D4.1 First open-source release of the SDK toolset

<table>
<thead>
<tr>
<th>Project Acronym</th>
<th>5GTANGO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Title</td>
<td>5G Development and Validation Platform for Global Industry-Specific Network Services and Apps</td>
</tr>
<tr>
<td>Project Number</td>
<td>761493 (co-funded by the European Commission through Horizon 2020)</td>
</tr>
<tr>
<td>Instrument</td>
<td>Collaborative Innovation Action</td>
</tr>
<tr>
<td>Start Date</td>
<td>01/06/2017</td>
</tr>
<tr>
<td>Duration</td>
<td>30 months</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D4.1 First open-source release of the SDK toolset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workpackage</td>
<td>WP4</td>
</tr>
<tr>
<td>Due Date</td>
<td>M10</td>
</tr>
<tr>
<td>Submission Date</td>
<td>06/04/2018</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Status</td>
<td>To be approved by EC</td>
</tr>
<tr>
<td>Editor</td>
<td>Wouter Tavernier (IMEC)</td>
</tr>
<tr>
<td>Contributors</td>
<td>Wouter Tavernier, Askhat Nuriddinov, Thomas Soenen (IMEC), Manuel Peuster, Stefan Schneider (UPB), Panagiotis Stavrianos, Dimosthenis Kyriazis (UPRC), Peer Hasselmeyer (NEC), Anton Roman (QUO), Eleni Fotopoulou, Anastasios Zafeiropoulos, Thanos Xirofotos (UBI), Jose Bonnet (ALT)</td>
</tr>
<tr>
<td>Reviewer(s)</td>
<td>Jose Bonnet (ALT), Sonia Castro (ATOS)</td>
</tr>
</tbody>
</table>

**Keywords:**

| SDK |
Deliverable Type

<table>
<thead>
<tr>
<th>R</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEM</td>
<td>Demonstrator, pilot, prototype</td>
</tr>
<tr>
<td>DEC</td>
<td>Websites, patent filings, videos, etc.</td>
</tr>
<tr>
<td>OTHER</td>
<td></td>
</tr>
</tbody>
</table>

Dissemination Level

<table>
<thead>
<tr>
<th>PU</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including the Commission Services)</td>
</tr>
</tbody>
</table>

Disclaimer:

This document has been produced in the context of the 5GTANGO Project. The research leading to these results has received funding from the European Community’s 5G-PPP under grant agreement n° 761493.

All information in this document is provided “as is” and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors’ view.
Executive Summary:

The architecture of 5GTANGO’s Service Development Kit (SDK) follows the design principles from WP2, described in D2.2, and further refines them into a coherent end-to-end workflow. The SDK is one of the three core architectural components of 5GTANGO along with the Service Platform (SP), in charge of Management and Orchestration of the multi-PoP (Point of Presence) infrastructure, and the Validation & Verification (V&V) platform, in charge of harmonising extensive testing of services and components under a range of pre-defined scenarios and conditions. This deliverable documents the first release of the 5GTANGO SDK, which aims at providing the developer of software-based network services and virtual functions with a set of models, tools and processes to do so.

The SDK builds further on the groundbreaking work of SONATA, extending its strengths, modularity and applicability towards different target platforms, and increases its stability as well as its toolset. The entire service development workflow starts with the setup of a development workspace and project environment. The developer subsequently assembles descriptors for Virtual Network Functions (VNFs) and/or for a Network Service (NS). The latter interconnects VNFs and defines a Virtual Network Function Forwarding Graph (VNFFG) on top of it. Descriptors in 5GTANGO follow an ETSI-aligned data model (schema) encoded in YAML files. The process of writing descriptors for VNFs and services is usually tedious and error-prone, so 5GTANGO has developed a descriptor generation tool which is able to generate descriptors based on high-level objectives and service information using a web interface. VNF descriptors refer to container or Virtual Machine (VM) images required to actually execute desired middlebox functionality (e.g., NAT, firewall, DPI, etc.). The produced files can be manually fine-tuned and validated with respect to syntax, policy and particular semantic aspects such as loops in the VNFFG using a 5GTANGO-specific validation tool. Once all individual files and descriptors are available, the modular packaging component can produce service packages for the 5GTANGO platform or for example for the Open Source MANO platform (OSM) or any future-envisioned format. Produced packages can then be onboarded and deployed on a execution platform. This can be either a Virtual Infrastructure Manager under the control of the 5GTANGO SP, or the 5GTANGO emulator. The latter can be executed on a local developer PC. This drastically shortens the feedback loop for the developer to deploy and test a service in runtime context. The 5GTANGO emulator further evolves the SONATA emulator by extending it with VM support and a range of other features. The 5GTANGO traffic generator enables ad hoc generation of network traffic (control and/or data traffic) empowering the developer with tools to rapidly verify runtime behaviour under particular traffic conditions. QoS and testing functionality in future SDK releases will build further on this functionality in order to develop performance testing and QoS model building of such software-based network services. These models can serve the Verification & Validation platform extensively test, assess and complete these models under a range of (potentially) standardised test scenarios. The 5GTANGO platform is unique in its support for service-and function specific management functionality enabling developer-based code injection to modify the control of services and functions in a developer-defined manner. These components are envisioned to be extended to support re-usable process templates for state management of running services and components. In future SDK releases, Specific Manager (SM) models will be developed supporting re-usable scaling, migration and failure resiliency workflows.

All 5GTANGO SDK components are released as open-source, and are readily accessible from GitHub repositories. This deliverable provides information on the used APIs, interfaces and used repositories for external users and developers willing to test, contribute or further strengthen the functionality and stability of the 5GTANGO SDK.
Contents

List of Figures vii

List of Tables ix

1 Introduction 1
   1.1 Prototype Features Overview ................................ 1

2 SDK components 4
   2.1 Schema for Descriptors ..................................... 5
      2.1.1 Descriptor and schema types ............................. 6
      2.1.2 Relationship between descriptors ....................... 11
   2.2 Descriptor Generator ....................................... 11
      2.2.1 Future extensions .................................... 13
   2.3 Workspace and Project Management .......................... 14
      2.3.1 Workspace management ................................ 14
      2.3.2 Project management .................................. 14
   2.4 SDK Validation tool ......................................... 15
      2.4.1 Operation mode ...................................... 16
      2.4.2 Syntax checks ...................................... 16
      2.4.3 Integrity checks .................................... 16
      2.4.4 Topology ........................................ 17
      2.4.5 CLI ............................................ 17
      2.4.6 Service API ...................................... 19
      2.4.7 Using the validation tool from 5GTANGO Portal ........ 19
   2.5 Image Generator ........................................... 19
      2.5.1 Image generation functionality ......................... 22
   2.6 Packaging ................................................... 23
      2.6.1 Package Format Specification ........................... 23
      2.6.2 Packaging Tool ....................................... 26
   2.7 Access Tool ................................................ 30
      2.7.1 Functionality ....................................... 30
      2.7.2 Future Plans ....................................... 30
   2.8 Emulator .................................................... 31
      2.8.1 Integrating with Service Platform ....................... 31
      2.8.2 Local Testing Environment ............................. 32
      2.8.3 Adding VM Support to the Emulator .................... 32
      2.8.4 Testing MANO Systems Against the Emulator ........... 33
   2.9 Development and testing of Specific Management functionality .... 34
   2.10 Traffic generation tool .................................... 37
      2.10.1 CLI commands ..................................... 37
      2.10.2 Example of use .................................... 38
      2.10.3 REST API definition ................................. 38
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Sections</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.11</td>
<td>QoS and Test Model</td>
<td>38</td>
</tr>
<tr>
<td>2.12</td>
<td>State Management Capabilities</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Interfaces</td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>SDK internal interfaces</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>SDK external interfaces</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>Source Code</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>Conclusion</td>
<td>47</td>
</tr>
<tr>
<td>A</td>
<td>Bibliography</td>
<td>48</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Overview of main components of the 5GTANGO v0 release .......................... 3
2.1 SDK workflow ...................................................................................... 5
2.2 Overview of available descriptor types and their relationship ...................... 11
2.3 Descriptor generator GUI: Wizard for providing high-level information and generating descriptors ............................................................... 12
2.4 Descriptor generator GUI: Generated descriptors ready for download ............ 13
2.5 tng-sdk-validate reference diagram .......................................................... 16
2.6 Image conversion process ......................................................................... 20
2.7 Container-based image instantiation vs. VM-based image instantiation .......... 22
2.8 Internal architecture of tng-sdk-package component ....................................... 27
2.9 Automatically generated OpenAPI documentation of tng-sdk-package’s REST API endpoints .............................................................................. 29
2.10 SDK access tool process ........................................................................... 30
2.11 Component diagram showing the newly developed VimEmuWrapper that allows the service platform’s infrastructure adaptor to connect to the northbound REST interface of the emulation platform. ................................................................. 32
2.12 Architecture of the extended Containernet project (Containernet 2.0) that adds support for hybrid emulations with Docker containers and VMs to 5GTANGO’s emulation platform. ......................................................... 33
2.13 Initial results of an experiment that tests the behavior of OSM when connected to large-scale, emulated multi-PoP topologies ........................................ 34
2.14 Testing process of specific management functionality .................................... 36
2.15 tng-sdk-traffic general architecture diagram ................................................. 37
2.16 VNF Resource Efficiency [28] ................................................................... 39
2.17 Elasticity Profile [28] ................................................................................. 40
2.18 Profiling Toolkit within 5GTANGO ............................................................. 41
2.19 Indicative visualisation .............................................................................. 41
3.1 SDK interface overview ............................................................................. 45
List of Tables

2.1  SDK component overview ................................................................. 5
2.2  tng-sdk-validate REST endpoint overview ...................................... 19
2.3  tng-sdk-package REST endpoint overview ...................................... 29
2.4  Traffic generator REST API .............................................................. 38
4.1  SDK code repository overview .......................................................... 46
1 Introduction

The core mission of 5GTANGO is to enhance the programmability of 5G networks through three main parts: i) an Software Development Kit (SDK), ii) a Service Platform, and iii) a Validation and Verification Store. An adequate synergy between these three aims for enabling rapid design, implementation, testing and deployment of novel networked services following a DevOps workflow. The SDK plays a crucial role in this workflow, as all development starts at the SDK.

