D3.2 Intermediate release of SDK prototype and documentation

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	D3.2 Intermediate release of SDK prototype and documentation
Workpackage	WP3 Service Programmability and Toolset
Due Date	November 30th, 2016
Submission Date	December 23rd, 2016
Version	1.0
Status	To be approved by EC
Editor	Wouter Tavernier (imec)
Contributors	Wouter Tavernier, Steven Van Rossem (imec), Lus Conceio, Tiago Batista (UBI), Michael Bredel (NEC), Geoffroy Chollon (TCS), Daniel Guija, Muhammad Shuaib Siddiqui (i2CAT), Manuel Peuster (UPB)
Reviewer(s)	Sharon Mendel-Brin (NOKIA), Sonia Castro (ATOS)

Keywords:

SDK, DevOps, Software Development Kit

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

Executive Summary:

The first phase of the SONATA project resulted in a release of the first version of the Software Development Kit (SDK) and Service Platform (SP). These SONATA building blocks complement each other in the context of next generation of mobile networks and telecommunication standards referred as 5G, focusing on **optimal (re-)use of the available network and cloud infrastructure** in order to provide telecom services based on a mix of software- and hardware-based infrastructure.

The **SDK** is built up as a set of **small independent tools** which can be combined in one or multiple workows to develop a SONATA service which is composed of software-based Network Functions (NFs). This design and programming model enables agile development, involving quick and iterative cycles of development, with the possibility of rapidly transitioning between development and operations (**DevOps**), which is one of the key characteristics of the SONATA approach. The software design of the SDK re-uses existing workflows and concepts in software development such as the use of workspaces, project folders and packaging techniques. An overview of the available components and functionality after the first phase of the project were provided in D3.1 [5]. The SONATA SP is the primary target platform for the services developed by the SDK, however re-use and extensibility towards using the same SDK in order to develop and test NFV services on other MANO platforms is foreseen in the future. The initial SP design, its functionality and corresponding API was documented in D4.1 [6], while updates and novel components of the SP are documented in D4.2 [7].

This deliverable documents the updated design, additional features, as well as new components of the SDK in the intermediate SONATA release. The **structure of this document therefore follows as closely as possible the structure of D3.1** [5], assuming re-use of the existing SDK design, and focusing on the documentation of changes and additional components.

The SONATA **programming model** is built around **descriptors** for packages, network services, and network functions. The **schema** of these descriptors for this release are being updated to account for novel features including container implementations, automated testing, or improved licensing support. **Validation functionality** was made in the first release of the project in order to validate syntax of descriptors. In this release this functionality has been further extended with a dedicated tool `son-validate` which goes beyond pure syntax checking and also checks a range of semantic aspects, e.g. detection of loops in service graphs. Baseline functionality of the SDK is built around CLI functionality for creating workspaces, project spaces and interaction with the SP. These have now been extended with **versioning support for workspaces**, and modified tools (i.e., `son-access`) for **interacting with the (gatekeeper of the) SP** and associated service and NF catalogue(s). The latter impacts the deprecation of the use of the SDK-specific `son-catalogue` of phase 1. As reported in D3.1 [5] and related scientific publications, the ability to emulate the SP and associated infrastructure is one of the core value propositions of the SDK. In this intermediate release these main assets have been even further exploited through bug-fixes, **improved APIs, monitoring and debugging capabilities**. Additional **debugging and monitoring functionality** in this release enables easy inspection and visualisation of network interfaces and links, as well as functionality to generate particular types of (test) traffic for debugging purposes. `son-analyze` functionality has now been integrated with Jupyter Notebook technology [2] enabling the use of bleeding-edge statistical functionality in scientific Python libraries. The strong set of features combining the SDK functionality related emulation, monitoring and analysis introduces a **new performance profiling tool** `son-profile`. The latter eases identifying performance trends under given degrees of resource restrictions. This enables developers to pre-assess performance in order to optimize scaling behaviour of services on a range infrastructure hardware environments.

Contents

List of Figures

List of Tables

1 Introduction

The first version of the SONATA SDK, which was documented in D3.1 [5], provided SDK components and **tools for setting up a SONATA programming environment**, a SONATA platform **emulator** and an initial set of tools for **monitoring and analysing services** developed using the SDK. The first SDK interacted with the initial version of the Service Platform documented in D4.1 [6] via the gatekeeper of the Service Platform. The developed SDK tools provided one of the first integrated set of tools which enabled to deploy services combining both network and cloud resources in a controlled and uniform way.

Since the release of the first SONATA SDK, the architecture and corresponding interfaces have slightly changed, improved and refined, and additional components have been developed. These architectural updates have been documented in D2.3 [4], while updates to the Service Platform are documented in D4.2 [7]. This deliverable focuses on the updates of SONATA Software Development Kit following this renewed architecture and SP updates.

1.1 Structure of the deliverable

The document is structured in a very similar way to D3.1 [5]. The section Section 2 refines the development workflow in relation to the different components of the SDK. Next, the main design aspects of new components and updates to existing SDK components are described in Section 3. 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 validation and profiling tool. Each of these sections also sketches a perspective of further planned features for the considered SDK components. Section 4 concludes the document, capturing the most important updates of the SDK design, and summarizes future plans on enhancing the basic SONATA Software Development Kit for last phase of the project. 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 Updated SDK design and new components

The SONATA SDK is built as a set of light-weight (CLI-focused) tools helping the developer to setup the development environment, describe the developed service and network functions, and test them in a controlled environment before uploading them to a full-fledged Service Platform.

The main design of the SDK, as well as its components and corresponding developer workflow were described in D3.1 [5]. In this section we refine this process, indicate changes into existing components and interfaces, and introduce a number of additional SDK tools.

The main development object SONATA's SDK is addressing is the **service**, which consists of a **combination of Network Functions** (NFs) interconnected through a forwarding graph. In addition to NFs, a service can also require Service and Function Specific Managers (SSM and FSMs). The details on each of these components are captured under **Descriptors** (which have been slightly updated, see Section 3.1). Descriptors might refer to information available in a catalogue which is accessed via a (security-enabled) Gatekeeper, which is part of a Service Platform. Note that this renewed process does not any longer support the (unsecure) previous `son-catalogue` to be part of the local developer SDK installation.

A SONATA service is developed through a set of environment and packaging tools. The SONATA development environment requires a workspace to be set up (`son-workspace`) in which different SONATA projects can be developed. From this release on, `son-workspace` supports versioning of workspace environments. A SONATA project is the construction environment (`son-project`) for a particular SONATA service within a workspace. Particular SDK tools will support the development of individual VNFs and FSMs and SSMs (`son-vnf` and `son-ssm/fsm`) in future releases. The **newly introduced `son-validate` tool** of this release **enables validation of syntax as well as selected semantics** of service components in an ad-hoc manner. When a project is ready to be deployed, it will be packaged into a service package (`son-package`). The latter can be (and ultimately will be) deployed (or pushed) on an actual Service Platform using the **newly developed `son-access` tool**. This tool bundles previous `son-push` functionality as well as functionality for **accessing a catalogue** behind the Gatekeeper of the SP. In order to enable quick iterations in development and testing without requiring the setup of a full-fledged Virtual Infrastructure Management solution such as OpenStack, together the service platform, an SP emulator (`son-emu`) is part of the SDK and enables to locally deploy a service on an environment which is highly similar to the SP. Within the emulation environment (but also on the SP), monitoring tools (`son-monitor`) and **newly developed (performance) profiling tools** (`son-profile`) can be used to verify functionality or performance of developed components. Subsequently, the resulting data might be analysed using the updated analysis tools (`son-analyze`) in order to improve or update the resulting descriptors and ultimately the service package.

2.1 Updated SDK interfaces

Figure 2.1 illustrates the interrelation of existing, new and updated SDK components as well as parts of relevant parts of the SP. New components are coloured in green, existing components appear in yellow, significantly updated components are indicated in white, planned components are orange, and deprecated components are grey. Every interface corresponds to a reference point with a particular label and will be documented further in this deliverable:

- The development of SONATA services occurs on a **project folder** (PRJ) in a **workspace** (WKSP) on the **filesystem**, built by the updated `son-workspace` (the new functionality is described later in this document in Section 3.2).
- The **interface between the SDK and the SP** is mediated via the **Gatekeeper**, following the **GK REST API** documented in D4.1 [6] and updated in D4.2 [7], where the SDK side is implemented by `son-access` (integrating now functionality from the deprecated `son-push`).
- The **SDK emulator** `son-emu` is now controllable through an "**EMU REST API**", which is used by the `son-monitor` component and the newly designed `son-profile` component, enabling performance benchmarking of service components.
- The interface between the `son-monitor` component and the `son-analyze` tools follows a **Prometheus-based "MON REST API"**. `son-monitor` might access (real-time) monitoring data on the SP using a Websocket-based interface (documented Section 3.6) enabling seamless DevOps processes alternating between the SDK and SP.
- The **interface to the catalogue is now mediated via the Gatekeeper**, using the **GK REST API** which is documented in Section 3.3.

2.2 Updated SDK workflow

Having small, light-weight SDK components enables multiple developer workflows. This section will however detail the *canonical* workflow. This refines the SDK workflow documented in Deliverable D3.1. The main process is depicted in Figure 2.2, which follows the following steps:

1. In order to be able to deploy a developed service, either a **Service Platform** or a Service Platform emulator (`son-emu` as indicated in the figure), must be **initialized** by the operator of the platform. In the described workflow it is started before the service is deployed. However, this could be done just before step 7 as well.
2. Next, a SONATA development **workspace must be created** via `son-workspace` before a project can be created (this will be done automatically if this step is omitted) by the developer in the SDK.
3. In order to prepare the development workspace for the development of an individual SONATA project, a **project space is created** via `son-project` by the developer in the SDK.
4. A service can be built using different pre-existing VNFs, SSMs and other **artifacts which might be fetched from a catalogue** accessible through the Gatekeeper from the SP using `son-access`.
5. **Project development** using a range of manual actions and/or tools (which might be provided in the second phase of the project, such as `son-vnf` and `son-ssm`) by the developer in the SDK.
6. The newly released `son-validate` can be used to assess syntax of descriptors, or to perform a range of consistency checks across services, components or packages.

