



ONEedge.io

A Software-defined Edge Computing Solution

---

# D3.1. Software Report - a

Software Report v.1.0

31 July 2020

## Abstract

This report summarizes the design of the technology components that have been implemented as part of the First Innovation Cycle (M4-M9), as well as the full details of each of the software requirements that are being addressed as part of the development of such components. For each Software Requirement, this document provides a full description, a list of detailed requirements and specifications, a description of its architecture and components, the data model, and relevant changes applied to the API and Interfaces.



Copyright © 2020 OpenNebula Systems SL. All rights reserved.



This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No 880412.



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.



## Deliverable Metadata

<b>Project Title:</b>	A Software-defined Edge Computing Solution
<b>Project Acronym:</b>	ONEedge
<b>Call:</b>	H2020-SMEInst-2018-2020-2
<b>Grant Agreement:</b>	880412
<b>WP number and Title:</b>	WP3. Product Innovation
<b>Nature:</b>	R: Report
<b>Dissemination Level:</b>	PU: Public
<b>Version:</b>	1.0
<b>Contractual Date of Delivery:</b>	31/7/2020
<b>Actual Date of Delivery:</b>	31/7/2020
<b>Lead Authors:</b>	Vlastimil Holer, Rubén S. Montero and Constantino Vázquez
<b>Authors:</b>	Sergio Betanzos, Ricardo Díaz, Christian González, Alejandro Huertas, Jorge M. Lobo, Ángel L. Moya, Jan Orel, Petr Ospaly and Cristina Palacios
<b>Status:</b>	Submitted

## Document History

Version	Issue Date	Status <sup>1</sup>	Content and changes
1.0	31/7/2020	Submitted	First final version of the report

<sup>1</sup> A deliverable can be in one of these stages: Draft, Peer-Reviewed, Submitted and Approved.



## Executive Summary

The purpose of deliverable D3.1 is to offer a summary of the design of the technology components that have been implemented in the First Innovation Cycle (M4-M9), as well as to provide the full details of each of the software requirements that are being addressed as part of the development of such components.

For each Software Requirement, this document provides a full description, a list of detailed requirements and specifications, a description of its architecture and components, the data model, and relevant changes applied to the API and Interfaces.

During the First Innovation Cycle (M4-M9), the project mostly focused on those software requirements needed to achieve our first milestone in M9, which is the base functionality needed for a single-host edge deployment.

The work carried out during this First Innovation Cycle involved software requirements from components CPNT1, CPNT2, CPNT3, CPNT4 and CPNT5, with a special focus on laying the technological foundation of ONEedge. These are some of the main new features that have been implemented as part of this process:

- Development of a new tool to achieve fully automatic EdgeNebula upgrades.
- A new driver to interact with Firecracker VMM.
- Redesign and implementation of a new monitoring system.
- Improvement of the network interface with VMware services.
- Extended functionality for NUMA support.
- OneFlow being re-written to improve scalability and response time.
- Improvement of the Graphical User Interface (Sunstone).
- Improvement of the Amazon EC2 and Packet drivers.
- Several extensions of the Infrastructure Provision and Deployment tools.
- Better support for deploying a Kubernetes cluster.
- New integration with the Docker Hub marketplace.
- New version of the Kubernetes appliance.



## Table of Contents

<b>1. Edge Instance Manager (CPNT1)</b>	<b>5</b>
[SR1.2] Automatic Product Upgrade	5
<b>2. Edge Workload Orchestration and Management (CPNT2)</b>	<b>16</b>
[SR2.1] Integration with Serverless Hypervisor	16
[SR2.3] Secure and Scalable Distributed Monitoring	21
[SR2.5] Integration with Remote VMware vCenter Service	27
[SR2.6] VNF Support	30
[SR2.8] Complete Service Flows	35
[SR2.9] Web UI extensions	44
<b>3. Edge Provider Selection (CPNT3)</b>	<b>45</b>
[SR3.4] Driver Maintenance Process	45
<b>4. Edge Infrastructure Provision and Deployment (CPNT4)</b>	<b>49</b>
[SR4.1] Reliable Edge Resource Provision	49
[SR4.2] Usability, Functionality and Scalability of Provision	53
[SR4.3] Provision Template for Reference Architectures	57
<b>5. Edge Apps Marketplace (CPNT5)</b>	<b>62</b>
[SR5.2] Built-in Management of Application Containers Engine	62
[SR5.3] Integration with Application Containers Marketplace	64
[SR5.4] New Edge Applications Marketplace Entries	69



