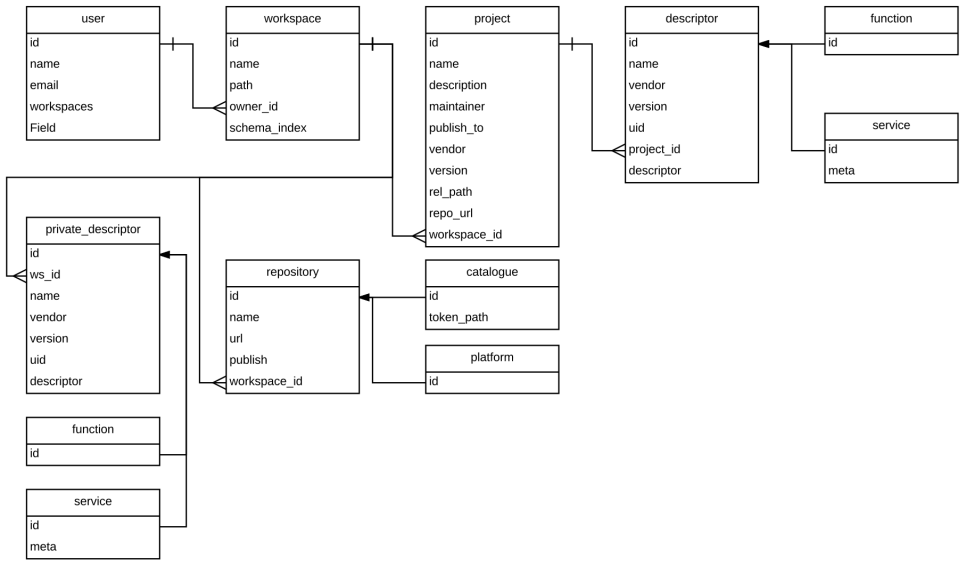

Figure 4.11: Son-editor's database schema

Descriptor Files

All descriptors are not only written to the database but also to the file system before if needed. This way the son-cli tools could be used to create a service packages based on the descriptor files written to disk. The other reason for not storing exclusively to the database was to have the ability to recreate the database from the file system. This is currently used in two cases. The first one is migrating the database to a new schema, which was often necessary during the development. The second one is importing previously created workspaces in bulk without having to upload everything through the frontend editor. By simply copying the workspace into the users folder and restarting the backend server it will automatically scan the unknown files and import them to the database. Further, this approach enables importing a project from GitHub by cloning the project folder to disk an importing it.

4.4.2.3 GitHub Integration

The backend also handles the main functionality of the GitHub Integration features.

GitHub OAuth Login

To avoid creating separate accounts for every user of the editor the GitHub OAuth API is used for the login process. This not only reduces the need for a separate account, but it also reduces the burden to create a secure place to store passwords.

In the OAuth process the developer first needs to register his application with GitHub and provide a callback URL. When the users want to login using GitHub, they are redirected to the authorization page to confirm giving the application access to the requested data. GitHub then calls the callback URL with an authorization code that can in turn be used to retrieve the user information. To identify the developer's application they must provide a client ID with the authorization call and the client secret with the second call.

GitHub Sharing

Another feature we use GitHub for is the project sharing mechanism. The GitHub integration enables the user to clone remote projects, initialize and publish local projects and use the git pull, commit and push implementations to keep the projects in sync between GitHub and the son-editor.

4.4.2.4 API Specification

The backend offers a REST API used by the frontend. This API uses the following URL structure to represent endpoints for workspaces, projects, and the different descriptors. Each of these endpoints implements the corresponding CRUD operations to manipulate the linked artifacts.

```
|/workspaces/
|---|<wsID>
    |---|/projects/
        |---|<pjID>
            |---|/functions/
            |   |---<vnfID>
            |---|/services/
                |---<servID>
```

The complete API specification of the son-editor backend component is publicly available in a document hosted in a GitHub repository [9].

4.4.3 Editor Frontend

The editor frontend represents the user interface to interact with son-editor. It is implemented in JavaScript using a couple of modern UI and frontend libraries. Figure 4.12 shows its general structure and its code modularization. The frontend offers several views to manipulate different artifacts of a SONATA network service project. Starting from the main page, a user can create, update, delete and configure workspaces in which projects can be created. These workspaces correspond to the workspaces used by the son-cli tools. In the project view, network service projects can be created, updated, deleted or be shared on GitHub. The frontend then offers two different editors. One to manipulate network service descriptors (NSD) and one to manipulate virtual network function descriptors (VNFD).

4.4.3.1 NS Editor

The NSD editor offers a graph-based view to compose and manipulate NSDs. All existing components (VNFs, connection points, E-LAN and E-Line connections) for constructing a NS are displayed on the left side of the editor and can be drag and dropped to the editing area on the right side. By clicking the connection points of two components the user can create connections between them to define forwarding graphs. Besides this graphical representation, the user can always switch to the text-based editor view in which the descriptors can be directly manipulated. This allows, e.g., the annotation of descriptor elements with properties not defined when the editor was implemented. Figure 4.13 shows an example graph of a network service consisting of three VNFs, three connection points and a management E-LAN.

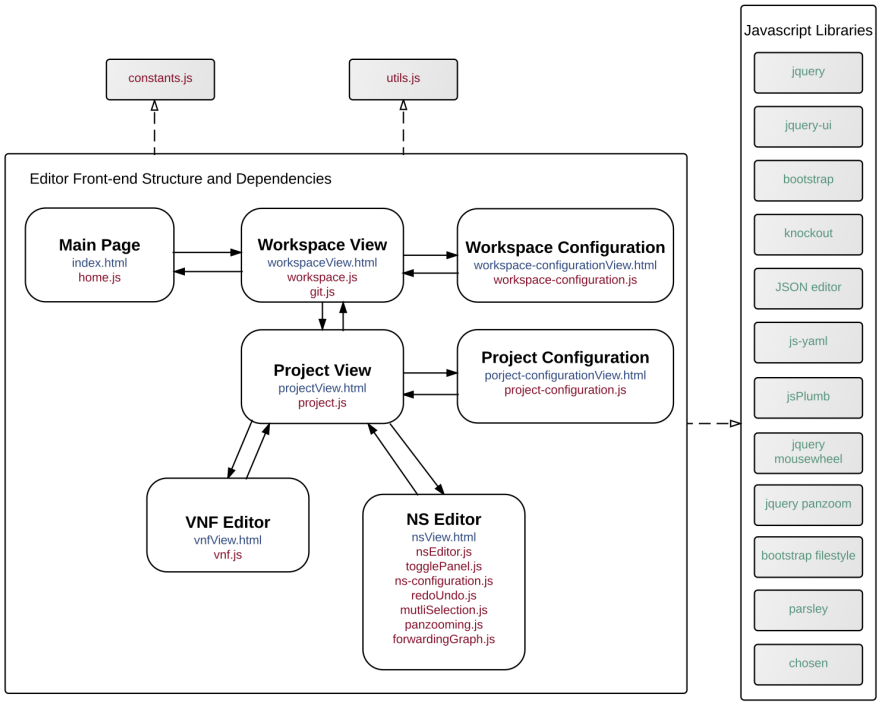

Figure 4.12: Son-editor frontend components and views as well as used frameworks and libraries

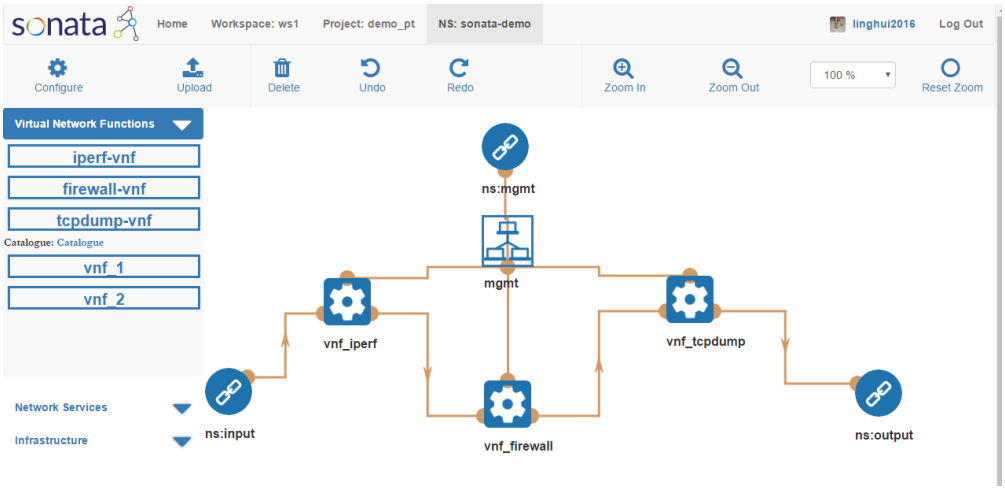

Figure 4.13: NSD editor view showing a graphical representation of a example network service

4.4.3.2 VNF Editor

The VNFD editor offers a form-based view to manipulate VNFDs. This view is not statically programmed but is automatically generated based on the schemas configured for the given projects. As default, the editor only shows input fields for properties marked defined as *required* by the schemas. Using this, the interface presented to the user is simplified which lowers the burden for new users. However, if additional, optional fields are required for the VNFD, they can be added to the form by selecting them from a drop-down interface. The VNFD editor also uses the schemas to do simple live validation of the user inputs as shown in Figure 4.14. On top of the screen the missing inputs are presented to the user. This reduces the risk of adding bugs to VNFDs.

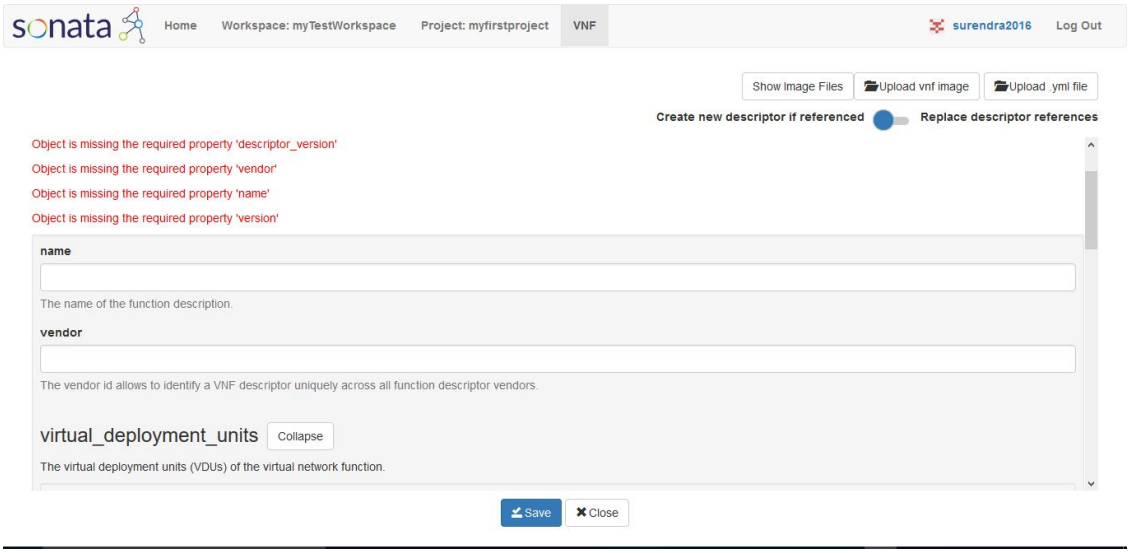

Figure 4.14: Form-based VNFD editor with live validation of user inputs

4.4.4 Installation

Even though the editor components can be deployed on separated hosts as shown in Figure 4.8, the default way to install and run it is as a self-contained Docker container to simplify its distribution and deployment. Figure 4.15 shows this scenario in which backend and frontend are executed in a single container which can be accessed by users using their default browser.

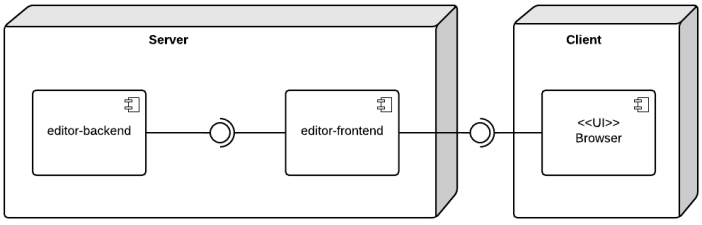

Figure 4.15: Typicall deployment of son-editor as single Docker container containing back- and frontend

The following steps are required to deploy the editor:

1. Since the editor uses OAuth to authenticate its users, a OAuth application token is required to run it. To retrieve such a token (from GitHub), go to *GitHub Settings - OAuth applications* and *Register a new application*.
   a) Chose an application name: *SONATA Editor*
   b) Configure the URL of your installation: `http://localhost/` or `http://your-domain.com`
   c) Configure the authentication callback URL: `http://localhost/backend/login`
   d) *Save* and collect the generated *ClientID* and *ClientSecret* for step 4
2. Clone the repository:
   a) `git clone https://github.com/sonata-nfv/son-editor-backend`
3. Switch to *son-editor-backend* folder:
   a) `cd son-editor-backend/`
4. Add GitHub OAuth *ClientID* and *ClientSecret* to *config.yaml*
   a) `vim config.yaml`
5. Build and run container:
   a) `docker-compose up`

Finally you can access the editor with your browser using `http://localhost/`.

4.5 son-emu

The SONATA emulator is one of the oldest components of the SONATA SDK. Since the last release we added the following new features to it. Beside these additional features, several smaller changes and many bugfixes were made.

4.5.1 OpenStack-like API Endpoints

Besides testing network services locally using son-emu and its dummy gatekeeper, it is highly desirable to be able to integrate our multi-PoP emulation platform with real-world orchestration solutions, like the SONATA service platform and let these solutions control the emulated infrastructure. To do so, we identified two different approaches to enable the easy integration between our emulated environment and real-world MANO systems. The first approach is to implement customized driver plugins that support our emulation platform for each MANO system. The second approach is to add already standardized interfaces to our emulation platform which can then directly be used by off-the-shelf MANO installations. Since we want to support as much MANO solutions as possible and keep the development overhead low, we decided to go for the second approach. More specifically, we realized it by adding OpenStack-like interfaces to our platform, because practically every MANO system comes with driver plugins for the OpenStack APIs.

