OSM White Paper

OSM VNF ONBOARDING GUIDELINES

A White Paper prepared by the OSM VNF Onboarding Task Force

Version 1.0 - June 2019

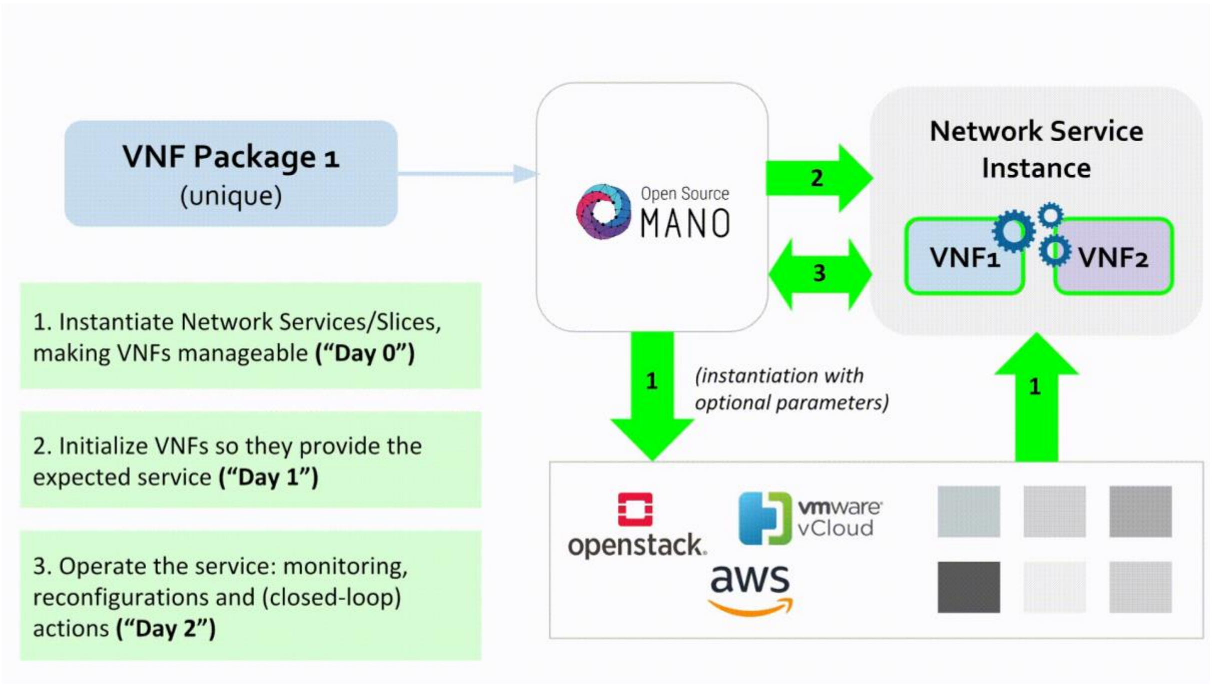
ETSI
06921 Sophia Antipolis CEDEX, France
Tel +33 4 92 94 42 00
info@etsi.org
www.etsi.org

Table of Contents

Introduction

The complete onboarding process implies producing a VNF Package that will be part of the OSM catalogue for its inclusion in a Network Service.

The onboarded VNF should aim to fulfil the lifecycle stages it requires to function properly, which the NFV MANO layer should then be able to automate. The resulting package should thus include all the requirements, instructions and elements to achieve these lifecycle stages, which are: basic instantiation (a.k.a. "Day 0"), service initialization (a.k.a. "Day 1") and runtime operations (a.k.a. "Day 2").


VNF Package 1
(unique)
1. Instantiate Network Services/Slices, making VNFs manageable ("Day 0")
2. Initialize VNFs so they provide the expected service ("Day 1")
3. Operate the service: monitoring, reconfigurations and (closed-loop) actions ("Day 2")
Open Source MANO
2
3
1
(instantiation with optional parameters)
Network Service Instance
VNF1
VNF2
1
openstack.
vmware vCloud
aws

More information on the delimitation of these stages can be found in the *OSM Scope and Functionality* document.

This document provides guidelines for building the VNF Package to achieve each lifecycle stage successfully and is structured as follows:

1. Onboarding requirements
2. Day 0: VNF Instantiation & management setup
3. Day 1: VNF Service initialization
4. Day 2: VNF Runtime operations
5. Known issues

Onboarding requirements

Each lifecycle stage targets specific configurations in the VNF. These are: management setup during instantiation (Day 0), service initialization right after instantiation (Day 1) and re-configuration during runtime (Day 2).

In order to provide a VNF with as many capabilities for each lifecycle stage as possible, the following specific requirements should be addressed.

Day 0 requirements

During the Day 0 stage, the VNF is instantiated and the management access is established so that the VNF can be configured at a later stage. The main requirements to achieve this are:

Description of each VNF component

The main function of every VNF component (VDU) should be clearly described in order to facilitate the understanding of the VNF. For example:

VDU	Description
vLB	External frontend and load balancer
uMgmt	Universal VNF Manager (EM)
sBE	Service Backend of the platform

Defining NFVI requirements

These requirements refer to properties like the number of vCPUs, RAM GBs and disk GBs per component, as well as any other resource that the VNF components need from the physical infrastructure. For example:

VDU	vCPU	RAM (GB)	Storage (GB)	External volume?
vLB	2	4	10	N
uMgmt	1	1	2	N
sBE	2	8	10	Y

For some VNFs, the Enhanced Platform Awareness (EPA) characteristics need to be defined when the VNF requires performance capabilities which are “higher than default” or any particular hardware architecture from the NFVI. Popular EPA attributes include:

- Computer performance attributes:
  - CPU Pinning
  - NUMA Topology Awareness
  - Memory Page Size
- Data plane performance attributes:

– PCI-Passthrough

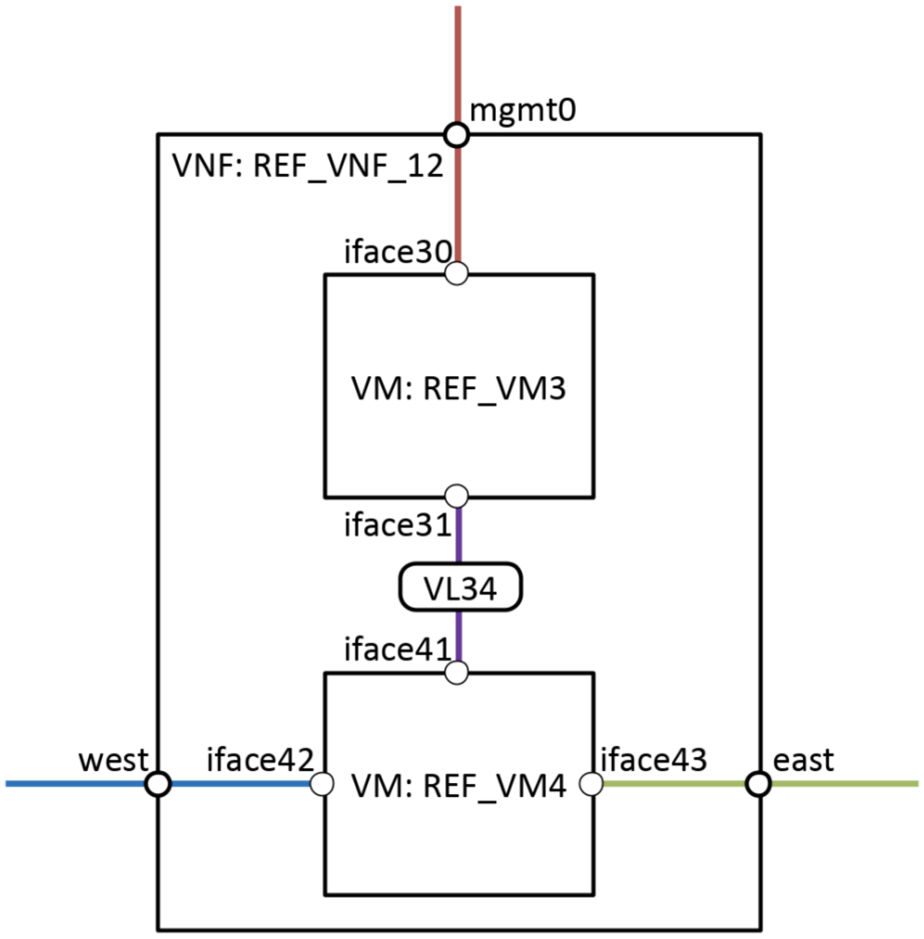
– SR-IOV

For example, vLB and sBE VDUs could require:

- 2 dedicated vCPUs
- Large-size memory pools
- SR-IOV for eth1 and eth2

Topology and management definition

Ideally, a diagram should be used to quickly identify components and internal/external connections.



Additional topology examples, along with sample descriptor files, can be found here.

Images and cloud-init files

The images for each component should be available in the format that corresponds to the main supported hypervisor. This image should contain the minimal configuration that makes it generic (not scenario-specific) and with **no hardcoded parameters** that are relevant to the service. Furthermore, cloud-init files can be used to inject this minimal configuration to the VNF. Some examples:

```
# Cloud-init using cloud-config format

#cloud-config
hostname: vnfc01
chpasswd:
  list: |
    ubuntu:ubuntu
  expire: False
ssh_pwauth: True
```

```
# Cloud-init using bash format for CentOS

#!/bin/bash
hostnamectl set-hostname vnfc01

cat <<EOF > /tmp/ipcfg
DEVICE=eth1
BOOTPROTO=dhcp
HWADDR=00:19:D1:2A:BA:A8
ONBOOT=yes
EOF

echo -y | cp /tmp/ipcfg /etc/sysconfig/network-scripts/ifcfg-eth0
systemctl restart network
```

Identifying the instantiation parameters

The VNF Day 0 configuration may require some parameters passed at instantiation time in order to fulfil the needs of the particular environment or of other VNFs in the Network Service. These parameters should be identified as early in the process as possible.

Day 1 requirements

The main objective of the Day 1 stage is to configure the VNF so it starts providing the expected service. To achieve this, the main requirements are:

Identifying dependencies between components

This may be required to identify instantiation parameters or special timing requirements. Examples of dependencies between components include:

- Components needing parameters from other components or from the infrastructure to complete the parameter configuration.
- Components depending on others for their configuration to be initialized.

Defining the required configuration for service initialization

This initial configuration will run automatically after the VNF is instantiated. It should activate the service delivered by the VNF and should be initially prepared in the language that the VNF supports. Once it is defined, it will need to be incorporated by the mechanism that the generic VNF Manager implements. For example:

```
# A Python script (NETCONF/YANG in the example)

from ncclient import manager
import sys

config = """
   <config>
    <interface-configurations xmlns="...">
     ...
    </interface-configurations>
   </config>
"""

host = {'name':'VNF1', 'ip': '192.168.0.1'}
interface_list = ['eth1', 'eth2']

m = manager.connect(host=host['ip'], username='ws', password='ws')

for interface in interface_list:
   response = m.edit_config(target='candidate', config=config.format(interface=interface))
   commit = m.commit()
   print(commit)

m.close_session()
```

```
# An Ansible playbook (VyOS module in the example)

- hosts: all
  tasks:
  - name: Configure the VNF initial NAT Rules
    vyos_config:
      lines:
        - set nat destination rule 1 inbound-interface eth0
        - set nat destination rule 1 destination port 80
        - set nat destination rule 1 protocol tcp
        - set nat destination rule 1 translation address {{destination_ip}}
```

Identifying the need for instantiation parameters

The VNF Day 1 configuration may require some parameters passed during instantiation in order to fulfil the needs of the particular environment or of other VNFs in the Network Service. These parameters should be identified as early in the process as possible.

Day 2 requirements

The main objective of Day 2 is to be able to **re-configure** the VNF so its behaviour can be modified during runtime, as well as be able to monitor its main KPIs and run scaling actions over it. To achieve this, the main requirements are:

Identifying dependencies between components

This process may be required to identify if a VNF component requires a parameter coming from other components for fulfilling runtime operations successfully.

Defining all possible configurations for runtime operations

The set of configurations should be available to be triggered from the orchestrator during the VNF runtime, either manually by the operator or automatically, based on a determined state. Once that set of configurations has been defined, it needs to be incorporated by the mechanism that the generic VNF Manager implements. Just as in Day 1, the set of configurations can be provided by Python scripts, Ansible playbooks, VNF-specific commands that run over SSH, REST API calls, or whatever the VNF makes available to expose its main operations.