# 1. Edge Instance Manager (CPNT1)

## [SR1.2] Automatic Product Upgrade

### Description

The OpenNebula (EdgeNebula) upgrades were always relatively simple to manage, as most of the steps could be automated - product packages or database schema upgrades. The problematic part of upgrades is updating the configuration files for newer versions. EdgeNebula comes with more than 70 individual configuration files, through which administrators can adjust the behaviour of various parts. Every custom change in a configuration file in the old version has to be reevaluated again on upgrade and manually applied to the new version of file. Moreover, the migration of custom change does not have to be straightforward as syntax or semantic of configuration files changes during the time.

To be able to achieve the fully automatic EdgeNebula upgrades, we had to implement a missing mechanism for automatic upgrades of **configuration files** to the new versions. This new mechanism deals with administrators' custom changes and migrates them to the new version of configuration files fully automatically. Problematic or conflicting customizations are reported back to the administrators to resolve them semi-automatically.

New configuration upgrade mechanism is provided via a new dedicated CLI tool `onecfg` and is part of EdgeScape.

### Requirements and Specifications

#### Files Types Classification

It is necessary to review all existing configurations we use and their file formats to understand what and how to automatically upgrade. Following 4 file formats were identified:

- YAML (with OR without strict order in arrays)
- Shell with only variables declaration
- OpenNebula specific INI-like format
- XML

Overview of existing files and their formats provide Table 1.1.

FILE	FORMAT TYPE
<code>/etc/one/auth/ldap_auth.conf</code>	YAML strict
<code>/etc/one/auth/server_x509_auth.conf</code>	YAML
<code>/etc/one/auth/x509_auth.conf</code>	YAML
<code>/etc/one/az_driver.conf</code>	YAML
<code>/etc/one/az_driver.default</code>	XML
<code>/etc/one/cli/*.yaml</code>	YAML strict
<code>/etc/one/defaultrc</code>	Shell
<code>/etc/one/ec2_driver.conf</code>	YAML
<code>/etc/one/ec2_driver.default</code>	XML
<code>/etc/one/ec2query_templates/*.erb</code>	XML
<code>/etc/one/econe.conf</code>	YAML
<code>/etc/one/hm/hmrc</code>	Shell
<code>/etc/one/monitord.conf</code>	OpenNebula specific
<code>/etc/one/oned.conf</code>	OpenNebula specific



/etc/one/oneflow-server.conf	YAML
/etc/one/onegate-server.conf	YAML
/etc/one/onehem-server.conf	YAML
/etc/one/packet_driver.default	XML
/etc/one/sched.conf	OpenNebula specific
/etc/one/sunstone-logos.yaml	YAML strict
/etc/one/sunstone-server.conf	YAML
/etc/one/sunstone-views.yaml	YAML
/etc/one/sunstone-views/**/*.*.yaml	YAML
/etc/one/tmrc	Shell
/etc/one/vcenter_driver.conf	YAML
/etc/one/vcenter_driver.default	XML
/etc/one/vmm_exec/vmm_exec_kvm.conf	OpenNebula specific
/etc/one/vmm_exec/vmm_exec_vcenter.conf	OpenNebula specific
/etc/one/vmm_exec/vmm_execrc	Shell
/var/lib/one/remotes/datastore/ceph/ceph.conf	Shell
/var/lib/one/remotes/etc/datastore/ceph/ceph.conf	Shell
/var/lib/one/remotes/etc/datastore/fs/fs.conf	Shell
/var/lib/one/remotes/etc/im/firecracker-probes.d/probe_db.conf	YAML
/var/lib/one/remotes/etc/im/kvm-probes.d/pci.conf	YAML
/var/lib/one/remotes/etc/im/kvm-probes.d/probe_db.conf	YAML
/var/lib/one/remotes/etc/im/lxd-probes.d/pci.conf	YAML
/var/lib/one/remotes/etc/im/lxd-probes.d/probe_db.conf	YAML
/var/lib/one/remotes/etc/market/http/http.conf	Shell
/var/lib/one/remotes/etc/tm/fs_lvm/fs_lvm.conf	Shell
/var/lib/one/remotes/etc/vmm/firecracker/firecrackerrc	YAML
/var/lib/one/remotes/etc/vmm/kvm/kvmrc	Shell
/var/lib/one/remotes/etc/vmm/lxd/lxdrc	YAML
/var/lib/one/remotes/etc/vmm/vcenter/vcenterrc	YAML
/var/lib/one/remotes/etc/vnm/OpenNebulaNetwork.conf	YAML
/var/lib/one/remotes/vmm/kvm/kvmrc	Shell
/var/lib/one/remotes/vnm/OpenNebulaNetwork.conf	YAML