As illustrated in Figure 4.16, our platform automatically starts OpenStack-like control interfaces for each of the emulated PoPs which allow MANO systems to start, stop and manage VNFs. This approach has a good level of abstraction and we have been able to implement the needed set of API endpoints required by the SONATA and OSM orchestrators in less than *3.8k* lines of Python code. Specifically, our system provides the core functionalities of OpenStack's *Nova*, *Heat*, *Keystone*,

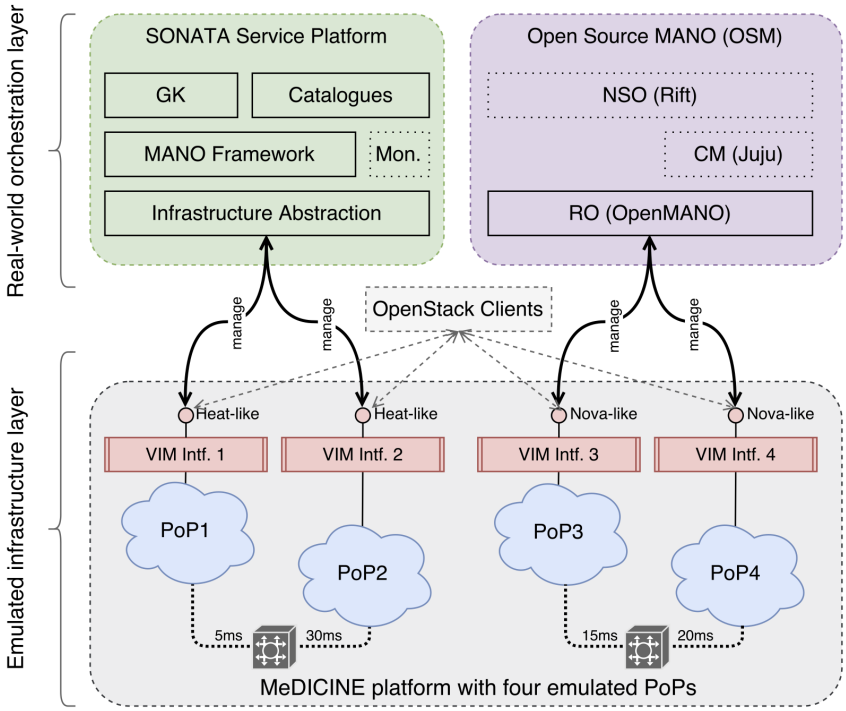

Figure 4.16: Integration concept to run real-world MANO systems on top of the SONATA emulator controlling service deployments

Glance, and *Neutron* APIs. Even though not all of these APIs are directly required to manage VNFs, all of them are needed to let the MANO systems believe that each emulated PoP in our platform is a real OpenStack deployment.

From the perspective of the MANO systems, this setup looks like a real-world multi-VIM deployment, i.e., the MANO system's southbound interfaces can connect to the OpenStack-like VIM interfaces of each emulated PoP. A demonstration of this setup was submitted to and accepted by IEEE NetSoft 2017.

4.5.2 Demonstration Dashboard

Son-emu as such is a system-level tool that offers multiple command line interfaces to the user. This offers a high-level of control and the possibility to script and automate experiment setups. However, to get a quick overview about the emulated system and especially to demonstrate the emulator we decided to add a graphical web-based dashboard to the emulator. This dashboard is served by an internal web server which is also responsible to host the REST APIs of the emulator. The dashboard as such is implemented in JavaScript and polls the REST interfaces to visualize the system's state as shown in Figure 4.17. The dashboard shows a list of emulated PoPs, their labels, internal names and responsible software switches. In addition it shows all running containers that are deployed on the emulated infrastructure. This makes it easy to explain an audience what is happening inside the emulation platform and for a service developer to get a quick overview of the deployed service in the SDK emulator environment. The information given in the dashboard, can be used, similar to what a Network Service Record (NSR) file would contain if the service was deployed in the Service Platform.

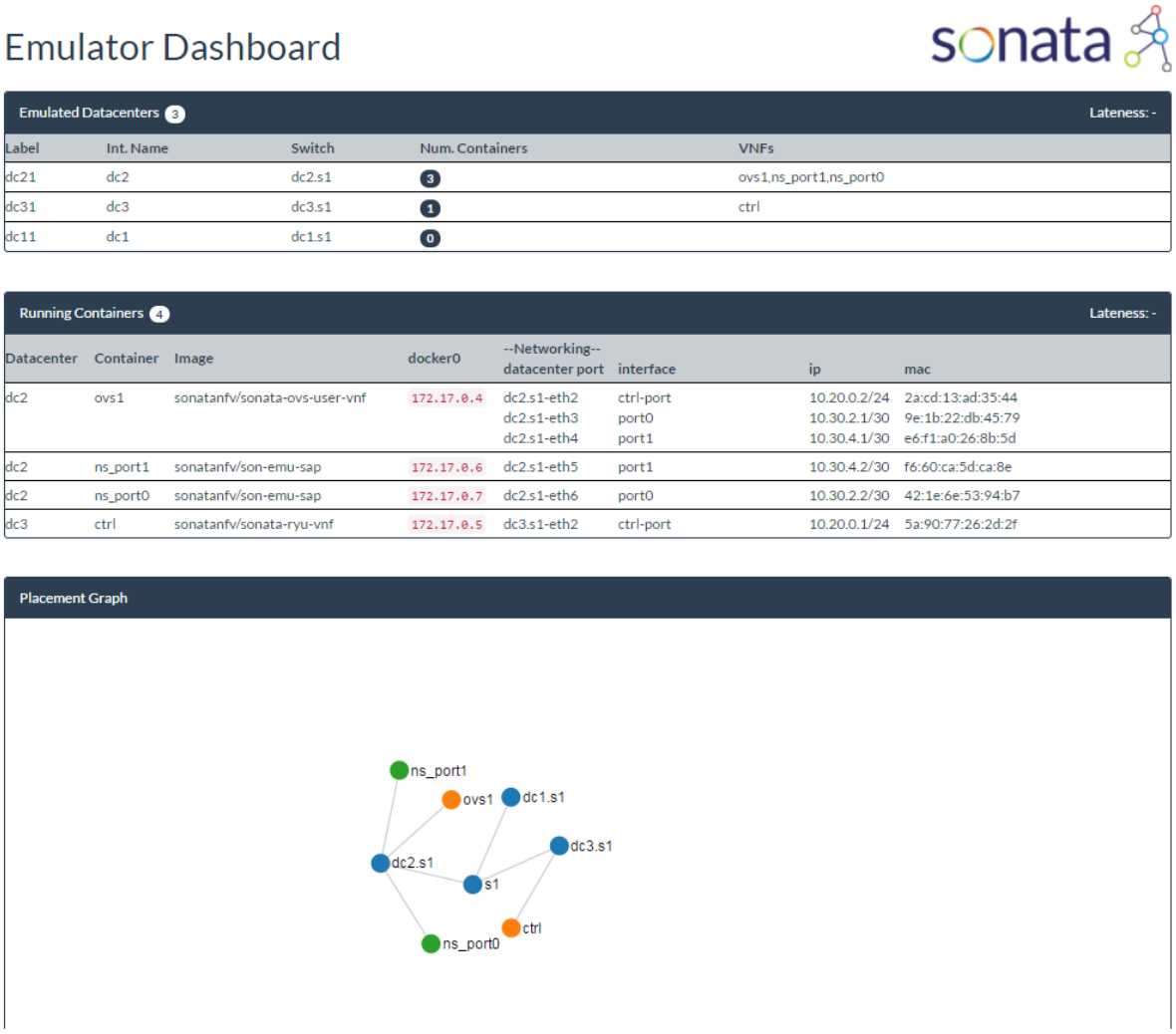

Figure 4.17: Screenshot of the emulator dashboard

4.5.2.1 Dashboard Structure

The dashboard has following structure, as can be seen in Figure 4.17:

- **Emulated Datacenters**: The list of datacenters that are specified in the topology file used to start the emulator. The placement of each VNF in the service unto one of the datacenters.
- **Running Containers**: All VNFs that the service package contains and their deployment information, such as networking info (IP, MAC, interface names), image, name and to which datacenter interface they are connected.
- **Placement Graph**: A graphical representation of the placement of each VNF in the service unto one of the datacenters in the emulated infrastructure topology. The graph is built using the D3js library [8].

4.5.2.2 VNF Configuration Terminal

To make it more easy for a service developer to test and debug the emulated service, the configuration of the deployed VNFs is a crucial step. The SONATA SDK environment provides several means to do VNF configuration. In order to execute CLI commands inside a deployed Docker VNF in son-emu, different options exist, the technical details are given in Appendix B. The easiest option is to open a VNF terminal window from the dashboard:

By double-clicking on a VNF in the placement graph on the son-emu dashboard, a terminal window is started. These xterm windows can be forwarded using X11. This xterm window opens a prompt inside the VNF, allowing a developer to quickly check the internal state and configure the VNF.

4.5.3 Host-based Service Access Points

The SONATA SDK environment offers several ways to inject and analyse test traffic to/from the emulated service. Three types of SAP (Service Access Point) are implemented in the emulator. The type is specified in the Service and VNF descriptor (NSD or VNFD)

- **Internal SAP**: This is deployed as a dedicated Docker VNF, chained to the service VNFs as defined in the NSD.
- **External SAP**: This is deployed as a virtual interface on the son-emu host. NAT rules are installed using iptables, which make it possible to connect this interface and the service's endpoint to the outside world (eg. to connect a traffic generating process, running on the host).
- **Management SAP**: These VNF interfaces are typically connected to an isolated management network. they are used to access and configure the VNFs. They can be replaced by the default VNF interfaces to the docker0 switch, as described in next section.

4.5.4 Dummy Gatekeeper Updates

The Dummy Gatekeeper is the API endpoint where SONATA Service Packages can be pushed to, to be deployed on the emulated infrastructure topology. Several flags have been created, to enhance the deployment options. These deployment flags are specified in the topology file, further technical info is given in the appendix: Section B.2. As a summary we give a brief overview of the new deployment flags and their functionality:

- `deploy_sap` when set to `True`, each connection point (SAP) in the NSD is deployed according to its defined type, as described in previous section.
- `auto_deploy` when set to `True`, each pushed service package will automatically be deployed (this avoids the need to explicitly ask the dummy gatekeeper to deploy the latest pushed package).
- `auto_delete` when set to `True`, any currently deployed service will automatically be deleted when a new one is asked to be deployed. This configures `son-emu` to only deploy one service at a time.
- `docker_management` when set to `True`, the VNF interfaces that are configured in the NSD/VNFD as `type=management` are not instantiated, but are replaced with the interfaces to the `docker0` switch, which are anyway deployed by Docker.
- `sap_vnfd_path` is the pathname of the VNFD YAML file that describes which Docker VNF to use as service endpoint. Each internal connection point in the NSD is deployed with this VNFD.

4.5.5 Generic Configuration and Scaling Support

The total of implemented features described above, further enhance the SONATA SDK emulation environment. They allow that configuration and deployment can be manipulated by a management function as illustrated in Figure 4.18. This resembles the typical functionalities that would be implemented by a Service/Function Manager when the service is deployed in a MANO platform:

- The VNF **configuration** mechanism described in the section above, is used to test the VNF management interfaces.
- The REST API of the SONATA emulator can be used to manipulate the **chaining** of traffic through the VNFs and to test possible **scaling strategies** by deploying or removing any other VNFs. The details of the REST API were already described in Section 3.4.2 of [4] and are maintained on the GitHub wiki page: `https://github.com/sonata-nfv/son-emu/wiki/REST-API-command-overview`

This enables a speed-up in development time, needed to create, debug or modify the configuration or scaling procedures of the NFV-based service, before deployment in production through a MANO platform.

4.5.6 Experimenting with Placement Strategies

Another feature of the emulation platform is the possibility to experiment with different placement strategies for VNFs and services in a simple way. The only thing a SDK and emulator user needs to do for this is replacing the default placement class in the dummy gatekeeper module with a customized placement strategy which can be done with a couple of Python statements.

```
class RoundRobinDcPlacement(object):
    """
    Placement: Distribute VNFs across all available
    PoPs (DCs) in a round robin fashion.
    """
```

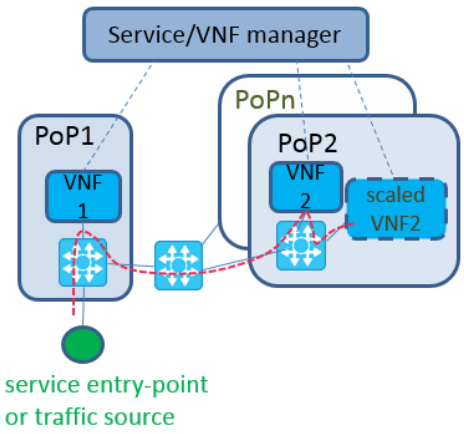

Figure 4.18: General SONATA emulator functionality

```
def place(self, nsd, vnfds, saps, dcs):
    c = 0
    dcs_list = list(dcs.itervalues())
    for id, vnfd in vnfds.iteritems():
        vnfd["dc"] = dcs_list[c % len(dcs_list)]
        c += 1  # inc. c to use next DC
```

The above code listing shows the default placement algorithm of the dummy gatekeeper which equally distributes the VNFs across all available PoPs. The entire logic is implemented in a single placement method: `place(self, nsd, vnfds, saps, dcs)` which gets the NSD, a list VNFDs and a list of SAPs to be deployed as its input. Additionally a list with all available PoPs (or emulated data centres) is given to this method so that all required information about the service instance to be deployed as well as about the underlining infrastructure to deploy on is available. To assign a VNF to a particular data centre, the algorithm needs to assign the target data center object to the `vnfd["dc"]` field of the VNF to be placed (see: `vnfd["dc"] = dcs_list[c % len(dcs_list)]`).

This simple placement interface gives developers a easy way to try out different placement strategies on top of SONATA's emulation platform.