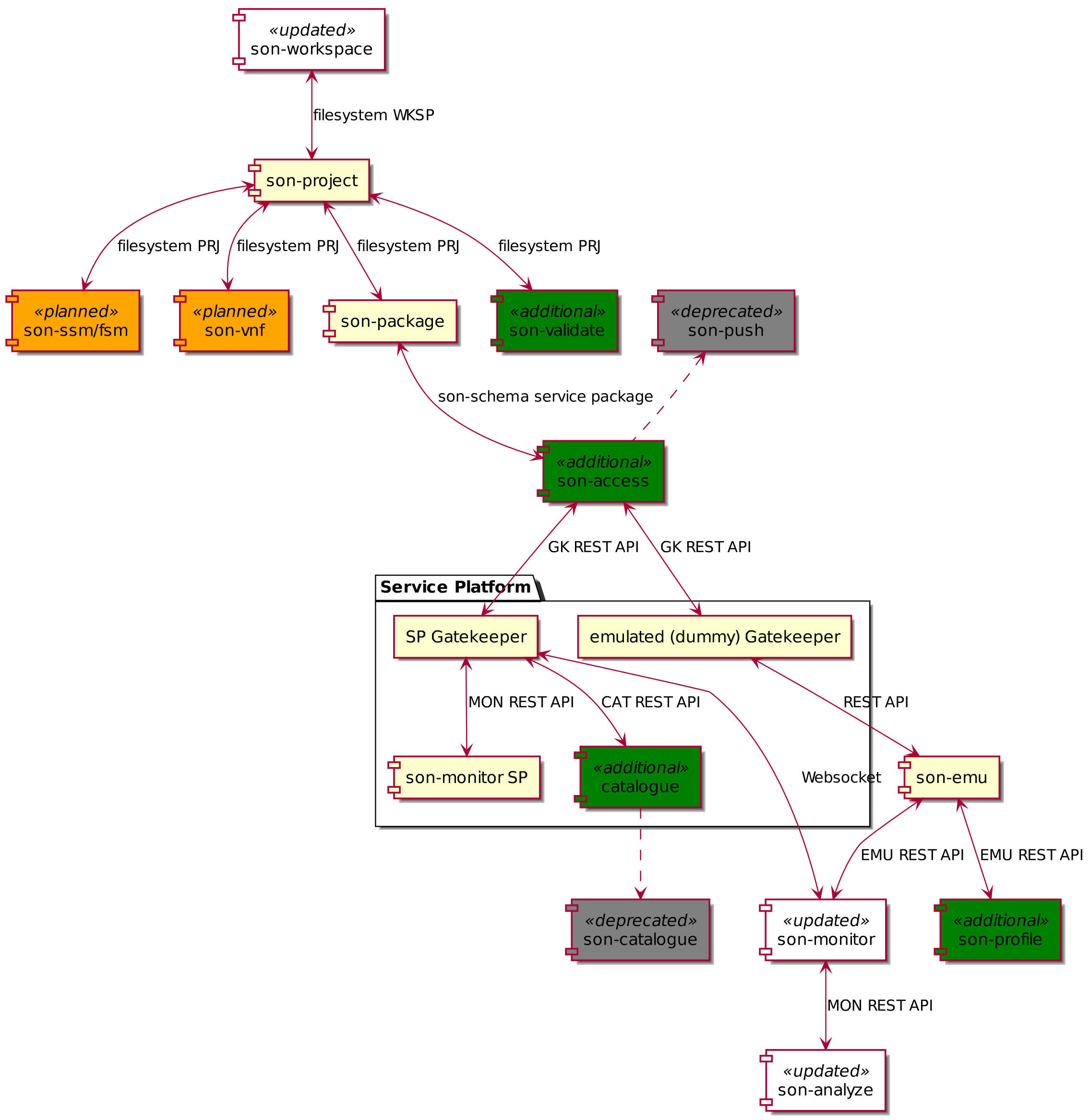

Figure 2.1: Interfaces between the main SDK components

7. The necessary descriptor files of the service are **bundled into a package** which can be deployed using `son-package` by the developer in the SDK.
8. The resulting package is **uploaded to the gatekeeper** of the (emulated) Service Platform via the `son-access` tool by the developer in the SDK. As a result, an identifier is received from this interaction.
9. Once uploaded, the received identifier can be used by the developer to actually trigger the (emulated) Service Platform to **deploy** it, again using the `son-access` tool from the SDK.
10. The SP itself might now induce internal (re)-deployment actions in order to make sure the service gets up and running.
11. In order to monitor particular functional or performance-related service, VNF or SSM parameters, a **deployed service can be monitored** using `son-monitor` from the SDK.
12. The SDK translates the requested monitoring actions into **platform specific instructions** (e.g., using the MON REST API interface of the emulator).
13. The (emulated) Service Platform actually starts the requested **monitoring actions** on the infrastructure and/or on the VNFs.
14. **Monitoring data is generated** within the infrastructure or from the VNFs and is sent back to the Service Platform.
15. The (emulated) Service Platform returns the resulting **monitoring data to the SDK** in the form of a stream, file or other format.
16. The SDK tool `son-analyze` can be used to **analyse and visualize** the resulting data, helping the developer to improve the service design and re-start the process at an earlier step of this process.
17. The new SDK tool `son-profile` can generate different test setups in order to **assess performance and performance trends** under one or more multiple resource constraints.

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

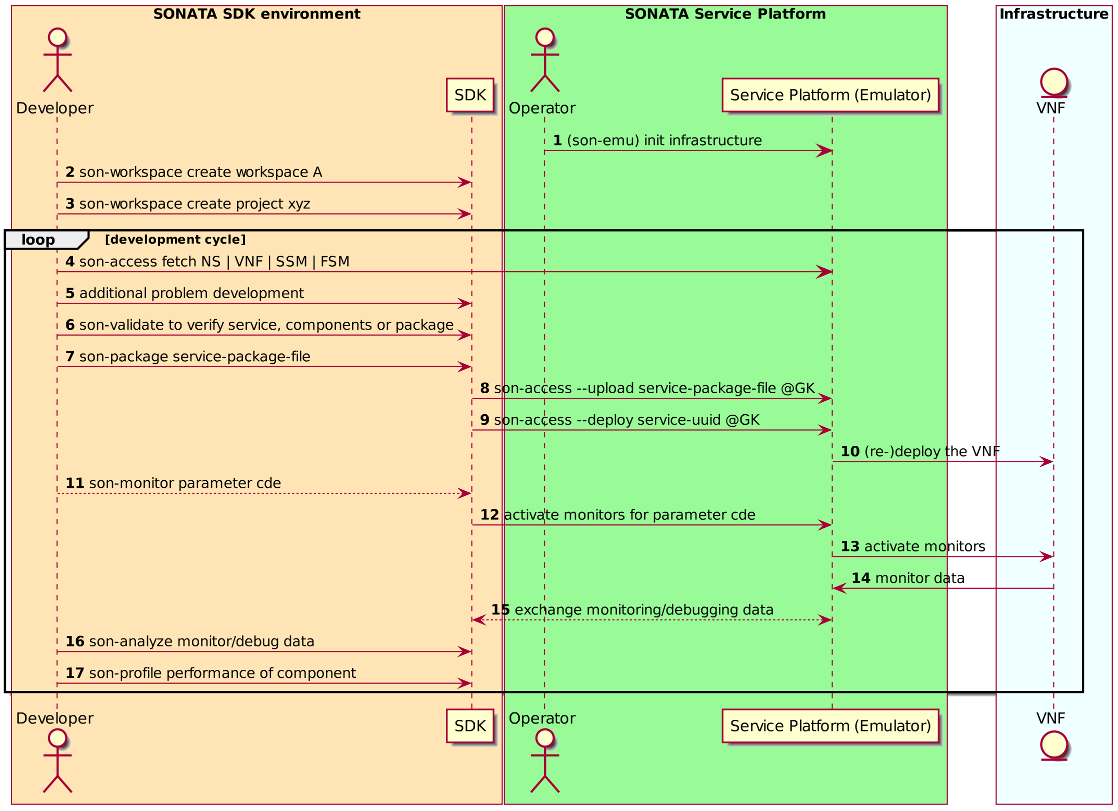
SONATA SDK environment
SONATA Service Platform
Infrastructure
Developer
SDK
Operator
Service Platform (Emulator)
VNF
1 (son-emu) init infrastructure
2 son-workspace create workspace A
3 son-workspace create project xyz
loop [development cycle]
4 son-access fetch NS | VNF | SSM | FSM
5 additional problem development
6 son-validate to verify service, components or package
7 son-package service-package-file
8 son-access --upload service-package-file @GK
9 son-access --deploy service-uuid @GK
10 (re-)deploy the VNF
11 son-monitor parameter cde
12 activate monitors for parameter cde
13 activate monitors
14 monitor data
15 exchange monitoring/debugging data
16 son-analyze monitor/debug data
17 son-profile performance of component

Figure 2.2: SDK development workflow

3 Component design

This section will go deeper into the additional features of this intermediate release of the SONATA SDK and provide details on some of the planned features for the final release. In order to avoid excessive verbosity, the sections are expected to be read as an add-on or set of changes of already existing SDK components compared to D3.1, rather than a full recapitulation of individual design, functionality and tests.

3.1 son-schema

The SONATA Schemata are used to specify the various descriptors used by the SONATA system. As described in detail in deliverables D2.2 [3], D2.3 [4], and D3.1 [5] already, schemata are implemented in YAML using the JSON Schema standard [9]. They are based on the ETSI NFV descriptor specification and can be used by the SDK, the catalogues, and the Service Platform to validate and verify the actual descriptors.

In the following section we describe the updates and improvements over our first schema implementation and provide an outlook on future features.

3.1.1 Network function descriptor schema

Network Function Descriptors (VNFDs) are used to specify and provide meta-data to virtualized network functions. To this end, they contain technical information, like virtual machine images, that constitutes a network function. This information is used by the Service Platform to manage the lifecycle of network functions.

The next version of the SONATA VNFDs contains various improvements to adapt to state-of-the-art developments of other descriptors, like HOT and Tosca, and to adapt the most recent ETSI NFV specification, which improved and matured over the last year. In addition we corrected some shortcomings that became obvious during our implementation phase. Namely, the improvements are:

- **Differentiation between different interfaces**: In the older version of the VNFD schema, there was only one type of interface specification. In order to differentiate whether a interface is public or private, we added internal (private), external (public) and management annotations to ingress and egress ports. This simplifies the integration, say of the FSMs and SSMs, which can be only connected to internal interfaces, and increases security as, say management interfaces, are not reachable from public IP addresses any more.

- **IPv6 support**: As we are running out of IPv4 addresses, modern systems should support the larger address space of IPv6. To reflect this requirement, we added IPv6 support to the VNFD respectively. To this end, we added a new interface type that can handle IPv6 addresses. Thus a VNF now can make use of IPv6 for all of its interfaces, and VNF components can be interconnected using IPV6. Support of IPV6 by the Service Platform is part of the Service Platform's future planned features.

- **Container VDU support**: The upcoming version of the Service Platform will support container-based VIMs like Docker and Kubernetes. Thus, the VNFD has to support not only virtual machines as a basis for Virtual Deployment Units (VDUs) but also container images, say for Docker. To this end, we added a new VDU image type, i.e. Docker, to support Docker images. In the next version, a VNF has to be virtual-machine-based or container-based only. However, on a longer run, we foresee mixed version containing containers and virtual machines in parallel as well.

- **Automated testing support**: In order to further support the DevOps approach, automated testing is crucial. To this end, the next version of VNFDs contain test section that link to tests which can be executed by the Service Platform, similar to unit tests in software development.

3.1.2 Network service descriptor schema

Network Service Descriptors (NSDs) are used to specify complex network services that comprise multiple VNFs described by multiple VNFDs. Similar to the VNFD schema, we adapt the NSD schema to meet the most recent ETSI NFV specification and corrected some of the shortcomings we spotted during implementation. Namely, the improvements are:

- **NSD as VNF support**: In order to improve the re-use of existing resource we enable recursiveness within the descriptors, meaning that an NSD can not only reference VNFDs as components, but also other NSDs. To this end, we adapted the identifiers used by the descriptors and - on the Service Platform and catalogues side - introduced a resolver that resolves and integrates references artifacts.

- **Automated testing support**: Similar to the VNFDs, the next version of NSDs contain test section that link to tests which can be executed by the Service Platform, similar to unit tests in software development.

3.1.3 Package descriptor schema

Package Descriptors (PDs) are used to provide additional meta-data to NFV packages, which contain VNFDs, NSDs, and further artifacts, like images and scripts. The packages are very complete already and less changes are necessary. However, a few improvements have been made, as detailed below:

- **License support**: Efficient licensing is a crucial business requirement. In order to support licensing better, the package descriptor can now link to licenses, licensing scripts, and external licensing systems which can be used to implement and enforce license policies. This offers a path to address all kinds of licensing in very general way.

3.2 son-cli

`son-cli` SDK component is a set of command line tools that are 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 focuses on the new features, improvements and implementation details of `son-cli` tools for SONATA second year. Overall information on the component can be found in deliverables D3.1 [5] and D2.2 [3].

3.2.1 son-access

The `son-access` is a new component introduced to SONATA Year 2 plan for the SDK side. This component comes to replace the SDK Catalogue component from the SDK. A new feature for SONATA Year 2 plan is to have one multipurpose catalogue entity for the whole SONATA ecosystem. This catalogue entity has been decided to be on the Service Platform (SP) side, implemented by the Service Platform Catalogue. For this reason, SDK Catalogue has been removed from the SDK and the functionalities it enabled are going to be kept on the SP Catalogue, which must be an accessible component to the SDK. Furthermore, `son-access` is a key component inside the `son-cli` that enables communication from the SDK to the SP bringing those functionalities from SDK Catalogue to the SP Catalogues. `son-access` also points to introduce security features, in the communication with the SP.

The main goal of `son-access` component is to provide a secured connection between SDK end-users and the Service Platform, using their credentials to access the Platform and being able to submit and request stored package files and descriptors from the SP Catalogue.

3.2.1.1 New features and improvements

The `son-access` component is a Year 2 new component that is being developed from scratch, but it also re-uses some components that were responsible for the communication of the SDK Catalogue with `son-cli` components. These changes include improvements and implementation of some new features.

Some SDK components were affected by the removal of the SDK Catalogue. This and changes in the SDK architecture lead to the need of migrating and upgrading the functionality of the following `son-cli` components to `son-access` sub-components as further described in this section:

- `son-push`: the component was responsible for submitting SONATA Packages (`son-package`) to the Service Platform.
- `CatalogueClient`: the component was required in order to communicate with the SDK Catalogue and it allowed to get/post descriptors to SDK Catalogue.
- `son-publish`: the component was required for the `CatalogueClient` and the SDK Catalogue. It was responsible for submitting ‘SDK Project’ elements to the SDK Catalogue.

New `son-access` component architecture is composed of three main sub-components. Figure 3.1 shows how `son-access` is internally composed. In order to implement SDK Catalogue functionalities in the SP Catalogue, each sub-component is responsible for different processes:

- Access: enables secured API.
- Push: responsible for upstream communication to the SP.
- Pull: responsible for downstream communication from the SP.

Currently, `son-access` supports new features and present some improvements over former `son-cli` components:

Access (Authentication and authorization) The 'access' sub-component is responsible for security functions of `son-access` component. As its name suggests, it enables the SDK to gain secured access to the Service Platform. It implements the methods to authenticate end-users of

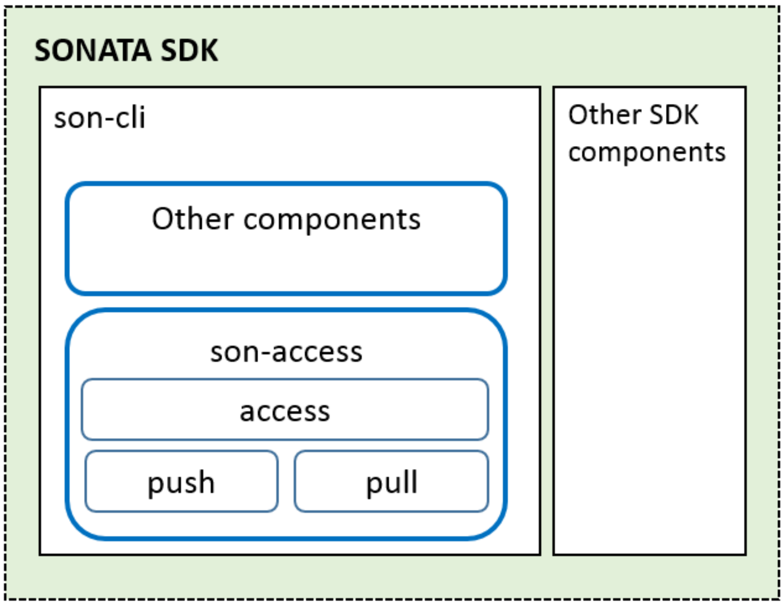

Figure 3.1: Initial version architecture for `son-access` component implementation

SDK to the SP Gatekeeper. Authentication and authorization strongly depends on the User management component on the Gatekeeper in the SP side. SDK end-users, such developers, are required to register to the SP, or they are available to log in to the SP using a GitHub account associated to the SONATA project. When an end-user has a valid account in the SP, they can start the current `son-access` workflow to connect from the SDK to the SP. Figure 3.2 shows the workflow between each component:

1. When authenticating to the SP, SDK end-users (developers) must insert their 'username' and 'password' to the `son-cli` (credentials granted when registering). Using "Social login" credentials such GitHub is still not available from `son-access`.