**Table 1.1.** List of EdgeNebula configuration files and their file types

### Configuration Files Manipulation

To be able to implement automatic configuration files upgrade, we need to be able to manipulate each configuration file (read, update and write changes) and preserve the structure (and comments) as much as possible.

While reading the configuration files is not a problem, updating them and storing the changes back is not that common for the configuration file formats we use.

The abstraction classes to manipulate each file format are implemented within (OneScope::Config::Type namespace):

- Simple - for plain files (also use for XML)
- Augeas - generic class to manage files via Augeas (<https://augeas.net>)
- Augeas::ONE - for OpenNebula specific format via Augeas interface
- Augeas::Shell - for Shell format via Augeas Interface
- Yaml - for YAML serialized data structures
- Yaml::Strict - for YAML serialized data with strict ordered arrays

This allows to implement the most suitable approach for each file format to read, update and write files. Each class provides a set of methods to manipulate the files as shown on Table 1.2.



METHOD	DESCRIPTION
initialize([NAME])	Class constructor with optional file NAME (not loaded by default)
content	Accessor method into in-memory data structure representing file content.
load([NAME])	Load from file NAME content into memory
save([NAME])	Save memory content to file
delete	Delete file from disk (based on NAME provided on load)
exists?([NAME])	Check if file NAME exists
copy(OBJECT)	Copy configuration content from another OBJECT into the current object.
to_s	Serialize memory configuration content as string in file format
same?(OBJECT)	Compares current configuration with configuration from another OBJECT
similar?(OBJECT)	Compares current configuration with configuration from another OBJECT
diff(OBJECT)	Provides a list of differences between current and another OBJECT
patch(DIFF, MODE)	Apply list of differences from DIFF to current object
hintings(DIFF, [REP])	Format list of different parts for human beings

**Table 1.2.** Configuration class methods

Each configuration file must use an appropriate class matching its file format. First object needs to be instantiated and a particular file loaded.

All manipulation with content is over content method accessor, which provides raw data (e.g., Array or Hash structure for YAML formats, String for plain files) or Augeas object.

When changes are done, configuration object content is saved into a file.

Example configuration file manipulation from Ruby code is presented on Figure 1.1:

```
#!/usr/bin/ruby

require 'bundler/setup'
require 'onescape'

# create object to manage YAML configuration file
cfg = OneScape::Config::Type::Yaml.new('/etc/one/sunstone-views.yaml')

# load / update / save configuration file only if exists
if cfg.exists?
  cfg.load
  cfg.content['default'] << ['user', 'cloud']
  cfg.save
end
```

**Figure 1.1.** Example code of configuration file manipulation



### Automatic Change Management

We expect that the majority of changes among different versions of the same file can be automatically detected and applied.

To achieve this approach, we need to be able to compare 2 versions of the same file format (A and B), identify differences between A and B files and propose individual steps needed to be done on file A to get into content equivalent with B. Also we must be able to apply such steps of differences on A to get into state B.