4.6 son-profile

The basic idea of son-profile is to deploy network services on SONATA's emulation platform and do some load testing under different resource constraints. During these tests a variety of metrics can be monitored which allows service developers to find bugs, investigate problems or detect bottlenecks in their services. The main purpose of son-profile is to automate big parts of this workflow to support network service developers as much as possible. The general idea of introducing a profiling functionality in the SONATA SDK was already given in section 3.5 of [4]. In this section we further detail the actual implementation. We introduce two modes of operation for the `son-profile` command:

- **Active Mode**: Create a series of service packages, each with a specific resource limitation and defined functional tests. These packages can be automatically deployed on the emulator or MANO platform where the functional tests are executed and metrics are gathered.

- **Passive Mode**: This mode assumes that a service is pre-deployed in the SONATA emulator. The defined functional tests are executed and metric are gathered and statistically analyzed. This has the advantage that resource limitations, test traffic generation and specialized monitoring can be installed by leveraging the REST API and other unique features of the SONATA emulator.

Both modes are further detailed in the next sections.

4.6.1 Active Mode

The *active* mode of `son-profile` allows the user to specify profiling experiments and a service package to which these experiments should be applied. Based on this, `son-profile` generates a couple of new service packages each of them representing exactly one resource configuration that should be tested during the profiling experiments. One of the main updates to the previous version of `son-profile` is its improved modularization as explained in the following.

4.6.1.1 Modularized Controller Architecture

Figure 4.19 describes our system as well as the *active* profiling workflow. In the first step, a user creates a *profiling experiment descriptor (PED)* that contains all necessary information to perform a profiling experiment. In particular, it references the network service that should be profiled, e.g., a service package or service descriptor, and it includes descriptions of all service configurations that should be tested, e.g., different resource assignments for the used VNFs. The PED is used to trigger our profiling system by using its command line (CLI) interface (step 1 in Figure 4.19). The profiler reads the PED and forwards the request to its *descriptor engine*. This module takes the network service referenced by the PED file and embeds (or extends) it with additional measurement VNFs, called *measurement points* (MP). Our system offers default *measurement point* VNFs that contain standard networking test tools, like *iperf* or *hping*. A user can replace these by any custom measurement VNF which may, for example, contain domain-specific, proprietary traffic generators. After the embedding step, one copy of the new service description, for each configuration specified in the PED, is generated (step 2). This includes resource configurations, like number of cores assigned to a VNF. The *descriptor engine* itself offers a plugin interface for service description generators so that our profiler becomes service descriptor agnostic and can be extended to third-party description formats.

In the third step, the previously generated service configurations are deployed one after the other on the target platform(s) using the *platform driver* modules (step 3). These drivers act as a client to the target platform (usually `son-emu`) and form an abstraction layer between specific MANO northbound interfaces and our internal control mechanisms.

Once a service instance is up and running, the traffic generators in the additionally deployed *measurement point* VNFs are activated and start to stimulate the service. After this, the service instance is destroyed and removed from the platform before the next service configuration is deployed. We call the deployment, execution, and test of a single service configuration a *profiling round*.

During a *profiling round*, performance data is collected in two ways (step 4). First, service-internal performance metrics are monitored, including log files inside the *measurement points* and VNFs (if enabled). Second, platform metrics, like packet counters on virtualized interfaces, are collected through the platform's monitoring APIs. The latter are platform-specific and may not be available on each target platform.

As a last step (step 5), all measured data, collected from various sources, is aggregated and stored in a unified, table-based format. Each row in this table represents exactly one of the tested service

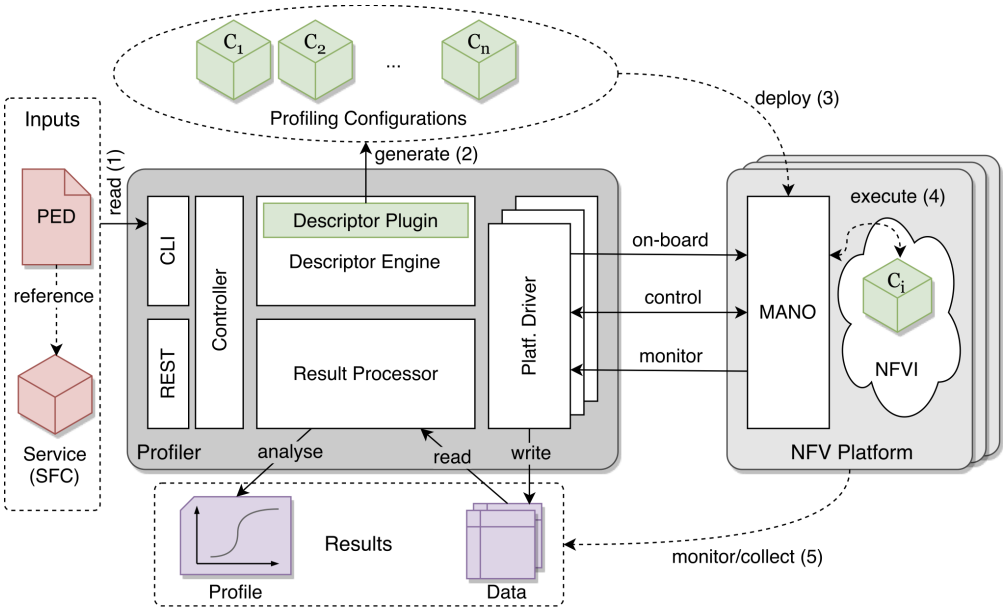

Figure 4.19: System architecture of our profiling system interacting with several NFV platforms

configurations. These tables are then passed to a post-processing module which automatically triggers user-defined analysis scripts that can, for example, perform statistical analysis on the collected datasets.

4.6.1.2 Package Generation

Based on the experiment descriptors and the service specification, our profiling system will generate one service configuration for each combination of parameters that should be tested. The generation of these configurations highly depends on the service descriptor technology used by the target platforms. This is why we use a plugin design to generate the descriptions which allows to implement specific generators for third-party description approaches.

One of the main functionalities of these generators is to extend the network service descriptor of the service under test with additional measurement VNFs. This can be achieved with two different approaches shown in Figure 4.20. The first approach extends the main service graph as such by appending the additional VNFs to the connection points specified in the PED (a). The second approach, in contrast, does not modify the network service descriptor as such but embeds it into another service descriptor that contains the measurement VNFs (b). As shown in the figure, the second approach has a much cleaner design and simplifies the generator implementation. However, it requires that the target platform supports hierarchical service structures.

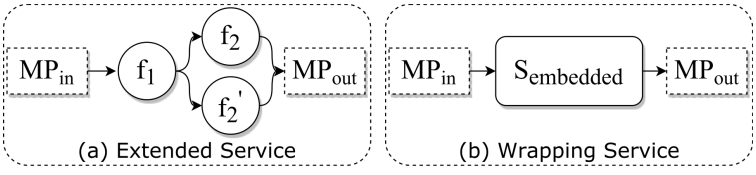

Figure 4.20: Test service generation examples. Extended service descriptor (a) and embedded service (b)

4.6.2 Passive Mode

The *passive* execution mode of `son-profile` is used to dynamically control and monitor a service which is pre-deployed in the SONATA emulator. Resource allocation can be dynamically adjusted. Functional tests can be generated and the set of monitored metrics can be specified. During the tests, the monitored metrics will be statistically analyzed and a summary of the measured results will be generated, giving an indication of the VNF's performance and used resources. In Figure 4.21 we see the results of such a passive profile run. Each measurement point indicates one functional test, where the VNF or service was loaded with a fixed traffic rate during a specified period. On the X-axis the average input load is given, on the Y-axis the measured average CPU load. Error bars indicate the 95% confidence interval of the measurement point. We can see that the confidence interval increases as the host is experiencing more load. This is an important indication, as an overloaded host will not generate representative performance data of the VNF [10]. The use of the Passive Profiling Mode is documented on our GitHub wiki page [1].

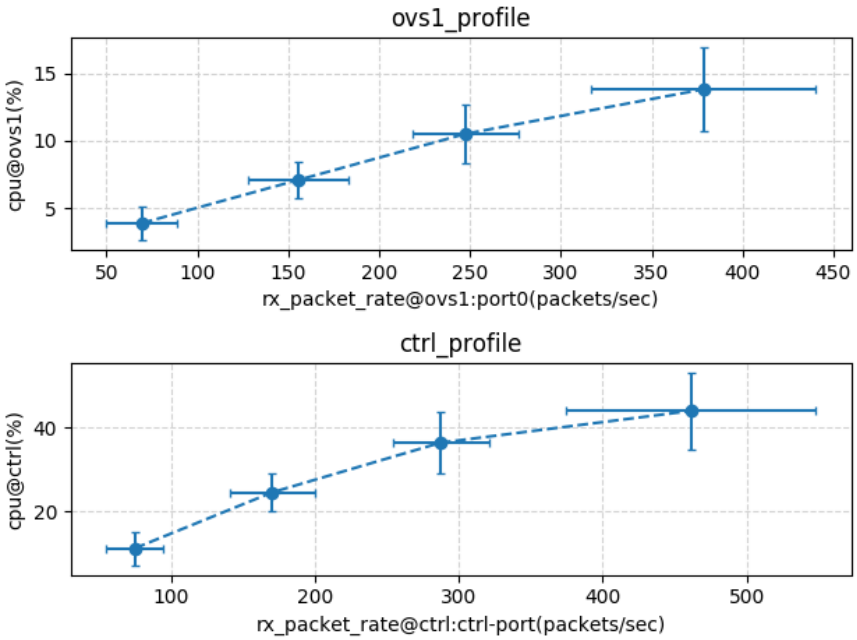

Figure 4.21: Example of passive profiling results

Passive mode options

The basic usage the passive profiling mode is via following CLI command, starting a profiling run on a pre-deployed service with a specified PED file:

```
son-profile -p ped_ctrl.yml --mode passive
```

This is the complete list of options that are implemented for this command:

- `--mode passive`: required to indicated passive mode.
- `-p ped_file.yml`: required, path to the PED file with the profile configuration.
- `--no_display`: optional, if specified no metrics are dynamically displayed during the test runs. (no use of the *ncurses* library to adapt the terminal window).

[1] `https://github.com/sonata-nfv/son-cli/wiki/son-profile:-passive-mode`

- `--graph_only`: optional, if specified no test runs are generated, only a graph is generated from an existing result file.

- `--results_file -r`: optional, path where the results are stored (default: test_results.yml in the working directory).

4.6.2.1 Profiling use-cases

The relation between the VNF's performance and allocated resources is valuable input for resource and capacity planning. It helps to optimally estimate the cost or configuration needed for the required service performance, eg. how many VNF instances or CPU power is needed. Likewise, VNF scaling algorithms can benefit from the performance profile, since they need to do the same estimation.

Configuration and scaling procedures can be tested in the SDK with support of the SONATA emulator as already explained in Section 4.5.5. This was also demonstrated with a vEPC example service at the SOFTNETWORKING 2017 conference and the related presentations are shared on the SONATA GitHub [2]. Service or Function Specific Managers can be plugged into the SONATA SP to execute custom configuration and scaling actions.

VNF profiling helps to decide on the most optimal scaling strategy for a service or VNF to follow. For a scale-in/out procedure for example, it can be tested if state migration and load-balancing mechanisms work for newly added or removed VNFs. Vertical/horizontal scaling can be tested in the SDK by actually changing your service descriptor/graph, adding for example a load-balancer and re-profiling it. Additionally, representative performance monitoring of scaled VNFs can be done by installing the emulator on more performant hardware nodes. The modular architecture and the emulator's REST API allow that `son-cli` and `son-emu` can be installed on different nodes, without sacrificing functionality and monitoring capabilities.

Vertical scaling

Either the `son-profile` PED file or directly the `son-emu` REST API can be used to manipulate the resource allocation of a VNF deployed in the emulator. This is illustrated above in Figure 4.21. Both `son-profile` and `son-monitor` tools are used to automate this, as shown in Figure 4.22. A PED file can be used to automatically monitor a VNF under a defined input workload and a varying allocated CPU or memory share. This is a vertical scaling VNF test to automatically measure the performance under varying resources. The Grafana window in [10] shows the different monitored metrics as time-series. The CPU allocation is shown on the lower graph. A test was done with 10-20-30-40-50% of CPU share allocated to the VNF under test. As the CPU allocation increases, the processed packet rate and throughput (upper graphs) also increase. The smaller window is a plotted summary of this monitored data, generated by the `son-profile` tool, as explained earlier. Each measurement point represents the average value of the VNF performance during the five defined CPU shares. We can see that at the last measurement point, the performance does not increase linearly any more. This phenomenon and detection system was published in [10]. This is reported by the profiling tool as an overloaded host, as the requested 50% of CPU share could not be allocated fully. The resulting performance profile can be used at VNF deployment and scaling, eg. to pick the correct CPU allocation for the required performance.

[2] `https://github.com/sonata-nfv/son-tutorials/tree/master/softnetworking/Presentations`

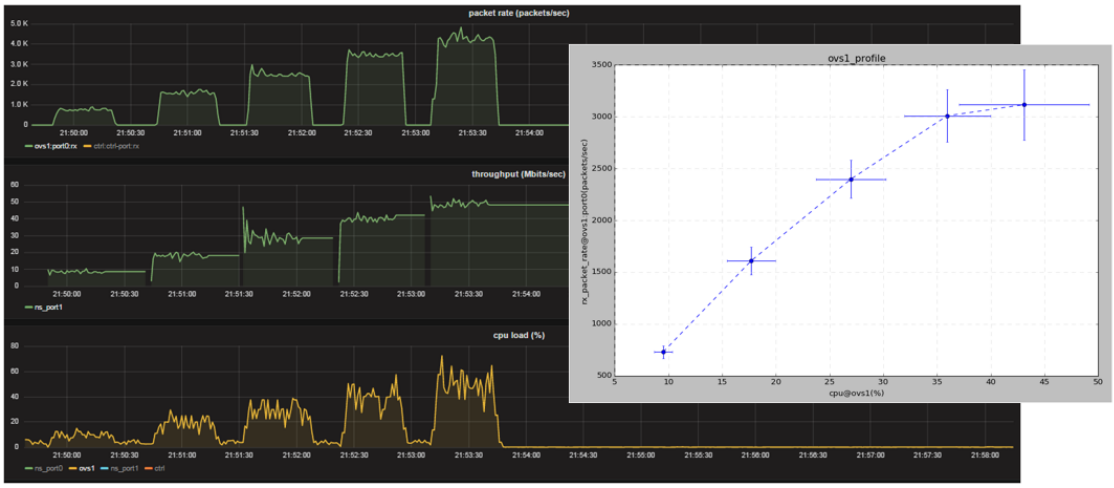

Figure 4.22: Testing a VNF under varying allocated resources using the SONATA SDK

Horizontal scaling

The test-concept for horizontal scaling in the SONATA SDK is illustrated in Figure 4.23. A scale-out action of a VNF or service can for example include the addition of a load-balancer. To test and debug this action, the editor can be used to change the number of VNFs in the service graph. This change is stored in the descriptor and the service package. To test the scaled-out service, a load-balancer VNF can be included in the service itself and modified service package can be deployed on the emulator via the Dummy Gatekeeper. Alternatively, the emulator's REST API can be used to deploy extra VNFs and install flow-rules steering traffic to the different VNFs. That way, the networking configuration of the virtual datacenters in the emulator can be manipulated to route test traffic to any deployed VNF. Similarly, the placement of the scaled VNFs unto different emulated datacenters can also be controlled by plugging-in a customized placement function in the dummy gatekeeper module of the emulator.