2. Then `son-access` sends developers' credentials to SP User Management API, which evaluates the identity of the end-user.

3. If the end-user credentials are valid, SP User Management returns an Access Token (JSON Web Token or JWT) to `son-cli`. This Access Token includes the end-user identity, using OpenID Connect 1.0, which is a simple identity layer on top of the OAuth 2.0 protocol, and grants authorization based on the role assigned to the end-user (in this case, the developer). Each Access Token has an expiration time, and a developer can access the platform while the Token or session is valid. Once the Access Token is received, it will be included in every message header that is sent to the SP.

If the end-user credentials are not valid, the SP returns an invalid login message.

Push (former `son-push` and `son-publish`) This sub-component re-uses and merges code from `son-push` and `son-publish` and implements functions to enable the upstream communication to the SP using an Access Token. Currently, these functions are:

- Publish SONATA packages (`son-package`) to the SP. When a `son-package` is submitted to the SP Catalogue, it is processed by the Gatekeeper. User Management module in

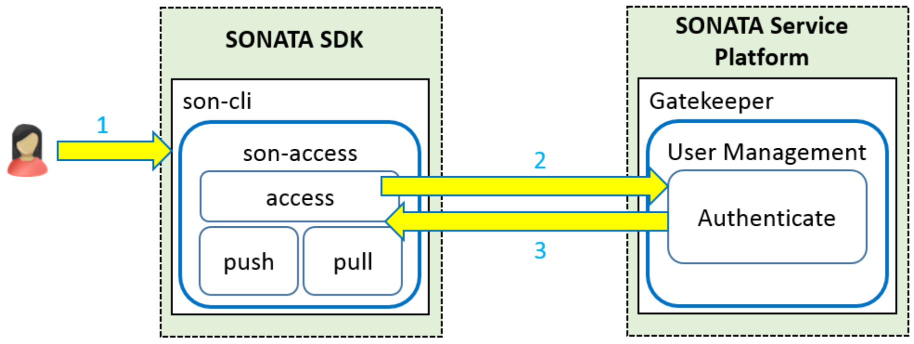

Figure 3.2: Main workflows between SDK and SP Gatekeeper through son-access

the Gatekeeper validates the Access Token. Then, the content of the `son-package` is split by type and stored on the corresponding catalogue according each descriptor type. Finally, `son-package` file itself is stored on the file catalogue. If all the processes succeed, the Gatekeeper is responsible for returning a response containing the identifier for each stored descriptor and file to the `son-access` component.

Pull (former `CatalogueClient`) This sub-component re-uses and merges code from `Catalogue Client` and `son-publish` and implements functions to enable the downstream communication from the SP using an Access Token. Currently, these functions are:

- Request SONATA packages (son-packages) from the SP Catalogues. Using an identifier for the package file (the name trio convention), `son-access` can request a `son-package` that is already stored in the SP Catalogue. User Management module of the Gatekeeper validates the Access Token of the request and checks its authorization. If it is authorized the Gatekeeper requests the `son-package` file from the SP Catalogue, and returns the file to the `son-access` component.
- Request descriptors from the SP Catalogues. In the same way as `son-packages`, using an identifier for a descriptor, or querying to the Gatekeeper API, a SDK end-user can request one or multiple descriptors from the SP Catalogue. The process from the SP is the same for the `son-package`' request, however it requests Network Services Descriptors (NSDs), Virtual Network Function Descriptors (VNFDs) and Package Descriptors (PDs).

3.2.1.2 Planned features

While the component is currently under development, for SONATA Year 2 next releases, `son-access` component plans to complete next functionalities. Figure 3.3 shows the complete workflow between each component:

Access This sub-component will add a security layer to the communications with the SP and will perform automatically required security processes for each interaction. To achieve this goals, next features will be addressed:

- Integration with rest of `son-cli` components, specially `son-workspace`, which will include end-user and configuration settings, such log-in credentials.
- Improvements on Access Token management, automatically including Access Token on each message header to authenticate and authorize developers.

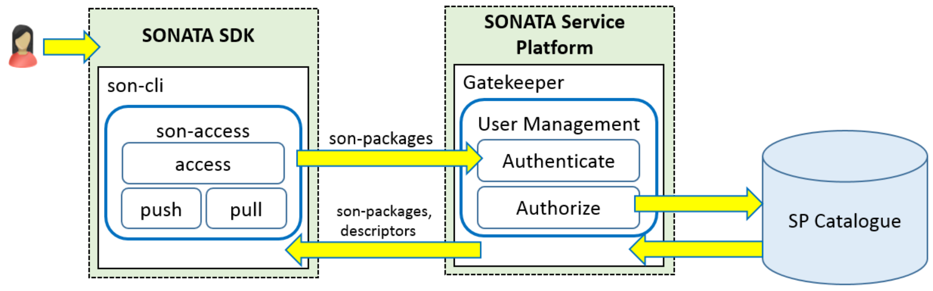

Figure 3.3: SDK and SP Catalogues components interaction workflows

Push This sub-component will be integrated with the other sub-components and will face next changes:

- Submit descriptors such Service and Function descriptors to the SP Catalogue. This feature is supported partially on the SP Catalogue side, but Gatekeeper API requires package files instead of descriptors.
- Enable and improve other former `son-push` functionalities.

Pull As the other sub-components, this will also be integrated and receive improvements:

- Enable downstream functions to retrieve one or multiple descriptors from the SP.
- Enable requests to retrieve `son-packages` from the SP and store them in developers' custom file system or storage system according to their preferences.

3.2.2 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. The former specification and design of `son-package` is described in D3.1 [5] Previously, the packaging process involved an interaction with the SDK `son-catalogue`, however as this component was integrated inside the Service Platform, this interaction will be performed using the `son-access` tool, in order to retrieve external dependencies. Moreover, the validation of project and its components that were integrated inside `son-package` will be performed by the new `son-validate` tool.

3.2.2.1 Planned features

Integration with `son-access` To be able to solve external dependencies, `son-package` must interact with `son-access` which will be responsible for retrieving service and function descriptors from the Service Platform Catalogues. This will not affect the logic and workflow process of building a service package, but a modification of the inner `son-package` client to interact with `son-access`. To be noted that a cache system for the external dependencies will still exist and work according to the same logic of the previous implementation.

Integration with `son-validate` The validation process of services, functions and the final package itself will also be outsourced to the `son-validate` tool. As `son-validate` is being developed specifically for this purpose, but also providing additional validation features other than syntax (service integrity and network topology), it is more proficient to use this tool to perform the validations.

3.2.2.2 Planned workflow

As a result from the foreseen modifications, the workflow of `son-package` will be as illustrated in Figure 3.4.

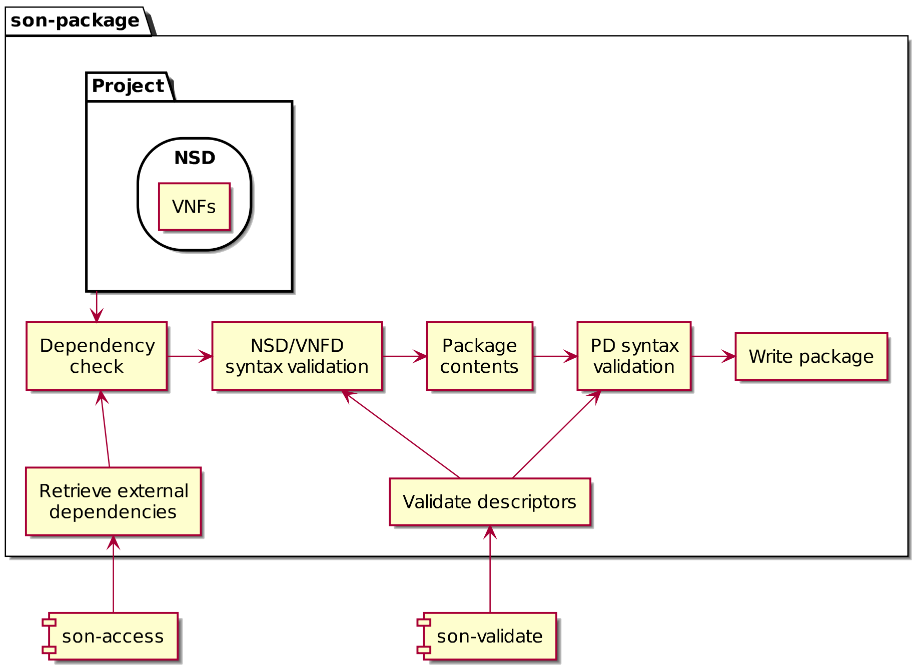

Figure 3.4: son-package tool

3.2.3 son-push

The `son-push` tool as part of the first release of the SONATA SDK will cease to exist as a stand-alone component. As documented earlier in this section, push-functionality will be integrated into the new `son-access` component.

3.2.4 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. A comprehensive description of its functionalities and design were specified in D3.1 [5]] and its new features are specified here.

3.2.4.1 New features

versioning The new requirements of the SDK platform, namely developer authentication, resulted in drop of support to previous workspace versions. As a result, a new versioning systems was implemented to force the update of `son-workspace` to the new version.

3.2.4.2 Planned features

configuration parameters New configuration parameters are being implemented aiming to include the developer credentials and addresses to multiple SONATA Service Platforms. The developer authentication credentials to access a specific Service Platform must be stored in its workspace. Moreover, a developer should be able to access multiple Service Platforms and as such, the workspace configuration must hold multiple credential entries.

3.2.5 son-validate

The `son-validate` is a new CLI tool for the SDK with the purpose of aiding the development of services and functions. This tool was mainly developed to support the validation of SDK projects, however it is designed to also be utilised outside the SDK scope. Individual service and function descriptors can be validated independently, without requiring a developer workspace, which makes it adequate to be used by the SONATA's Service Platform. The `son-validate` addresses the following validation scopes:

- **Syntax**
- **Integrity**
- **Network Topology**

3.2.5.1 Syntax

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

3.2.5.2 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.

3.2.5.3 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 3.5 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.

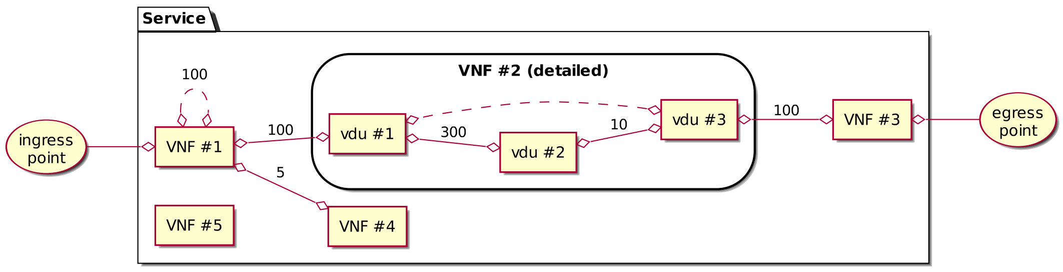

Figure 3.5: Example of a Service Network Topology

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.

3.3 son-catalogue

The `son-catalogue` component is composed by the API implementation of the SDK Catalogues and the database for the services and functions descriptors. Development of the SDK Catalogues is ceased due to the removal of the component in the SDK. The key functionality of the SDK Catalogues component was to offer a local API and storage for developers to easily manage descriptors.

While this component is no longer supported in the SDK, it has been stated to support SDK Catalogues functionalities in the Service Platform Catalogues in order to have one multipurpose catalogue in the SONATA ecosystem.

3.3.1 Updates and improvements

In SONATA Year 1 version, there were two different catalogues in the SONATA ecosystem, the Service Platform Catalogues and the SDK Catalogues. However, they both had almost the same functionalities but for different purposes. For Year 2, it was planned to have one multipurpose catalogue for the SONATA ecosystem, and then to remove the SDK Catalogues component from the SDK side. On October 31th, SDK Catalogues was removed as a component from the SONATA SDK.

Year 1 Service Platform Catalogues was focused on Platform storage for descriptors while SDK Catalogues was intended to be a local storage in developers' machine.

While development of this component has ended in release v1.0 version, code is still available in the GitHub repositories although there is no new updated planned for Year 2. Current status of the component involves that any development and/or new update will be implemented on Service Platform (SP) Catalogues instead, and SDK Catalogues code will remain available on the repositories until Year 2 Service Platform Catalogues can fully replace SDK Catalogues functionalities.