Deliverable D2.2 [5] provided some first views and insights on the high-level processes and components of the SDK. This document details the designs and first release implementation of the main 5GTANGO SDK components, refining, correcting and further improving these designs into concrete tools. After the first project year, we are proud to already fully support an end-to-end workflow using novel 5GTANGO components, building further on the foundations of SONATA project. The resulting release v0 of the SDK is fully aligned with the released functionality in the other WPs, namely the V&V and SP functionality. A project wide overview of release v0 is therefore given in Sec. 1.1.

The philosophy of the 5GTANGO SDK is to provide a useful blend of independent, light-weight tools which could be used independently on their own, as well as in synergy with each other, assisting the global goal of service development for the 5GTANGO Service Platform. The SDK aims for easing the life of the developer of network functions, services, descriptors and tests in an easy-to-understand, quick and robust manner. Although the first priority of these tools is to assist in development targeting 5GTANGO SP, it is a strong objective of this work to remain as extensible as possible, and maximally compatible with other platforms, such as OSM. We are convinced that this further strengthens the quality and sustainability of the developed tools. Although some components are inherited from SONATA, many components are novel, and existing components are drastically redesigned, streamlined, and customised focusing on additional support for testing, validation and verification.

The rest of the document is structured as follows. Sec. 2 provides the core of the document, by providing an overview of the SDK, its workflow and processes, followed by an in-depth documentation of the individual components. Sec. 3 details the interfaces of the SDK towards other 5GTANGO parts, as well as the interfaces used between the individual SDK components. Sec. 4 provides pointers to the developed software code and its usage, while final Sec. 5 provides concluding thoughts and pointers for future work in the rest of the project.

1.1 Prototype Features Overview

This sub-section lists the features that the first version of the 5GTANGO platform (including the service development kit, the service platform, the V &V component) will support.

These features are:

1. Developers may use the SDK tools to build a package containing a Network Service (NS), VNFs that implement it and Tests to test that NS or VNF;

2. Developers may upload packages;
3. Developers may query the status of the package un-packaging and validation process;

4. Packages are un-packaged and their content (including the file containing the package) is stored in the Catalogue;

5. The package’s developer is notified of the successful package un-packaging and validation process;

6. The V&V is notified about the existence of a new package;

7. Tests and Network Services are automatically associated through a tagging mechanism built into their descriptors;

8. The V&V automatically selects Network Services to be tested:
   1. Network Service instantiation is requested to the Service Platform;
   2. the Network Service instance data is read from the Repositories;
   3. the Network Service instance is used in a pre-defined way;
   4. the results of the normal execution of the Network Service instance are collected;
   5. the Network Service instance is terminated;

9. The following features are ready and available via APIs, although not yet via the Portal (see Fig. 1.1):
   1. Network Slices comprising a number of Network Services can be defined;
   2. Policies can be defined;
   3. SLAs can be defined;

Details on all these features can be seen in the deliverables D3.2 [11], D4.1 [12] and D5.1 [13]. The high level architecture supporting these features is shown in Fig. 1.1.
Figure 1.1: Overview of main components of the 5GTANGO v0 release
2 SDK components

The 5GTANGO SDK is composed of several components, each one with its own specific purpose, usable in a largely independent manner. The global workflow, as described in this section, ensures that they are complementary and add value to the global development and testing duties of the service or VNF developer. The components and functionality were described from a high level point of view in [5], and are harmonised and restructured slightly to ease the manner in which they can be presented.

The functionality of the SDK can be roughly bundled in the following parts. Project environment tools are responsible for setting up the local developer workspace environment on the developer’s computer, for example preparing the file system and folder structure to store environment information, authorization tokens, etc. In addition to the workspace, individual developer projects can be created as well to store all development artifacts related to a project in its own folder structure. Developer schema are used to formalise the structural requirements of involved 5GTANGO artifacts. In many cases descriptors are required, for example following a YAML format. These schemas will detail the data model and required format to specify them. The development tools are tools which actually assist the programmer in creating images, descriptors for services or VNFs and specific managers for controlling services. These tools usually result into additional files within the prepared project folder structure. Packaging and access tools enable to combine all single project artifacts into a single package or archive which can be uploaded to execution environments such as the emulator, the V&V store, or the Service Platform. Validation tools are used to execute a number of formal checks on produced packages without actually deploying them. This usually involves syntactical checking of the descriptor files, as well as executing checksums to ensure that no low-level bit errors have occurred. Emulation tools enable to deploy service packages on a locally emulated environment, i.e. the developer’s computer. Emulation tools enable to quickly run services, without requiring to go through several external systems. Finally, debugging tools assist the developer to inspect state, conditions or bugs in deployed services, presumably the emulator (although other execution environments might be targeted as well).

Tbl. 2.1 gives an overview of currently developed SDK components in the context of this categorisation. Fig. 2.1 illustrates how these different tools together form a streamlined development workflow. Each of the main steps in this workflow is supported by a set of SDK tools and components. Those components indicated with a black background have been partly implemented, while those with a white background have been discussed in this deliverable, but are planned for a future SDK release. One could see the depicted workflow as the typical canonical way of working. However, multiple iterations could happen and each of the developed tools is built as such that they can be used in an independent manner.
2.1 Schema for Descriptors

In 5GTANGO, there are various types of descriptors ranging from specifications of VNFs and network services to SLA and slice descriptors. These descriptors provide important information to the 5GTANGO components, e.g., the service platform or the V&V. Deliverable D2.2 [14] provides an introduction and overview of 5GTANGO’s function and service descriptors.

All descriptors are specified as structured YAML files, which are both human- and machine-readable. The structure of each descriptor type is defined by a schema, specifying mandatory and optional fields as well as the format of the YAML file. Existing descriptors can easily be validated against the schemas to make sure their structure is correct. The corresponding GitHub repository [2] contains schemas for each descriptor type along with example descriptors. Furthermore, a validation tool enables quickly checking the validity of new descriptors.

Table 2.1: SDK component overview

<table>
<thead>
<tr>
<th>SDK tool</th>
<th>Category</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema</td>
<td>developer schema</td>
<td>5GTANGO descriptor, record, and package specifications and schemas (data models)</td>
</tr>
<tr>
<td>project creator</td>
<td>project environment tools</td>
<td>tool to manage network service projects</td>
</tr>
<tr>
<td>descriptor generator</td>
<td>development tools</td>
<td>tool to rapidly create descriptors based on high-level objectives</td>
</tr>
<tr>
<td>image converter</td>
<td>development tools</td>
<td>tool to rapidly create VIM-dependent images</td>
</tr>
<tr>
<td>sm tester</td>
<td>development tools</td>
<td>tool to rapidly test/generate specific managers (e.g., SSM, FSM)</td>
</tr>
<tr>
<td>packager</td>
<td>packaging &amp; access tools</td>
<td>tool to create package from project folder</td>
</tr>
<tr>
<td>uploader</td>
<td>packaging &amp; access tools</td>
<td>tool to upload package to execution environment or to access catalogues</td>
</tr>
<tr>
<td>validator</td>
<td>validation tools</td>
<td>descriptor validation tool</td>
</tr>
<tr>
<td>VIM emulator</td>
<td>emulator tools</td>
<td>tool to deploy services locally on a Mininet-based environment</td>
</tr>
<tr>
<td>traffic generator</td>
<td>debugging tools</td>
<td>tool to easily generate custom traffic for testing purposes</td>
</tr>
</tbody>
</table>
2.1.1 Descriptor and schema types

In addition to the different descriptor-specific information, all descriptors contain fields specifying
the name, author, vendor, and version of the descriptor, which can be used for identification.
Moreover, each descriptor contains a link to the corresponding schema to allow easy validation and
management of different versions of schemas.

---

desciption: "https://raw.githubusercontent.com/sonata-nfv/tng-schema/master\n/function-descriptor/vnfd-schema.yml"

vendor: "eu.5gtango"
name: "ubuntu-vnf"
version: "0.9"
author: "eu.5gtango"
description: "A default ubuntu VNF descriptor"

In the following, we provide a brief overview of the different schema and descriptor types and
their most important fields:

2.1.1.1 Virtual Network Function descriptor (VNFD)

A VNFD specifies the involved virtual deployment units (VDUs), e.g., an Ubuntu image with
associated resource requirements. It also defines how these VDUs are interconnected and what
connection points the VNF exposes to the outside world.

```
virtual_deployment_units:
- id: "vdu01"
  vm_image: "ubuntu16-04.qcow2"
  resource_requirements: ...
# connection points of the VDU inside the VNF
connection_points:
- id: "mgmt"
  interface: "ipv4"
  type: "internal"
- id: "input"
  interface: "ipv4"
  type: "internal"
- id: "output"
  interface: "ipv4"
  type: "internal"

# VNF connection points to the outside world (similar to VDU conn. points)
connection_points: ...

# virtual links connecting the VDUs’ connection points with each other
# and with the VNF’s connection points
virtual_links:
- id: "mgmt"
  connectivity_type: "E-LAN"
```
connection_points_reference:
  - "vdu01:mgmt"
  - "mgmt"

dhcp: True
-id: "input"
connectivity_type: "E-Line"
connection_points_reference:
  - "vdu01:input"
  - "input"

dhcp: True
-id: "output"
connectivity_type: "E-Line"
connection_points_reference:
  - "vdu01:output"
  - "output"

dhcp: True

2.1.1.2 Network Service descriptor (NSD)

NSDs specify the involved VNFs inside the service and define how the VNFs are connected (in a forwarding graph). In 5GTANGO, network services may also be recursive and contain other network services.

# involved network functions

network_functions:
  - vnf_id: "vnf1"
    vnf_vendor: "eu.5gtango"
    vnf_name: "ubuntu-vnf"
    vnf_version: "0.9"

# NS connection points to the outside world (similar to VNFD)

connection_points: ...