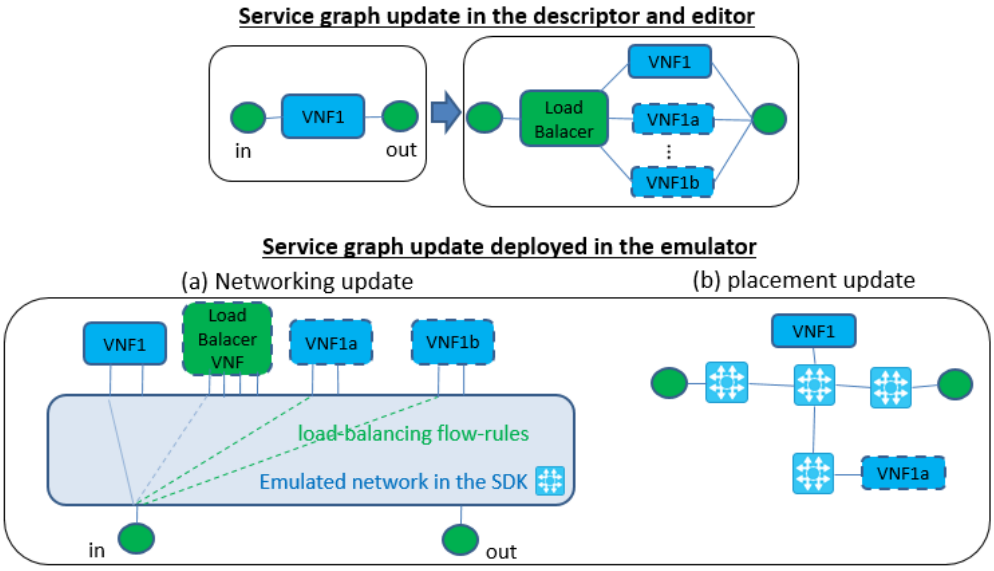

Figure 4.23: Various scale-out possibilities in the SONATA SDK

4.6.3 Profile Experiment Descriptor (PED file) updates

To support automatic functional testing and monitoring, the PED file was introduced. It is a flexible, YAML-based configuration file that holds all the necessary info to configure the VNFs, specify monitored metric and generate traffic load. The PED files define how a given network service should be automatically profiled. The structure of this PED file was already documented in section 3.5.2 of [4]. In this section we highlight the main updates compared to previous release. Several new fields are introduced in the PED file to fully support the passive profiling mode:

The main parts of the PED file are described below. Some example PED files and CLI commands to use the son-profile tool are given in the appendix: Appendix C

- `service_experiments`: In the passive mode, only service experiments are executed.
- `time_limit`: each test run will last this period of seconds. During this period the specific test traffic is generated and metrics are gathered. Statistics such as average and confidence interval are calculated over this interval for each metric.
- `measurement_points`: Each item in reflects a VNF in the service for which several configurations need to executed during each test run.
  - `name`: Name of the VNF in the emulator for which the configuration will be adapted.
  - `configuration`: List of general configuration commands that are executed once, at the beginning of the profiling.
  - `cmd`: Configuration command that is executed at the start of each test run.
  - `cmd_order`: VNF order for which the 'cmd' needs to be executed.
- `resource_limitations`: For each of the VNFs in the service, resource limitations can be specified. These include CPU and MEM limits.

The next fields are used to specify the metrics that need to be monitored during the profiling run. They are given in the format of an MSD file. This file was introduced in section 3.6.2 of [4]. The format of the MSD file is documented on our GitHub wiki page: `https://github.com/sonata-nfv/son-cli/wiki/son-monitor:-msd-file`

- `input_metrics`: MSD file that describes the metrics to be monitored for the 'input' traffic of the service. These metrics reflect the generated traffic or resource constraints that are given as input to the service during the test runs. These metrics can be monitored during the test using Grafana.
- `output_metrics`: MSD file that describes metrics to be monitored during the test runs. These metrics reflect the effects or load induced by the generated traffic. These metrics can be monitored during the test using Grafana.

Additional fields are defined to configure and plot the results at the end of the profile tests.

- `profile_calculations`: For each of the items in this field, a plot is made at the end of the profiling run. Each plot needs a specification of which metrics to put on the X/Y axis.

4.7 son-monitor

In this section we describe the different updates done in the SDK monitoring framework. The actual implementation efforts of `son-monitor` are located in different repositories, mainly `son-cli` and `son-emu`. The main architecture of son-monitor was already presented in section 3.6 of [4]. The green boxes in Figure 4.24 indicate where in the architecture updates have been implemented, compared to previous release. Technical implementation details are given in Appendix D. We give a short description of each update:

- **Metrics Gateway and Websocket interface**: A mechanism is developed to stream data from the SONATA Service Platform to the SDK. This is further described in the section below.
- **Test VNFs**: New traffic generating scripts are going to be implemented to support the development of the Pilot Services in the SONATA project.
- **External Service Access Points**: As indicated in Section 4.5.3, the emulator now support External Access Points that can be used to attach traffic generating processes or scripts that running outside of the emulation environment, on the host itself or on a different machine.
- **VNF Resource Starvation Monitor**: This implements a detection mechanism to timely stop the profile run when the host machine gets overloaded. A dedicated monitor function can be started that monitors the CPU usage of a Docker VNF. By statistically analyzing the CPU usage during a short timeframe, resource starvation can be detected [10]. This can occur for example during a profile run, when a VNF gets less allocated resources then initially requested, because the physical node is saturated. In this case, the monitored VNF performance is not valid any more since it only represents an undefined fraction of the total requested resources. If detected in time, the profile run should stop and monitored metrics should be omitted from the performance profile result.
- **Profile and Monitoring Descriptor**: Several bug fixes and small updates were implemented that allow the correct execution and monitoring of profiling tests.

4.7.1 Streaming Monitor Data from the Service Platform

So far, SDK environment was only capable of monitoring and analyzing services deployed in the SDK emulator. Now we also implemented the functionality to stream monitored data from a service deployed in the Service Platform to the SDK for further analysis. Next to the types of metrics that should be monitored for a service, the SONATA Service Platform (SP) also allows to specify multiple alerts (thresholds) inside the service descriptor. The Monitoring Manager inside the SP will send the Alert messages, triggered by the specified alert thresholds. This is the default monitoring system inside the SP. As reaction to an alert, or just to check, the service developer using the SDK can also request monitoring data from the SP. This way, the tools in the SDK can help to further analyse or debug the service performance. By using credentials, access control is implemented by the Service Platform Gatekeeper. This way, an SDK user gets only access to an allowed set of deployed services and VNFs (eg. only the services uploaded and instantiated by this specific SDK user). The SONATA SDK has two possibilities to retrieve monitoring data from the SP. These two ways of monitoring refer directly to the Gatekeeper API, as documented in in D4.3 (section 2.1.5):

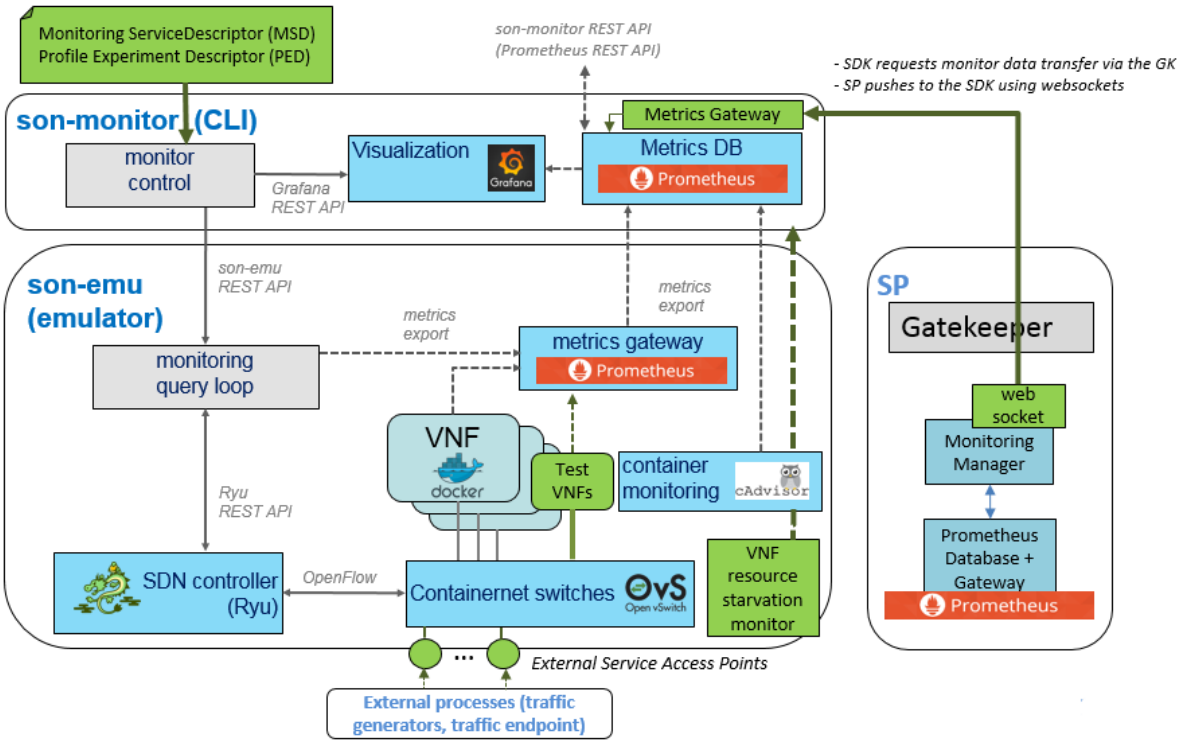

Figure 4.24: son-monitor updates

- **Asynchronous request**: Using the REST API of the Gatekeeper, a request can be made to the Monitoring Manager to return a list of historical monitored metrics stored in the metrics database (eg. metric gathered during the last 5 mins). The `son-monitor query` command implements this in the SDK.
- **Synchronous stream**: This is implemented by streaming (during a limited timeframe) live monitored data. In this section we further explain this feature. The `son-monitor stream` command implements this in the SDK.

As illustrated in Figure 4.24 , a websocket on the Service Platform is made available on request from the SDK. The SDK can then connect to this websocket and receive pushed metrics from services deployed in the Infrastructure controlled by the SP. Access control, authZ/authC is implemented via the Gatekeeper and the SDK `son-access` tool. The Gatekeeper API remains the single-point-of-access where the SDK can connect to, to retrieve data from the SP. The received stream of metrics from the websocket on the Service Platform needs is stored in the Prometheus DB in the SDK. This is also described in D2.3 [2] (section 3.2.5) and D4.2 [6] (section 7). On the SDK-side a dedicated Metrics Gateway is started which can connect to the websocket and store any received metric in the SDK Prometheus Database. Once the streamed data is stored in the SDK Prometheus DB, it can be queried for further analysis.

To support the transfer of monitoring data from the SP to the SDK, a new set of `son-monitor` CLI commands has been implemented. In the VNF descriptor (VNFD), a metric is part of a specific Virtual Deployment Unit (VDU). This VDU is the VM image that is deployed and for which metrics are exported. In the SONATA service model, a VDU can have multiple instances. At scale-out for example, the number of VDU instances can be increased. A VDU instance is referred to as a VNF Component (VNFC) in the VNF Record (VNFR). An exported metric belongs to a specific VNFC.

Due to the specifics of this ETSI-based SONATA service model, a metric is therefore uniquely defined by the tuple `vnf_instance_id`, `vdu_id` and `vnfc_id`. The developer should inspect the records and descriptors of the deployed service to identify the available metrics and the deployed VNFC's. In the VNFR, it can be found which VNFC ID is of interest to monitor. In the VNFD, the available metrics are listed. All options of the commands are listed in Section D.3 .A summary of these commands is given below:

Table 4.7: son-monitor commands

CLI command	Description
son-monitor query	Query the Prometheus DB in the Service Platform or the SDK for historically monitored metric values. A JSON formatted string is returned, containing the requested metric data. When addressing the SONATA Service Platform, the request is authenticated via the Gatekeeper and the command queries the correct UUID's for the VNF and VDU instance where the metric belongs to. Using those UUID's, the Gatekeeper's asynchronous monitoring API can be addressed.
son-monitor stream	Connect to the monitoring manager in the Service Platform and receive the live monitored metric values via a websocket stream. The specified metric and VNF are filtered from the output stream. The filtered output value is then exported to the SDK Prometheus database. This command is blocking, it waits to exit until the websocket is closed. To avoid the blocking, the command can be executed in the background. The request is authenticated via the Gatekeeper and the command queries the correct UUID's for the VNF and VDU instance where the metric belongs to. Using those UUID's, the Gatekeeper's synchronous monitoring API can be addressed.
son-monitor stream stop	Stop all running websocket streams and stop exporting to Prometheus.

4.8 son-analyze

This section presents the majors contributions to the `son-analyze` tool following the D3.2 release.

4.8.1 `son-access`'s new authentication

The `son-analyze` tool has been updated to follow the latest `son-access` update. `Son-access` has introduced a new authentication mechanism to increase security while accessing the SONATA service platform and the SONATA SDK. As the `son-analyze` library has to connect to the SONATA GateKeeper to retrieve historical monitoring data, these new features needed to be integrated.