The approach is similar to what is known in the Unix world with command line tools diff and patch for managing changes in plain files. For all OpenNebula configuration file types we implement such functionality for:

- **diff** (method diff) to provide differences between 2 configuration objects
- **patch** (method patch) to apply differences on A to get into state equivalent to B

As an extra safety validation feature and to be able to replace files with new ones in cases there only minor user change (e.g., in comment), we also implement comparison functions to detect which files are:

- **same** (method same?) - same on block level
- **similar** (method similar?) - have same semantic, but are NOT same on block level

Example use of identification of differences between 2 configuration files and application of the differences to the file is presented on Figure 1.2.

```
#!/usr/bin/ruby

require 'bundler/setup'
require 'onescape'

# object to manage YAML configuration
cfg1 = OneScape::Config::Type::Augeas::ONE.new('/etc/one/oned.conf')
cfg2 = OneScape::Config::Type::Augeas::ONE.new('/etc/one/oned.conf-new')

# load files
cfg1.load
cfg2.load

# identify differences between cfg1 and cfg2
diff = cfg1.diff(cfg2)

# dump diff structure to terminal
STDERR.puts diff

# apply changes from cfg2 to cfg1 and save
cfg1.patch(diff)
cfg1.save
```

Figure 1.2. Example code of configuration file manipulation

### Complete Configurations Upgrade

Upgrade of configuration files on the users' deployments must be done for all files transactionally (it is not possible to upgrade only a part of the configuration files). Classes which manipulate with individual configuration files needs to be incorporated into a framework which would allow to:





- **bulk generate, persist and apply** upgrade differences between all available conf files
- include **custom upgrade code** for operations which cannot be identified automatically
- upgrade in a safe **sandbox** and copy final state to production locations only on success
- **track version** of configuration

### Command and Control Tool

The CLI tool `onecfg` is required for administrators to be able to check the configuration state, see available upgrades, run the bulk configuration upgrade and deal with potential conflicts introduced by heavy configuration customizations. Such a tool hides all complexities of manipulation with configuration files, bulk patching of files following the differences or sandboxing the operations behind a simple to use interface.

### Integration with Packager

EdgeNebula is going to be distributed as traditional operating system packages (rpm, deb). Both packagers provide their own ways and defaults how to deploy newer configuration files for changed ones during package upgrade. The new configuration file can be automatically deployed (and old version backed up) OR old configuration file can be left untouched (and new version is deployed under different name), while the non-customized configuration files are replaced with their newer versions automatically. The packager is a disruptive component here in terms of configuration files during the upgrade process.

It is necessary to use our own mechanism in the EdgeNebula packages to backup the state of the source configuration files before any package upgrade with files happens. And force our automatic configuration upgrade which preserves the customizations to use the backed up clean and trustworthy source state.

### **Architecture and Components**

The upgrade mechanism is implemented by following linked components integrated into a stand-alone simple to use command line tool `onecfg`:

- **Configuration Manipulation Classes:** interact with single configuration file (type)
- **Migrator of Configurations:** manages the upgrade of all configurations at once
- **Version Tracker:** tracks version of current configuration files

It provides insight into the configuration state on the EdgeNebula front-end deployment and also orchestrates all steps of the configuration files automatic upgrade process, which is the crucial part of the tool functionality. Tool completely avoids doing manual configuration upgrades for a non-heavily customized environment. It is expected to be integrated into a more complex upgrade automation process, which deals with upgrading of all parts (e.g., packages, database).

Figure 1.3 shows the interactions among the components during the successful upgrade of configuration.