SDK Catalogues enabled an API that supported Network Service and Virtual Network Functions descriptors (NSDs, VNFDs) management and storage on the developers' local machine. The SDK `son-cli` component was fully integrated with the SDK Catalogues in order to push / pull descriptors. SP Catalogues do also support management and storage of NSD and VNFD. In addition, they also support Package Descriptors (PD) and SONATA packages (`son-package`) storage. The Service Platform Gatekeeper controls access to the SP Catalogues from outside the Platform and other components within the Platform.

In SONATA Year 2, SP Catalogues will replace SDK Catalogues and, in order to meet this requirement and fill SDK Catalogues functionalities in the SP Catalogues, a new component will grant access from the SDK to the SP Catalogues. This new component is called `son-access`, and it will remain inside the `son-cli` component (see Section A.4). Updates on the Gatekeeper will be required in order to grant access to developers from the SDK to the SP Catalogues.

3.3.2 Planned features

SDK Catalogues is no longer supported and no features or updates are planned for Year 2 phase. The adaptation of SDK Catalogues functionalities in the SP Catalogues is the main goal for the catalogue component of the SONATA ecosystem. This will require other components to receive updates in order to successfully support these functionalities. Next list shortly describes involved components and planned new features or improvements:

- SDK `son-access`: A new component is being developed under the `son-cli` component in the SDK. This component will replace the SDK Catalogues component in the SDK side, including updates on the `CatalogueClient` and former `son-push` component. It will grant secured access to the SP Catalogues through the Gatekeeper and will be responsible for enabling functionalities for submitting and requesting packages to the SP Catalogues. More details about this new component can be found at Section A.4.

- SP Gatekeeper: It will act as a bridge between SDK `son-access` and the SP Catalogues. It features an API that allows users and developers external to the Platform to manage descriptors and packages from the SP Catalogues. It currently supports requests for descriptors and packages, but only supports submitting packages. The API might be updated to also support submitting descriptors to the SP Catalogues. More information can be found on Deliverable D4.2 [7].
- SP Catalogues: This component is not currently having any change in regard to its core implementation but will be the component responsible for storing descriptors and SONATA packages from SDK users. It will fully replace the SDK Catalogues functionalities and it will be secured through the Gatekeeper. In order to gain access and use them, SDK end-users will be required to be registered to the SONATA Platform. SDK end-users will be free to use any file system or database locally to save and manage their descriptors.

3.4 son-emu

Our emulation platform called `son-emu` allows network service developers to locally prototype and test complete network services in realistic end-to-end multi-PoP scenarios, like already described in Section 4.2 of [3] and Section 3.4 of [5]. The following sections give insights about new features available for `son-emu` and future plans for this component.

3.4.1 New features and improvements

- **PoP resource isolation models:** The emulator offers a plug-in interface to assign custom resource models to PoPs. These resource models are then called whenever new compute instances are requested in a particular PoP and can compute resource limits, such as CPU cores, CPU shares, or memory, which are then applied to the started container. This mechanism allows developers to emulate resource isolation between PoPs and to simulate cloud-like overbooking of individual PoPs. There are two example resource models available which are described and evaluated in [15].
- **Gracefully shut down services through the SONATA dummy gatekeeper interface:** The REST interface of the dummy gatekeeper was extended to allow a graceful shutdown of a previously instantiated service. A shutdown is requested by performing a HTTP *DELETE* request on the REST endpoint of the service instance.
- **VNFD-based container resource limit support:** Resource configurations, such as number of CPU cores or memory limits that can be defined within a VNFD, are now applied to the VNF containers running in the emulator when a service is deployed using our dummy gatekeeper. This improves the compatibility of our dummy gatekeeper to the SONATA descriptors. This feature is also important for the foreseen integration between `son-emu` and the planned profiling tool, called `son-profile`.
- **Round-robin placement in dummy gatekeeper:** The dummy gatekeeper now distributes the VNFs of a deployed service evenly across all available PoPs.

- **REST API extension to set CPU limits:** Resource allocation can be dynamically altered during runtime. The resource configuration of the VNFs, deployed as Docker containers, is exposed via the REST API.

- **Isolated E-LAN networks are supported and deployable from the dummy-gatekeeper:** LANs are isolated using a specific VLAN tag for each specified E-LAN network in the service descriptor file. The OVS switches in the emulated infrastructure network and datacenters are configured to be used in stand-alone mode so the E-LAN packets are forwarded based on the switches' learning capabilities and VLAN tags of their interfaces. For E-Line links, the interfaces have also unique VLAN tags configured. By using unique VLAN tags on interfaces belonging to an E-LINE or E-LAN network, it is ensured that the learning virtual switches isolate network traffic between E-Line and E-LAN networks. More specifically, flooded packets on an E-LAN network are isolated and not forwarded to other E-Line or E-LAN networks. E-Line links have dedicated OpenFlow entries in the switches that enable service function chaining.

- **Xterm auto-start for son-emu-cli:** The command `son-emu-cli xterm <vnf_name>` can start an xterm window that attaches to the specified Docker VNF (not yet supported when son-emu-cli is executed locally and son-emu runs in an isolated VM but this can be solved by creating a SSH connection to the VM and executing the xterm commands in the VM).

- **The son-emu REST API returns the names of connected interfaces of a deployed VNF:** In particular the end of the veth pair which is connected to the emulated PoP switch. (The other end is included in the container namespace and therefore not visible from the host). This makes it possible to retrieve the correct interface name that should be monitored, with e.g. Wireshark, when trying to monitor the network traffic flowing through the emulated service on the host.

- **Bug fixes:**
  - Added priority field to son-emu-cli networking interface.
  - Dummy gatekeeper now returns `status 201` when a package is uploaded.
  - Dummy gatekeeper does not block if a start command of a VNF container fails.
  - CLI compute list output now shows the correct management interface IPs.

3.4.2 REST API

Table 3.1 and Table 3.3 document the REST API endpoint which is provided by `son-emu` and can be used to control parts of the emulations. This API is also used by the `son-emu-cli` tool. All URLs shown in the tables have to be prefixed with `/restapi`. All parameters in Table 3.1 have to be included as JSON into the messages body.

The update of resource allocations of VNFs deployed in `son-emu` can be controlled via a PATCH request to a specific VNF. The resource parameters that can be controlled for Docker containers are exposed: cpu, memory and block IO [8].

Table 3.1: `son-emu` compute and network request interface endpoint

Action	Entity	Http method	Path	Parameter
query	compute	GET	/compute/{dc_label} /{compute_name}	
instantiate	compute	PUT	/compute/{dc_label} /{compute_name}	• container specification • network specification • image name • start command • resource specification
change resource allocation	compute	PATCH	/compute/{dc_label}/{compute_name}	• resource specification
terminate	compute	DELETE	/compute/{dc_label} /{compute_name}	
query	compute	GET	/compute/{dc_label}	
query	datacenter	GET	/datacenter/{dc_label}	
query	datacenter	GET	/datacenter	
install	network	PUT	/network/{vnf_src_name} /{vnf_dst_name}	• src interface • dst interface • weight • match • bidirectional • cookie • priority
remove	network	DELETE	/network/{vnf_src_name} /{vnf_dst_name}	

Table 3.2 documents the REST API which controls the export of network metrics related to a VNF interface or a specific flow in the installed Service Function Chains. This REST API is used by `son-monitor` to install custom monitoring metrics. To allow easier implementation of further extensions, {`vnf_interface`}, {`metric`} and {`cookie`} will be implemented as JSON parameters in the message body, as part of a future update.

```
{metric} : tx_packets, tx_bytes, rx_packets, rx_bytes
{cookie} : chosen integer to identify (a set of) flows installed in the
network topology to steer the network traffic                   in the
service
```

Table 3.2: `son-emu` monitor request interface endpoint

Action	Entity	Http method	Path	Description
export network interface metric	monitor	PUT	/monitor/vnf/{vnf_name} [/{vnf_interface}] /{metric}	export traffic rate of specified vnf interface (take first interface of the vnf if no interface is specified)

Action	Entity	Http method	Path	Description
stop exporting metric	monitor	DELETE	/monitor/vnf/{vnf_name} [/{vnf_interface}] /{metric}	
install & export flow metric on a link	monitor	PUT	/monitor/link/{vnf_src_name} /{vnf_dst_name}	monitor traffic rate of a flow on a specific link in the service. The link is specified by the src and dst. After the flow metric is installed, it is exported, specified by its cookie. Parameters: • src interface • dst interface • match • metric • cookie • priority
stop monitoring flow metric on a link	monitor	DELETE	/monitor/link/{vnf_src_name} /{vnf_dst_name}	
export flow metric of vnf finterface	monitor	PUT	/monitor/vnf/{vnf_name} /{vnf_interface} /{metric}/{cookie}	export traffic rate of a flow at a vnf interface, specified by the cookie
stop exporting flow metric	monitor	DELETE	/monitor/vnf/{vnf_name} /{vnf_interface} /{metric}/{cookie}	

Table 3.3: `son-emu` response interface endpoint

Action	Entity	Http method	Path	Responses
query	compute	GET	/compute/{dc_label} /{compute_name}	• 200: compute instance information dict. • 500: error
instantiate	compute	PUT	/compute/{dc_label} /{compute_name}	• 201: none • 500: error
terminate	compute	DELETE	/compute/{dc_label} /{compute_name}	• 200: none • 500: error
query	compute	GET	/compute/{dc_label}	• 200: compute instances list • 500: error

Action	Entity	Http method	Path	Responses
query	datacenter	GET	/datacenter/{dc_label}	• 200: datacenter information dict. • 500: error
query	datacenter	GET	/datacenter	• 200: datacenter list • 500: error
install	network	PUT	/network/{vnf_src_name} /{vnf_dst_name}	• 200: chain info dict. • 500: error
remove	network	DELETE	/network/{vnf_src_name} /{vnf_dst_name}	• 200: chain info dict. • 500: error
export interface/flow metric	monitor	PUT	/monitor/{vnf_name} [/{vnf_interface}]/{metric}[/{cookie}]	• 200: info string. • 500: error
stop exporting metric	monitor	DELETE	/monitor/vnf/{vnf_name} [/{vnf_interface}]/{metric}[/{cookie}]	• 200: info string. • 500: error
install flow metric	monitor	PUT	/monitor/link/{vnf_src_name} /{vnf_dst_name}	• 200: flow dict. • 500: error
uninstall flow metric	monitor	DELETE	/monitor/link/{vnf_src_name} /{vnf_dst_name}	• 200: info string. • 500: error

3.4.3 Planned features

Most improvements planned for `son-emu` for the second half of the project target its integration with the new `son-profile` component. However, one major feature that will be investigated is the integration between `son-emu` and SONATA's service platform, as described in the next section. By integrating `son-emu` as a dedicated infrastructure, controlled by the SONATA service platform, we must also keep the design as modular as possible to enable re-use with other service platforms and compatibility with SONATA SDK features.

3.4.3.1 Integration with WP4 Service Platform Orchestrator and Infrastructure Adaptor

son-emu emulates environments with an arbitrary number of PoPs that can be controlled by one, or more, MANO systems. In the most simple case the build-in MANO layer, called *dummy gatekeeper*, is used to deploy SONATA services in the emulated environment. However, such deployments would be even more realistic if the MANO system of a *real* service platform could be connected to the emulator. One option to do this would be to add additional wrappers for son-emu to the *infrastructure abstraction* component of SONATA's service platform. Another option is to extend son-emu so that it is able to *fake* standard cloud interfaces for each of its emulated PoPs, for example, exposing a REST API that behaves like OpenStack Heat, like shown in Figure 3.6. This option comes with some obvious benefits. First, SONATA's service platform does not need any adaption and we can focus on the son-emu component to do the integration. Second, having standard cloud interfaces in place might allow to easily integrate our emulation platform with other MANO systems, such as OSM [12] which increases the usefulness and re-usability of son-emu substantially.

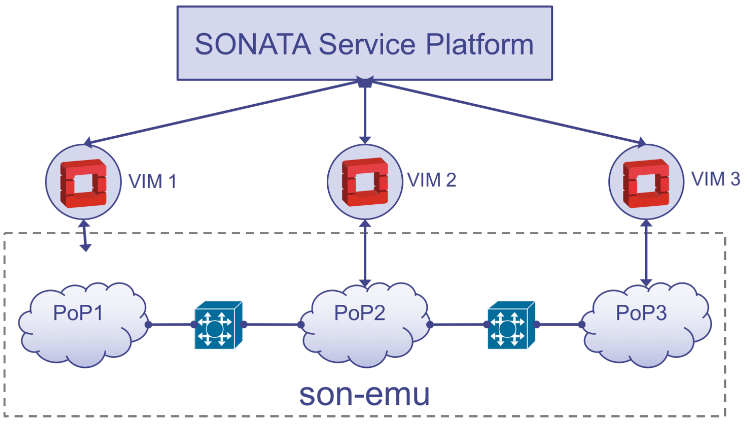

Figure 3.6: Envisioned integration between son-emu and SONATA's service platform orchestrator

To create a setup like shown in Figure 3.6 we plan to do some experiments to identify and prototype the subset of needed OpenStack API endpoints we have to re-implement to allow the minimal needed functionality to deploy a service on son-emu by using the MANO system of SONATA's service platform.