4.8.2 Support for MSD and NSD files

The `son-analyze` core library has been updated to support services' NSD and MSD files. This feature allows the service developer to explore his service from those files and facilitate the construction of queries. For example, the `nsd_links/link_ids` fields describing a probe in the MSD file may be directly used to build a query to retrieve the last week historical data for this specific metric. Whereas, the support of MSD files allow to write generic functions to increase code reuse. For example a function that takes a service MSD, retrieves all the VNFs metrics to build an aggregated view / summary of the overall resource usage.

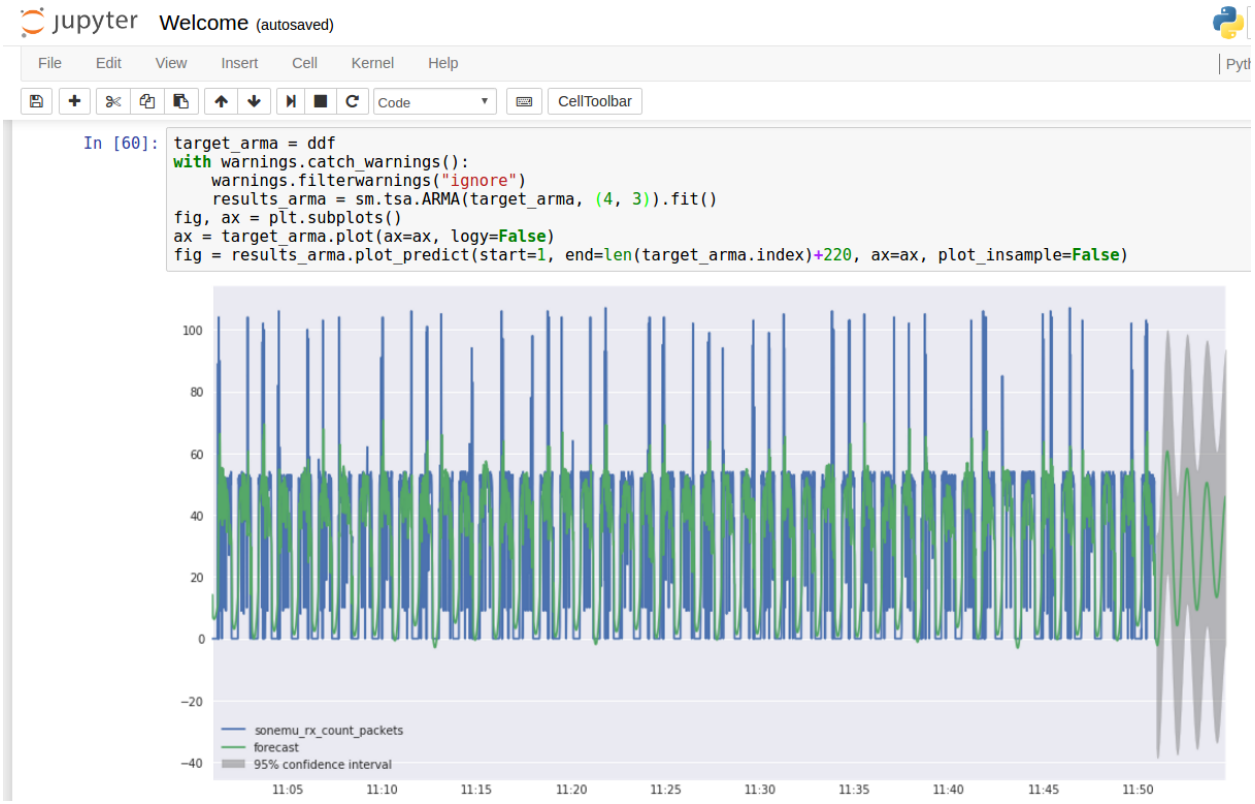

Figure 4.25: Forecasting the sonemu_rx_count_packets metric using son-analyze

4.9 son-sm

This section describes tools that we provided to support developers for creating Functions-Specific Managers (FSMs) and Service-Specific Managers (SSMs). FSMs and SSMs are container-based programmes, implemented by network service developers, aiming at improving the programmability of the lifecycle management of Virtualized Network Functions (VNFs) and network services, respectively.

FSM/SSM supporting tools include an FSM/SSM template and some examples. The template provides a naming pattern for FSMs/SSMs and takes care of their registration into the service platform. FSM/SSM examples, on the other hand, show developers how to use the template and provide some functionalities such as fetching the monitoring data from the Service Platform's monitoring plugin, (re)configuring VNFs, and service placement. The following sub-sections explain these tools more in details.

4.9.1 FSM/SSM Template

FSMs/SSMs are Service Platform's external plugins implemented by developers that are managed by a special Service Platform's plugin, called Specific Manager Registry (SMR). SMR is responsible for managing the lifecycle of FSMs/SSMs including onboarding, instantiation, updating, and termination. To provide a sophisticated management, SMR requires collecting a couple of metadata from FSMs/SSMs which should be provided by the FSM/SSM developers. These metadata are then used to register FSMs/SSMs into the Service Platform. Furthermore, to avoid SSM/FSM name collision, developers should provide unique names for their FSMs/SSMs.

To this end, the FSM/SSM template helps developers to provide the required metadata and unique names for their FSMs/SSMs. These functionalities are described in the following sections.

4.9.1.1 Registration

The registration procedure consists of two steps including collecting the metadata during the FSM/SSM implementation and forwarding them to the SMR during the FSM/SSM instantiation. Table 4.8 includes the required metadata and a short explanation of them.

Table 4.8: FSMs/SSMs metadata

Metadata	Description	Required for	Provided by
`specific_manager_type`	the FSM/SSM type which could be either fsm or ssm.	FSMs/SSMs	Developer
`service_name`	the name of the service that the FSM/SSM belongs to	FSMs/SSMs	Developer
`function_name`	the name of the VNF that the fsm belongs to	FSMs	Developer
`specific_manager_name`	the actual fsm/ssm name (e.g., placement, scaling)	FSMs/SSMs	Developer
`id_number`	an id number to differentiate FSMs/SSMs developed for the same purpose)	FSMs/SSMs	Developer
`version`	version of the FSM/SSM	FSMs/SSMs	Developer
`description`	the FSM/SSM description	FSMs/SSMs	Developer
`uuid`	the FSM/SSM unique identifier	FSMs/SSMs	SMR
`sfuuid`	the unique identifier of service/VNF that the FSM/SSM belongs to	FSMs/SSMs	FLM/SLM
`update_version`	needs the be populated by `true` if the FSM/SSM is an updated version of existing FSM/SSM	FSMs/SSMs	Developer

The actual registration process happens during the FSM/SSM instantiations. Once the FSM/SSM container is started, the template comes into play and sends all the metadata provided by the developers to the SMR through the message broker. The SMR receives the metadata, generates a unique identifier for FSM/SSM and store a record of it. If no error happens during the registration, SMR sends a `Registration_OK` response to the FSM/SSM template and the template, then, triggers the actual function of the FSM/SSM. Figure 4.26 depicts the workflow of FSM/SSM registration.

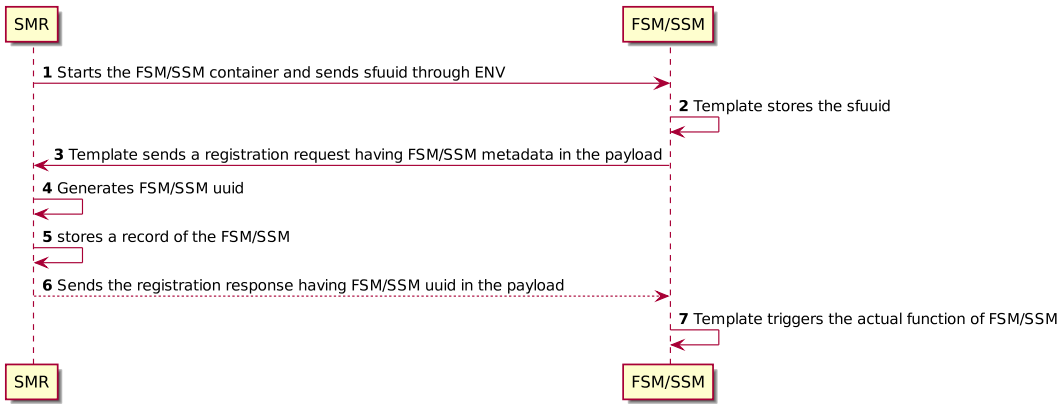

Figure 4.26: Sequence diagram for FSM/SSM registration

4.9.1.2 Naming Pattern

The Template also provides a naming pattern that helps developers to provide meaningful and unique names for FSMs/SSMs in order to avoid name collision in the Service Platform. The pattern contains multiple metadata that is combined to create the FSM/SSM unique names.

The SSM and FSM names are a combination of the following items, respectively:

- SSM name: son, [`specific_manger_type`],[`service_name`],[`specific_manager_name`], [`id_number`]
- FSM name: son, [`specific_manger_type`],[`service_name`],[`function_name`],

[`specific_manager_name`], [`id_number`]

The description of each item is already given in Table 4.8. Note that, the name provided by the template should be included in the NSD/VNFD as the FSM/SSM id. The following example shows how FSM names are embedded into a VNFD.

```
function_specific_managers:
  - id: "sonfsmservice1firewallplacement1"
    description: "placement FSM for firrewall"
    image: "hadik3r/sonfsmservice1firewallplacement1"

  - id: "sonfsmservice1firewallscaling1"
    description: "scaling FSM for firrewall"
    image: "hadik3r/sonfsmservice1firewallscaling1"
```

The name created by the template will be sent to the SMR during the FSM/SSM registration, and the SMR combines the name with the service UUID. This combination constructs a unique name which avoids the FSM/SSM names collision in the Service Platform.

4.9.2 SSM/FSM Examples

We have provided some examples of FSMs/SSMs to support developers in creating FSMs/SSMs with different functionalities. Developers can create their desire FSMs/SSMs by adjusting the examples to their needs and requirements. These examples include the following:

4.9.2.1 Dumb SSM

This example shows how the developer can use the FSM/SSM template. It provides the FSM/SSM metadata and uses the template for registration. Developers can use this example to create any SSM by overwriting the `on_registration_ok` function of this example. An FSM version of this example is also provided which can be used for FSM development.

4.9.2.2 Placement SSM

The placement SSM, besides using the template for registration, subscribes to the corresponding RabbitMQ topic, receives placement request and sends the placement decision to the placement Executive (Service Platform's plugin). Figure 4.27 shows the workflow of Placement SSM.

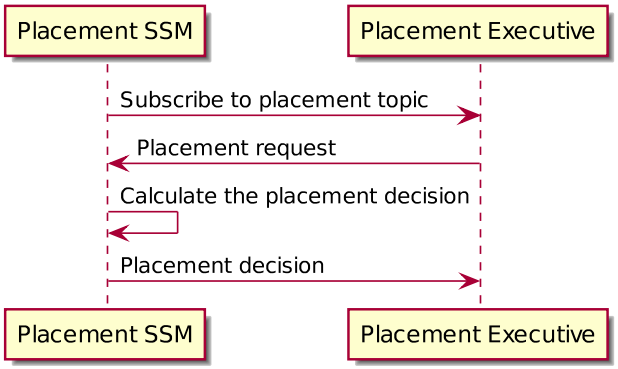

Figure 4.27: Sequence diagram for placement SSM example

4.9.2.3 Configuration FSM

This example shows how an FSM can retrieve the IP address of a VNF which can be used for (re)configuration of VNFs. VNFs' IP addresses are embedded in the VNF Record which can be requested from the Service Platform plugins, Function Lifecycle Manager (FLM) or Service Lifecycle Manager (SLM). To this end, configuration FSM subscribes to FLM generic topic and requests the VNFR. Once the FSM receives the VNFR, it retrieves the corresponding IP address from it. Figure 4.28 shows the workflow of configuration FSM.

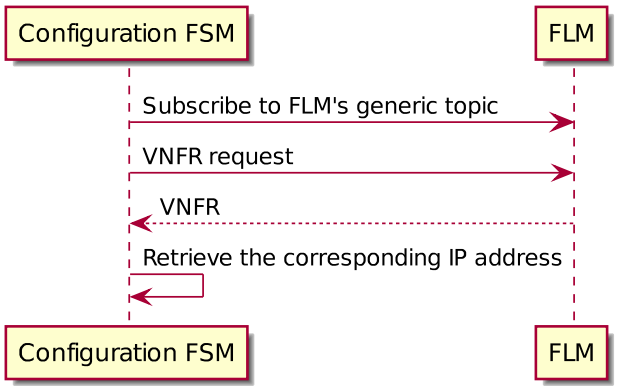

Figure 4.28: Sequence diagram for configuration FSM example

4.9.2.4 Monitoring FSM

The monitoring example, besides performing the registration, subscribes to the Monitoring topic for receiving monitoring alerts and reacts to the alerts regarding the CPU usage higher than 85%. The workflow of monitoring FSM example is shown in Figure 4.28

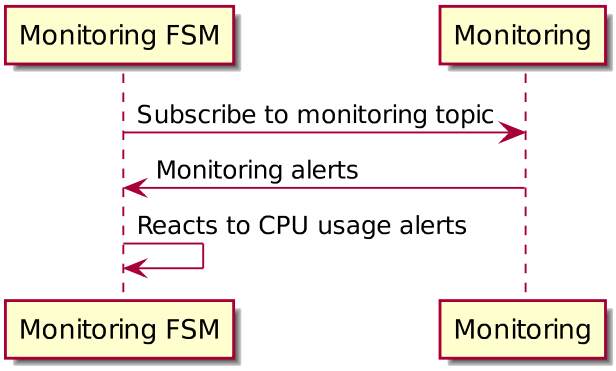

Figure 4.29: Sequence diagram for monitoring FSM example

5 The future of the SONATA SDK

This deliverable documents the last release of the SDK during the SONATA project. However, as the codebase of the SDK is open-source, the SDK will continue to evolve and will be extended in various ways beyond the duration of the project. This section therefore will sketch: i) how to contribute as a developer to the SDK, ii) where to find future documentation related to the SDK, and iii) an initial vision and set of potential features which might be included in future releases of the SDK.