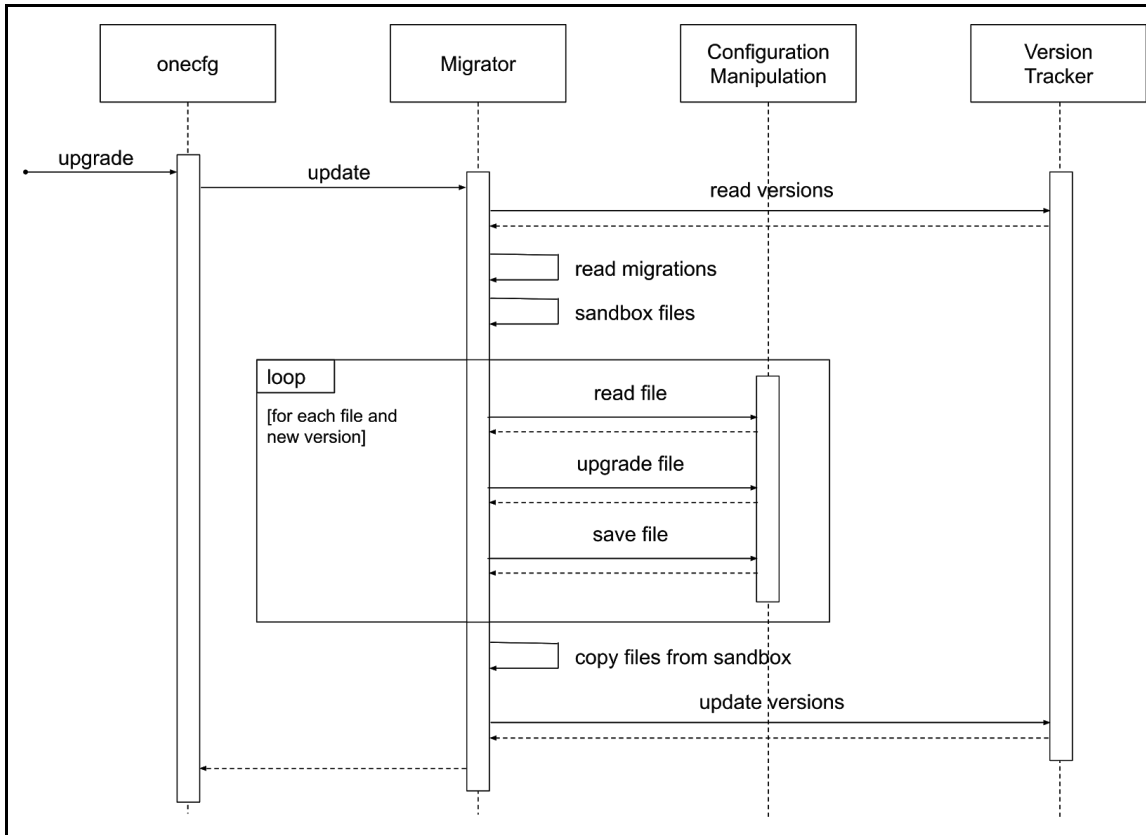


Figure 1.3. Sequence diagram of upgrade

## Data Model

### File Change Descriptor

Differences between configuration files (objects) as returned by diff() method of each configuration class are represented as a list of elemental operations necessary to follow when change is going to be applied on similar configuration. Table 1.3 describes metadata included in a single elemental operation.

KEY	DESCRIPTION
path	Path (location) in configuration file to make change as array
state	Action to take with change, possible values <ul style="list-style-type: none"> <li>• <b>set</b> - set existing parameter with different new value</li> <li>• <b>ins</b> - insert new configuration parameter</li> <li>• <b>rm</b> - remove configuration parameter</li> </ul>
key	Configuration parameter name
value	Configuration parameter value (for <b>set/ins</b> )
old	Previous configuration parameter value (for <b>set/rm</b> ) to detect user changes
extra	Hash with additional operation metadata <ul style="list-style-type: none"> <li>• <b>Multiple</b> - True/False if parameter has multiple occurrences</li> <li>• <b>Index</b> - Index with location of the change in array</li> </ul>

Table 1.3. Elemental change operation data



Example of simple change in configuration file oned.conf, which adds new TIMEOUT parameter into section DB is described by file comparison command diff on Figure 1.4.

```
# diff oned.conf oned.conf-new
76c76,77
< DB = [ BACKEND = "sqlite" ]
---
> DB = [ BACKEND = "sqlite",
>       TIMEOUT = 2500 ]
```

Figure 1.4. Difference between plain files (command diff)

Same change is described by the comparison mechanism of the configuration classes by a structure presented in Figure 1.5. It contains a single elemental operation with metadata following structure from the Table 1.3.

```
[{"path"=>["DB"], "key"=>"TIMEOUT", "value"=>"2500", "state"=>"ins",
"extra"=>{"multiple"=>false}}]
```

Figure 1.5. Change in file as described by configuration class

A method `hintings()` of each configuration class transforms the machine difference structure above into human readable text. Same change from Figures 1.4 and 1.5 is processed by `hintings` method on Figure 1.6.

```
ins DB/TIMEOUT = "2500"
```

Figure 1.6. Human readable change in configuration file

### Complete Upgrade Descriptor

Single upgrade descriptor of all changes for all configuration files from one version to another is a YAML document. It includes a list of all files - their names, filesystem metadata, raw content and file change descriptor (as returned by `diff()` method of each configuration type class). The descriptor tracks not only configuration files which have changed between product versions, but only existing unchanged files and files which are newly created or deleted in target version.