# virtual links connect the VNFs’ and NS’ connection points (similar to VNFD)

virtual_links: ...

forwarding_graphs:
  - fg_id: "fg01"
    number_of_endpoints: 2
    number_of_virtual_links: 2
    constituent_virtual_links:
      - "input-2-vnf1"
      - "vnf1-2-output"
    constituent_vnfs:
      - "vnf1"
    network_forwarding_paths:
      - fp_id: "fg01:fp01"
        policy: "none"
        connection_points:
2.1.3 Package descriptor

As further described in Sec. 2.6, 5GTANGO’s packages can contain files of different flavors (e.g., belonging to both OSM and 5GTANGO) through layering. The NFV Advanced Package Descriptor (NAPD) describes which files are part of the package and to which flavor they belong.

```
package_content:
- source: "Definitions/mynsd.yaml"
  algorithm: "SHA-256"
  hash: "d0e782829355a07c2ddcaaa765c80b507e60e6167067c950dc2e6b0da0dbd8b"
  content-type: "application/vnd.5gtango.nsd"  # MIME type of file
- source: "scripts/scale/scale.sh"
  algorithm: "SHA-256"
  hash: "e0e782829355a07c2ddcaaa765c80b507e60e6167067c950dc2e6b0da0dbd8b"
  content-type: "text/x-shellscript"  # MIME type of file
  tags:  # (optional) list
    - "eu.5gtango"
    - "etsi.osm"
```

2.1.4 SLA template descriptor

An SLA template descriptor incorporates metrics such as specific objectives or quality attributes, parameters as well as expressions (i.e., rules) between parameters. The proposed YAML schema, based on ISO/IEC DIS 19086-2, aims to specify the main building blocks of an SLA template and also presents an expression (i.e., function) that allows any service provider to specify any metric included in a template (e.g., availability, response time, etc.).

As SLA templates are tailored to a service provider, they reference network services inside the service provider’s catalogue using its UUID (rather than vendor/name/version). This allows to define separate SLA template descriptors for the same network service being used by different service providers, i.e., stored in their individual catalogues.

```
# involved network service
ns:
  uuid: ns001
  description: a network service description

# high level SLOs
objectives:
- slo_id: req001
  slo_name: NetworkServiceAvailability
  slo_definition: Availability is the percentage of time, in a specific time
interval, during which a NS can be used for the purpose that it was originally designed and built for.

slo_unit: "\%"
slo_value: '99.95'

# corresponds to how to measure the objective metric:
  - metric_id: mtr001
    metric_definition: Total time the NS is alive

rate:
  parameterWindow: 1 year
  sampleInterval: sample rate every 1 hour

# a function under which the specific metric of the SLA should obey expression:
  expression_statement: prmtr001/prmtr002
  expression_language: ISO80000
  expression_unit: hour

# expressing in detail each parameter inside the expression parameters: ...

2.1.1.5 Policy descriptor

A Network Service Runtime Policy Descriptor (RPD) is a deployment template attached optionally to a network service referencing a set of enforcement rules, upon which certain actions are taken in order to meet some objectives described by specific SLAs. In 5GTANGO, a policy descriptor is attached to a specific network service. A network service may have more than one policy, but only one is active during instantiation. The basic ingredients of a policy descriptor is a set of policy rules that contain conditions and actions. Conditions may be recursive, i.e., contain other sub-conditions.

collectiveRules:
  - name: "highTranscodingRateRule"
    salience: 1
    inertia:
      value: 30
      duration_unit: "m"
    duration:
      value: 10
      duration_unit: "m"
    aggregation : "avg"
    conditions:
      condition: AND
    rules:
      - id: vnf1.CPULoad
        field: vnf1.CPULoad
        type: double
input: number
operator: greater
value: '70'

actions:
- action_type: "Infrastructure"
  name: "ApplyFlavour"
  value: "3"
  target: "vnf1"

2.1.1.6 Slice descriptor

A slice descriptor contains references to all network services belonging to a slice.

# involved network services
network_services:
- ns_id: "ns_sonata"
  ns_vendor: "eu.5gtango.slice-descriptor"
  ns_name: "5gtango-demo"
  ns_version: "0.1"

# similar to connection points in NSD and VNFD
service_interface_points: ...

qos:
  5qi_value: "6"

2.1.1.7 Test descriptor

A test descriptor defines the type and configuration parameters of a test as well as testing tags, which are used for test execution.

# test_configuration
test_type: TTCN3
test_execution_files: "path/to/binaries"
test_configuration_parameters:
- parameter_name: "TTCN3_config_file"
  parameter_definition: "location of the TTCN3 configuration file"
  parameter_value: "simple_VNF_test.ttcn3"
  content_type: "text/plain"
test_category:
- category_type: performance

# test tags for execution
test_execution:
- test_tag: moongen
  tag_id: tag_001
- test_tag: io_bandwidth
  tag_id: tag_002
2.1.2 Relationship between descriptors

Fig. 2.2 provides an overview of the available descriptors and illustrates their relationship. A package contains all and any relevant files and can thus reference any of the other descriptors. A network slice references included network services and a network service references included network functions. Finally, test descriptors may refer to services to be tested.

2.2 Descriptor Generator

As mentioned in Sec. 2.1, 5GTANGO uses a variety of descriptors to store important information about services, packages, slices, etc. in a structured way. Containing lots of details, descriptors are often quite complex and difficult to write manually and/or from scratch. Usually, different descriptors of the same type have many similarities, e.g., most VNFs have the same three connection points (input, output and management). However, simply copying and pasting from existing descriptors can easily lead to errors if not all fields are adjusted correctly.

To alleviate the process of creating correct descriptors, 5GTANGO introduces a new descriptor generator (also see D2.2 [14]). The descriptor generator takes high-level information about VNFs and NSs and generates corresponding VNFDs and NSDs. In doing so, users can provide any information in a simple, graphical user interface, upload existing VNFDs to be reused, or use the provided, sensible default values (Fig. 2.3).

Overall, the descriptor generator greatly relieves developers from performing repetitive, error-prone tasks and thus accelerates and simplifies the process of creating descriptors. To create a valid sample descriptor, only a single click is necessary. Clearly, the generated descriptors can still be edited and adjusted, e.g., to handle more complex situations. In fact, the web interface of the
Figure 2.3: Descriptor generator GUI: Wizard for providing high-level information and generating descriptors
The descriptor generator is written in JavaScript and HTML and can completely run on client side or inside a docker container with minimal installation effort. The GitHub repository always provides up-to-date details about installation, usage, and CI/CD [1]. When generating NSDs and VNFDs with the descriptor generator, it assumes that all VNFs are connected in a linear chain and have three connection points (input, output, management). Typical services do fulfill this assumption. For the generation process, the tool uses “default descriptors” (e.g., VNFs with a single Ubuntu VDU) that are adjusted and extended as required. These default descriptors are maintained in the tng-schema repository and are fetched directly from there to ensure consistency with the latest schemas.

### 2.2.1 Future extensions

Currently, the descriptor generator focuses on network service descriptors (NSDs) and virtual network function descriptors (VNFDs) as these can be quite complex and are mandatory for deploying services in 5GTANGO. In future extensions, the descriptor generator will support further descriptor types to enable the quick and easy generation of descriptors listed in Sec. 2.1. We also plan to integrate functionality to automatically validate generated descriptors against their schemas before downloading them, which is useful if the generated are modified online before the download. The descriptor generator should also be integrated with the packaging and uploading tools to provide a seamless experience for developers using the SDK. Ideally, new descriptors can then be generated, validated, packaged, and uploaded with just a few clicks.

We also plan a command-line version of the descriptor generator that may help advanced users generate descriptors more quickly without the graphical interface. Instead, high-level information may be specified in form of YAML configuration files or passed to the generator directly.

More functionality and options such as supporting different kinds of VNFs (e.g., CPU- vs. memory-bound), different forwarding graphs (not just linear chains), etc. will also be considered in the future. However, more features and flexibility also may increase complexity. Thus, we will carefully consider this trade off to ensure the descriptor generator remains simple to use.
2.3 Workspace and Project Management

Workspace and project management are tightly coupled as the configuration files in the workspace are used by the projects. They are split into different Python modules and can be used with the commands `tng-workspace` and `tng-project`, respectively.

2.3.1 Workspace management

The SDK workspace and project management in 5GTANGO is built upon SONATA’s `son-work-space` tool (see SONATA’s deliverable D3.3 [30]) but further evolved for an even simpler and more flexible workflow.

After creating descriptors for all involved VNFs and NSs (see Sec. 2.1 and Sec. 2.2), developers can start creating a project with their descriptors and associated files. Configuration files, e.g., required for project creation are stored in the 5GTANGO workspace. If not available, a workspace can easily be created using the `tng-workspace` tool.

When creating a new workspace with the `tng-workspace` command, a file structure and configuration files are generated automatically (usually at `~/.tng-workspace`). This workspace is then used for all projects unless specified otherwise. The typical file structure of the workspace is the following:

```
+- .tng-project
 | +- catalogues
 | +- configuration
 | +- platforms
 | +- projects
 |   +- config.yml
 | +- workspace.yml
```

The file `workspace.yml` holds metadata of the workspace as well as links to the tng-schema repository and access credentials for the service platforms. Further configuration files should be added in the corresponding directories. As an example, `config.yml` inside the `projects` folder is used when adding files to a project with `tng-project --add` for mapping yml-files to the correct MIME type.

2.3.2 Project management

Once the workspace is created, new projects can be created with the `tng-project` command. While projects had a strict structure in SONATA, 5GTANGO’s projects are more flexible. 5GTANGO projects still provide a recommended structure with sample files but also allow adding any files anywhere within the project in order to be compatible with other platforms (not just 5GTANGO). The typical project structure is the following:

```
+- projectname
 | +- dependencies
 | +- deployment
 | +- sources
 |   +- nsd
 | +- vnfd
 | +- project.yml
```
Similar to git, developers can simply add files to the project with `tng-project --add`, where the file type is detected automatically and wildcards are allowed. Removing files works similarly with `tng-project --remove`.

The project.yml is automatically generated when creating a new project with the `tng-project` command, e.g., with `tng-project -p p1` (see the generated `project.yml` below). It includes metadata about the package (name, vendor, version, maintainer, description), a version field, and most importantly a list of all files belonging to the project.