5.1 How to contribute to the development of the SONATA SDK

As mentioned above, SONATA follows an open-source model. The SONATA software, which is available under Apache 2.0 license, is hosted publicly on GitHub. Thus, everyone is invited to contribute code, bug-fixes, and even extensions to the project. Moreover, contributions to documentation and tests are more than welcome.

In order to contribute code in general, a (outside) developer has to follow regular GitHub project process. To this end, one has to fork the project, implement changes on a clone of the fork, and finally create and send pull requests back to the SONATA main branch. Pull requests are then tested automatically by the SONATA Jenkins infrastructure. When all tests are past, one of the lead developers of the (sub-) project has to accept the pull request and merge it with the master branch. Lead developers today are the original SONATA members that once created the (sub-) project and the respective GitHub repository. However, any other person can become a lead developer by election, once a reasonable credibility and trust has been established with the current lead developers. Thus, SONATA opens itself to an external community. And since the SONATA project has created interest in other projects, like 5GTANGO, we expect the SONATA infrastructure, like the Jenkins test system, to be in place at least for the next 2.5 years.

5.2 Where to find up-to-date documentation material

Recent documentation as well as code can be found at the GitHub website at `https://github.com/sonata-nfv`. One may find additional sources of information in the respective README files of each and every repository as well as in the corresponding GitHub Wiki pages. All the relevant and immediate information goes in there. In addition, the SONATA website at `http://www.sonata-nfv.eu` acts as a resource for additional information such as contact information and mailing lists.

5.3 Future work

The SONATA SDK has been developed in a rapidly changing environment. As a result, components might be extended to accommodate a changed environment, as well as to improve their feature set. Below we give a short overview per component on potential future work.

Table 5.1: Potential ideas for extending the SDK components beyond the final release

SDK component	Potential future changes and/or extensions
`son-schema`	• The schema defining the SONATA programming model in the form of its various descriptors, is likely to be updated frequently. Likely extensions are changes in order to unify the descriptors with those from ETSI NFV, OSM, ONAP, or other project which happen to be the most adopted one.
`son-editor`	• Integration with additional tools, enabling direct activation from one `son-editor`-based Integrated Development Environment.
`son-package`	• Support for creating packages for alternate Service/MANO Platforms, such as OSM or ONAP.
`son-validate`	• Validation of additional (in-)consistencies in the tested packages such as inconsistencies in the Network Function Forwarding Graph. • Sanity check of FSM and SSM code.
`son-access`	• Support for interaction with alternate Service or MANO Platforms.
`son-emu`	• Support for alternate execution environments besides Docker containers.
`son-monitor`	• Support for test automation, and integration with validation functionality. • Creation of customized monitoring agents that can be hot-plugged into a service. • Creation of traffic generating VNFs that can be hot-plugged into a service.
`son-profile`	• Automatic derivation of performance extrapolation trends and/or suggestions for improving performance. • Service-integrated, automated performance checks against pre-defined values.
`son-analyze`	• Including statistical analysis optimizations for particular use cases. • Analysis of captured packet streams (in addition to captured metric values)

6 Conclusion

The SONATA SDK environment comprehends a set of tools and features that **support the end-to-end lifecycle of NFV-based telecom services**. This is shown in Figure 6.1. Using the Service Package, a service can be deployed both in the SDK emulator as in the SONATA Service Platform itself. This enables a DevOps-way-of-working where NFV-based services can be quickly created, edited, updated and validated.

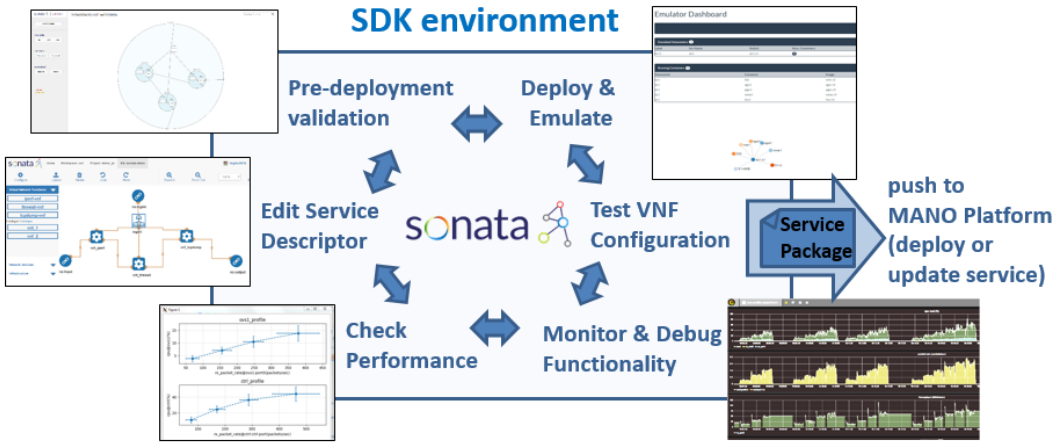

Figure 6.1: The SONATA SDK environment

The service functionality is audited in several stages of the development cycle in the SDK. The process is started with the setup of a workspace in the developer's machine, where different services can be edited and grouped into user-defined projects. The files in this workspace describe the total network service and can be manipulated manually or with the graphical editor tools provided by the project. Once the VNF and service descriptors are ready, the resulting project is packaged. Additionally, validation tools make sure that the network service graph and package is valid and secure, before being deployed on the emulator or service platform. Next, once deployed (on the emulator or service platform), validation includes configuration, monitoring, profiling and analysis of the overall service functionality. This complete set of features decreases development time and allow the creation of a fully validated service package, ready to be deployed on a production-focused Service Platform. In order to support production-ready environments, a wide range of features in this SDK release **focus on security-related aspects** such as user authorization, authentication, and the possibility to sign and verify developed service and NF packages.

Although most of the SDK tools focus on the SONATA Service/MANO Platform, many tools are **prepared to interoperate beyond the scope of SONATA borders**. The SDK emulator (`son-emu`), for example, has been extended to support OSM. Other tools, such as the packaging component are prepared to be extended to other platforms beyond the initial scope of SONATA. Particular attention has been paid to documentation, and assuring good processes are in place enabling external contribution in order to extend the lifetime of the SDK beyond the SONATA project.

A Technical Details of son-cli

A.1 son-access

A.1.1 Usage

The next lines are a short guide that briefly describes the main commands to properly use `son-access` component

```
usage: son-access [optional] command [<args>]
      The supported commands are:
         auth     Authenticate a user
         list     List available resources (service, functions, packages, ...)
         push     Submit a son-package or request a service instantiation
         pull     Request resources (services, functions, packages, ...)
         config   Configure access parameters
positional arguments:
command               Command to run

optional arguments:
-h, --help            show this help message and exit
-w WORKSPACE_PATH, --workspace WORKSPACE_PATH
                      Specify workspace to work on. If not specified will
                      assume '/root/.son-workspace'
-p PLATFORM_ID, --platform PLATFORM_ID
                      Specify the ID of the Service Platform to use from
                      workspace configuration. If not specified will assume
                      the ID in 'default_service_platform'
--debug               Set logging level to debug
```

The `son-access` tool supports five different subcommands to deal with authentication, listing of resources, uploading of resources, requesting of resources and configuration of access parameters.

- Authentication - `auth`

```
usage: son-access [..] auth [-h,-p] -u USERNAME -p PASSWORD
```

- Authenticate a user

```
optional arguments:
-h, --help            show this help message and exit
-u USERNAME, --username USERNAME
                      Specify username of the user
-p PASSWORD, --password PASSWORD
                      Specify password of the user
--logout              Ends access token lifespan
```

- Configure parameters - `config`

```
usage: son-access [..] config [-h] (--platform_id SP_ID | --list) [--new]
                              [--url URL] [-u USERNAME] [-p PASSWORD]
                              [--token TOKEN_FILE] [--default]
```

- Configure access parameters

```
optional arguments:
-h, --help            show this help message and exit
--platform_id SP_ID   Specify the Service Platform ID to configure
--list                List all Service Platform configuration entries
--new                 Create a new access entry to a Service Platform
--url URL             Configure URL of Service Platform
-u USERNAME, --username USERNAME
                      Configure username
-p PASSWORD, --password PASSWORD
                      Configure password
--token TOKEN_FILE    Configure token filename (deprecated)
--default             Set Service Platform as default
```

- List resources - `list`

```
usage: son-access [..] list [-h] resource_type
```

- List available resources (services, functions, packages, ...)

```
positional arguments:
resource_type  (services | functions | packages)

optional arguments:
-h, --help     show this help message and exit
```

- Submit packages - `push`

```
usage: son-access [..] push [-h]
(--upload PACKAGE_PATH [--sign] | --deploy SERVICE_ID)
Submit a son-package to the SP or deploy a service in the SP

positional arguments:
--upload PACKAGE_PATH        Specify package path to submit
--upload PACKAGE_PATH --sign Specify package path to sign and submit
--deploy SERVICE_ID          Specify service identifier to instantiate

optional arguments:
-h, --help  show this help message and exit
--sign      Indicates if the package will be signed with user's private key
```

- Request resources - `pull`

```
usage: son-access [..] pull [-h] (--uuid UUID | --id VENDOR NAME VERSION)
                            resource_type
```

- Request resources (services, functions, packages, ...)

```
positional arguments:
resource_type         (services | functions | packages)

optional arguments:
-h, --help            show this help message and exit
--uuid UUID           Query value for SP identifiers (uuid-generated)
--id VENDOR NAME VERSION
                      Query values for package identifiers (vendor name
                      version)
```

Some examples on how to configure a new platform, authenticate a user, submit a package file and retrieve resources:

```
son-access config --platform_id sp1 --new --url http://127.0.0.1:5001 --default
son-access auth -p sp1 --username tester --password 1234
son-access list services
son-access push --upload samples/sonata-demo.son
son-access push --upload samples/sonata-demo.son --sign
son-access -p sp1 push --upload samples/sonata-demo.son
son-access pull packages --uuid 65b416a6-46c0-4596-a9e9-0a9b04ed34ea
son-access pull services --id sonata.eu firewall-vnf 1.0
son-access -p sp1 push --deploy 65b416a6-46c0-4596-a9e9-0a9b04ed34ea
```

A.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
```

A.3 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
```

A.4 son-validate

son-validate can be used through the CLI or as a micro-service running inside a docker container.

A.4.1 son-validate CLI

The CLI interface is designed for developer usage, allowing to validate SDK projects, package descriptors, service descriptors and function descriptors. It receives the following arguments:

```
usage: son-validate [-h] [-w WORKSPACE_PATH]
      (--project PROJECT_PATH | --package PD | --service NSD | --function VNFD)
      [--dpath DPATH] [--dext DEXT] [--syntax] [--integrity]
      [--topology] [--debug]

Validate a SONATA Service. By default it performs a validation to the syntax,
integrity and network topology.

optional arguments:
  -h, --help            show this help message and exit
  -w WORKSPACE_PATH, --workspace WORKSPACE_PATH
                        Specify the directory of the SDK workspace for
                        validating the SDK project. If not specified will
                        assume the directory: '$HOME/.son-workspace'
  --project PROJECT_PATH
                        Validate the service of the specified SDK project. If
                        not specified will assume the current directory.
  --package PD          Validate the specified package descriptor.
  --service NSD         Validate the specified service descriptor. The
                        directory of descriptors referenced in the service
                        descriptor should be specified using the argument '--
                        path'.
  --function VNFD       Validate the specified function descriptor. If a
                        directory is specified, it will search for descriptor
                        files with extension defined in '--dext'
  --dpath DPATH         Specify a directory to search for descriptors.
                        Particularly useful when using the '--service'
                        argument.
  --dext DEXT           Specify the extension of descriptor files.
                        Particularly useful when using the '--function'
                        argument
  --syntax, -s          Perform a syntax validation.
  --integrity, -i       Perform an integrity validation.
  --topology, -t        Perform a network topology validation.
  --debug               sets verbosity level to debug
```

The different levels of validation, namely syntax, integrity and topology can only be used in the following combinations:

- syntax `-s`
- syntax and integrity `-si`
- syntax, integrity and topology `-sit`

The son-validate tool can be used to validate one of the following components:

- **project** - to validate an SDK project, the `--workspace` parameter must be specified, otherwise the default location `$HOME/.son-workspace` is assumed.
- **service** - in service validation, if the chosen level of validation comprises more than syntax (integrity or topology), the `--dpath` argument must be specified in order to indicate the location of the VNF descriptor files, referenced in the service. Has a standalone validation of a service, son-validate is not aware of a directory structure, unlike the project validation. Moreover, the `--dext` parameter should also be specified to indicate the extension of descriptor files.
- **function** - this specifies the validation of an individual VNF. It is also possible to validate multiple functions in bulk contained inside a directory. To if the `--function` is a directory, it will search for descriptor files with the extension specified by parameter `--dext`.

Some usage examples are as follows:

validate a project: `son-validate --project /home/sonata/projects/project_X --workspace /home/sonata/.son-workspace`

validate a service: `son-validate --service ./nsd_file.yml --path ./vnfds/ --dext yml`

validate a function: `son-validate --function ./vnfd_file.yml --dext yml`

validate multiple functions: `son-validate --function ./vnfds/ --dext yml`

A.4.2 son-validate Service

son-validate can be executed as a service, providing a RESTful interface to validate objects and retrieve validation reports. son-validate API service can be executed in two distinct modes: `stateless` or `local`. Stateless mode will run as a stateless service only and can be instantiated at any remote location. Local mode is designed to run in the developer OS, providing additional functionalities. It aims to provide automatic monitoring and validation of local SDK projects, packages, services and functions. Automatic monitoring and validation can be enabled in workspace configuration, specifying the type of validation and which objects to validate. This functionality watches for changes in the specified objects automatically triggering the validation process as required.