Defining key performance indicators

The metrics that are relevant to the VNF should be specified, whether they are supposed to be collected from the infrastructure (through the VIM) or directly from the VNF (or its Element Manager, through any API, MIB or command that the VNF exposes). Some examples include:

- Metrics typically collected from the VIM/NFVI:
  - CPU Usage
  - Memory Usage
  - Network activity (bandwidth, drops, etc.)
  - Storage consumption
- Metrics collected from the VNF/EM (examples):
  - Active transactions/sessions/connections
  - Active users
  - Size of the database or a particular table
  - Application status

Defining closed-loop operations

Closed-loop operations are actions triggered by the status of a particular metric. The main use cases include:

- Auto-scaling: a VNF component scales horizontally (out/in) to match the current demand. Some typical definitions that must be clear are:
  - How the VNF will load-balance the traffic once it scales.
  - Which components should scale, in what quantity, and based on which metric threshold or status.
  - How much time the system should wait between scaling requests.
- Auto-healing: a VNF component is re-instantiated, reloaded or re-configured based on a service status. Some typical definitions that must be clear are:
  - Under which conditions the system should trigger an auto-healing action.
  - Which elements should be affected and at what level (re-instantiation, hard-reload, soft-reload, process restart, etc.)

Day 0: VNF Instantiation & management setup

Description of this phase

The objective of this section is to provide the guidelines for including all the necessary elements in the VNF Package for its successful instantiation and management setup, so it can be further configured at later stages.

The way to achieve this in OSM is to prepare the descriptor so that it accurately details the VNF requirements, prepare cloud-init scripts (if needed), and identify parameters that may have to be provided at later stages to further adapt to different infrastructures.

Day-0 Onboarding guidelines

Building the initial package

The most straightforward way to build a VNF package from scratch is to use the existing script available at the OSM Devops repository. From a Linux/Unix-based system:

Clone the OSM DevOps repository and access the tools folder.

```
git clone https://osm.etsi.org/gerrit/osm/devops.git
cd devops/descriptor-packages/tools
```

Run the generator script with the desired options.

```
./generate_descriptor_pkg.sh [options] [name]
```

Most common options are:

Parameter	Scope	Description	Values
-t	package	descriptor type	vnfd
-a	package	create package for the descriptor	-
-N	package	keep folder after tar is built	-
-c	package	create folder structure inside package	-
-d	package	destination of the folder	path
–nsd	package	create folder structure for NSD as well	-
–image	vdu	image name	name
–vcpu	vdu	vCPU number	#
–memory	vdu	RAM size	[mb]
–storage	vdu	disk size	[gb]
–cloud-init-file	vdu	cloud-init file name	name
–interfaces	vdu	interface number (additional to management)	#

–vendor	vnf	vendor name	name

For example:

```
./generate_descriptor_pkg.sh -t vnfd -N -c -d /home/ubuntu \
-a --image haproxy_ubuntu --vcpu 2 --memory 4096 --storage 10 \
--cloud-init-file init_lb --interfaces 2 --vendor ACME --nsd vLB
```

Note that we are adding the 'nsd' keyword to also create an NS Package that refers to this VNF Package, to be able to instantiate it and test it out. So the above example will create, in the /home/ubuntu folder:

- vLB_vnfd → VNFD Folder
- vLB_vnfd.tar.gz → VNFD Package
- test_vnf01_nsd → NSD Folder
- test_vnf01_nsd.tar.gz → NSD Package

The VNFD Folder will contain the YAML file which models the VNF. This should be further edited to achieve the desired characteristics.

Modelling advanced topologies

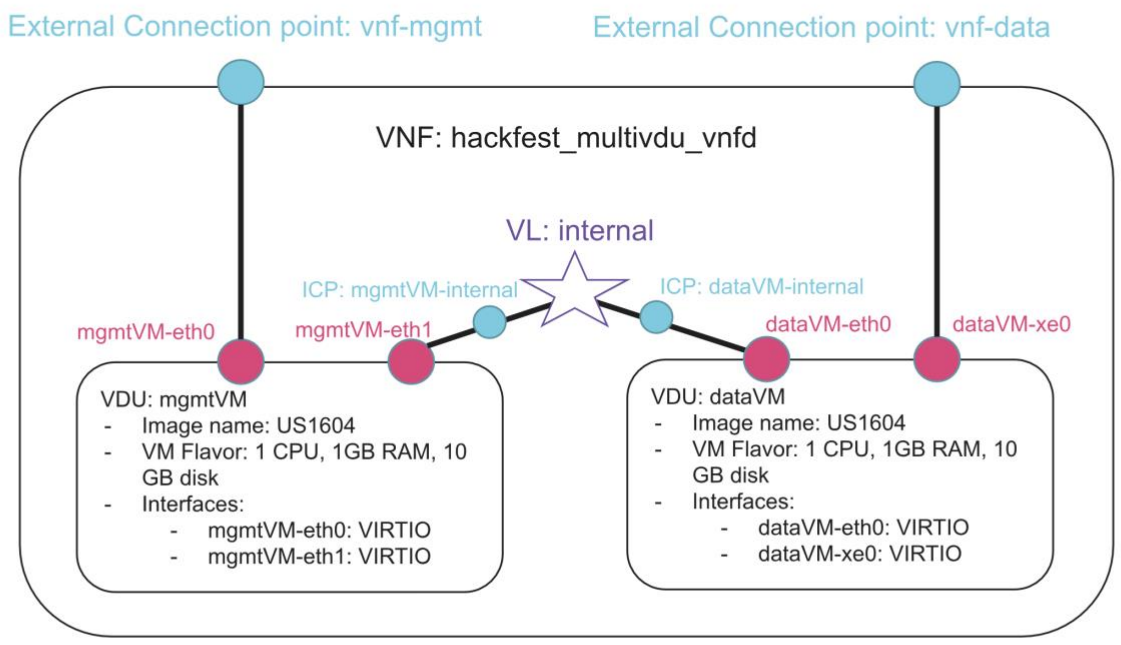
Most topology types, along with sample descriptor files, can be found here.

When dealing with multiple VDUs inside a VNF, it is important to understand the differences between external and internal connection points (CPs) and virtual link descriptors (VLDs).

Component	Definition	Modelled at
Internal VLD	Network that interconnects VDUs within a VNF	VNFD
External VLD	Network that interconnects different VNFs within an NS	NSD
Internal CP	Element internal to a VNF, maps VDU interfaces to internal VLDs	VNFD
External CP	Element exposed externally by a VNF, maps VDU interfaces to external VLDs	NSD

As VNF Package builders, we should clearly identify interfaces that i) are internal to the VNF and used to interconnect our own VDUs through internal VLDs, and ii) those we want to expose to other VNFs within a Network Service, using external VLDs.