```yaml
descriptor_extension: yml
files:
- path: p1\sources\vnfd\vnfd-sample.yml
tags:
  - eu.5gtango
type: application/vnd.5gtango.vnfd
- path: p1\sources\nsd\nsd-sample.yml
tags:
  - eu.5gtango
type: application/vnd.5gtango.nsd
package:
  description: Some description about this sample
  maintainer: Name, Company, Contact
  name: 5gtango-project-sample
  vendor: eu.5gtango
  version: '0.1'
version: '0.5'
```

The file list contains the file path, the MIME type, and tags for each file. Tags are optional and specify the target platform (or platforms) belonging to each file. Hence, `tng-project` can be used to create and organize projects for 5GTANGO but also for other platforms like OSM. When creating a package, `tng-package` uses the tags for creating different layers corresponding to the different target platforms. For details about layering, see the package specification [7].

Finally, when all files are added to the project, it is ready to be packaged and uploaded to the service platform. For always up-to-date information and details refer to the GitHub repository [3].

### 2.4 SDK Validation tool

This tool was already existing in Sonata SDK tool kit, however in 5GTANGO the validation tool has been extracted from the `son-c1i` tool to be developed as a single application. This decision was made to make easier all the repository management tasks and to keep it working as an individual tool which can be used separately from the rest of the SDK.

The validation of the structure of the packages which was included in the in the original Sonata validation tool has been extracted from it and included in the `tng-sdk-package` for the sake of coherence and to make its maintenance easier.

This tool enables four types of validations:

- **Structure**: in the current version the only structure validation done is over the SDK folder itself. The tool checks that all the relevant files and folder structure is

- **Syntax**: just a syntactic check based on the defined in the `tng-schema` repository.
- **Integrity**: it verifies that all the elements the Descriptor contains are valid and all the required elements to setup the topology are present.

- **Topology**: validates the network topology logic and tries to find design defects.

In Sonata all the logic behind the integrity and topology were embedded in the validator code. In 5GTANGO this functionality will be progressively extracted from it and added as rules written in a Domain-specific language so that it is easier to add new rules and checks.

### 2.4.1 Operation mode

The `tng-sdk-validate` has two operation modes as the original tool in Sonata. It can be executed from the console using a CLI but it also can be sued as a service through a REST API. The service API in turn can be executed in two different modes which will be described later.

### 2.4.2 Syntax checks

The Service and Function descriptors are syntactically validated against the schema templates, available at the tng-schema repository. This functionality has been directly ported from Sonata so it was just adapted to 5GTANGO. The tool gets an updated version of the Schemas and check if the Descriptor is compliant with it.

### 2.4.3 Integrity checks

Integrity checks are applied both to Service and Function Descriptors.

#### 2.4.3.1 Service integrity

Service descriptors typically contain references to multiple VNFs, which are identified by a composition of the vendor, name and version of the VNFs. 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.

2.4.3.2 Function integrity

Similarly to service descriptors, VNFs may also contain multiple sub-components, 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.

2.4.4 Topology

The tng-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 tng-validate tool comprises the following validation mechanisms.

- **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.
- **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. The tng-validate tool analyses the network graph and returns a list of existing cycles to help the developer in the topology.
- **node bottlenecks** - checks for possible network congestion in the links. It takes into account the bandwidth specified for the interfaces, the weights which are assigned to the edges of the network graph in order to assess possible bottlenecks in the path.

2.4.5 CLI

The allows to interact with the validation tool from a console.

The output of the help option of the tng-sdk-validate application (included below) shows a comprehensive reference of the CLI syntax which hardly differs from original son-validate command. It also includes examples of use.

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

5GTANGO SDK validator
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.

--project PROJECT_PATH
  Validate the service of the specified SDK project. The project directory must be specified using the argument '--workspace' otherwise $HOME/.tng-workspace is assumed

--package PACKAGE_FILE
  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' for integrity and topology checks.

--function VNFD
  Validate the specified function descriptor. If a directory is specified, it will search for descriptor files with extension defined in '--dext'. It is possible to validate multiple functions contained in the same directory.

--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' and '--service' options.

--syntax, -s
  Perform a syntax validation.

--integrity, -i
  Perform an integrity validation.

--topology, -t
  Perform a network topology validation.

--debug
  Sets verbosity level to debug

Example usage:

tng-sdk-validate --project /home/sonata/projects/project_X
                     --workspace /home/sonata/.son-workspace
  tng-sdk-validate --service ./nsd_file.yml --path ./vnfds/ --dext yml
  tng-sdk-validate --function ./vnfd_file.yml
  tng-sdk-validate --function ./vnfds/ --dext yml

It is important to note that the options only can be used in the following combinations since most complex validation checks always requires to pass more basic levels of validation:

- only syntax: -s,
- syntax and integrity: -si,
- syntax, integrity and topology: -sit

2.4.5.1 Examples of use

Relevant examples of use of the tng-sdk-validate command are included below:

# this validates the project indicated in the path. By default it performs a complete validation of syntax, integrity and topology.
$ tng-sdk-validate --project /home/sonata/projects/project_X \
   --workspace home/sonata/.son-workspace

# performs a complete validation of the service defined in the specified file.
$ tng-sdk-validate --service ./nsd_file.yml --path ./vnfds/ --dext yml

# performs a complete validation of the function defined in the specified file.
$ tng-sdk-validate --function ./vnfd_file.yml

# performs a complete validation of the function defined in all the files in the
# specified folder.
$ tng-sdk-validate --function ./vnfds/ --dext yml

2.4.6 Service API

Additionally to the CLI which enables the use of tng-sdk-validate from a console terminal, it also
exposes a REST API so that the validation can be used as a service by other tools.

It has to modes of operation: - *stateless*: in this mode the tool works as a service which can be
used by other applications. - *local*: this mode is intended to be use by developers to validate the
file descriptors as they are created/modified.

2.4.6.1 API overview

tbl. 2.2 gives an overview of the endpoints exposed by tng-sdk-validate.

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
<th>Endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>validate SDK project, service or function</td>
<td>POST</td>
<td>/api/val/v1/validate/{object_type}</td>
</tr>
<tr>
<td>retrieve list of validated objects</td>
<td>GET</td>
<td>/api/val/v1/report</td>
</tr>
<tr>
<td>retrieve validation report for specific object</td>
<td>GET</td>
<td>/api/val/v1/report{result}/{resource_uuid}</td>
</tr>
<tr>
<td>retrieve validated topology graph</td>
<td>GET</td>
<td>/api/val/v1/report{topology}/{resource_uuid}</td>
</tr>
<tr>
<td>retrieve validated forwarding graph structure</td>
<td>GET</td>
<td>/api/val/v1/fwgraphs/{resource_uuid}</td>
</tr>
</tbody>
</table>

In order to access to the last version of the documentation of both the CLI and the REST API
refer to the tng-sdk-validate repository [4].

2.4.7 Using the validation tool from 5GTANGO Portal

The validation tool in Sonata had a specific web-based GUI. This functionality is going to be
integrated in the 5GTANGO Portal so the son-validate GUI from Sonata is not going to be
ported to 5GTANGO.

The Portal will be able to be used only as a front-end for tng-sdk-validate service by config-
uring the validation screen as the only one available so there is no need to maintain the GUI as a
single entity.

2.5 Image Generator

The core functionality of Network Functions is stored in the images referred in their corresponding
descriptors. The SONATA platform is currently heavily focused on the OpenStack Virtual Infra-
structure Management system which on its turn relies on QEMU machine emulation. As a result,
the images of VNFs to be deployed via the SP need to be provided in a compatible format such as `qcow2` or `raw`.

Until this SDK release, the main deployment technology used in the 5GTANGO VIM emulator Sec. 2.8 are Docker containers. The disk images corresponding to these are Docker images. As a result, VNFs which are tested in the emulator based on Docker images cannot directly be reused in the VIM supported by the SP.

In order to make the development and deploying of VNFs easier and less error-prone 5GTANGO introduces VNF image management tool `tng-sdk-img`. In the first phase of the project, the main focus of this tool will be conversion Docker containers to QEMU-based images. This enables developers to quickly deploy Docker-based VNFs not only on the SDK emulator, but also on the SP.

According to 5GTANGO Schema VNFs are distributed as packages compatible to the ETSI VNF package format which is based on the TOSCA CSAR standard. The package contains a descriptor which specifies the involved virtual deployment units (VDUs) with associated resource requirements, how these VDUs are interconnected and what connection points the VNF exposes to the outside world. The VDU is defined by it’s name, image and image format. The conversion tool reads this information and creates a virtual machine for each VDU.

The principle of the image conversion process is shown in figure Fig. 2.6. The virtual machine image is created on the basis of Ubuntu Xenial Cloud Image by default, but any different base image can be specified as a CLI argument if needed. The base image must have `cloud-init` installed since further provisioning is to be done with this utility.

The usage of the tool is as follows:

```
$ tng-sdk-img convert VNFD [BASE_IMAGE]
```

The conversion tool performs different checks. First, the conversion tool checks whether the image has the Docker format. Second, the presence of a specified Docker image in the Docker Hub is checked. If the result of checking is positive, the conversion tool creates and provisions a virtual machine image for this particular virtual deployment unit.

During provision the Docker Engine is installed on this virtual machine. Next, appropriate Docker image is pulled from Dockerhub. In order to upload local Docker images to Dockerhub,
the instructions from Docker Documentation can be used. The conversion functionality configures
the system to run docker container after boot automatically. The Docker container is run with
host network to have direct access to all network interfaces. The network cost (i.e., reduction in
throughput and increase in latency) of this setup will be evaluated for different VNFs at a later
stage.

The number of network interface cards (NICs) that a VNF requires in order to be adequately
connected to the other VNFs in the considered network service, is defined in the VNFD. The
considered VIM and subsequently the used hypervisor must instantiate the correct number NICs
to correctly instantiate the VNF in the infrastructure. It is therefore the role of the SP to propagate
this information. Although the produced VNF image does not directly determine the number of
required or expected interfaces, its stored configuration on the virtualised disk must do so (e.g.,
the `/etc/network/interface` file). Even if the machine corresponding to the considered image
is booted, exposed interfaces by the hypervisor might not be activated by default. Therefore, in
order to support multiple interfaces, `netplugd` is installed and configured within the image. This
is a daemon which responds to network link events from the Linux kernel and configures network
interfaces. When a carrier signal is detected on an interface, it runs a script to bring the interface
up.

The disk space overhead induced by the conversion process is about 1 GB on top of the used
Docker image (evenly split between the overhead of the base Ubuntu Xenial Cloud Image and the
Docker Engine).

In Fig. 2.7 we illustrate the usage of the tool for a Snort VNF with 2 NICs. This VNF is part
of a network service consisting of 2 symmetric VNFs, each with 2 NICs, and 1 service entry and 1
service output point, as indicated in the figure below. Each of the VNFs is currently assumed to be
available as Docker container image. In order to convert the image of Snort VNF (and analogously
for the other VNFs), we can use the following command:

```
$ tng-sdk-img convert snort-vnf.yml
Checking requirements
OK
Downloading base image
OK
Starting conversion
Processing vdu01
Creating seed
OK
Creating VDU
OK
Deleting seed
OK
Deleting base image
OK
Conversion finished
$ ls
snort-vnfd.yml  snort-vnf_vdu01.qcow2
```

Fig. 2.7 illustrates the difference of instantiating this network service in the SDK emulator using
the two Docker containers vs. the instantiation of the 2 VNFs on a single virtualized server using
2 VMs on deployed physical server in the infrastructure controlled by the SP.
Note that it is the responsibility of the configuration platform, for example the Mininet or Emulator API or of the hypervisor (e.g., Xen) to create virtual interfaces, links and associated switch configuration. The main difference is that in the VM-based instantiation the entire machine architecture is emulated, while only isolated, dedicated namespaces are used in the Docker-based instantiation. As a consequence, the resulting images of the VNFs are significantly smaller for the Containers than those for the VMs. The latter additionally need to store the OS and Docker engine to be executed on the virtualised machine within the image.

The current conversion tool is to be used to convert already developed VNFs implemented as Docker images. It solves discontinuity in development workflow by having different images for 5GTANGO Service Platform and SDK.

### 2.5.1 Image generation functionality

Current release focuses on conversion functionality between Docker and VM images. However, in the future, we envision the generation of images based on a range of configuration formats, such as:

- **software configuration and installation script** (e.g., bash script or Dockerfile): this feature aims to create a VM or Docker image based on an installation script or Dockerfile. This is a direct extension of existing conversion functionality.

- **Click Modular Router configuration** [21]: Click is a software architecture for building flexible, modular and configurable routers in *nix-based systems based on re-usable packet components. Their configuration is based on a configuration script which will serve as basis for the generated container or VM image.

- **RUMP kernel configuration** [20]: Rump Kernels enable to build efficient ad-hoc, *NIX-based (uni-)kernels with limited, but optimized driver and/or network stacks enabling lightweight NFs.

---

![Diagram](image.png)

Figure 2.7: Container-based image instantiation vs. VM-based image instantiation
LING kernels (Erlang on Xen) [22]: Erlang on Xen is a highly compatible Erlang VM capable of running as Xen guest OS enabling ultra fast boot and processing times. Erlang has been used in many cases as a robust application platform for distributed network applications including cloud applications, control and data plane architectures including Ericsson carrier-grade routers or backend of Facebook.

2.6 Packaging

Packages are the first class artifact used to exchange network services, VNFs, tests or test results between 5GTANGO’s main components. These packages are stored in a 5GTANGO-specific package format that extends existing cloud and VNF package standards, like TOSCA CSAR [17] or ETSI’s package format defined in ETSI GS NFV-SOL 004 [19]. The 5GTANGO package concepts have been initially introduced in D2.2 [5] and are further detailed in the following sections.

2.6.1 Package Format Specification

5GTANGO’s package format aims to be the most generic, reusable, feature complete, and compatible package format in the NFV landscape. To achieve this, we build up on the existing CSAR [17] and ETSI SOL004 [19] standards and extend them to explicitly support the following additional features:

- **Multiple package types**: 5GTANGO packages can contain single VNFs, complex network services, or test results only. This is a major difference to SONATA packages, but aligns them more with the view of ETSI [18], OSM [15], ONAP [23], or OpenBaton [16].

- **Layering**: 5GTANGO packages can contain different flavors of descriptors/artifacts for the same service, e.g., one NSD for 5GTANGO SP and one for OSM. This allows us to target different platforms with the same package and bypasses problems that occur if you try to automatically translate descriptors. We call this concept layering. Each layer is identified by one or more tags.

- **Artifact references**: To support slim packages, it is possible to reference artifacts, e.g., disk images or any other file, from within the 5GTANGO package. The unpackager will try to resolve these references when the package is unpacked, e.g., download the referenced disk image. To reference artifacts, we follow an approach that is inspired by Unix filesystem symlinks: For each reference, a file with the ending *.ref is created as a placeholder for the referenced artifact inside the package. This file contains all information required to resolve the reference, e.g., URIs, protocol information, checksums, or credentials of an FTP server. The benefit of this approach is that it fully compatible with the CSAR and ETSI standards, which would not be the case if we try to put reference information inside manifest files or package descriptors.

- **Package references**: 5GTANGO packages can reference other 5GTANGO packages using the vendor.name.version triple known from SONATA. With this, test result packages can reference the services to which they belong. Checksums allow to verify the integrity of references packages.

Even though, these features are not considered by the existing standards, our packages are still fully backwards compatible with them. The following sections discuss the general package structure and our advanced package descriptor format as two highlights of our package format. The full 5GTANGO package specification can be found online [7].
2.6.1.1 Package Structure

The 5GTANGO package structure is compatible with TOSCA’s CSAR v1.0 package structure which includes a **TOSCA-Metadata** directory that contains meta data files which describe the content of the packages. In addition to this, ETSI SOL004 V2.3.1 defines that there MUST be an additional manifest file that contains ETSI standard-specific information and has the extension *.mf. Based on these, the structure of 5GTANGO packages looks as follows:

**Minimal example:**

```
| +- TOSCA-Metadata
  |   +- TOSCA.meta
  |   +- NAPD.yaml
  +- Definitions
    |   +- mynsd.yaml
    +- mynsd.mf
```

**General example:**

```
| +- TOSCA-Metadata
  |   +- TOSCA.meta
  |   +- NAPD.yaml
  +- Definitions
    |   +- mynsd.yaml
    ..
    +- Images
      |   +- cloudimage.ref
    +- Scripts
    +- Tests
    ..
    +- mynsd.mf
```

---

2.6.1.2 NFV Advanced Package Descriptor

5GTANGO’s package format is designed to be as compatible as possible to the existing ETSI work. However, the 5GTANGO project still aims to introduce new features and innovations in the package format and NFV packaging concept. To do so, we add an additional descriptor file, called **NFV Advanced Package Descriptor** (**NAPD.yaml**) to 5GTANGO packages which contains all additional information that is required to implement novel package features that go beyond the CSAR or ETSI standard, e.g., tagging of artifacts. The **NAPD.yaml** can be viewed as the successor of the SONATA-NFV package descriptor file [29]. This concept, of extending the standard compliant manifest files by an additional package descriptor files is also used by other projects, like OpenBaton or ONAP to overcome the shortcomings of the current standards.

The NAPD file contains ALL information of the ETSI manifest file plus additional 5GTANGO specific information. With this, it is a candidate to replace the ETSI manifest file. The **NAPD.yaml** file uses YAML and its schema is defined in the **tng-schema** GitHub repository, just like any other
descriptor of the 5GTANGO project. The following examples show how a minimal or a more complex NAPD looks like:

Minimal example:

```yaml
descriptor_schema: >
  "https://raw.githubusercontent.com/sonata-nfv/tng-schema/
   master/package-specification/napd-schema.yml"
vendor: "eu.5gtango"
name: "example-package"
version: "0.1.1"
package_type: "application/vnd.5gtango.package.nsp"
maintainer: "Manuel Peuster, Paderborn University"
release_data_time: "2017.01.01T10:00+03:00"
```

Complex example:

```yaml
descriptor_schema: >
  "https://raw.githubusercontent.com/sonata-nfv/tng-schema/
   master/package-specification/napd-schema.yml"
vendor: "eu.5gtango"
name: "example-package"
version: "0.1.1"
package_type: "application/vnd.5gtango.package.nsp"
maintainer: "Manuel Peuster, Paderborn University"
release_data_time: "2017.01.01T10:00+03:00"
description: "This is an example 5GTANGO package."
logo: "icons/upb_logo.png"
package_content:
- source: "Definitions/mynsd.yaml"
  algorithm: "SHA-256"
  hash: "d0e7828293355a07c2dc4c98af5d392b035f073c950dc2e6b0da0dbd8b"
  content-type: "application/vnd.5gtango.nsd"
- source: "scripts/scale/scale.sh"
  algorithm: "SHA-256"
  hash: "e0e7828293355a07c2dc4c98af5d392b035f073c950dc2e6b0da0dbd8b"
  content-type: "text/x-shellscript"
tag:
  - "eu.5gtango"
  - "etsi.osm"
- source: "scripts/cloud_init/cloud_init.cfg"
  algorithm: "SHA-256"
  hash: "f0e7828293355a07c2dc4c98af5d392b035f073c950dc2e6b0da0dbd8b"
  content-type: "application/yaml"
tag:
  - "etsi.osm"
- source: "images/ubuntu_xenial.ref"
  algorithm: "SHA-256"
  hash: "x0e7828293355a07c2dc4c98af5d392b035f073c950dc2e6b0da0dbd8b"
```
Tags are entirely optional and will help to filter files for different target platforms (layering feature that is described at the beginning).

2.6.1.3 Official 5GTANGO Package Specification

Instead of providing the full specification of the 5GTANGO package format in this document, we made it available online on a dedicated GitHub page “5GTANGO Package Specification (latest)” [7]. This has two advantages: On the one hand, the specification can be a living document which can be updated when needed. On the other hand, the package specification is available to the public from day zero, making it easy to pick up and use by other projects.

2.6.2 Packaging Tool

To simplify the creation and extraction of 5GTANGO packages, we develop a packager tool as part of 5GTANGO’s SDK. This tool is called tng-sdk-package and is a complete rewrite of the SONATA packaging tool. The reason for this rewrite is that the 5GANGO packaging tool does not only support and simply the package creation but can also be used to unpack packages. For this, the packaging tool can be deployed as a micro service and used, e.g., by the service platform to unpack packages. The clear benefit of this design, compared to the SONATA design in which we used different tools for this, is that only one common codebase needs to be maintained that deals with packages. Thus, code changes, which may be a result of a change in the package format, need only be done in a single component of 5GTANGO. The following sections describe the architecture of tng-sdk-package in more detail.

2.6.2.1 Architecture

Fig. 2.8 shows the internal architecture of the tng-sdk-package tool. On a higher level, the design can be split into three parts. First, the interfaces that are used to call and use the packaging component. A CLI interface for local usage by a developer as well as a REST interface to be used by other services is provided. Both interfaces are described in more detail in Sec. 2.6.2.2 and Sec. 2.6.2.2. Second, the packager modules which take care about the actual packaging process. Third, the storage backend modules which are responsible to store and/or upload unpackaged artifacts.

Packager Modules

Once the packager tool receives the request to pack or unpack a package, its PackageManager creates a new, asynchronous Packager thread and hands over all required information about the request. This makes 5GTANGO’s packager tool asynchronous and allows to run multiple pack/unpack requests in parallel. It also ensures that interfaces are not blocked if fat packages with multiple GByte(s) of size are handled. Once a pack/unpack request has finished, the tng-sdk-package can call a callback URL to inform waiting components about the finalization of the process.

Instead of only implementing a single Packager component that is designed to handle the 5GTANGO package format, we opted for a more generic and extensible design which allows the easy integration of further Packager implementations that support other package formats. Fig. 2.8 shows the TangoPackager and the OsmPackager as an example for this extensible design. Each Packager has to implement methods to pack and unpack packages.
Figure 2.8: Internal architecture of tng-sdk-package component

Storage Backends

If tng-sdk-package is used to unpack a given package, it needs to store the extracted artifacts somewhere. One option for this is to just store them inside the local file system or in the file system of the tng-sdk-package container if the packager tool is used as a micro service. However, the downside of this setup would be that other components need to download the artifacts from the packager service and upload them to other locations, like a catalogue, from where they can be accessed by other components of the service platform or the V&V. This is why we again opted for a more flexible design and allow the implementation of different StorageBackend components. Besides the default FileSystemSB, we will implement a StorageBackend that can directly talk to 5GTANGO’s catalogues and thus directly upload all artifacts to them. This means a requesting component, e.g., the Gatekeeper just needs to submit a package to the tng-sdk-package service and trigger the unpack procedure. Once all artifacts from the package are extracted, they are automatically uploaded to the catalogues, and the packager informs the Gatekeeper about the location of the uploaded artifacts in the catalogues via the callback mechanism.

2.6.2.2 CLI Interface

The following listing shows the CLI of tng-sdk-package that can be used by a developer to locally create or unpack network service, VNF, or test packages. It is also possible to start the packager as a service which offers a REST API by using the -s flag.