A.4.2.1 Configuration

The Dockerfile for the son-validate service lives at the root of son-cli project at directory: `tools/validator.Dockerfile` since it has dependencies with some modules of this project. Configuration is done using the following environment vars inside the Dockerfile:

- `VAPI_HOST`: the binding IP address for the service, default is 0.0.0.0
- `VAPI_PORT`: the listening port for the service, default is 5001
- `VAPI_CACHE_TYPE`: type of caching to be used, default is `redis`
- `VAPI_ARTIFACTS_DIR`: working directory, where temporary artifacts will be stored (auto removed on program exit). Default is `./artifacts`
- `VAPI_DEBUG`: set verbose level to debug, default is `False`

A.4.2.2 Run son-validate API service

son-validate-api has the following usage:

```
usage: son-validate-api [-h] [--mode {stateless,local}] [--host HOST]
                        [--port PORT] [-w WORKSPACE] [--debug]

SONATA Validator API. By default service runs on 127.0.0.1:5001

optional arguments:
  -h, --help            show this help message and exit
  --mode {stateless,local}
                        Specify the mode of operation. 'stateless' mode will
                        run as a stateless service only. 'local' mode will run
                        as a service and will also provide automatic
                        monitoring and validation of local SDK projects,
                        services, etc. that are configured in the developer
                        workspace
  --host HOST           Bind address for this service
  --port PORT           Bind port number
  -w WORKSPACE, --workspace WORKSPACE
                        Only valid in 'local' mode. Specify the directory of
                        the SDK workspace. Validation objects defined in the
                        workspace configuration will be monitored and
                        automatically validated. If not specified will assume
                        '/home/lconceicao/.son-workspace'
  --debug               Sets verbosity level to debug
```

Please notice that specified arguments will override environment variables, if defined.

Run in stateless mode

To execute son-validate as a local service simply run: `son-validate-api --mode stateless`

Run in local mode

To execute son-validate as a local service simply run: `son-validate-api --mode local`

A.4.2.3 Workspace configuration

When running in local mode, automatic monitoring and validation of objects may be set up in the workspace configuration, under the `validate_watchers` key. Example for validating a project and a service:

```
validate_watchers:
    ~/sonata/sdk-projects/sample-project:
        type: project
        syntax: true
        integrity: true
        topology: true
    ~/sonata/sdk-projects/objects/nsds/sample-nsd.yml
        type: service
        syntax: true
```

A.4.2.4 API

The service API accepts the following requests:

- `/validate/object_type` [POST]: validate an SDK `project`, a `package`, a `service` or a `function` specified by object_type
- `/report` [GET]: provides a dictionary of available validated objects
- `/report/result/<resource_id>` [GET]: provides validation results of
- `/report/topology/<resource_id>` [GET]: provides the validated network topology graph of
- `/fwgraphs` [GET]: provides the validated forwarding graphs structure of
- `/resources` [GET]: retrieves the cached validation resources
- `/watches` [GET]: retrieves watched resources

A.4.3 Event configuration

son-validate enables the customization of validation issues to be reported by a user-defined level of importance. Each possible validation event can be configured to be reported as `error`, `warning` or `none` (to not report). Event configuration is defined in the file `eventcfg.yml`. For now, it can only be configured statically but in the future we aim to support a dynamic configuration through the CLI and service API.

The validation events are described in Table Table A.1.

Table A.1: Event codes and description

Code	Description
evt_project_service_invalid	invalid service descriptor in project
evt_project_service_multiple	multiple service descriptors in project
evt_package_format_invalid	invalid package file format
evt_package_struct_invalid	invalid package file structure
evt_package_signature_invalid	invalid package signature
evt_pd_stx_invalid	invalid package descriptor (PD) syntax
evt_pd_itg_invalid_reference	invalid references in PD
evt_pd_itg_invalid_md5	invalid file MD5 checksums
evt_service_invalid_descriptor	invalid descriptor file
evt_nsd_stx_invalid	invalid service descriptor (NSD) syntax
evt_nsd_itg_function_unavailable	referenced function unavailable
evt_nsd_itg_function_invalid	invalid function
evt_nsd_itg_badsection_cpoints	section 'connection_points' contains invalid references
evt_nsd_itg_badsection_vlinks	section 'virtual_links' contains invalid references
evt_nsd_itg_undeclared_cpoint	use of undeclared connection point
evt_nsd_itg_unused_cpoint	unused connection point
evt_nsd_top_topgraph_failed	failure on building topology graph
evt_nsd_top_topgraph_disconnected	topology graph is disconnected
evt_nsd_top_badsection_fwgraph	section 'forwarding_graphs' contains invalid references
evt_nsd_top_fwgraph_unavailable	section 'forwarding_graphs' not defined
evt_nsd_top_fwgraph_cpoint_undefined	undefined connection point in forwarding graph
evt_nsd_top_fwgraph_position_duplicate	duplicate position index in forwarding graph
evt_nsd_top_fwgraph_cpoints_odd	number of connection points in forwarding graph is odd
evt_nsd_top_fwpath_invalid	forwarding path incompatible with defined topology
evt_nsd_top_fwpath_cycles	cycles found in forwarding path
evt_nsd_top_fwpath_inside_vnf	direct path linking interfaces of the same VNF
evt_nsd_top_fwpath_disrupted	disrupted forwarding path
evt_function_invalid_descriptor	invalid function descriptor file
evt_vnfd_stx_invalid	invalid function descriptor (VNFD) syntax

Code	Description
evt_vnfd_itg_badsection_cpoints	section 'connection_points' contains invalid references
evt_vnfd_itg_badsection_vdus	section 'virtual_deployment_units' contains invalid references
evt_vnfd_itg_vdu_badsection_cpoints	section 'connection_points' in VDU contains invalid references
evt_vnfd_itg_badsection_vlinks	section 'virtual_links' contains invalid references
evt_vnfd_itg_undeclared_cpoint	use of undeclared connection point
evt_vnfd_itg_unused_cpoint	unused connection point
evt_vnfd_itg_undefined_cpoint	undefined connection point
evt_vnfd_top_topgraph_failed	failure on building topology graph
evt_duplicate_cpoint	duplicate connection point
evt_invalid_descriptor	invalid SONATA descriptor

B Technical Details of son-emu

The following sections give more details of the implementation of different updates in `son-emu`.

B.1 Dashboard VNF terminal

In order to execute CLI commands inside a deployed Docker VNF in son-emu, different options exist:

- execute the CLI command from the containernet prompt in son-emu.

```
containernet> vnf_name command
```

- A terminal window can be started for a VNF using the Docker CLI:

```
docker exec -it vnf_name /bin/bash
```

- An xterm terminal window can be started by using son-cli:

```
son-monitor xterm -n vnf_name1 vnf_name2
```

- An xterm window can be started via the REST API of son-emu:

```
GET http://<son-emu-rest-api>/restapi/monitor/term
```

- By double-clicking on a VNF in the son-emu dashboard, a call to the son-emu REST API described above is done.

xterm windows can be forwarded using X11. The detailed configuration is described below. This information is also maintained in the wiki page on GitHub: `https://github.com/sonata-nfv/son-emu/wiki/VNF-configuration-terminal`.

B.1.1 X11 settings for forwarding xterm windows

In case the SONATA SDK is running in a VM, the xterm window can be forwarded to the host using X11. This way an xterm can be started from the son-emu dashboard web interface.

B.1.1.1 SONATA SDK VM settings

X11 needs to be directed to the IP address of the host, by setting the 'DISPLAY' environment variable. `export DISPLAY=<IP_of_host>:0`

B.1.1.2 Host settings

Allow connections from remote hosts () to the Xserver on the host.

- Windows:

An X server needs to be running, eg. Xming Server The IP address of the SONATA VM must be allowed to connect to Xming by adding the IP to: 'C:\Program Files\xming\X0.hosts'

- Linux:

`xhost +` (allow connections from all)
`xhost +<IP_of_VM>` (only allow specific IP)

B.1.1.3 SSH terminal

When accessing the SONATA SDK VM via SSH, X11 can be allowed by using: `ssh -X <IP_of_VM> ...` or via Putty: activate X11 Forwarding in the connection -> SSH -> X11 settings.

B.2 Son-emu Topology File

This information is also maintained in the wiki page on GitHub: `https://github.com/sonata-nfv/son-emu/wiki/son-emu-topology-file`.

This page describes specific details on the creation of a new topology file to startup the SONATA emulator. Example topology files are in the folder 'src/emuvim/examples'.

B.2.1 Create the containernet instance

The following line of code in the topology file creates a containernet instance:

```
net = DCNetwork(controller=RemoteController, monitor=True, enable_learning=False)
```

B.2.1.1 Optional flags

- `monitor` when set to `True`, this flag starts a cAdvisor and Prometheus Pushgateway Docker container alongside son-emu. cAdvisor will monitor general metrics of the deployed Docker VNFs. The Pushgateway will collect specific flow counters that are exported from the son-emu virtual network implemented by OpenvSwitch.

This needs to be enabled for `son-monitor` to work.

- `enable_learning` when set to `True`, this flag configures the OpenvSwitch networking in `son-emu` with `fail mode=standalone`. This means the 'son-emu' network topology will function as a self-learning switch.

B.2.2 Add a SONATA dummy Gatekeeper endpoint

This defines a Gatekeeper endpoint , similar to the one in the SONATA SP. A REST API is started here that accepts SONATA service packages.

```
sdkg1 = SonataDummyGatekeeperEndpoint("0.0.0.0", 5000, deploy_sap=True,
                                      auto_deploy=True, docker_management=True,
                                      auto_delete=True, sap_vnfd_path=sap_vnfd_path)
```

B.2.2.1 Optional Flags

These flags each implement a special deployment functionality:

- `deploy_sap` when set to `True`, each connection point (SAP) in the NSD is deployed according to its defined type:
  - `internal` : deployed as a dedicated Docker VNF, chained to the VNFs as defined in the NSD.
  - `external` : deployed as a virtual interface on the 'son-emu' host. NAT rules are installed using iptables, which make it possible to connect this interface and the service's endpoint to the outside world (eg. to connect a traffic generating process, running on the host).
  - `management`: as defined by the `docker_management` flag (see below)
- `auto_deploy` when set to `True`, each pushed service package will automatically be deployed (this avoids the need to explicitly ask the dummy gatekeeper to deploy the latest pushed package).
- `auto_delete` when set to `True`, any currently deployed service will automatically be deleted when a new one is asked to be deployed. This configures `son-emu` to only deploy one service at a time.
- `docker_management` when set to `True`, the VNF interfaces that are configured in the NSD/VNFD as `type=management` are not instantiated, but are replaced with the interfaces to the `docker0` switch, which are anyway deployed by Docker.
- `sap_vnfd_path` is the pathname of the VNFD .yml file that describes which Docker VNF to use as service endpoint. Each internal connection point in the NSD is deployed with this VNFD.

C Technical Details of son-profile

C.1 son-profile Active Mode

Further information is maintained in the GitHub wiki page: `https://github.com/sonata-nfv/son-cli/wiki/son-profile:-active-mode`

C.1.1 Prerequisites and Setup

The active mode of `son-profile` requires an remote host or VM on which the emulator is installed and password-less SSH access is possible to control the remote profiling host. To simply this setup for the user, we provide a pre-configured Vagrant-based example VM that comes with everything included to setup the required environment.

To set up the VM download the Vagrantfile and run the VM build process:

```
wget https://github.com/sonata-nfv/son-cli/raw/master/src/son/profile/
        misc/son-emu-vagrant/Vagrantfile
vagrant up
```

Wait until the process has finished and you have a VM running which can be used as a remote host for son-profile.

C.1.2 Example Usage

The following CLI examples illustrate the usage of son-profile.

Start the default active mode with a specified PED file:
`son-profile --mode active -p example_ped1_small.yml`

Start the active mode with a specified PED file and config file
`son-profile --mode active -p example_ped1_small.yml -c config.yml`

The SONATA service package containing the service that should be profiled needs to be located in the same folder as the used PED file that references the package. An example package can be found here: `https://github.com/sonata-nfv/son-cli/blob/master/src/son/profile/tests/misc/sonata-fw-vtc-service.son`

Results generated by the experiments have to be written into the `/mnt/share` folder of the VNFs or measurement points and are then automatically collected and copied to the local result folder after each profiling run.

C.1.3 Example Configuration File

son-profile active mode requires a configuration file, which describes the remote hosts that are controlled by son-profile and should be used to profile the service. An example `config.yml` file that uses the previously mentioned Vagrant VM looks like this:

```
descriptor_version: "0.1"
target_platforms:
```

```
  vagrant_vm:
    address: "127.0.0.1"
    package_port: 5000
    ssh_port: 2222
    ssh_user: "vagrant"
    ssh_key_loc: "~/.ssh/id_rsa"
```

C.1.4 Example PED File

```
#
# This is an example for a profiling experiment descriptor (PED).
# It defines profiling experiments for the sonata-fw-vtc-service-emu example service.
#
descriptor_version: 0.1
# SONATA-like identifier (just in case we need it)
vendor: "eu.sonata-nfv"
name: "sonata-fw-vtc-profile-experiment"
version: "0.1"
author: "Manuel Peuster, Paderborn University, manuel.peuster@uni-paderborn.de"
description: "This is an example profiling experiment descriptor (PED)."

# path to the package of the service we want to profile
service_package: "./sonata-fw-vtc-service.son"

#
# First type of experiments: Service level experiments
#
service_experiments:
  - name: "service_throughput"
    description: "iperf test for entire service"
    repetitions: 1
    time_limit: "120"
    # NSD to be used (SONATA-like vendor.name.version reference)
    service: "eu.sonata-nfv.sonata-fw-vtc-service.0.1"
    # additional containers for traffic generation/measurements (like SAPs)
    measurement_points:
      - name: "mp.output"
        connection_point: "ns:serviceout"
        container: "mpeuster/p2-mp"
        cmd_start: ${"iperf -s"}
        cmd_stop: null
      - name: "mp.input"
        connection_point: "ns:servicein"
        container: "mpeuster/p2-mp"
        cmd_start: ${"iperf -c 1.1.1.1 -t 120", "iperf -c 1.1.1.1 -t 120 -u"}
        cmd_stop: null
    # resource configurations to be tested during profiling run (defined per VNF)
    resource_limitations:
      # again: SONATA-like references
```

```
- function: "eu.sonata-nfv.fw-vnf.0.1"
  cpu_bw: "${0.05 to 0.1 step 0.05}" # Omnet++ style parameter study macros
  cpu_cores: 1
  mem_max: "${64, 128}" # Omnet++ style parameter study macros
  mem_swap_max: null
  io_bw: null
- function: "eu.sonata-nfv.vtc-vnf.0.1"
  cpu_bw: "${0.05 to 0.1 step 0.05}"
  cpu_cores: 1
  mem_max: "${256, 512}"
  mem_swap_max: null
  io_bw: null
- function: "mp.input"
  cpu_bw: 0.2
  cpu_cores: 1
  mem_max: 512
  mem_swap_max: null
  io_bw: null
- function: "mp.output"
  cpu_bw: 0.2
  cpu_cores: 1
  mem_max: 512
  mem_swap_max: null
  io_bw: null
```

C.2 son-profile Passive Mode

Further information is maintained in the GitHub wiki page: `https://github.com/sonata-nfv/son-cli/wiki/son-profile:-passive-mode`

C.2.1 Example Usage

The following CLI examples illustrate the usage of `son-profile`.