Metadata of each single managed configuration file are listed in Table 1.4.

KEY	DESCRIPTION
action	Type of change operation in new file version: <ul style="list-style-type: none"> <li>• <b>[undefined]</b> - no change happens</li> <li>• <b>create</b> - introduced new file to be created</li> <li>• <b>delete</b> - file is deleted in new version</li> <li>• <b>apply</b> - upgrade file by applying change</li> </ul>
class	Partial name of configuration type class which manages file, e.g.: <ul style="list-style-type: none"> <li>• <b>Simple</b></li> <li>• <b>Augeas::ONE</b></li> <li>• <b>Augeas::Shell</b></li> </ul>



	<ul style="list-style-type: none"> <li>• <b>Yaml</b></li> <li>• <b>Yaml::Strict</b></li> </ul>
owner	File user owner on filesystem (e.g., oneadmin)
group	File group owner on filesystem (e.g., oneadmin)
mode	File permissions on UNIX filesystem as a string (e.g., 0640)
content	String with raw content of the new file version (or old version on delete)
change	Change descriptor as returned by diff() (only if change happens)

**Table 1.4.** Version change metadata of single managed file

Upgrade descriptor is a Hash structure with only single 'patches' key element (for historic reason and will be subject to change in the next development iteration). Under the 'patches' there is a Hash structure with metadata of all managed files. Absolute filenames are in the structure keys and version change metadata of single managed file (Table 1.4) are in the values. An example of a complete upgrade descriptor is available in Figure 1.7.

```

---
patches:
  /etc/one/cli/oneprovision.yaml:
    class: Yaml::Strict
    owner: root
    group: root
    mode: '0644'
    action: apply
    change:
      - path:
          - :NAME
          key: :size
          value: 15
          old: 25
          state: set
          extra: {}
      - path:
          - :NAME
          key: :expand
          value: true
          state: ins
          extra: {}
    content: |
      ---
      :ID:
        :desc: Provision identifier
        :size: 36
      :NAME:
        :desc: Name of the Provision
        :size: 15
        :left: true
        :expand: true
  ...
  /etc/one/cli/onegroup.yaml:
  ...
  /etc/one/cli/onehost.yaml:
  ...

```

**Figure 1.7.** Example of generated upgrade descriptor



Although not strictly a data representation, the complete upgrade descriptor specified as set of default contents and automatic changes in the configuration files (Figure 1.7) might not cover all cases. Programmatic migrator for the configuration parts which are not possible to handle automatically is also necessary. Such a programmatic upgrade descriptor is a Ruby code, which is dealing with data changes in the configuration files content.

Example of programmatic upgrade descriptor as a code is shown on Figure 1.8. It contains a pre-upgrade section (def pre\_up), which prepares the state of configuration directories for the upgrade (i.e., create new directories, fix owners or permissions). Upgrade part gets each configuration file as an instance of old and new customized versions of the configuration file.

The code inside should update the data content of each file as required (figured example sets parameter MONITORING\_INTERVAL\_HOST to value 60 in configuration file /etc/one/oned.conf).

```

module Migrator

  # Preupgrade steps
  def pre_up
    @fops.mkdir('/var/lib/one/remotes/etc')
    @fops.chown('/var/lib/one/remotes/etc', 'oneadmin', 'oneadmin')
    @fops.chmod('/var/lib/one/remotes/etc', 0o750)
  end

  # Upgrade steps
  def up
    process('/etc/one/oned.conf', 'Augeas::ONE') do |old, new|
      new.set('MONITORING_INTERVAL_HOST[1]', 60)
    end
  end

end

end

```

Figure 1.8. Example of programmatic upgrade descriptor

## API and Interfaces

### API

No API changes are required.