3.4.3.2 Modular architecture of the emulator's environment

The service platform developed in WP4 can be used to control the orchestration and monitoring of services on emulated environments in son-emu. This will allow better integration of all modular aspects of a SONATA network service in the SDK environment, such as SSM/FSM development and monitor data gathering/analysis. Furthermore, the development workflow, as described earlier in Section 2, must be respected.

In a first phase, the SDK environment, the emulator and the service platform will run locally on the developer's machine. To keep the SDK environment as light-weight as possible, not all modules of the Service Platform need to be deployed. A selection will be made to maximize development support. An example in this context might be message bus access for SSM/FSM support and orchestration functionality to deploy a SONATA service package on son-emu via the default SONATA Gatekeeper.

At a later stage, we will investigate how we can exploit the modular approach further, leading to an environment where the SDK, Service Platform and Emulator are all running on different physical node, as illustrated in Figure 3.7. `son-emu` can be installed in a dedicated VM in this case. Challenges in this setup are:

- The Emulator is treated as an additional Infrastructure where service can be deployed. When pushing a service from the SDK, the developer should be able to configure `son-emu` with a custom PoP topology and choose it as deployment environment.
- Keep access open to the `son-emu` REST API. Access control via the Gatekeeper should allow the SDK to use this REST API for various development features implemented in `son-monitor` or `son-profile`.

Further SONATA emulator and SDK enhancements will be implemented, keeping in mind this modular approach. The `son-emu` REST API will be the main interface for SDK development features to communicate with the emulator.

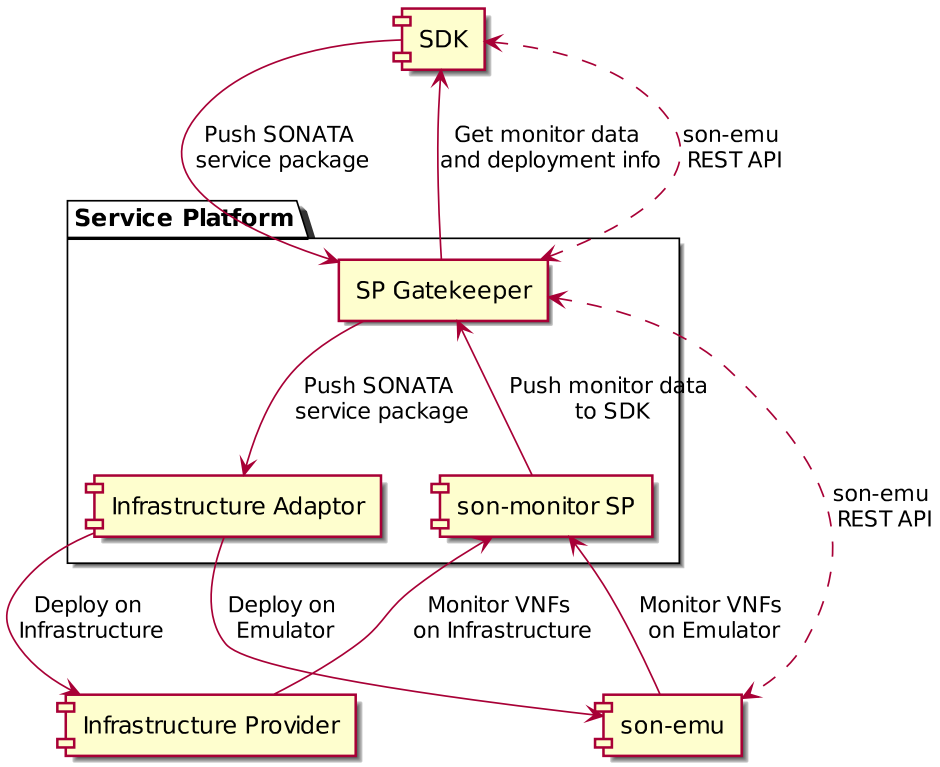

Figure 3.7: Modular architecture of the SDK, Service Platform and son-emu

3.5 son-profile

`son-profile` is a new tool planned to be part of SONATA's SDK to allow network service developers to automatically profile and test network services before they are deployed to production. The tool will be released in the second half of the SONATA project and is currently under development. 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 following sections will shed some light on `son-profile` early component design and implementation details since the high-level design and the general concept of NFV profiling was already introduced in Section 3.2.3 of deliverable D2.3 [4].

3.5.1 Design of son-profile

We designed `son-profile` to be as modular as possible so that other users can easily modify and extend it, e.g., to be compatible with other service descriptor formats or to interact with other execution platforms or monitoring systems. Figure 3.8 shows our initial design and the involved components as well as the artefacts read or written by the tool. It also shows how `son-profile` integrates with other components of SONATA to control the execution of profiling experiments.

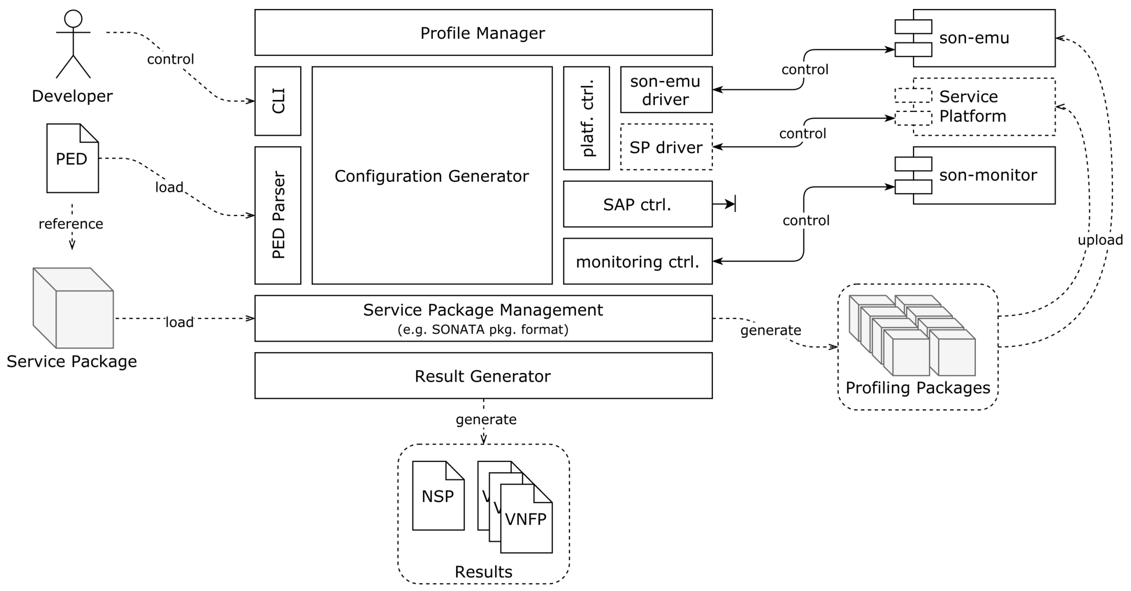

Figure 3.8: Detailed component breakdown of son-profile tool

The first component in the figure is the *Profile Manager* which is the central component of the tool which is responsible to control the workflow and instruct other modules. On the input side, `son-profile` offers a *CLI* which can be used by a developer to start or stop profiling experiments which are defined in so called *Profile Experiment Descriptors (PED)*. These YAML-based descriptors are read by the *PED Parser* module which is able to detect and unroll Omnet++-like [11] parameter macros inside these descriptors. This gives a developer a convenient interface to easily define complicated parameter studies by editing a single file. These inputs are then passed to the *Configuration Generator* which is in charge of computing all necessary service configurations that should be tested during the planned profiling run. This is typically done by computing the Cartesian product of all configuration parameters to be tested according to the given PED file.

After all configurations are properly generated, the *Service Package Management* module is activated and reads the input package that contains the service to be profiled and that is referenced by the given PED file. Based on the input package, the *Service Package Manager* module then generates one new service package for each generated configuration. The resulting *Profiling Packages* are then written to a temporary folder and are deployed one after each other during the actual profiling phase. It is worth mentioning that the encapsulation of the entire service package management into a single component has the advantage that only this component has to be replaced to make `son-profile` compatible to other service description and packaging formats, like OSM [12].

Using the generated *Profiling Packages*, `son-profile` connects to or starts an execution platform (default: `son-emu`) to deploy the first *Profiling Package*. These profiling services also contain additional helper VNFs that are called *Service Access Points (SAP)* and are connected to the ingress and egress points of the service under test and allow to inject traffic into the service chain, e.g., iperf (cf. D2.3 [4]). This setup is then executed for a pre-defined time and measurements are recorded by either `son-monitor` or based on log files shared by the containers. After these measurements are done, the service is terminated and the next service configuration (next *Profiling Package*) is deployed and tested. To control this entire process, `son-profile` is able to interface with external components. First, it can interface with execution environments, like `son-emu` or any other service platform, as long as a driver module for it was created. Second, `son-profile` can directly interface with the SAP containers, e.g., by SSH, to control the traffic generators and their properties. Third, `son-profile` interfaces with `son-monitor` to kick-off the monitoring process for each profiling run.

After all profiling runs have been performed, the recorded monitoring data and log files are collected and processed by the *Result Generator* which creates the final performance profiles, e.g., *Network Service Profiles (NSP)* and *VNF Profiles (VNFP)*. This component can again be replaced, e.g., by machine learning algorithms that analyse the results instead of writing the raw values to disk.

3.5.2 Profiling experiment description

Profiling experiment descriptors (PED) are YAML-based description files that define how a given network service should be automatically profiled. Figure 3.9 shows an entity relationship diagram of a PED as it is used in our current prototype. It may evolve over time and will be stable when the first functional version of " son-profile" is released. Each PED has some high-level descriptions fields, such as vendor, name, version, and thus follows the general design of all SONATA description files. In addition, a PED file contains a *service_package* field which contains a path to the service package to be used. In each PED file, a developer can define several service and/or function experiments. A service experiment is used to test or profile a complete network service chain which may consist of multiple VNFs. A function experiment, in contrast, is used to test or profile a singe function (a VNF) that is part of the given service. Each experiment contains a reference that either references the service (NSD) or function (VNFD) that should be used. Further, an experiment contains parameters, like experiment *duration* or a *time_limit* to define how long each configuration will be tested. Inside each experiment, a developer specifies the service access points (SAP) that should be connected to the service or function in order to send test traffic through them. Additionally, resource limitations can be specified to test VNFs with different resource configurations, e.g., test *VNF-A* with 1, 2, 3, and 4 cores to obtain information about its scaling behaviour (cf. D2.3 [4]).

3.5.3 VNF profiling case study

We executed a series of initial profiling experiments to validate the feasibility our approach [14]. Figure 3.10 shows some example results of our experiments to indicate how profiling results for a single VNF, in this case the well-known *Snort IDS* [1], could look like. The VNF was installed in an Ubuntu 14.04-based Docker container and the experiments have been executed on a machine with Intel(R) Core(TM) i7-960 CPU @ 3.20 GHz, 4 physical cores, hyper threading, and 24 GB RAM. The error bars in all presented results represent 95% confidence intervals.

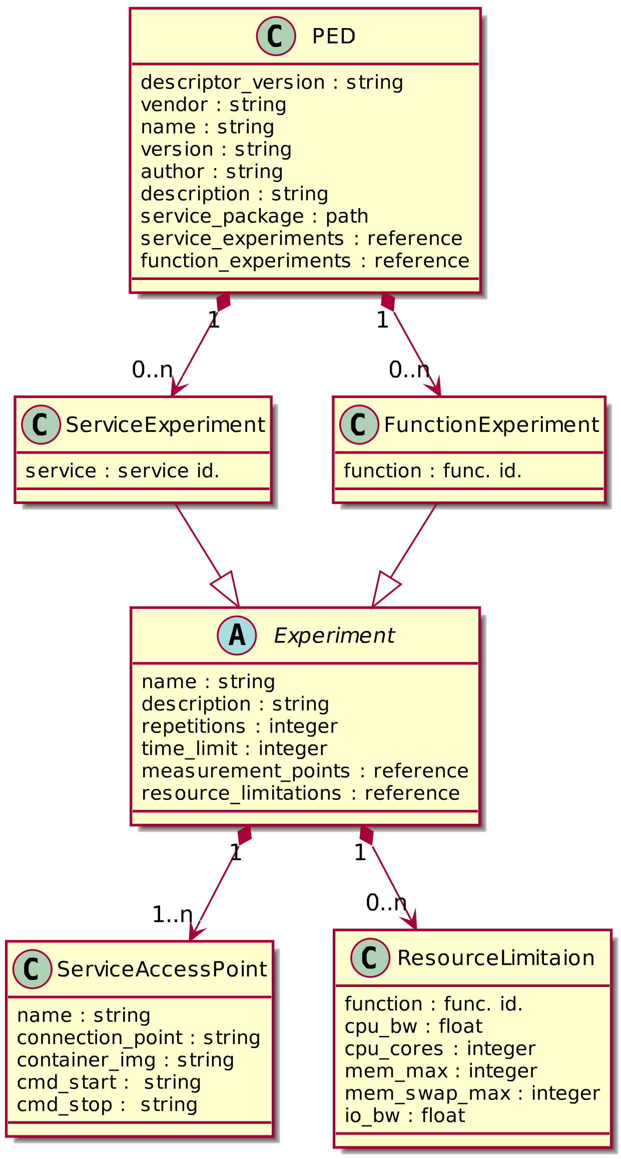

Figure 3.9: Entity relationship diagram of a profiling experiment descriptor (PED)

All experiments use iperf-based service access point (SAP) containers to send traffic through the Snort intrusion detection VNF and measure its throughput. We profiled two major versions of *Snort*, namely *version 2.9* installed from Ubuntu's package repositories and *version 3.0alpha* which is a preliminary release available as source code. Both versions are used in their default configuration and we profiled them under changing CPU limitations.