In this example from the 5th OSM Hackfest, we are building the following Multi-VDU topology:



The VNFD would look like this:

```
vnfd:vnfd-catalog:
  vnfd:
  -  ...
    # An external CP should be used for VNF management
    mgmt-interface:
      cp: vnf-mgmt

    # External CPs are exposed externally, to be referred at the NSD
    connection-point:
    -  id: vnf-mgmt
      name: vnf-mgmt
      short-name: vnf-mgmt
      type: VPORT
    -  id: vnf-data
      name: vnf-data
      short-name: vnf-data
      type: VPORT

    # Internal VLDs are defined globally at the VNFD
    internal-vld:
    -  id: internal
      name: internal
      short-name: internal
      type: ELAN
      internal-connection-point:
      -  id-ref: mgmtVM-internal
      -  id-ref: dataVM-internal

    # Inside the VDU block, multiple VDUs, their interfaces and CPs are modelled
    vdu:
    -  id: mgmtVM
      ...

      # VDU Interfaces map to either an external or internal CP
      interface:
      -  name: mgmtVM-eth0
         position: '1'
         type: EXTERNAL
         virtual-interface:
            type: VIRTIO
         external-connection-point-ref: vnf-mgmt
      -  name: mgmtVM-eth1
         position: '2'
         type: INTERNAL
         virtual-interface:
            type: VIRTIO
         internal-connection-point-ref: mgmtVM-internal

      # Internal CPs are modelled inside each VDU
      internal-connection-point:
      -  id: mgmtVM-internal
         name: mgmtVM-internal
         short-name: mgmtVM-internal
         type: VPORT

    -  id: dataVM
      ...
      # VDU Interfaces map to either an external or internal CP
      interface:
      -  name: dataVM-eth0
         position: '1'
         type: INTERNAL
         virtual-interface:
            type: VIRTIO
         internal-connection-point-ref: dataVM-internal
      -  name: dataVM-xe0
         position: '2'
         type: EXTERNAL
```

```
            virtual-interface:
              type: VIRTIO
            external-connection-point-ref: vnf-data

          # Internal CPs are modelled inside each VDU
          internal-connection-point:
          -  id: dataVM-internal
             name: dataVM-internal
             short-name: dataVM-internal
             type: VPORT
```

As an additional reference, let's take a look at this Network Service Descriptor (NSD), where connections between VNFs are modelled using external CPs mapped to external VLDs like this:

```
nsd:nsd-catalog:
  nsd:
  -  ...
    # External VLDs are modelled globally
    vld:
    -  id: mgmtnet
       name: mgmtnet
       short-name: mgmtnet
       type: ELAN
       mgmt-network: 'true'
       vim-network-name: mgmt
       vnfd-connection-point-ref:

       # Mapping between VNF's external CPs and the external VLD occurs here:
       -  vnfd-id-ref: hackfest_multivdu-vnf
          member-vnf-index-ref: '1'
          vnfd-connection-point-ref: vnf-mgmt
       -  vnfd-id-ref: hackfest_multivdu-vnf
          member-vnf-index-ref: '2'
          vnfd-connection-point-ref: vnf-mgmt
```

Modelling specific networking requirements

Even though it is not recommended to hard-code networking values in order to maximize the VNF Package uniqueness, there may be some freedom for doing this at internal VLDs, especially when they are not externally accessible by other VNFs and not directly accessible from the management network.

The **IP Profiles** feature allows us to set some subnet specifics that can become useful. Further IP Profile settings can be found at the OSM Information Model Documentation. The following VNFD extract can be used as a reference:

```
vnfd:vnfd-catalog:
  vnfd:
  -  ...
    # IP profiles let us set subnet parameters like disabling a default GW
    ip-profiles:
    -  name: ip1
       description: ip1
       ip-profile-params:
         ip-version: ipv4
         dns-server: 8.8.8.8
         gateway-address:
         subnet-address: 192.168.100.0/24
         dhcp-params:
           enabled: true

    # The IP Profile name is then applied at the VLD level
    internal-vld:
    -  id: internal
        ip-profile-ref: ip1
        ...
```

Specific IP and MAC addresses can also be set, although this practice is not recommended unless we use it in isolated connection points.

```
vnfd:vnfd-catalog:
  vnfd:
  -  ...
    # A specific IP address can be set at the VLD; it requires the subnet to be predefined by using an I
P Profile
    internal-vld:
    -  id: internal
       ip-profile-ref: p1
       ...
       internal-connection-point:
       -  id-ref: mgmtVM-internal
          ip-address: 192.168.100.100
    ...
    vdu:
    -  id: mgmtVM
       ...

       # A specific MAC address can also be set at the interface level
       interface:
       -  ...
          mac-address: '01:02:03:01:02:03'
```

Building and adding cloud-init scripts

Cloud-init basics

Cloud-init is normally used for Day 0 operations such as:

- Setting a default locale
- Setting an instance hostname
- Generating instance SSH private keys or defining passwords
- Adding SSH keys to a user's .ssh/authorized_keys so they can log in
- Setting up ephemeral mount points
- Configuring network devices
- Adding users and groups
- Adding files

Cloud-init scripts are referred at the VDU level. These can be defined inline or can be included in the **cloud_init** folder of the VNF package, then referred in the descriptor.

For inline cloud-init definition, follow this:

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
  vdu:
  - ...
   cloud-init: |
    #cloud-config
    ...
```

For external cloud-init definition, proceed like this:

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
  vdu:
  - ...
   cloud-init-file: cloud_init_filename
```

Its content can have a number of formats, including #cloud-config and bash. For example, any of the following scripts sets a password in Linux.

```
#cloud-config
hostname: lb_vdu
password: osm2018
```

```
chpasswd: { expire: False }
ssh_pwauth: True
```

```
#cloud-config
hostname: lb_vdu
chpasswd:
 list: |
   ubuntu:osm2018
 expire: False
```

Additional information about cloud-init can be found in this documentation.

Parameterizing Cloud-init files

Beginning in OSM version 5.0.3, cloud-init files can be parameterized by using double curly brackets. For example:

```
#cloud-config
hostname: lb_vdu
password: {{ password }}
chpasswd: { expire: False }
ssh_pwauth: True
```

Such variables can then be passed at instantiation time by referring the VNF index it applies to, as well as the name and value of the variable.

```
osm ns-create ... --config "{additionalParamsForVnf: [{member-vnf-index: '1', additionalParams:{pass
word: 'secret'}}]}"
```

When dealing with multiple variables, it might be useful to pass a YAML file instead.

```
osm ns-create ... --config-file vars.yaml
```

Please note that variable naming convention follows Jinja2 (Python identifiers), so hyphens are not allowed.

Support for Configuration Drive

Besides cloud-init being provided as userdata through a metadata service, some VNFs will require storing the metadata locally on a configuration drive.