### CLI

All configuration files management done by administrators (or any dependent automation mechanism) is strictly over the newly introduced CLI tool `onecfg`. Tool provides the set of subcommands, which are briefly listed in Table 1.5. For each subcommand a complete list of parameters can be displayed when `--help` is passed.

SUBCOMMAND	DESCRIPTION
<b>onecfg generate</b>	This subcommand is intended for OpenNebula developers to generate a complete upgrade descriptor for changes which can be identified and applied fully automatically.

**onecfg init**

Initialize version tracking state for the configuration files. If tooling loses track of the current version (due to several product upgrades without triggering configuration upgrade), the version tracking state can be reinitialized based.

**onecfg status**

Status shows the current configuration version:

```
# onecfg status
--- Versions -----
OpenNebula:  5.10.1
Config:      5.10.0

--- Available Configuration Updates -----
No updates available.
```

Lists available configuration version upgrades.

```
onecfg status
--- Versions -----
OpenNebula: 5.10.1
Config:     5.6.0

--- Backup to Process -----
Snapshot:   /var/lib/one/backups/config/backup
(will be used as one-shot source for next update)

--- Available updates -----
New config: 5.10.0
- from 5.6.0 to 5.8.0 (YAML,Ruby)
- from 5.8.0 to 5.10.0 (YAML,Ruby)
```

**onecfg upgrade**

Upgrades the configuration files:

```
# onecfg upgrade --verbose
INFO  : Checking updates from 5.8.0 to 5.10.0
ANY   : Backup stored in
'/tmp/onescape/backups/2019-12-12_15:14:39_18278'
INFO  : Updating from 5.8.0 to 5.10.0
INFO  : Incremental update from 5.8.0 to 5.10.0
INFO  : Update file '/etc/one/vcenter_driver.default'
INFO  : Skip file '/etc/one/cli/oneprovision.yaml' - missing
INFO  : Update file '/etc/one/cli/onegroup.yaml'
INFO  : Update file '/etc/one/cli/onehost.yaml'
INFO  : Update file '/etc/one/cli/oneimage.yaml'
...
ANY   : Configuration updated to 5.10.0
```

No changes on the filesystem will be done if no upgrade is available nor if the process fails to upgrade a particular file with even a single error.

To deal with error situations during upgrade (introduced by incompatible customizations by cloud administrators), the tool provides a set of patch modes which instructs the upgrade process how to deal with error situations (globally, for specific file, for specific file and version). Following patch modes are available:

- **skip** - skip failing operation
- **force** - apply upgrade on most suitable place in file
- **replace** - replace conflicting customization with distr. one

Patch modes are specified during **onecfg upgrade** run following way, e.g.:



```
# onecfg upgrade \
  --patch-modes skip:/etc/one/oned.conf \
  --patch-modes skip,replace:/etc/one/oned.conf:5.10.0 \
  --patch-modes force:/etc/one/sunstone-logos.yaml:5.6.0 \
  --patch-modes replace:/etc/one/sunstone-server.conf \
  --patch-modes skip:/etc/one/sunstone-views/admin.yaml:5.4.1 \
  --patch-modes skip:/etc/one/sunstone-views/admin.yaml:5.4.2 \
  --patch-modes skip:/etc/one/sunstone-views/kvm/admin.yaml
```

**onecfg validate**

Reads and validates syntax of all managed configuration files.

```
# onecfg validate --verbose
INFO : File '/etc/one/vcenter_driver.default' - OK
INFO : File '/etc/one/ec2_driver.default' - OK
INFO : File '/etc/one/az_driver.default' - OK
INFO : File '/etc/one/auth/ldap_auth.conf' - OK
INFO : File '/etc/one/auth/server_x509_auth.conf' - OK
...
```

**onecfg diff**

Reads all managed configuration files and identifies all user customizations. List of customization is provided as a human readable change descriptor (see Figure 1.6) with user inserted, removed and set parameters.

```
# onecfg diff
/etc/one/cli/oneimage.yaml
- ins ID/adjust = true
- set USER/size = 8
- set GROUP/size = 8
- ins NAME/expand = true

/etc/one/oned.conf
- set DEFAULT_DEVICE_PREFIX = "\"sd\""
- set VM_MAD[NAME = "vcenter"]/ARGUMENTS = "\"-p -t 15 -r 0 -s sh vcenter\""
-   rm          VM_MAD[NAME      = "vcenter"]/DEFAULT      =
  "\"vmm_exec/vmm_exec_vcenter.conf\""
- ins HM_MAD/ARGUMENTS = "\"-p 2101 -l 2102 -b 127.0.0.1\""
- ins VM_RESTRICTED_ATTR = "\"NIC/FILTER\""
```