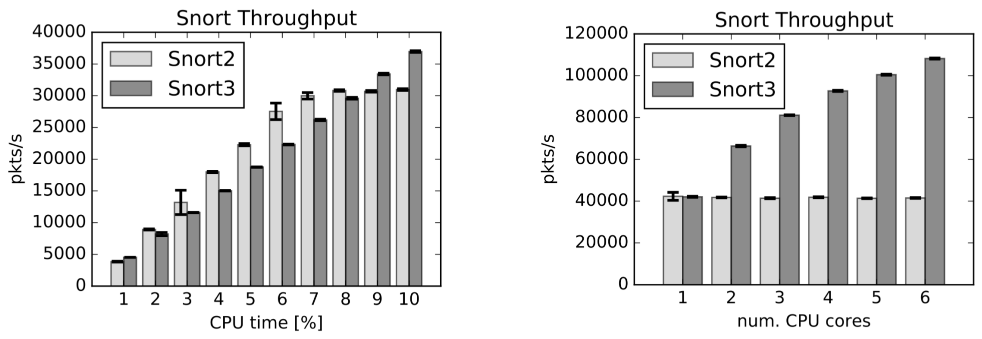

Figure 3.10: Example profiling experiment: Throughput comparison of two major snort versions and changing CPU configurations (CPU time (a) and CPU cores (b))

Figure 3.10 shows the averaged results of 25 repetitions of these experiments. The results provide insights about the runtime behaviour of these two different *Snort* versions. Figure 3.10(a) shows their behaviour under very limited CPU time allocations (son-profile can connect (e.g. via SSH) and execute the specified commands and start/stop for example traffic generation.

- The specified measurement points should be exported by the monitoring framework.
- Any Resource Limitation for VNFs deployed on `son-emu` should be controllable via `son-emu` 's REST-API.

3.6 son-monitor

In the scope of the entire SONATA project, `son-monitor` related work in WP3 is focused on providing helpful and user-friendly ways to monitor developed network services and making this monitor data available for further analysis and debugging of the service. A demonstration of the SONATA SDK monitoring features was published in [17]. In its current state, `son-monitor` features can gather and store a wide set of metrics on services which are deployed on the `son-emu` emulator. A service developer can use the SONATA SDK to view and further analyse these metrics for troubleshooting, debug or optimization purposes.

In future updates of the SONATA framework, we plan to enhance the monitoring functionality with the following new features:

- Together with `son-emu` and `son-profile`, additional test functionality can be implemented by using dedicated Test-VNFs with programmable features such as traffic generation and analysis. Part of the work in `son-monitor` will be dedicated to the development of such Test-VNFs and how to control them from the SDK. (e.g. the PED file described in Section 3.5.2)
- `son-monitor` could leverage standard metric monitoring by also providing ways to monitor non-numerical data such as packet dumps, log files, configuration files or VNF state information.

- From the SDK, son-monitor should be able to receive monitor data of services deployed in the actual Infrastructure, using the Service Platform, as described later on in this section.

The current status and further updates regarding son-monitor will be detailed in next sections. More detailed implementation details and usage is given in Appendix C.

3.6.1 Son-monitor architecture

The planned updates regarding the SDK monitoring features are illustrated in Figure 3.11. The different functional blocks in the modular architecture of the SONATA SDK are shown. Compared to previous deliverable D3.1, these updates are already implemented:

- **Prometheus Gateway + Database**: To respect the modular approach of the SONATA framework, the Prometheus Gateway is implemented as a Docker container running together with son-emu (on the same node). The Prometheus Database is also a Docker container but can be started on any other node (where the SDK is installed). This implementation can later be made compatible with the Service Platform's Prometheus Database, which can be configured to also gather metrics from son-emu 's Gateway.

- **Visualisation using Grafana**: To provide a more user-friendly way of monitoring and debugging, monitored metrics can be visualized using Grafana [10]. This visualizes the queried metrics from the Prometheus database through a web-based GUI. The Grafana framework is started as a Docker container. A Monitoring Service Descriptor (MSD) file describes which metrics need to be visualized and how they should be combined on different graphs. The son-monitor control code parses this MSD file and translates this to a Grafana Dashboard. Grafana is then configured to display this dashboard, using its standard REST API. By default the Grafana REST API and web GUI are available at http://localhost:3000 on the SDK.

- **Network metric exporters in son-emu**: Next to the compute, storage and network related metrics gathered by cAdvisor in son-emu, additional network metrics are gathered and exported using a query loop. This loop is using an SDN controller to query via the OpenFlow protocol the packet and byte counters of the (virtual) network interfaces that are installed in the emulated service. Next to the interface counters, also specific flow counters can be installed to allow a more finer-grained network traffic monitoring. The set of interface and flow counters to be monitored can be configured from the MSD file or the son-emu REST API. All metrics gathered in son-emu are pushed to a Prometheus Push Gateway where they await to be pulled by the external Prometheus Database in the SDK.

The green boxes in Figure 3.11 depict the updated planned functionality which are further explained in next sections.

3.6.1.1 Son-monitor REST API

Once monitored metrics are available and stored in the Prometheus Database, they can be queried and used for further analysis, for example by the son-analyze tool-set. As mentioned in Section 2 and on Figure 3.11 data queries can be done by using the standard Prometheus HTTP API [16].

By default the Prometheus REST API is available at http://localhost:9090 on the SDK.

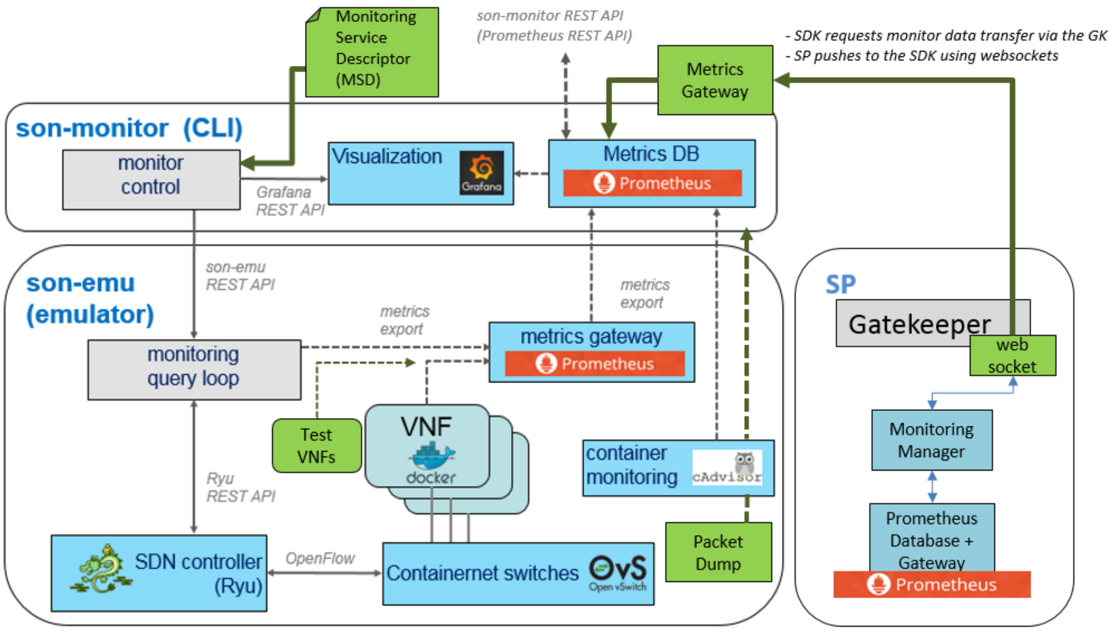

Figure 3.11: Different functional blocks in the SDK monitoring framework

3.6.2 Monitor service descriptor

To help a service developer monitor a wide set of metrics, a dedicated descriptor is devised. This Monitor Service Descriptor (MSD) file instructs the SDK monitoring framework to gather and display a custom set of metrics. The standard SONATA descriptor files in the service package (NSD and VNFD) define the metrics that should be gathered when the service is in production. The MSD on the other hand, defines a broader or different set of metrics that are only needed during development. It is a dedicated, customizable file since the monitor requirements of a service in development might be different compared to a service running in production. The MSD file can be used to dynamically change the monitored parameters of a deployed service in `son-emu` without having to interrupt or re-deploy the service. It is given as input to `son-monitor` who will parse it, initialize the export of the required metrics, translate it to a Grafana dashboard and configure the correct Prometheus queries to get the values out of the Database.

The structure of the file is given in Figure 3.12. It can be seen that two major classes of metrics can be specified:

- **VNF metrics**: This type of metrics is related to a VNF, interface or service as a whole, grouped per metric type. This can be used to compare the same metric type for different VNFs or interfaces.

Table 3.4: Valid VNF Metrics for the MSD file

Parameter	Possible values/Type	Description
metric_type	`["cpu", "mem"]`	Compute related metrics
metric_type	`["packet_loss", "jitter"]`	Metrics that are exported from a dedicated Test-VNF
metric_type	`["packet_rate", "byte_rate", "packet_count", "byte_count"]`	Network metrics for a specific interface
vnf	`"vnf_name:interface"`	vnf_name+interface as used in the NSD

Parameter	Possible values/Type	Description
direction	`["tx", "rx"]`	Monitor the egress or ingress traffic

- **NSD link metrics**: This type of metrics is related to the network/flow statistics in the emulated service, grouped per metric type. This can be used to compare in detail different types of network traffic/flows on the same or different links/chains. If the link is not a pre-defined chain in the NSD, it cannot be monitored.

Table 3.5 explains the NSD link Metrics which differ from the VNF metrics above.

Table 3.5: Valid NSD Link Metrics for the MSD file

Parameter	Possible values/Type/Format	Description
link_id	string	link id that is also used in the NSD
metric_type	`["packet_rate", "byte_rate", "packet_count", "byte_count"]`	Network metrics for a specific interface or flow
source	`"vnf_name:interface"`	the source of the monitored traffic
destination	`"vnf_name:interface"`	the destination of the monitored traffic
match	string (OpenFlow based match entry)	Specify flow using the same syntax as OpenVSwitch [13]. If match is empty, total interface counter is used.

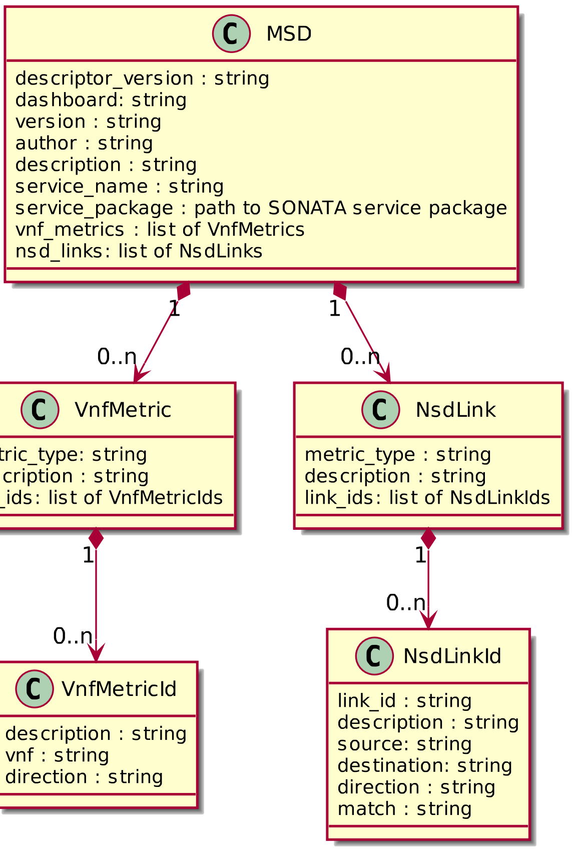

Figure 3.12: Entity relationship diagram of a Monitoring Service Descriptor (MSD)

3.6.3 Planned features

Future development in WP3's `son-monitor` will be related to the topics explained below.

3.6.3.1 Dump packets on any VNF interface deployed in son-emu

A `son-monitor` command starts a packet dump tool (e.g. tcpdump or wireshark) and looks up the correct interface/veth pair that is used for the specified VNF/interface in the descriptor. The packet stream can be saved (as a .pcap file) for further re-use and analysis. As indicated on Figure 3.11, the packet dump functionality should be executed on the same node where the VNF interface is. This way, packet capture speed is optimal and the captured stream can be filtered before exported to the SDK. In practice the packet dump functionality will be first implemented as a micro-service on `son-emu`, started via its REST API. The resulting .pcap file can then be transferred to the SDK side.

Further functionality in this context can focus on the export of other non-numerical data from a service such as log files, configuration files or VNF state information (e.g. gathered via SSH login to the VNFs). For the export of this data from the Service platform, special care should be taken to sufficiently filter out privacy-related parameters (such as ip/mac-addresses and clear text payload). The gathered files in `son-emu` or other Infrastructures can be transferred to the SDK and stored there for further analysis or re-use.