```
$ tng-package -h
    [-f FORMAT] [-v] [-i ignore-checksums]
    [-no-autoversion] [-s] [-d dump-swagger]
    [-d dump-swagger-path DUMPSwagger_PATH]
    [-a SERVICE_ADDRESS] [-p SERVICE_PORT]
```

5GTANGO SDK packager

optional arguments:
Besides the default input and output arguments, the CLI offers a couple of convenience options that are described in the following:

- **--ignore-checksums**: When a package is unpacked, the packager validates the checksums of all contained artifacts to ensure integrity. However, for development purposes, or if a package was created by hand, this validation can be disabled.

- **--no-autoversion**: When a package is created, the packager automatically increments its version number. This is a convenience function to ensure that the generated package can directly be on-boarded to a service platform without name collisions. This option allows to disable this behavior.

- **--dump-swagger**: The built-in REST API does automatically generate an OpenAPI (formerly Swagger) API specification and serves it as webpage at the API endpoint. With this flag, the OpenAPI definition can be dumped to a file and used for documentation purposes.

The following listing gives two examples that show how `tng-sdk-package` can be used to pack and unpack a 5GTANGO network service package:

```bash
# create a NS package from a NS project
$ tng-package -p path/to/mynsproject
Packaged: mynsproject.0.3.11.tng in 31.2s

# unpack a given NS package
$ tng-package -u mynsproject.0.3.11.tng
Unpackaged to: mynsproject/ in 15.4s
```
2.6.2.3 REST API Specification

The REST API interface of tng-sdk-package is used to interact with the packaging tool when it is deployed as a micro service. This API allows to pack and unpack network service, VNF, and test packages as well as querying the status of packaging and unpackaging processes. Following the agile principles of the 5GTANGO project, the REST API implementation automatically generates its own OpenAPI-based documentation and serves it as a webpage as shown in Fig. 2.9.

A snapshot of this dynamically generated API documentation is also available on GitHub [8]. Tbl. 2.3 gives a brief overview of the provided REST API endpoints without going into too much detail about the used data models, since they are continuously developed and updated. However, the reader can find the full specification, including data models, of the latest API version online [8].

Table 2.3: tng-sdk-package REST endpoint overview

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
<th>Endpoint</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpacking</td>
<td>POST</td>
<td>/api/v1/packages</td>
<td>upload a package to unpack</td>
</tr>
<tr>
<td>unpacking status</td>
<td>GET</td>
<td>/api/v1/packages/{package_process_uuid}</td>
<td>query status</td>
</tr>
<tr>
<td>packaging</td>
<td>POST</td>
<td>/api/v1/projects</td>
<td>upload a project to pack</td>
</tr>
<tr>
<td>packaging status</td>
<td>GET</td>
<td>/api/v1/projects/{package_process_uuid}</td>
<td>query status</td>
</tr>
</tbody>
</table>
2.7 Access Tool

Once a package is produced by the SDK packaging tool, it should be uploaded (onboarded) on the V&V and SP via a REST API. This API is documented in [?] and can be used using manual CLI or library tools such as CURL. As current SP and V&V follow the APIs as defined in the SONATA project, for this first release of the SDK the son-access tool will be reused for this purpose. In the following sub-sections, we briefly recapitulate its functionality and usage.

2.7.1 Functionality

The son-access tool enables a secured communication between the SDK and SP based on authentication and authorization for potential end-users of the SP features. Mainly, the son-access is composed of the following components:

1. **Access** sub-component, implementing the authentication and authorization automatic procedures for the interactions with the SP features. The Access sub-component is fully integrated with the Gatekeeper API for the authentication procedure, but it also keeps compatibility with platforms that do not enable authentication layers.

2. **Push** sub-component, enabling the submission of the Package files to the SP Catalogues with authentication procedures if enabled.

3. **Pull** sub-component, enabling the retrieval of the Package files, Network Service (NSD) and Function Descriptors (VNFD) from the SP Catalogues and their storage to a preferable configured file system.

The current architecture is depicted in Fig. 2.10 in the flow diagram along with the complete workflow between each component. It is of paramount importance to note that only the necessary components of the Service Platform are depicted.

The corresponding APIs, that are already implemented, are listed in the wiki repository [31].

2.7.2 Future Plans

For the second release of the 5GTANGO SDK, all developer-triggered interaction among 5GTANGO components relying on REST APIs will be harmonised into a single SDK component. This involves access to multi-purpose catalogues, including the retrieval of all the hosted metadata and descriptors.
from the new services/frameworks introduced in the 5GTANGO infrastructure, as well as the SP, V&V framework and potentially other components.

2.8 Emulator

5GTANGO’s multi-PoP NFVI emulation platform is developed as part of the OSM [15] project under the name vim-emu [24]. This project is based on the open source project Containernet [25] which is actively developed by Paderborn University. The following sections detail the planned extensions that enable the use of the existing emulation platform in the 5GTANGO context. Besides this, represent all emulator-related developments also a direct cooperation between the 5GTANGO and the OSM project which actively contributes to the dissemination of the 5GTANGO project as such.

2.8.1 Integrating with Service Platform

One of the main shortcomings of the existing emulator platform is the limited integration with 5GTANGO’s service platform (former SONATA service platform). This shortcoming needs to be addressed to allow service developers to use the emulated NFVI together with the service platform as a rapid prototyping setup. Further, does this integration allow to use the emulation platform as test execution platform for a 5GTANGO V&V that is connected to a 5GTANGO service platform instance.

As already described in D2.2 [5] will the integration between service platform and emulator be done by implementing an additional wrapper component for the *infrastructure abstraction* component of the service platform. This wrapper can then directly talk to the native REST API of the emulator and control the deployment, configuration, chaining, and termination of VNFs inside the emulated multi-PoP infrastructure.

Fig. 2.11 shows the design of this new wrapper component in context of the infrastructure adaptor on the left and the emulator on the right. The wrapper component is developed in Java, inherits from the *AbstractWrapper* class, and implements the same methods as implemented by the existing OpenStack wrapper component. This also acts as a proof of concept to show the multi-VIM capabilities of 5GTANGO’s platform.

More specifically the implemented methods are used as follows:

- **deployService**: Deploys a given NSD by calling **deployFunction** for each contained VNF.
- **deployFunction**: Prepares the function deployment and waits for a placement decision of the MANO framework.
- **deployFunctionOnVIM**: Deploys a single VNF in form of a Docker container on a specific PoP in the emulated environment.
- **getResourceUtilisation**: Returns the resource utilization of the given PoP. Since the emulator has no real resources, like vCPUs, it retrieves simulated values from the emulator API.
- **scaleFunction**: Can add additional instances of a VNF.
- **removeService**: Removes all VNFs of a service.

The remaining methods are currently not considered to be used by the emulator, since they deal with image management of virtual machines disk image, which is unnecessary in the emulator case, which relies on Docker images that are pre-registered in the local emulator installation.
2.8.2 Local Testing Environment

Besides the integration with the service platform, the emulator can also act as a stand alone tool to locally execute simplified test cases on the developers laptop. For this, a translation and adaptor component needs to be created that interconnects a test control system, e.g., a TTCN-3 execution environment, with the emulator API. In addition, test adaptors in form of Docker containers need to be implemented that allow to inject test traffic into a VNF or a service running inside the emulated environment.

Such an environment will be very helpful for VNF and service developers because it allows quick turn-around times, because the on-boarding and deployment steps of a real service platform can be skipped. However, such a test tool is limited to functional tests and is also limited in terms of special platform or hardware requirements, like SR-IOV.

2.8.3 Adding VM Support to the Emulator

As described in D2.2 [5] was our emulation platform always designed to use Docker container-based VNFs. This makes the entire emulation platform very lightweight and easy to use. Especially when the limits of local hardware resources is considered, e.g., a standard laptop. Nevertheless, this design decision comes with the downside that the emulation platform can not be used for prototyping of VNF software that can only be installed and integrated in VM-based VNFs but not in containers.

5GTANGO will remove this shortcoming of the emulator by adding support for Qemu-based (KVM) virtual machines to the emulator. This extension will allow to use VMs and Docker-containers in a single service chain running inside the emulated platform as shown in Fig. 2.12. To achieve this, Containernet (the base of the emulation platform) will be extended and integrated.