The support for this is available at the VNFD model, as follows:

```
supplemental-boot-data:
   boot-data-drive: 'true'
```

Guidelines for EPA requirements

Most EPA features can be specified at the VDU descriptor level as requirements in the guest-epa section, which will be then translated to the appropriate request through the VIM connector. Please note that the NFVI should be pre-configured to support these EPA capabilities.

Huge Pages

Huge pages are requested as follows:

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
   vdu:
   -  ...
      guest-epa:
         mempage-size: LARGE
         ...
```

The mempage-size attribute can take any of these values:

- LARGE: Require hugepages (either 2MB or 1GB)
- SMALL: Doesn’t require hugepages
- SIZE_2MB: Requires 2MB hugepages
- SIZE_1GB: Requires 1GB hugepages
- PREFER_LARGE: Application prefers hugepages

CPU Pinning

CPU pinning allows for different settings related to vCPU assignment and hyper threading:

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
   vdu:
   -  ...
      guest-epa:
         cpu-pinning-policy: DEDICATED
         cpu-thread-pinning-policy: AVOID
         ...
```

CPU pinning policy describes the association between virtual CPUs in the guest and the physical CPUs in the host. Valid values are:

- DEDICATED: Virtual CPUs are pinned to physical CPUs
- SHARED: Multiple VMs may share the same physical CPUs.
- ANY: Any policy is acceptable for the VM

CPU thread pinning policy describes how to place the guest CPUs when the host supports hyper threads. Valid values are:

- AVOID: Avoids placing a guest on a host with threads.

- SEPARATE: Places vCPUs on separate cores, and avoids placing two vCPUs on two threads of the same core.
- ISOLATE: Places each vCPU on a different core, and places no vCPUs from a different guest on the same core.
- PREFER: Attempts to place vCPUs on threads of the same core.

NUMA Topology Awareness

This policy defines if the guest should be run on a host with one NUMA node or multiple NUMA nodes.

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
  vdu:
  -  ...
    guest-epa:
      numa-node-policy:
        node-cnt: 2
        mem-policy: STRICT
        node:
          - id: 0
            memory-mb: 2048
            num-cores: 1
          - id: 1
            memory-mb: 2048
            num-cores: 1
      ...
```

node-cnt defines the number of NUMA nodes to expose to the VM, while mem-policy defines if the memory should be allocated strictly from the 'local' NUMA node (STRICT) or not necessarily from that node (PREFERRED). The rest of the settings request a specific mapping between the NUMA nodes and the VM resources and can be explored in detail in the OSM Information Model Documentation

SR-IOV and PCI-Passthrough

Dedicated interface resources can be requested at the VDU interface level.

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
  vdu:
  -  ...
    interface:
    -  name: eth0
       position: '1'
```

```
          type: EXTERNAL
          virtual-interface:
            type: SR-IOV
```

Valid values for type, which specifies the type of virtual interface between VM and host, are:

- PARAVIRT : Use the default paravirtualized interface for the VIM (virtio, vmxnet3, etc.).
- PCI-PASSTHROUGH : Use PCI-PASSTHROUGH interface.
- SR-IOV : Use SR-IOV interface.
- E1000 : Emulate E1000 interface.
- RTL8139 : Emulate RTL8139 interface.
- PCNET : Emulate PCNET interface.

Managing alternative images for specific VIM types

The image name specified at the VDU level is expected to be either located at the images folder within the VNF package, or at the VIM catalogue.

Alternative images can be specified and mapped to different VIM types, so that they are used whenever the VNF package is instantiated over the given VIM type. In the following example, the ubuntu1604 image is used by default (any VIM), but a different image is used if the VIM type is AWS.

```
vnfd:vnfd-catalog:
  vnfd:
  - ...
    vdu:
    -   ...
        image: ubuntu1604
        alternative-images:
        -   vim-type: aws
            image: ubuntu/images/hvm-ssd/ubuntu-artful-17.10-amd64-server-20180509
```

Updating and testing instantiation of the VNF package

Once the VNF Descriptor has been updated with all the Day 0 requirements, its folder needs to be repackaged. For example, in Linux/UNIX, it would be something like: tar -cvfz vLB_vnfd.tar.gz vLB_vnfd/

A Network Service package containing at least this single VNF needs to be used to instantiate the VNF. This could be generated with the *devops tool* described earlier.

Remember the objectives of this phase: 1. Instantiating the VNF with all the required VDUs, images, initial (unconfigured) state and NFVI requirements. 2. Making the VNF manageable from OSM (OSM should have SSH access to the management interfaces, for example)

To test this out, the NS can be launched using the OSM client, like this:

osm ns-create --ns_name [ns name] --nsd_name [nsd name] --vim_account [vim name] \
--ssh_keys [comma separated list of public key files to inject to vnfs]

At launch time, extra ***instantiation parameters*** can be passed so that the VNF can be adapted to the particular instantiation environment or to achieve a proper inter-operation with other VNFs within the specific NS. More information about these parameters will be revised during the next chapter as part of Day 1 objectives, or can be reviewed here.

The following sections will provide details on how to further populate the VNF Package to automate Day 1/2 operations.

Day 1: VNF Service initialization

Description of this phase

The objective of this section is to provide the guidelines to include all necessary elements in the VNF Package. This allows the exposed services inside the VNF to be automatically initialized right after the VNF instantiation.

The main mechanism to achieve this in OSM is to build a Proxy Charm and include it in the descriptor.

Day 1 Onboarding guidelines

Adding Day 1 primitives to the descriptor

This type of initial action will run automatically after instantiation and should be specified in the VNF descriptor. These can be defined at two different levels:

- VDU-level: for a specific VDU, used when a VDU needs configuration, which is different than the VDU used for managing the VNF.
- VNF-level: for the "management VDU", used when the configuration applies to the VDU exposing an interface for managing the whole VNF.

Initial primitives must include a primitive named config that passes information for OSM VCA to be able to authenticate and run further primitives into the VNF. The *config primitive* should provide, at least, the following parameters:

- ssh-hostname: Typically used with the "rw_mgmt_ip" variable, which is automatically replaced by the VNF or VDU management IP address specified in the corresponding section.
- ssh-username: The username used for authentication with the VDU.
- ssh-password or ssh-public-key: A static password (not recommended unless it is changed afterwards) or a public-key from the OSM host. [TODO: confirm that if none is provided, there is an automatic injection with feature 1429?]

In addition to the *config primitive*, more initial primitives can be run in the desired order so that the VNF initializes its services. Note that each of these additional actions will be detailed later in the proxy charm that implements them.

The following example shows VNF-level initial primitives: both the expected *config* primitive in the beginning and also the *configure-remote* and *start-service* to be run in addition right after initialization.