3.6.3.2 Test-VNF functionality

The test VNFs can be deployed in the service's endpoints on the emulator or can be specified in the service descriptor and deployed on the SP. They can be used to execute profiling or functional testing, where Test-VNFs are attached to the service for the duration of the test. In function of the example services shared via the SONATA GitHub repository, basic functionalities implemented in Test-VNFs will include:

- Packet generation at the input of a VNF/service.
- Packet monitoring and analysis at the output of a VNF/service.
- Metric export from a Test-VNF to Prometheus.
- SSH login to execute scripts/programs from the SDK.

3.6.3.3 Support for son-profile

As described in Section 3.5, profiling functionality can be implemented in the SONATA framework. The development done in `son-monitor` will further support this:

- The PED file defines a set of metrics to be monitored. These values can be queried from the SDK Prometheus database provided by `son-monitor`.
- The PED file defines a set of commands that need to be executed on the Test-VNFs to generate a certain traffic load. `son-monitor` will provide Test-VNFs which allow execution of commands by SSH login.

3.6.3.4 Web socket connection to the Service Platform

As illustrated in Figure 3.11, a websocket on the Service Platform will be 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 Service Platform. Access control, authZ/authC is implemented via the Gatekeeper. The received stream of metrics from the websocket on the Service Platform needs to be stored in the Prometheus DB in the SDK. This is also described in D2.3 [4] (section 3.2.5) and D4.2 [7] (section 7). On the SDK-side a dedicated Metrics Gateway will be developed which can connect to the websocket and store any received metric in the SDK Prometheus Database.

3.6.3.5 Functional testing for NFV Services

Using the `son-profile` and `son-monitor` features in the SONATA SDK it is possible to automate the execution of several commands, dynamically changing the traffic load in a VNF or service and meanwhile monitor the performance. To further extend this to functional testing, an 'assert' function should exist to check if measured metrics are within pre-defined boundaries. The PED file could be extended with these asserts and boundaries to fully implement a functional test for a VNF or service.

Automated functional testing for SONATA services can be implemented with following assumptions:

- The SDK can control a deployed Test-VNF by SSH and execute the needed commands to start the functional tests (e.g. traffic generation).
- The needed monitor data is available in the SDK Prometheus database.
- An extended PED file holds boundary data for the monitored metrics.
- The SDK can query the monitored data and check if the value is between the pre-defined boundaries (assertions).

3.7 son-analyze

3.7.1 New features and improvements

This sections presents the majors contributions to the `son-analyze` tool following the D3.1 [5] release.

3.7.1.1 License compliance

In the first proof of concept version, the R Lang ecosystem was the underlying base of the `son-analyze` tool. It used the native compiler to run the SONATA's R bindings and the RStudio environment for the graphical interface. RStudio featured the notion of notebook to store and share an analysis. This setup provided the biggest range of libraries and the best adoption in the open-source community. But this choice was incompatible with the SONATA's Apache 2 license. In particular, parts of the native R library is GPL licensed which infects most of the open-source libraries. Moreover, the RStudio environment is released under the Affero license. Because the `son-analyze` runs inside a container, the copyleft clause is not triggered. But, the situation was not so clear in terms of derivative work. To avoid license usage issues , the `son-analyze` tool was rewritten. Today, its code base is built on top of Python, SciPy and the Jupiter environment. The Figure 3.13 shows the current `son-analyze`'s architecture.

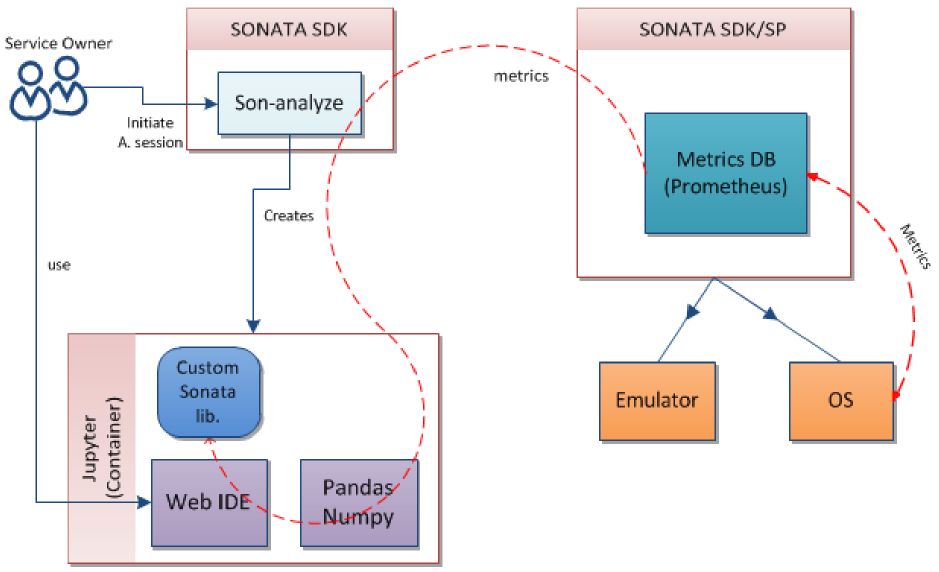

Figure 3.13: Son-analyze current architecture

3.7.1.2 Son-monitor bridge

To operate, the `son-analyze` tool requires data. When a service is run, the SONATA SP or the `son-monitor` tool gather VNFs' metrics. The first purpose of these metrics is to display to the service operator the current state of a service and its evolution across a relatively short period of time. As for the `son-analyze` tool, the service developer can inject those metrics in various statistical libraries to infer new pieces of information.

To make this happen, a Prometheus client was developed and shipped inside the `son- analyze` library. It comes with helper functions and structures to quickly parse and handle the data coming from the SONATA SP. With it, the service developer can query metrics to analyse or retrieve Network Service Descriptors or Network Service Records. By exploring the service's topology and components, the service developer will have a better understanding of the generated metrics. The choice of Prometheus, as a backend to store metrics, has been consistent between the SONATA SP and the SDK `son-monitor`. Because of it, the current gathering of metrics is compatible with the two environment. Now, with the upcoming Y2 release, this situation might slightly change as the SONATA SP will strengthen its security. As the Gatekeeper will adopt new authentication and authorization APIs, the flow of metrics will be gated behind a level of security. Thus, the next version of the `son-analyze` bindings will have to adapt to these changes.

3.7.2 Planned features

3.7.2.1 New SP bindings

Following the SONATA SP monitoring evolution explained in D4.2, the `son-analyze` tool might have to support the new flow mechanism for services' metrics. With this new feature, the metrics will be conveyed to the consumers in a live fashion as they are created by the probes in the SONATA SP. The need of streaming data has yet to be identified as a strong requirement for `son-analyze`. But supporting this feature might come handy to let the service developer creates online algorithms. At the very least, some kind of adaption will be required to comply with the new Gatekeeper version. In particular, it will roll out some new authentication and authorization mechanisms to enhance the SONATA SP security. The next Y2 `son-analyze` version will have to be compliant with them.

3.7.2.2 Adaptation loop

With the new addition of the `son-profile` tool in the Y2 release, a new lead is currently considered: an automated learning loop to maximise a service's QoS. As the `son-profile` automates the construction of test harness for services, it builds a performance profile depending on multiple constraints (for example, the amount of vCPU allocated to VNFs or the amount of traffic on a link). The service developer can then select the best profile that meet his requirements. For each profile, the resulting QoS (computed from the metrics) can be seen as a score. Using the `son-analyze` tool, the service developer can leverage this process by automatically discover the service's topology and constraints that maximize the service's score.

3.7.2.3 Analysis as a service

As a SDK tool, `son-analyze` is an optional step that the service developer can use to further scrutinise a service. In the current version, this tool is not integrated in an automated loop. Right now, the service developer must implement his analysis and update his service's topology or VNFs accordingly to his findings. Those two steps are manuals and, in particular for the first one, it isn't the scope of the `son-analyze` tool to automate them. Yet, it may be interesting to run an analysis in the SONATA SP to reach a better service adaptation.

A service developer can create, for example, an algorithm that predicts the service CPU load for the next hour by taking into account the past metrics and the concurrent number of the service's clients. By identifying the time ranges when the service's resource usage reaches its peak and bottom, the algorithm might learn an optimal scaling profile. But this algorithm lives in the SDK and it has no direct impact on the running service. Then by considering the dynamic variations of the algorithm results, it isn't possible for a service developer to manually apply the scaling decision all the time.

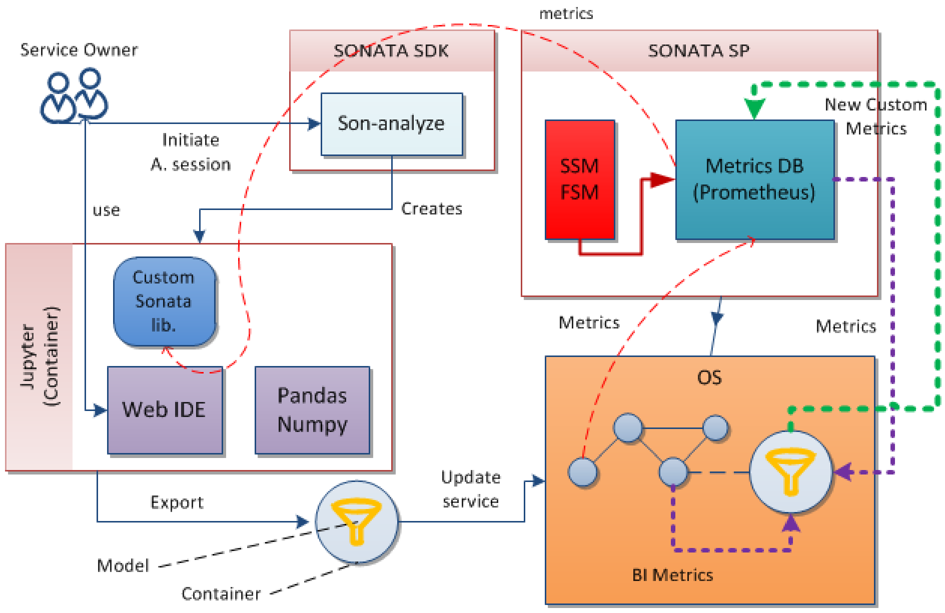

Figure 3.14: An analysis projection into the SONATA SP

Thus a possible new feature for the `son-analyze` tool could be to use the SDK CLI tools for compacting an analysis into a VNF and a service package. This package could then be instantiated in the SONATA SP, along side an already existing running service while configuring the necessary metrics flow. Thus the analysis will be fed with production metrics and make predictions on them. The Figure 3.14 displays this analysis projection in the SONATA SP. A SSM/FFSM could then grab this new piece of information to create a better adaptation decision.

3.7.2.4 Default analysis

To demonstrate the capability of the `son-analyze` tool, some basic algorithms will be supported. They will take as inputs a service topology and its metrics and computes results in two categories:

- Prediction: of the upcoming workload to scale a service ahead of time.
- Inference: of alerts threshold by detecting anomalies on a given metric.

4 Conclusion

For this intermediate SONATA release, we provided an update of the SONATA SDK as a collection of light-weight tools assisting in the development and testing on a local development machine. The programming model which unifies these tools is centred on the definition of schema for describing network services, network functions and associated management functionality.

This deliverable provides updates to existing schema and already existing SDK tools. The SDK enables now versioning of workspaces, has improved support and unification of monitoring both, emulated services as well as services deployed on the SP, and modified the way catalogues are accessed. In this new release, catalogues are securely accessible behind Gatekeeper functionality. The tool `son-validate` is a new SDK tool supporting syntactical and semantic validation and verification of services and components. In addition, developers might now extend the functional testing SDK capabilities with performance testing functionality as provided by the new `son-profile`.

Future releases of the SDK will reinforce and further extend existing components, improve usability (e.g., using GUOs), as well as introduce components for assisting developers in the development of Service and Function Specific Managers (SSM/SSM) or tools to improve interaction with other MANO platforms and service description formats.