Figure 2.12: Architecture of the extended Containernet project (Containernet 2.0) that adds support for hybrid emulations with Docker containers and VMs to 5GTANGO’s emulation platform.

with the virtualization library libvirt [27].

Our main requirement for the integration of VMs into the emulator is to be fully aligned with the existing Containernet and emulator APIs. Our design allows a user to add a fully-featured VM to an emulation topology with a single request that expects the path to the VM image to be used as additional parameter. Other parameters, like the node name or the IP addresses to be used, remain the same as in the existing implementations. This enables the seamless integration of VMs into existing emulation topologies. Additional parameters, like the hypervisor type, can be optionally passed to the underlying libvirt implementation.

A more challenging problem was the integration with the emulator’s interactive CLI that should allow a user to interact with all nodes (Docker containers and VMs) in the emulated network through a common CLI interface. In contrast to the CLI interaction scheme used in Containernet, which uses pipes to directly connect to the TTYs of the emulated hosts or containers, a direct interaction with VMs is not possible. To solve this, we use a network-based solution that adds a management network interface to each VM and connects to it using SSH. This solution solves the problem and gives seamless access to all VMs in the emulated topology (see Fig. 2.12). The downside of this approach is that it introduces the requirement that all used VMs need to have SSH installed and their access credentials have to be available to the emulation platform. We argue that this is an acceptable requirement since the majority of existing NFV and cloud orchestration solutions rely on such management interfaces in any case.

A proof-of-concept demonstration of this hybrid emulation platform is in preparation and will submitted to IEEE NetSoft 2018.

2.8.4 Testing MANO Systems Against the Emulator

Finally, the emulator can be used to perform large-scale multi-PoP experiments with MANO solutions. Where large-scale means testing MANO systems against network topologies that emulate a very high number of interconnected PoPs and their attached VIMs. This is something which is not possible with today’s lab-scale NFV testbed installations.

In a set of initial experiments, we analyzed the behavior of OSM in these large, emulated environ-
ments. We investigated the VIM attach procedure, which is used to connect OSM to a single PoP using the osm vim-create <vim-endpoint> command and the start up times of the emulation platform using a set of real-world topologies. Fig. 2.13 shows the total setup time breakdown to start the emulated infrastructure and to attach all emulated VIMs to OSM. The numbers behind the topologies indicate the number of nodes and links in each topology. The results show that the time required to attach the VIMs to OSM uses most of the test environment’s setup time, but the system can still be deployed and configured in less than 150s, even if the largest topology with more than 150 PoPs is used. The figure also shows the request times for all osm vim-create requests. It indicates that the attachment procedure becomes slightly slower when larger topologies are used.

A comprehensive study of the emulator’s scaling behavior is available in a paper [26] which is currently under review for publication at EuCNC 2018.

2.9 Development and testing of Specific Management functionality

5GTANGO allows developers to add Service and Function Specific Managers to the descriptors of its network service. This is a distinctive feature of SONATA and 5GTANGO, as it enables customisation of otherwise rigid orchestration and configuration capabilities available in other MANO platforms. These specific managers are processes which interact with the Service Platform using a well-defined API over the SP pub/sub bus [10] and incorporate service or function specific orchestration behaviour. Service Specific Managers (SSM) customise orchestration behaviour for NS life cycle events, while Function Specific Managers (FSM) customise VNF orchestration events. The developer adds them to the relevant descriptor, together with detailing which workflow they customise or extend. When the Service Platform needs to execute a network service life cycle event, it will first check whether one or more SSMs are associated with this service and workflow. If that is the case, the Service Platform will execute this SSM, i.e., process, instead of executing the generic workflow. A similar behaviour can be expected for network function life cycle events and FSMS. SM functionality enables NS- or VNF-specific placement, START- and STOP behaviour, configuration, and monitoring. In later phases dedicated events supporting scaling and failover logic will
be developed. Currently, the development of SMs fully relies on the input of the developer. Despite some initial SSM and FSM templates, the SONATA SDK does not provide any tools to assist the developer in the process of developing and testing SM functionality. This makes SM development a slow and error-prone process.

This release of the SDK was extended with an SM test engine, which drastically improves the involved workflow in developing and debugging specific management functionality. The current release focuses on the individual process blocks to support the generation of necessary SM code (extending and modifying the available SONATA SM templates) and descriptors, as well as the necessary environment context to actually test its execution. The real SM execution environment is provided by the Service Platform and the interaction of the SM with the Service Lifecycle component using the pub/sub bus. In order to support rapid testing of SM functionality, the Service Platform, the SLM and the pub-sub bus are emulated by enabling to generate the specific message content (payload) needed for testing the SM using the provided tool. The tool consists of two main parts: a generator and an executed component. Fig. 2.14 describes the overall process in developing and testing SMs using both components. We here describe briefly the high-level process, for in-detail command-line instructions, we refer the reader to the dedicated GitHub repository wiki pages.

The process is kickstarted by enabling the developer to generate a ready-made generic SSM or FSM template in the first step. Because this tool focuses on the development of the SM, the developer is in charge of manually deploying the associated VNFs for which the considered SM will be developed. The SM execution environment needs this deployment information in order to enable automated testing. A template of the required configuration can be generated (step 5), which should be modified by the developer to include adequate IP addresses and required interfaces (step 6). Depending on the particular events the SM developer wants to support, he/she can modify the appropriate event handlers of the provided SM template (step 8). A configuration FSM can for example add all necessary configuration steps. To prepare the execution environment to enable testing of the developed handler, it needs information about the corresponding messages expected on the pub-sub to actually feed the handler. In step 9, the generator component can be used to generate such payloads which can involve for example records for involved VNFs or services. After reception of this information (step 10), the developer can instruct the SM test executor to run the SM with given payload and or execution restrictions (e.g., only test particular handler) in step 11. The result of this execution (step 12) might capture bugs involving: syntax errors or runtime errors which might need a fix by the developer. In addition to direct execution errors, the developer might need to investigate the corresponding VNF(s) if the actions performed by the SM are actually correct (e.g., if configuration was done correctly). Based on the outcomes of step 12 and 13, the developer might change the code and iteratively restart the entire SM development process at step 3 (if we need a clean instance of the VNF because the running instance was misconfigured during the previous iteration) or step 8 (if the VNF is still healthy, step 9 and 10 can be skipped in this case). Once the entire development process is finished, the corresponding VNF descriptors can be adapted to include the information about the developed SSM in step 14.

Through the given tool, the entire SM development process can be performed without actually requiring an operational SP, neither an actual VIM. Although in subsequent SDK releases this process might be further automated, this is already an important milestone to ease the SM development process.
Figure 2.14: Testing process of specific management functionality
2.10 Traffic generation tool

The traffic generation tool has been introduced in 5GTANGO so it was not available in Sonata-NFV. This tool allows to generate real flows of network traffic in order to test and validate the NFV instances developed and deployed in an NFV platform based on 5GTANGO.

This SDK tool is a wrapper over existing and well-known traffic generation tools such as iperf. This way developer and testers will have a single tool which can cover a wide range of traffic simulation needs.

The tool can be used both from a CLI and also a REST API which enables the possibility of generation traffic flows from different services.

Diagram fig. 2.15 shows the internal architecture of the tng-traffic command. More external tools can be added in the future if there is a demand from the community project but scapy is very likely to cover a wide range of testing needs.

All the code and the most updated reference of the CLI and the REST API can be found in the project repository [9].

2.10.1 CLI commands

This is the output of the help option of the tng-sdk-traffic command:

```
$ tng-sdk-traffic -h
5GTANGO Traffic Generator
optional arguments:
-h, --help         show this help message and exit
-t TRAFFIC_TYPE, --type type of traffic to be generated.
                 The types can be: IP, UDP, TCP
                 validating the SDK project.
-b BANDWIDTH_IN_BPS, --bandwidth specifies the bandwidth of the generated traffic
                 in kbps. If not specified a bw of 500kbps is asumed.
```
-d DESTINATION_IP,  --destination destination IP or hostname of the traffic.
-o ORIGIN_IP,  --origin originate IP for the traffic, by default the tool will use the default settings of the system.
-c CONF_FILE  --configuration Optional configuration file to setup more advance traffic patterns.
--debug Sets verbosity level to debug

The syntax of the configuration is still not closed but it will be completely defined in repository documentation and the next SDK deliverable.

2.10.2 Example of use
The CLI follows this basic syntax:

```bash
$ tng-sdk-traffic -t <type of traffic> -b <BW> -d <dest IP> -c <optional conf file>
```

In the configuration file it would be possible to set advanced settings of the tools used to generate traffic. The creators of the services can include a customized configuration file to test their services once deployed.

For example, the command below generates an IPv6 flow to a defined IP:

```bash
#Generates a 25mbps flow over IPv6 to IP 2001:DB8::ABAB:25
$ tng-sdk-traffic -t ipv6 -b 25M -d 2001:DB8::ABAB:25'
```

2.10.3 REST API definition
The REST API enables the use of the traffic generation command from other tools of the project or any other external application. Tbl. 2.4 gives an overview of this REST API.

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
<th>Endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate traffic flow</td>
<td>POST</td>
<td>/api/traffigen/v1/flows</td>
</tr>
<tr>
<td>Retrieve traffic generation object</td>
<td>GET</td>
<td>/api/traffgen/v1/flows/{resource_uid}</td>
</tr>
<tr>
<td>Removes traffic generation object</td>
<td>DELETE</td>
<td>/api/traffgen/v1/flows/{resource_uid}</td>
</tr>
<tr>
<td>Creates traffic flow in existing object</td>
<td>POST</td>
<td>/api/traffgen/v1/flows/{resource_uid}/{flow_uid}</td>
</tr>
<tr>
<td>Start/Stop existing traffic flow in existing object</td>
<td>PUT</td>
<td>/api/traffgen/v1/flows/{resource_uid}/{flow_uid}</td>
</tr>
<tr>
<td>Retrieve status of a traffic generation flow</td>
<td>GET</td>
<td>/api/traffgen/v1/flows/{resource_uid}/{flow_uid}</td>
</tr>
<tr>
<td>Removes traffic generation flow</td>
<td>DELETE</td>
<td>/api/traffgen/v1/flows/{resource_uid}/{flow_uid}</td>
</tr>
</tbody>
</table>

2.11 QoS and Test Model
In this section, we are describing the approach followed in 5GTANGO for supporting profiling aspects of Virtual Network Functions (VNFs) and Network Services (NSs). Profiling has mainly to do with consideration of a set of performance aspects and aims at the extraction of useful insights regarding the behaviour of each software under various load conditions. The outcome of a profiling process can be a time series data set, a graph or a classification of the software based on predefined categories. Such an outcome can be used in various directions, including targeted deployment and runtime policies definition by the Service Providers as well as software update by Software Developers in case of identification of important bottlenecks in the software performance.

Given the initial specification of the profiling aspects tackled in 5GTANGO at D2.2, following, we
are providing elaborated specifications along with description of primary implementation aspects. The main objective is to identify the main performance metrics to consider and to formally bind the input and output of a profiling process in a way that can be applicable in any VNF or NS.

### 2.11.0.1 Profiling Model

As already stated, we are going to examine profiling aspects for both VNFs and NSs. In each case, different set of parameters are considered important to be measured and evaluated.

For VNFs profiling, performance evaluation considering different traffic load profiles is going to be realised. Profiling may regard various aspects, including:

- **Resources usage profiling**: how much resources are consumed (avg\_CPU, avg\_Memory, avg\_Storage) given the variation in the traffic load and the number of served users/sessions;

- **VNF specific metrics profiling**: how custom metrics (avg\_response\_time, avg\_requests\_per\_sec, transcoding\_rate) behave given the variation in the traffic load and the number of served users/sessions;

- **Elasticity profiling**: how scale operations in terms of scaling time and number of instances behave given the variation in the traffic load. Examination of the effect of scaling operations towards improvement of overall service provision is going to be also considered.

- **Availability and reliability profiling**: how the software behaves in terms of failures and failure recovery time, given the variation in the traffic load and the number of served users/sessions;

- **Deployment and migration profiling**: how the software behaves in terms of time required for deployment and migration, taking into account also the number of instances under migration.

In the following figures, some indicative examples of the parameters considered towards the examination of resources usage and elasticity profiling are provided, as they are defined in [28]. In the first figure Fig. 2.16, examination of the useful software output given the incremental addition of resources is examined. Minimum and maximum capacity per software is also identified.
In the second figure Fig. 2.17, the VNF capacity changes due to discrete configuration change actions, such as adding or removing a VNFC instance to/from a pool of fungible worker component instances are depicted. The time required for adding one or more instances along with the associated change in the total number of instances constitute the overall elasticity profile of the VNF.

For Network Services profiling, the main set of characteristics examined regard deployment and orchestration related metrics for the network service, as well as metrics related to the dependencies among the components (VNFs) constituting the overall NS. Analysis and identification of how graph relationships affect the overall performance is going to be performed. An indicative set of metrics regard deployment metrics (e.g. time required for the deployment or re-configuration of a service), end-to-end network metrics (e.g. end-to-end delay) and components interaction metrics (e.g. number of calls realised among components).

2.11.0.2 Profiling Toolkit

The profiling toolkit Fig. 2.18 consists of a set of components supporting the offline execution of profile creation processes, taking as input data made available through the set of tests realised per VNF or NS. Such data is available in the Test Result Repository of the V&V framework and regard mainly time series data. Upon request, such data are provided to the Profiling Component in the SDK, leading to the realisation of the associated analysis and the extraction of the relevant profiling information.

For supporting the various profile extraction processes, a set of data mining and analysis mechanisms are going to be designed and applied. Such processes regard extraction of descriptive statistics, realisation of regression/forecasting analysis and examination of correlation among parameters (e.g. effect of network traffic handled on the consumed compute resources). The set of analytic processes are going to be supported by data analytics tools based on R and Spark. Based on the requirements in terms of data volumes and computational needs, simple or big data computing frameworks are going to be exploited. In addition to the analytic processes, extraction of graphs based on a visualisation framework is going to be realised.

It should be noted that at the current phase, a simple profiling mechanisms setup is realised. This mechanisms is getting data from the time series database made available in 5GTANGO (InfluxDB) and examines the association among the resources consumption and the overall traffic load served. In addition to the analysis results, simple visualisations are produced based on the Graphana tool. Such an indicative visualisation is depicted at Fig. 2.19.
Figure 2.18: Profiling Toolkit within 5GTANGO

Figure 2.19: Indicative visualisation
2.12 State Management Capabilities

Many network functions need to store a certain amount of state. They need to keep track of past events which might influence their current and future behaviour. Examples include the set of currently open TCP connections flowing through a firewall, and the desired quality-of-service of a particular video stream. The type and the extent of state that a particular network function keeps, depends on the purpose of a function as well as on its implementation.

In addition to the function-internal state, network forwarding state needs to be considered. Network forwarding state refers to the rules which steer network traffic to a particular function at a specific location in the network. When migrating or scaling functions, the network forwarding state needs to be adjusted as well, in line with the changes done to the set and location of network function instances. A more thorough analysis of function state in relation to migration, upgrading and scaling can be found in deliverable D2.2 [5].

The 5GTANGO MANO system needs to support state management. Although the state itself will be managed by each individual VNF, the MANO system orchestrates forwarding state updates and the migration, upgrading, and scaling of functions. It therefore needs to trigger state externalization and recovery in line with service management schedules and needs to consider service level agreements at the same time. There is a clear requirement on the service platform to execute these functions at run-time. Nevertheless, the SDK needs to provide the support necessary to allow the developer to deal with state management. In particular, the SDK is concerned with the development of VNF and network service management aspects, of which state management is part and needs to be addressed properly.

Migration of VNFs requires the most basic state management functionality. This is due to migration requiring a direct transfer of state from the source VNF to the destination VNF. Splitting or merging of state is not required. For this reason, 5GTANGO will initially look at state management for VNF migration. Aspects of forwarding state migration from the old location to the new location are initially considered out of scope for the SDK, as this impacts the service platform only. Whether function specific forwarding state management is required or helpful and whether it could therefore benefit from function specific MANO extensions is currently unknown. If this turns out to be the case, 5GTANGO will consider providing appropriate support in the SDK.

On an abstract level, the steps involved in state migration are: 1) externalize state, 2) stop VNF, 3) start new VNF, 4) recover state. Such state migration must be orchestrated by the MANO platform and therefore involves the management system. As state externalization and recovery are typically specific for a VNF, 5GTANGO makes use of the function-specific management components of its architecture, namely the function specific managers (FSMs) and service specific managers (SSMs), for state management. These components will realize workflows for VNF migration, which trigger the execution of associated state migration functions.

The migration workflow is triggered by some event inside the service platform. Such events are triggered by the SLA and policy management components when they detect (imminent) violation of an SLA or a policy. The lifecycle manager receives this trigger and initiates the execution of the migration workflow. The associated events of a) externalizing state and b) recovering state are then issued to the FSM of the VNF to be migrated.

5GTANGO will therefore extend its lifecycle model with a migration workflow which can be extended and overwritten by service specific managers. Function specific managers can receive new event notifications for externalizing and for recovering state. The FSMs can then map these generic events to actions which are appropriate for their associated VNFs. Through these mechanisms, optimized VNF-specific state management is achieved.

5GTANGO will start working on state management capabilities in its second development sprint,
after the release of this deliverable.
3 Interfaces

This section describes the interfaces of the 5GTANGO Service Development Kit prototype. In particular, are described the interfaces among the internal components as well as the interactions of the 5GTANGO SP with external entities and stakeholders.

3.1 SDK internal interfaces

The SDK is basically a set of largely command-line (CLI) tools which work on a common workspace and project folder on a filesystem on the pc of the developer. As a result, the main internal SDK interface is the command line operating on a set of files in the filesystem. This is the case for: the workspace and project management tools, the descriptor generator, the image converter, the validator, the packager, and the access tool.

A number of SDK components can be activated as a service (daemon mode), and therefore can also be interacted with using a REST API. Those SDK components are the validator, the packager, the emulator and the traffic generation service. Running these components as a service enables to call their functionality in other contexts outside of the command line. The validator can be reused, for example from the SP or V&V when onboarding packages. Similarly, the unpackaging functionality of the packager can be used to fetch all dependencies during onboarding. The emulator and traffic generator service modes enable programmatically control them during the development and debugging phase.

Each of these APIs has been documented in this document and more extensively in their corresponding GitHub repositories. Fig. 3.1 gives an overview of these interfaces.

3.2 SDK external interfaces

From a higher level perspective, the SDK also interacts with other external components, such as the 5GTANGO Service Platform and the Verification and Validation Platform.

For these, the Access tool can be used to call the corresponding REST APIs of the SP (the Gatekeeper API is documented in [13]). The API allows the management of: users, sessions, VIMs, WIMs and KPIS, in addition onboarding and retrieving packages on the platform. In the future however, interaction might foreseen with external, third-party catalogues. Such functionality might enable to download components of network services or VNFs directly into the local developer work- or project space using the Access tool, or the Packager might interact with those catalogues in order to collect such components when building a fat package.
Figure 3.1: SDK interface overview
4 Source Code

All 5GTANGO developments are open source and hosted on GitHub. Following a microservice-based approach, the project creates separate repositories for individual, discernible components. All repositories are hosted within the SONATA GitHub organisation [6]. This was chosen due to the 5GTANGO project inheriting and extending the code of the SONATA platform. The components list shown below provides an overview of the mapping of each component to the repository it makes use of.

<table>
<thead>
<tr>
<th>Table 4.1: SDK code repository overview</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SDK tool</strong></td>
</tr>
<tr>
<td>schema</td>
</tr>
<tr>
<td>project creator descriptor generator</td>
</tr>
<tr>
<td>image converter</td>
</tr>
<tr>
<td>sm tester</td>
</tr>
<tr>
<td>packager</td>
</tr>
<tr>
<td>uploader</td>
</tr>
<tr>
<td>validation</td>
</tr>
<tr>
<td>VIM emulator</td>
</tr>
<tr>
<td>traffic generator</td>
</tr>
</tbody>
</table>
5 Conclusion

This document provided the base components and pillars of the 5GTANGO SDK. The SDK empowers the NFV developer with programming models, processes and a range of tools to assemble an NFV service. NFV programming are driven by descriptors and formats. The SDK descriptor generation functionality speeds up and eases the process of service and VNF descriptor writing. VNF image conversion and generation functionality aims to overcome existing barriers in VNF image formats. The developer process is defined by an end-to-end workflow starting from the local project environment containing all descriptors and executable VNF images. These local files are adapted, tested, onboarded and executed using SDK tools including validation and emulation functionality (or alternatively the V&V or SP). The emulator provides the developer with local service runtime testing including traffic generation and handling, which will be further extended in the future with advanced profiling and QoS model determination. Service- and function-specific management functionality can be tested in parallel using a dedicated test environment. Future releases of the SDK will further extend the SM concept with re-usable workflows with scaling, migration and failover mechanisms.
A Bibliography