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
  mgmt-interface:
    cp: vnf-cp0
  ...
  vnf-configuration:
```

```
initial-config-primitive:
-  name: config
   parameter:
   -  name: ssh-hostname
      value: <rw_mgmt_ip>
   -  name: ssh-username
      value: admin
   -  name: ssh-password
      value: secretpassword
   seq: '1'
-  name: configure-remote
   parameter:
   -  name: dest-ip
      value: 10.1.1.1
   seq: '2'
-  name: start-service
   seq: '3'
juju:
   charm: samplecharm
```

Instantiation parameters can be used to define the values of these parameters at a later time, during the NS instantiation. The following example shows a VDU-level parameter with variables. Note that when using VDU-level primitives, an interface must be specified as the "management interface" for that specific VDU.

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
  vdu:
  - ...
   interface:
   - external-connection-point-ref: vdu1_mgmt
     mgmt-interface: true
     ...
     vdu-configuration:
        initial-config-primitive:
        -  name: config
           parameter:
           -  name: ssh-hostname
              value: <rw_mgmt_ip>
           -  name: ssh-username
              value: admin
           -  name: ssh-password
              value: <password>
           seq: '1'
        - name: configure-remote
          parameter:
          - name: dest-ip
            value: <destination_ip>
          seq: '2'
        - name: start-service
          seq: '3'
        juju:
          charm: samplecharm
```

The values for the variables used at the primitive level are defined during instantiation, just like in the cloud-init case:

```
osm ns-create ... --config "{additionalParamsForVnf: [{member-vnf-index: '1', additionalParams:{password: 'secretpassword', destination_ip: '10.1.1.1'}}]}"
```

Remember that when dealing with multiple variables, it might be useful to pass a YAML file instead.

```
osm ns-create ... --config-file vars.yaml
```

Building a proxy charm

The charm is the element that implements the primitives or, put in other words, it provides the logic for the different configurations on the VNF. It is defined inside the “charms” folder of the VNF package. The contents of the charm must be built from a Linux machine.

Method 1: Building a proxy charm the traditional way

1. Install the charm tools and set up your environment. You might want to copy the export lines to your “~/.bashrc” profile file to automatically load them in the next session.

```
snap install charm
mkdir -p ~/charms/layers
export JUJU_REPOSITORY=~/charms
export LAYER_PATH=$JUJU_REPOSITORY/layers
cd $LAYER_PATH
```

b) A proxy charm includes, by default, the “VNF” and “basic” layers, which take care of the initial SSH connection to the VNF. Create the new personalized *layer* for your proxy charm:

```
charm create samplecharm
cd samplecharm
```

Note: Charm names do not support underscores.

c) Modify the basic files like this:

```
# layer.yaml file
includes:
  - layer:basic
  - layer:vnfproxy
```

```
# metadata.yaml file
name: samplecharm
summary: this is an example
maintainer: Gianpietro Lavado <gianpietro1@gmail.com>
description: |
  This is an example of a proxy charm deployed by Open Source Mano.
tags:
  - nfv
subordinate: false
series:
    - trusty
    - xenial
```

d) Modify the "actions.yaml" file, adding all actions/primitives and their parameters. Note that the value of these parameters is defined at the VNFD, either statically or by using variables, as explained earlier.

```
# actions.yaml file
configure-remote:
    description: "Configures the remote server"
    params:
        destination_ip:
            description: "IP of the remote server"
            type: string
            default: ""
start-service:
    description: "Starts the service of the VNF"
```

e) Create an "actions" folder and populate it with files representing each action. Filenames should match the name of the primitive, should be made executable with chmod +x and all must contain the following exact content.

```
# actions/configure-remote and actions/start-service files
cat <<'EOF' >> actions/set-server
#!/usr/bin/env python3
import sys
sys.path.append('lib')

from charms.reactive import main
from charms.reactive import set_state
from charmhelpers.core.hookenv import action_fail, action_name

"""
`set_state` only works here because it's flushed to disk inside the `main()`
loop. remove_state will need to be called inside the action method.
```

```
"""
set_state('actions.{}'.format(action_name()))

try:
    main()
except Exception as e:
    action_fail(repr(e))
EOF
```

f) Open the respective file at the 'reactive/' folder. This will be used to code, in Python, the actual actions that will run through SSH when each primitive is triggered. Note that any variable can be recovered in two ways:

- Using the config() function if the variable belongs to that specific primitive.
- Using the action_get('name-of-parameter') function to get any other parameter.

The following example provides an idea of the contents of a reactive file.

```
# reactive/samplecharm.py file
@when('actions.configure-remote')
def configure_remote():
    err = ''
    # Variables should be retrieved, if needed
    cfg = config()
    mgmt_ip = cfg['ssh-hostname']
    destination_ip = action_get('dest-ip')
    try:
        # Commands to be run through SSH should go here
        cmd = "vnfcli set license " + mgmt_ip + " server " + destination_ip
        result, err = charms.sshproxy._run(cmd)
    except:
        action_fail('command failed:' + err)
    else:
        action_set({'output': result})
    finally:
        remove_flag('actions.configure-remote')

@when('actions.start-service')
def start_service():
    err = ''
    # Variables should be retrieved, if needed
    try:
        # Commands to be run through SSH should go here
        cmd = "sudo service vnfoper start"
        result, err = charms.sshproxy._run(cmd)
    except:
        action_fail('command failed:' + err)
    else:
        action_set({'output': result})
    finally:
        remove_flag('actions.start-service')
```

g) If your proxy charm layer needs some extra dependencies, the debian or pip package should be added to the layer.yaml file. This is done through the 'packages' and 'python_packages' options inside the layer, for example:

```
includes:
- "layer:basic"
- "layer:ansible-base"
- "layer:vnfproxy"
options:
  basic:
    use_venv: false
    packages: ["build-essential","libssl-dev"]
    python_packages: ["pyyaml"]
```

h) Finally, build the charm with charm build and copy the resulting folder (in this case the ~/charms/builds/simplecharm directory) inside the charms folder of your VNF Package.

Further information about building charms can be found here.

Method 2: Using proxy charm generators

To date, the only supported generator is Ansible, which means that a proxy charm can be automatically populated based on an Ansible Playbook.