A Manual of son-cli tools

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

A.1 son-workspace

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

The `son-workspace` tool receives the following arguments:

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

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

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

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

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

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

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

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

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

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

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

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

`id`: Identification of catalogue server

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

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

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

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-validate

The `son-validate` tool can be used to validate the syntax, integrity and topology of SDK projects, services and functions. 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]

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.
```

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 stand-alone 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 son-access

The `son-access` tool is responsible for authenticate the developer to gain access to the Service Platform. Once authenticated, `son-access` allows the developer to submit packages to the Service Platform Catalogues and request packages and/or descriptors from the Service Platform Catalogues. This is a work in progress tool which might change its usage during development process.

The `son-access` tool receives the following arguments:

```
usage: son-access [-h]
                  [--auth URL] [-u USERNAME] [-p PASSWORD]
                  [--push TOKEN_PATH PACKAGE_PATH]
                  [--pull TOKEN_PATH PACKAGE_ID]
                  [--pull TOKEN_PATH DESCRIPTOR_ID]
                  [--debug]

  -h, --help             show this help message and exit
  --auth URL             requests an Access token to authenticate the user,
                         it requires platform url to login,
  -u USERNAME            username of the user,
  -p PASSWORD            password of the user
  --push TOKEN_PATH
```

```
       PACKAGE_PATH
--pull TOKEN_PATH
       PACKAGE_ID     requests a package or descriptor to the SP by its id,
       DESCRIPTOR_ID  requires path to the token file
--debug               increases logging level to debug
```

To authenticate a developer and receive an Access Token execute the following command:

```
son-access --auth <url> -u <username> -p <password>
```

Valid arguments for this command are:

- url: a string composed of IP address and port, e.g. 127.0.0.1:5001
- username: name for the user account to be authenticated
- password: password for the user account to be authenticated

To push a package execute the following command:

```
son-access --push /access/path /workspace/path
```

Valid arguments for this command are:

- /access/path: specific route to the file that contains the token
- /workspace/path: specific route to the package file to be submitted

To pull a package or a descriptor execute the following command:

```
son-access --pull /access/path <identifier>
```

Valid arguments for this command are:

- /access/path: specific route to the file that contains the token
- identifier: naming convention that identifies the resource, e.g. vendor.name.version

B Manual of son-emu-cli

The following sections describe the command line interface of `son-emu-cli`.

B.1 son-emu-cli compute

The compute client allows to control the instantiation of compute instances (containers) in an emulated datacenter (PoP).

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

son-emu compute

    Examples:
    - son-emu-cli compute start -d dc2 -n client -i sonatanfv/sonata-iperf3-vnf
    - son-emu-cli list
    - son-emu-cli compute status -d dc2 -n client


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

optional arguments:
  -h, --help            show this help message and exit
  --datacenter DATACENTER, -d DATACENTER
                        Datacenter to which the command should be applied.
  --name NAME, -n NAME  Name of compute instance e.g. 'vnf1'.
  --image IMAGE, -i IMAGE
                        Name of container image to be used e.g. 'ubuntu:trusty'
  --dcmd DOCKER_COMMAND, -c DOCKER_COMMAND
                        Startup command of the container e.g. './start.sh'
  --net NETWORK         Network properties of a compute instance e.g.
              '(id=input,ip=10.0.10.3/24),(id=output,ip=10.0.10.4/24)'
              for multiple interfaces.
  --endpoint ENDPOINT, -e ENDPOINT
                        UUID of the plugin to be manipulated.
```

B.2 son-emu-cli datacenter

The datacenter CLI can return and list information about the datacenters which are emulated in the current topology.

```
$ son-emu-cli datacenter -h
usage: son-emu-cli [-h] [--datacenter DATACENTER] [--endpoint ENDPOINT]
                   {list,status}

son-emu datacenter

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

optional arguments:
  -h, --help            show this help message and exit
  --datacenter DATACENTER, -d DATACENTER
                        Datacenter to which the command should be applied.
  --endpoint ENDPOINT, -e ENDPOINT
                        UUID of the plugin to be manipulated.
```

B.3 son-emu-cli network

The networking client allows to manipulate (install/remove) chaining rules of a running service.

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

son-emu network

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

optional arguments:
  -h, --help            show this help message and exit
  --datacenter DATACENTER, -d DATACENTER
                        Datacenter to in which the network action should be
                        initiated
  --source SOURCE, -src SOURCE
                        vnf name of the source of the chain
  --destination DESTINATION, -dst DESTINATION
                        vnf name of the destination of the chain
  --weight WEIGHT, -w WEIGHT
                        weight edge attribute to calculate the path
  --priority PRIORITY, -p PRIORITY
```

```
                        priority of flow rule
  --match MATCH, -m MATCH
                        string holding extra matches for the flow entries
  --bidirectional, -b   add/remove the flow entries from src to dst and back
  --cookie COOKIE, -c COOKIE
                        cookie for this flow, as easy to use identifier (eg.
                        per tenant/service)
  --endpoint ENDPOINT, -e ENDPOINT
                        UUID of the plugin to be manipulated.
```

B.4 son-emu-cli monitor

The monitoring CLI allows to setup monitoring metrics which are then received by the `son-monitor` tool.

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

son-emu monitor

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

optional arguments:
  -h, --help            show this help message and exit
  --vnf_name VNF_NAME, -vnf VNF_NAME
                        vnf name:interface to be monitored
  --metric METRIC, -m METRIC
                        tx_bytes, rx_bytes, tx_packets, rx_packets
  --cookie COOKIE, -c COOKIE
                        flow cookie to monitor
  --query QUERY, -q QUERY
                        Prometheus query
  --datacenter DATACENTER, -d DATACENTER
                        Datacenter where the vnf is deployed
  --endpoint ENDPOINT, -e ENDPOINT
                        UUID of the plugin to be manipulated.
```

C Manual of son-monitor

This appendix contains technical implementation details and manual how to use the different functionalities developed in `son-monitor`.

- **son-emu**: The extra implementation needed in son-emu to export monitored data.
- **SDK features** : How to start monitoring from the SDK when deploying a service on `son-emu`.

C.1 Ports used by son-monitor

The different processes and containers started to enable monitoring functionality in the SDK and son-emu also open up a number of ports:

Table C.1: Ports opened in the SDK by son-monitor

Port	Functionality
3000	Grafana web GUI and HTTP API
9090	Prometheus web GUI and HTTP API

Table C.2: Ports opened in son-emu by son-monitor

Port	Functionality
6653	Ryu OpenFlow controller port
8080	Ryu OpenFlow controller REST API
8081	cAdvisor metrics
9091	Prometheus Push Gateway

C.2 son-emu monitor features

At the start-up of `son-emu` a topology must be loaded. To enable all monitoring features, following settings should be followed:

```
def create_topology():
  # create topology
  net = DCNetwork(controller=RemoteController,monitor=True,enable_learning=True)
```

This enables:

- A remote controller (Ryu controller) to query via the OpenFlow protocol several network and flow metrics.
- The start-up of the cAdvisor and Prometheus Gateway Docker containers, to gather metrics.
- Enable learning capabilities for the deployed virtual switches in the emulated network topology of `son-emu` (needed to enable E-LAN networks)

C.3 SDK son-monitor features

The way to start any monitoring from the SDK has changed since previous deliverable D3.1. While it remains possible to use the son-monitor CLI, it is now recommended to use the MSD file as described in Section 3.6.2.

C.3.1 Automatic son-monitor initialization using the MSD file (recommended)

After a service has been deployed on the SDK emulator (son-emu), son-monitor can be used. Son-monitor uses the son-emu REST API and Prometheus framework.

`son-monitor msd -f file.yml` : this command reads an msd file, exports the requested metrics from son-emu and configures a Grafana dashboard. This is the recommended usage for son-monitor.

`son-monitor xterm [-n vnf_names]`: This command starts an xterm for all deployed docker VNFs in son-emu (if no names are specified, xterms for all vnfs are started)

This command sniffs the packets on a specified vnf interface (if no output file is specified, tcpdump is started in an xterm window)

```
son-monitor dump -vnf vnf_name:interface [-f filename.pcap]
son-monitor dump stop
```

C.3.2 Manual son-monitor initialization via the CLI

`son-monitor init` : this command starts the Grafana and Prometheus Database containers in the SDK.

`son-monitor init stop` : this command stops the Grafana and Prometheus Database containers in the SDK.

The commands executed in the MSD file can also be executed via the CLI:

Example1: Expose the tx_packets metric from son-emu network switch-port where vnf1 (default 1st interface) is connected. The metric is exposed to the Prometheus DB.

```
son-monitor son-monitor interface start -vnf vnf1 -me tx_packets
```

Example2: Install a flow_entry in son-emu, monitor the tx_bytes on that flow_entry. The metric is exposed to the Prometheus DB.

```
son-monitor flow_total start -src vnf1  -dst vnf2  \
-ma "dl_type=0x0800,nw_proto=17,udp_dst=5001"  -b -c 11 -me tx_bytes
```

Example3: Send a query to the Prometheus DB to retrieve the earlier exposed metrics, or default metric exposed by cAdvisor. The Prometheus query language can be used.

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

The `son-emu` REST API is addressed by son-monitor to export the requested metrics from a service deployed on `son-emu`. The son-monitor CLI that can be used to manually start monitoring actions from the SDK:

```
usage: son-monitor [-h] [--vnf_names [VNF_NAMES [VNF_NAMES ...]]] [--vim VIM]
                   [--vnf_name VNF_NAME] [--datacenter DATACENTER]
                   [--image IMAGE] [--dcmd DOCKER_COMMAND] [--net NETWORK]
                   [--query QUERY] [--input INPUT] [--output OUTPUT]
                   [--source SOURCE] [--destination DESTINATION]
                   [--weight WEIGHT] [--match MATCH] [--bidirectional]
                   [--priority PRIORITY] [--metric METRIC] [--cookie COOKIE]
                   [--file FILE]

                   {init,query,interface,flow_mon,flow_entry,flow_total,
                    msd,dump,xterm}
                   [{start,stop}]

    Install monitor features on or get monitor data from SONATA platform/emulator.

positional arguments:
  {init,query,interface,flow_mon,flow_entry,flow_total,msd,dump,xterm}
          Monitoring feature to be executed
              interface: export interface metric (tx/rx bytes/packets)
              flow_entry : (un)set the flow entry
              flow_mon : export flow_entry metric (tx/rx bytes/packets)
              flow_total : flow_entry + flow_mon
              init : start/stop the monitoring framework
              msd :  start/stop monitoring metrics from the msd
              (monitoring descriptor file)
              dump: start tcpdump for specified interface (save as .pcap)
              xterm: start an x-terminal for specific vnf(s)

  {start,stop}
         Action for interface, flow_mon, flow_entry, flow_total:
              start: install the flow entry and/or export the metric
              stop: delete the flow entry and/or stop exporting the metric
              Action for init:
              start: start the monitoring framework
              (cAdvisor, Prometheus DB + Pushgateway)
              stop: stop the monitoring framework
              Action for msd:
              start: start exporting the monitoring metrics from the msd
              stop: stop exporting the monitoring metrics from the msd


optional arguments:
  -h, --help            show this help message and exit
  --vnf_names [VNF_NAMES [VNF_NAMES ...]], -n [VNF_NAMES [VNF_NAMES ...]]
                        vnf names to open an xterm for
  --vim VIM, -v VIM     VIM where the command should be executed (emu/sp)
  --vnf_name VNF_NAME, -vnf VNF_NAME
                        vnf name:interface to be monitored
  --datacenter DATACENTER, -d DATACENTER
                        Datacenter where the vnf is deployed
  --query QUERY, -q QUERY
                        Prometheus query
  --source SOURCE, -src SOURCE
                        vnf name:interface of the source of the chain
  --destination DESTINATION, -dst DESTINATION
                        vnf name:interface of the destination of the chain
  --weight WEIGHT, -w WEIGHT
                        weight edge attribute to calculate the path
  --match MATCH, -ma MATCH
                        string to specify how to match the monitored flow
  --priority PRIORITY, -p PRIORITY
                        priority of the flow match entry, installed to get counter
                        metrics for the monitored flow
  --bidirectional, -b   add/remove the flow entries from src to dst and back
  --metric METRIC, -me METRIC
                        tx_bytes, rx_bytes, tx_packets, rx_packets
  --cookie COOKIE, -c COOKIE
                        integer value to identify this flow monitor rule
  --file FILE, -f FILE  service descriptor file describing
  monitoring rules or pcap dump file
```

C.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 Abbreviations

API	Application Programming Interface
CLI	Command Line Interface
CPU	Central Processing Unit
DB	Data Base
FSM	Function Specific Manager
GitHub	Git repository hosting service
IDS	Intrusion Detection System
JSON	JavaScript Object Notation
JWT	JSON Web Token
LAN	Local Area Network
MANO	Management and Orchestration
MSD	Monitor Service Descriptor
NSD	Network Service Descriptor
NSP	Network Service Profile
OVS	Open Virtual Switch
PED	Profile Experiment Descriptor
PoP	Point of Presence
REST	Representational State Transfer
SAP	Service Access Point
SDK	Software Development Kit
SP	Service Platform
SSM	Service Specific Manager
VLAN	Virtual LAN
VNF	Virtual Network Function
VNFD	VNF Descriptor
VNFP	VNF Profile
YAML	Yet Another Markup Language

E Bibliography

[1] Cisco. Snort Ids/Ips, 2016. Online at http://www.snort.org.

[2] Jupyter Community. Jupyter notebook. Website, Dec 2016. Online at http://jupyter.org/.

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

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

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

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

[7] SONATA consortium. D4.2: Service platform operational release and documentation. Website, December 2016.

[8] Docker resource allocation parameters that can be dynamically updated. Online at https://docs.docker.com/engine/reference/commandline/update/.

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

[10] The leading tool for querying and visualizing time series and metrics. Online at http://grafana.org/.

[11] OpenSim Ltd. Omnet++ Network Simulator. Website, 2016. Online at https://omnetpp.org.

[12] OSM. Google guice: Agile lightweight dependency injection framework, 2016. Online at https://osm.etsi.org/.

[13] Openflow based syntax for flow matching. Online at http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt.

[14] Manuel Peuster and Holger Karl. Understand Your Chains: Towards Performance Profile-based Network Service Management. In *5th European Workshop on Software Defined Networks (EWSDN'16)*. IEEE, 2016.

[15] Manuel Peuster, Holger Karl, and Steven van Rossem. MEdiCinE: Rapid Prototyping of Production-Ready Network Services in Multi-Pop Environments. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2016 IEEE Conference on*. IEEE, 2016.

[16] Http api for querying the prometheus database. Online at https://prometheus.io/docs/querying/api/.

[17] Steven Van Rossem, Wouter Tavernier, Manuel Peuster, Didier Colle, Mario Pickavet, and Piet Demeester. Monitoring and debugging using an Sdk for Nfv-powered telecom applications. In *IEEE NFV-SDN (NFVSDN2016)*. IEEE, 2016.