Start default passive mode with specified PED file:

```
son-profile -p ped_ctrl.yml --mode passive
```

Start default passive mode with specified PED file and path to store the results:

```
son-profile -p ped_ctrl.yml --mode passive -r profile_ctrl.yml
```

Start default passive mode with specified PED file but only draw the result graph from a specified results file:

```
son-profile -p ped_ctrl.yml --mode passive --graph-only -r profile_ctrl_demo.yml
```

C.2.2 Example PED file

The following example PED file is available in the SONATA GitHub Repository: `https://github.com/sonata-nfv/son-examples/tree/master/service-projects/sonata-ovs-user-service-emu`.

The different fields of the file are explained in Section 4.6.2.

```
---
```

```
#
# This is an example for a profiling experiment descriptor (PED).
# It defines profiling experiments for the sonata-ovs-user-service example service.
#
descriptor_version: 0.1
vendor: "eu.sonata-nfv"
name: "ovs-profile-experiment"
version: "0.1"
author: "name,email"
description: "This is an example profiling experiment descriptor (PED)."

# path to the package of the service we want to profile
service_package: "./sonata-ovs-user-service.son"

#
# For son-profile passive mode only Service level experiments are taken into account
#
service_experiments:
  - name: "service_throughput"
    description: "iperf test for entire service"
    repetitions: 1
    time_limit: "11"
    # NSD to be used (SONATA-like vendor.name.version reference)
    service: "eu.sonata-nfv.sonata-fw-vtc-service.0.1"
    # additional containers for traffic generation/measurements (like SAPs)
    measurement_points:
      - name: "ns_port0"
        connection_point: "ns:serviceout"
        container: "sonata-son-emu-sap:latest"
        configuration:
          - "ethtool -K port0 tx off"
          - "arp -s 10.20.30.40 11:22:33:44:55:66"
        cmd: 'iperf -c 10.20.30.40 -t999 -u -b${1,2,3,5}M'
        cmd_order: 2
      - name: "ns_port1"
        connection_point: "ns:servicein"
        container: "sonata-son-emu-sap:latest"
        configuration:
          - "ethtool -K port1 tx off"
        cmd: 'python iperf_server.py "-s -u -i1 -fm"'
        cmd_order: 1
#
# Additional resource limitations for the VNFs in the service.
#
     resource_limitations:
       - function: "ovs1"
         cpu_bw: "${0.4 to 0.8 step 0.1}"
         cpu_cores: 1
```

```
        #mem_limit: "${256, 512}m"
        mem_limit: 256m
      - function: "ns_port0"
        cpu_bw: 0.2
        cpu_cores: 1
        mem_limit: 512m
      - function: "ns_port1"
        cpu_bw: 0.2
        cpu_cores: 1
        mem_limit: 512m
#
# Metrics that need to be monitored during the profiling tests
#
    input_metrics: "msd_input.yml"
    output_metrics: "msd_output.yml"
#
# Metrics for which a plotted analysis is made
#
    profile_calculations:
      - name: "ovs1_profile"
        input_metric: "rx_packet_rate@ovs1:port0"
        output_metric: "cpu@ovs1"
      - name: "ctrl_profile"
        input_metric: "rx_packet_rate@ctrl:ctrl-port"
        output_metric: "cpu@ctrl"
```

D Technical Details of son-monitor

This information will be maintained in the GitHub wiki page: `https://github.com/sonata-nfv/son-cli/wiki/son-monitor:-getting-data-from-the-Service-Platform`

D.1 Configuration Parameters

The tools included in `son-monitor` require a list of configuration settings. It also depends on the `son-access` configuration file. This allows the tools to work with both local or remote instances of the emulator and the Service Platform. Below is the current list of variables for `son-monitor`:

```
SON_EMU_IP = '172.17.0.1'
SON_EMU_REST_API_PORT = 5001
SON_EMU_API = "http://{0}:{1}".format(SON_EMU_IP, SON_EMU_REST_API_PORT)

# Monitoring manager in the SP
SP_MONITOR_API = 'http://sp.int3.sonata-nfv.eu:8000/api/v1/'
# Gatekeeper api in the SP
GK_API = 'http://sp.int3.sonata-nfv.eu:32001/api/v2/'

# local port where the streamed metrics are served to Prometheus
PROMETHEUS_STREAM_PORT = 8082

# son-access config file (initialized by son-access workspace creation)
SON_ACCESS_CONFIG_PATH = "/home/steven/.son-workspace"

# tmp directories that will be mounted in the Prometheus
# and Grafana Docker containers by son-emu
tmp_dir = '/tmp/son-monitor'
docker_dir = '/tmp/son-monitor/docker'
prometheus_dir = '/tmp/son-monitor/prometheus'
grafana_dir = '/tmp/son-monitor/grafana'

# Prometheus config info
prometheus_server_api = 'http://127.0.0.1:9090'
prometheus_config_path = '/tmp/son-monitor/prometheus/prometheus_sdk.yml'
```

D.2 Metrics Gateway

As described in Section 4.7, a metrics gateway is implemented in the SONATA SDK to securely stream monitored data from the Service Platform to the SDK. In short, the procedure of streaming data can be summarized like this:

1. Start the metrics stream via the `son-monitor stream` command.

2. The gateway is started at port 8082 in the SDK host. The data can be checked at `http://:8082`
3. The Prometheus instance in the SDK is configured to get data from this gateway.
4. Connection to the websocket is made and data is exported via the gateway.
5. The gateway is automatically stopped when the websocket closes.

D.3 Updated son-monitor CLI

The CLI commands of the SDK's `son-monitor` tools have been updated. Each specific sub-command has its own help interface. The exact status can be found in the `son-monitor` GitHub repository. Some updated examples are given below:

```
$ son-monitor stream -h
usage: son-monitor stream [-h] [--metric METRIC] [--service SERVICE]
                          [--vnf_name VNF_NAME] [--vdu VDU_ID]
                          [--vnfc VNFC_ID] [--sp SP]
                          [{start,stop}]

Stream monitor data from the SONATA Service Platform.
(Authentication must be configured first via son-access)

positional arguments:
  {start,stop}          start/stop streaming metrics from the SONATA Service Platform

optional arguments:
  -h, --help            show this help message and exit
  --metric METRIC, -me METRIC
                        SP metric
  --service SERVICE, -s SERVICE
                        Service name that includes the VNF to be monitored
  --vnf_name VNF_NAME, -vnf VNF_NAME
                        vnf to be monitored
  --vdu VDU_ID, -vdu VDU_ID
                        vdu_id to be monitored
                        (optional, if not given, picks the first vdu)
  --vnfc VNFC_ID, -vnfc VNFC_ID
                        vnfc_id to be monitored
                        (optional, if not given, picks the first vnfc instance)
  --sp SP, -sp SP       Service Platform ID where the service is instantiated

$ son-monitor query -h
usage: son-monitor query [-h] [--vim VIM] [--service SERVICE]
                         [--vnf_name VNF_NAME] [--vdu VDU_ID] [--vnfc VNFC_ID]
                         [--metric METRIC] [--since START] [--until STOP]
                         [--query QUERY] [--datacenter DATACENTER]

Query monitored metrics from the Prometheus DB in the SDK or the Service Platform
```

```
(For querying the Service Platform, Authentication must be configured
first via son-access)

optional arguments:
  -h, --help            show this help message and exit
  --vim VIM, -vim VIM   Emulator or Service Platform ID where the service
                        is instantiated (default = emulator)
  --service SERVICE, -s SERVICE
                        Service name that includes the VNF to be monitored
  --vnf_name VNF_NAME, -vnf VNF_NAME
                        vnf to be monitored
  --vdu VDU_ID, -vdu VDU_ID
                        vdu_id to be monitored in the Service Platform
                        (optional, picks the first vdu if not given)
  --vnfc VNFC_ID, -vnfc VNFC_ID
                        vnfc_id to be monitored in the Service Platform
                        (optional, picks the first vnfc instance if not given)
  --metric METRIC, -me METRIC
                        The metric in the SDK or SP to query
  --since START, -si START
                        Retrieve the metric values since this start time
                        (eg. 2017-05-05T17:10:22Z)
  --until STOP, -u STOP
                        Retrieve the metric values until this stop time
                        (eg. 2017-05-05T17:31:11Z)
  --query QUERY, -q QUERY
                        raw Prometheus query for the emulator
  --datacenter DATACENTER, -d DATACENTER
                        Data center where the vnf is deployed in the emulator
                        (if not given, the datacenter will be looked up first)
```

D.4 Manual of son-analyze

The following section describes the command line interface of `son-analyze`.

```
$ son-analyze --help
usage: son-analyze [-h] [-v] [--docker-socket DOCKER_SOCKET]
                   {version,bootstrap,run,fetch} ...

An analysis framework creation tool for Sonata

positional arguments:
  {version,bootstrap,run,fetch}
    version             Show the version
    bootstrap           Bootstrap son-analyze
    run                 Run an environment
    fetch               Fetch data/metrics
```

```
optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         increase verbosity
  --docker-socket DOCKER_SOCKET
                        An uri to the docker socket (default:
                        unix://var/run/docker.sock)
```

D.5 Self-assessment of the SONATA SDK

As this deliverable documents the final release of the SDK within the SONATA project (as indicated earlier, future updates and release might happen outside the project), the goal of this section is to shortly document how the different objectives, as documented in the SONATA Description of Work, were addressed in the SDK, as a form of self-assessment.

SDK objectives from the SONATA project Description of Work (DoW)	Actions performed and components developed to achieve the objective.	Documentation source(s)
1. Design of a **Service Programming Model** for NFV, which allows for creating services by means of abstraction of fundamental functionality, such as interconnection of NFs, load balancing, scale-in and scale-out, function and service instantiation, etc. This includes involving the definition or selection of suitable programming and description languages, as well as the support of common patterns in the designed programming model. The model will further support the specification of runtime invariants and service monitoring requests that are to be realised in the operational environment.	**`son-schema`** defines the programming model including the description of interconnected NFs	Dedicated sections SONATA deliverables D3.1, D3.2 and D3.3, as well as the https://github.com/sonata-nfv/son-schema Git-repository.
	Service -and Function Specific Management (SSM and FSM) functionality enables placement and scaling, and templates have been defined for typical patterns.	FSM- and SSM architectural aspects are described in SONATA deliverables D2.1, D2.2 and D2.3. *SM templates can be found at https://github.com/sonata-nfv/son-sm, and are documented in Section 4.9.
	Monitoring and profiling descriptors (MSD and PED files) enable the specification of runtime monitoring metrics enabling testing	Dedicated sections on **`son-monitor`** and **`son-profile`** in SONATA deliverables D3.1, D3.2 and D3.3, as well as the https://github.com/sonata-nfv/son-cli Git-repository has dedicated pages on **`son-monitor`** and **`son-profile`**.
2. Development of a **Service Development Kit (SDK)** supporting the new programming model and providing a set of *well-integrated tools* and catalogues assisting the service development process.	The entire SDK consists of a set of light-weight tools (**`son-workspace`**, **`son-project`**, **`son-validate`**, **`son-package`**, **`son-access`**, **`son-emulator`**, **and `son-profile`**) which can be used in an integrated way using a consistent workflow on the command-line or from the GUI provided by son-editor.	The entire SDK toolset has been extensively documented in SONATA deliverables D3.1, D3.2 and D3.3
3. Development of **Supporting Tools for Service Development and Operations (DevOps)**, that will be integrated into the SDK. They will provide means for service developers to apply the DevOps concept by interacting with operators. Included will be a tool for analysing measured service runtime data received from the operator, *tools for debugging and profiling* service descriptions and network functions, and a tool for verifying interconnections of service components.	The entire SDK workflow has been design from initial phases, focusing on an DevOps workflow, enabling to use the same toolset on both the local emulator of the SDK as well as on the Service Platform/operations environment. The SDK son-monitor tool enables to monitor and analyse data coming from the Service Platform, and the **`son-analyse`** component provides extensive tools for statistical analysis of this data. The **`son-profile`** tool of the SDK supports a passive and an active mode, enabling to profile services and network functions on a range of infrastructure.	The integrated SDK workflow, supporting seamless deployment on both the local SDK emulator as well as on the Service Platform is documented in dedicated sections of D3.1, D3.2 and D3.3. The integration of monitoring and profiling functionality (provided by **`son-monitor`** and **`son-profile`**) is documented in dedicated sections in D3.3.

Figure D.1: Mapping SDK outcomes to the DoW objectives

E Abbreviations

API Application Programming Interface

config Configuration

FSM Function Specific Manager

MANO Management and Orchestration

MSD Monitoring Service Descriptor

NSD Network Service Descriptor

PED Profiling Experiment Descriptor

PEM Privacy-enhanced Electronic Mail

SDK Service Development Kit

SAP Service Access Point

SP Service Platform

SSM Service Specific Manager

SMR Specific Manager Registry

UUID Universally Unique Identifier

VDU Virtual Deployment Unit: Atomic deployment unit if a VNF (eg. VM/container image)

VNFC Virtualized Network Function Component: Instance of a VDU in a deployed service in the SONATA platform

VNFD Virtual Network Function Descriptor

F Bibliography

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

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

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

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

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

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

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

[8] D3js - data-driven documents. Online at `https://d3js.org/`.

[9] SONATA Project. Son-editor Api specification, 2017. Online at `https://github.com/sonata-nfv/son-editor-backend/blob/master/technical_document_editor.pdf`.

[10] Wouter Tavernier Didier Colle Mario Pickavet Piet Demeester Steven Van Rossem, Manuel Peuster. Automated Monitoring and Detection of Resource-limited Nfv-based Services. In *Network Softwarization (NetSoft), 2017 IEEE Conference on*. IEEE, 2017.

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