A sample Ansible playbook would look like this:

```
# This sample playbook applies to a VyOS router
# The hosts where the playbook will be executed will automatically contain the IP address of the VDU,
 in a /etc/ansible/hosts file at the charm container
- hosts: vyos-routers
  # note that the following setting is needed in most cases
  connection: local
  tasks:
  - name: configure the remote device
    vyos_config:
      lines:
        - set nat destination rule 1 inbound-interface eth0
        - set nat destination rule 1 destination port 80
        - set nat destination rule 1 protocol tcp
        - set nat destination rule 1 translation address {{dest_ip}}
```

Once the Ansible playbook has been tested against your VNF, the procedure to incorporate it in a charm is as follows:

1. Create your environment and charm in the traditional way (that is, steps (a) and (b) from the previous method)
2. Clone the devops repository elsewhere and copy the generator files to your charm root folder. For example: cp -r ~/devops/descriptor-packages/tools/charm-generator/* ./ [TODO: migrate to binary]
3. Install the dependencies of the generator with sudo pip3 install -r requirements.txt
4. Run the generator, which will populate all the required files automatically. It requires an Ansible playbook to be copied inside a new "playbooks" folder under the charm root directory, and the primitive to be named "playbook" (by default). The following example runs the generator with the minimal options.

```
python3 generator-runner.py --ansible --summary "Configures VNF using Ansible" \
--maintainer "Gianpietro Lavado <glavado@whitestack.com>" \
--description "Configures VNF using Ansible"
```

e) Adjust the "reactive" file as desired, for example, if you wish to pass some parameters to your playbook (which supports Jinja)

```
@when('actions.playbook')
def playbook():
  try:

    # getting a variables from the config primitive
    cfg = config()
    mgmt_ip = cfg['ssh-hostname']
    # a sample on passing a specific file to the playbook, considering that the charm will be located a
t the /var/lib/juju/agents folder
    config_file = charms.libansible.find('config.conf', '/var/lib/juju/agents/')

    # populating an object with the variables
    dict_vars = {'dest_ip': mgmt_ip,'config_file': config_file}

    # running the playbook along with the given variables
    result = charms.libansible.execute_playbook('playbook.yaml', dict_vars)
  except:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    err = traceback.format_exception(exc_type, exc_value, exc_traceback)
    action_fail('playbook failed: ' + str(err))
  else:
    action_set({'output': result})
    finally:
        remove_flag('actions.playbook')
```

f) Finally, build the charm with charm build and copy the resulting folder (in this case the "~/charms/builds/simplecharm" directory) inside the "charms" folder of your VNF Package.

Once the VNF is launched, the results from running the generator will be found inside the proxy charm *lxc* container, at the "/var/log/ansible.log" file. If not successful, it could indicate the need for other possible modifications which are applicable for certain VNFs.

Note: some VNFs will not pass some SSH pre-checks that Ansible performs in some operations (SFTP, SCP, etc.) In those cases, it has been noted that ansible_connection=ssh, which is a default set of the generator, needs to be disabled. This preset would need to be deleted from the lib/charms/libansible.py file, create_hosts function. [TODO: explore an enhancement to the Ansible Generator, to be as generic as possible]

Testing instantiation of the VNF package

Remember the objective of this phase: **to configure the VNF automatically so it starts providing the expected service**.

To test this out, the NS can be launched using the OSM client, like this:

```
osm ns-create --ns_name [ns name] --nsd_name [nsd name] --vim_account [vim name] --ssh_keys [c
omma separated list of public key files to inject to vnfs]
```

Furthermore, and as mentioned earlier, extra ***instantiation parameters*** can be passed so that the VNF can be adapted to the particular instantiation environment or to achieve a proper inter-operation with other VNFs in the specific NS.

For example, if using IP Profiles to predefine subnet values, a specific IP address could be passed to an interface like this:

```
osm ns-create ... --config '{vnf: [ {member-vnf-index: "1", internal-vld: [ {name: internal, ip-profile: {...
}, internal-connection-point: [{id-ref: id1, ip-address: "a.b.c.d"}] ] } ],
  additionalParamsForVnf...}'
```

When dealing with multiple fixed IP addresses, variables or other additions to the original descriptor, it might be useful to pass a YAML file instead.

```
osm ns-create ... --config-file ip-vars.yaml
```

As you can see, the parameters being defined during instantiation follow the information model structure. Further information and examples about these parameters can be reviewed here.

After deployment is done, proxy charms can be monitored and debugged by using the juju status and juju debug-log commands, respectively. If proxy charms need to be started in any particular order, please note that the order of proxy charm initialization follows the order in which 'constituent VNFs' are listed at the NSD, but the actual operations could be executed in a different order, depending on the time it takes for each proxy charm container to be ready.

Day 2: VNF Runtime operations

Description of this phase

The objective of this section is to provide the guidelines for including all the necessary elements in the VNF Package so that it can be operated at runtime and therefore, re-configured on demand at any point by the end-user. Typical operations include re-configuration of services, KPI monitoring and the enablement of automatic, closed-loop operations triggered by monitored status.

The main mechanism to achieve re-configuration in OSM is to build a proxy charm and include it in the descriptor. On the other hand, monitoring and VNF-specific policy management can be achieved by specifying the requirements at the descriptor (modifying monitored indicators and policies at runtime is not supported in OSM as of version 5.0.5)

Day 2 Onboarding guidelines

Adding Day 2 primitives to the descriptor

Day-2 primitives are actions invoked on demand, so the config-primitive block is used instead of the initial-config-primitive block at the VNF or VDU level.

For example, a VNF-level set of Day 2 primitives would look like this:

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
   mgmt-interface:
      cp: vnf-cp0
   ...
   vnf-configuration:
      ...
      config-primitive:
      -   name: restart-service
          parameter:
          -   name: offset
              default-value: 10
              data-type: STRING
      -   name: clean-cache
          parameter:
          -   name: force
              default-value: true
              data-type: BOOLEAN
      juju:
          charm: samplecharm
```

Building a proxy charm

Proxy charms for implementing Day 2 primitives are built exactly in the same way as when implementing Day 1 primitives.

Adding monitoring parameters

Collecting NFVI metrics

In order to collect NFVI-level metrics associated to any given VDU and store them in the OSM TSDB (using Prometheus software), a set of monitoring-params should be declared both globally and at the VDU level.

Only CPU and Memory are supported as of OSM version 5.0.5. For example:

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
   vdu:
   - id: "apache_vdu"
     ...
     monitoring-param:
       - id: "apache_cpu_util"
         # nfvi-metric name should match the supported set of metrics collectable by OSM through the VI
M connectors
         nfvi-metric: "cpu_utilization"
       - id: "apache_memory_util"
         nfvi-metric: "average_memory_utilization"
   ...
   monitoring-param:
   -   id: "apache_vnf_cpu_util"
       name: "apache_vnf_cpu_util"
       # only 'AVERAGE' aggregation is supported at this time
       aggregation-type: AVERAGE
       vdu-monitoring-param:
         # vdu-ref should match the id of the VDU
         vdu-ref: "apache_vdu"
         # vdu-monitoring-param-ref should match the nfvi-metric id
         vdu-monitoring-param-ref: "apache_cpu_util"
   -   id: "apache_vnf_memory_util"
       name: "apache_vnf_memory_util"
       aggregation-type: AVERAGE
       vdu-monitoring-param:
         vdu-ref: "apache_vdu"
         vdu-monitoring-param-ref: "apache_memory_util"
```

Collecting VNF indicators

As of OSM version 5.0.5, collection of VNF indicators is done by using proxy charms with the *metrics layer*. This is a simple method that has a couple of limitations:

- Metrics are collected every five minutes and this can't be changed.
- Only positive decimal values of *gauge* or *absolute* types can be collected.

At the charm level, the only file that needs to be created before building it is the “metrics.yaml” file at the root folder of the charm.

For example, the following file collects *active users* and *loads* values from a Linux machine.

```
# metrics.yaml file
metrics:
  users:
    type: gauge
    description: "# of users"
    command: who|wc -l
  load:
    type: gauge
    description: "5 minute load average"
    command: cat /proc/loadavg |awk '{print $1}'
```

More information on how to populate this file can be found in the Juju developer metrics documentation.

Once the charm has been created and included in the VNF Package, the descriptor needs to define the metrics to actually be collected by OSM. As with any charm, this can be done at a VNF or VDU level.

For example, at the VNF level (a VDU that represents the VNF):

```
vnfd:vnfd-catalog:
  vnfd:
  - ...
    mgmt-interface:
        cp: vnf-cp0
    ...
    vnf-configuration:
      ...
      juju:
        # this is the name of the proxy charm
        charm: metricscharm
      metrics:
        # metric names should match the ones specified at the metrics.yaml file
        - name: users
        - name: load
    ...
    monitoring-param:
    -   id: "ubuntuvdu_users"
        name: "ubuntuvdu_users"
        aggregation-type: AVERAGE
        vnf-metric:
          # vnf-metric-name-ref should match the metric name specified at VNF/VDU level
        vnf-metric-name-ref: "users"
    -   id: "ubuntuvdu_load"
        name: "ubuntuvdu_load"
        aggregation-type: AVERAGE
        vnf-metric:
          vnf-metric-name-ref: "load"
```

This other example does the same, but at a specific VDU level:

```
vnfd:vnfd-catalog:
  vnfd:
  - ...
    vdu:
    - ...
      interface:
      # remember that for any VDU that runs a charm, a management interface needs to be specified
      - external-connection-point-ref: vdu1_mgmt
        mgmt-interface: true
      ...
      vdu-configuration:
        ...
        juju:
          charm: metricscharm
        metrics:
          - name: users
          - name: load
    ...
    monitoring-param:
    -   id: "ubuntuvdu_users"
        name: "ubuntuvdu_users"
        aggregation-type: AVERAGE
        vdu-metric:
          vdu-ref: "ubuntuvdu1"
          vdu-metric-name-ref: "users"
    -   id: "ubuntuvdu_load"
        name: "ubuntuvdu_load"
        aggregation-type: AVERAGE
        vdu-metric:
          vdu-ref: "ubuntuvdu1"
          vdu-metric-name-ref: "load"
```

Adding scaling operations

Scaling operations happen at a VDU level and can be added with automatic triggers (*closed-loop* mode triggered by *monitoring-param* thresholds), or with a manual trigger.

In both cases, a scaling-group-descriptor section must be added to the VNF descriptor. The following example enables VDU scaling based on a manual trigger (OSM API or CLI).

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
  scaling-group-descriptor:
  -  name: "apache_vdu_manualscale"
     # the following counts refer to "scaled instances" only
     min-instance-count: 0
     max-instance-count: 10
     scaling-policy:
     -  name: "manual_policy"
        scaling-type: "manual"
     vdu:
     -  vdu-id-ref: apache_vdu
        count: 1
```

The following example defines a closed-loop scaling operation based on a specific monitoring parameter threshold.

```
vnfd:vnfd-catalog:
 vnfd:
 - ...
  scaling-group-descriptor:
  -  name: "apache_vdu_autoscale"
     min-instance-count: 0
     max-instance-count: 10
     scaling-policy:
     -  name: "apache_cpu_util_above_threshold"
        scaling-type: "automatic"
        threshold-time: 10
        cooldown-time: 120
        scaling-criteria:
        -  name: "apache_cpu_util_above_threshold"
           # this is the name of the monitoring-param to monitor
           vnf-monitoring-param-ref: "apache_vnf_cpu_util"
           # scale-in threshold
           scale-in-threshold: 20
           scale-in-relational-operation: "LT"
           # scale-out threshold
           scale-out-threshold: 80
           scale-out-relational-operation: "GT"
     vdu:
```

```
    - vdu-id-ref: apache_vdu
      count: 1
```

More information about scaling can be found in the OSM Autoscaling documentation

Testing instantiation of the VNF package

Each of the objectives of this phase can be tested as follows:

- **Enabling a way of re-configuring the VNF on demand**: primitives can be called through the OSM API, dashboard, or directly by running the following OSM client command: osm ns-action [ns-name] --vnf_name [vnf-index] --action_name [primitive-name] --params '{param-name-1: "param-value-1", param-name-2: "param-value-2", ...}
- **Monitor the main KPIs of the VNF**: if correctly enabled, metrics will automatically start appearing in the OSM Prometheus database. More information on how to access, visualize and troubleshoot metrics can be found in the OSM Performance Management documentation
- **Enabling scaling operations**: automatic scaling should be tested by making the metric reach the corresponding threshold, while manual scaling can be tested by using the following command (which also works when the “scaling-type” has been set to “automatic”): osm vnf-scale [ns-name] [vnf-name] --scaling-group [scaling-group name] [--scale-in|--scale-out]

Known issues

OSM Release FIVE

v5.0.5

- Instantiation “additional” parameters can’t be passed with the config object through the dashboard. Being addressed by this fix.
- Cloud-init will not allow parameter names with hyphens due to Jinja2 (Python Identifiers) implementation, but the error is misleading. Being addressed through this fix.
- Proxy charms may not be deleted after NS is terminated, making the NS fail to be deleted. Workaround is to delete the LXC container with juju remove-machine command, then force the deletion of the NS. Being addressed by this fix.

ETSI
06921 Sophia Antipolis CEDEX, France
Tel +33 4 92 94 42 00
info@etsi.org
www.etsi.org