**Table 1.5.** Subcommands of onecfg tool



## 2. Edge Workload Orchestration and Management (CPNT2)

### [SR2.1] Integration with Serverless Hypervisor

#### Description

A new driver to interact with Firecracker VMM has been implemented. This allows ONEedge to support light VMs called microVMs.

MicroVMs are fully integrated and support every basic operation (create, terminate, power-off, etc.). MicroVMs are also integrated with the storage stack (file based datastores) and network stack (linux bridge based drivers). Additional ONEedge features VNC support and contextualization support for Firecracker microVMs.

#### Requirements and Specifications

##### Seamless integration

MicroVMs are treated like normal VMs resources. Apart from hypervisor limitations, the integration allows to:

- Perform existing actions (e.g create, terminate, power-off, ...)
- Use existing storage drivers.
- Use existing networking drivers.
- Use existing contextualization methods.

As a result of this seamless integration, microVMs are fully supported for every OpenNebula service that uses VMs resources, like OneFlow. Also most of the storage and networking drivers are available without major changes.

##### Lightweight

The Integration with the serverless application provide lightweight VMs (microVMs) which facilitates the management of serverles workloads by:

- Increasing the VMs per host density.
- Reducing deployment time.
- Allowing to easily deploy application oriented images.

##### Isolation

As the microVMs will run in a multitenant environment, isolation between different microVMs must be achieved to ensure users security. The microVM process is isolated by using a hypervisor provided tool called Jailer.

##### Monitoring

A monitoring driver has been implemented for monitoring both MicroVMs and hypervisor nodes.



## Web Interface

The Sunstone web interface has been modified in order to adapt itself to microVMs. When a microVM template is being defined or updated the interface will take care of showing only available actions and restrict or change the available options for some fields.

## Architecture and Components

No new components were needed to implement the new hypervisor integration. The high level architecture of the new driver is defined in Figure 2.1 below.

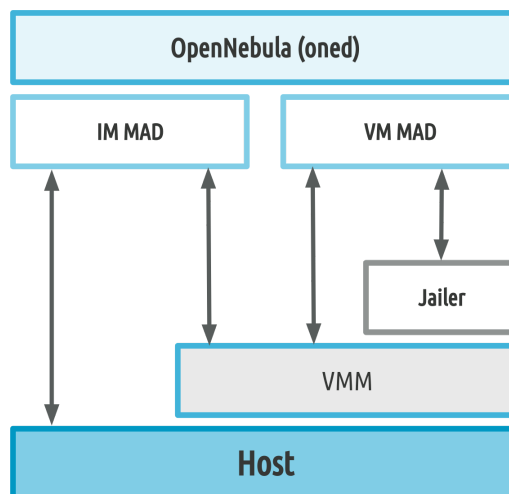


Figure 2.1. Overview of OpenNebula Host and Firecracker (jailer) components

The **Jailer** takes care of adding an extra layer of security by isolating the process using linux existing tools like chroot and cgroups.

The **VMM** (Firecracker) takes care of managing the microVM and provides a REST API to perform different actions over the microVM (e.g gracefully power-off, rescan disks, ...).

## Data Model

MicroVMs information is stored in the OpenNebula databases as an XML document with the same structure used for VMs. The XSD defining the schema can be found in the repository:

<https://github.com/OpenNebula/one/blob/master/share/doc/xsd/vm.xsd>

```

<VM>
  <ID>16</ID>
  <UID>0</UID>
  <GID>0</GID>
  <UNAME>oneadmin</UNAME>
  <GNAME>oneadmin</GNAME>
  <NAME>alpine-16</NAME>
  ...
  <DEPLOY_ID>one-16</DEPLOY_ID>
  ...
  <TEMPLATE>
    ...
    <CPU><![CDATA[0.1]]></CPU>
  
```

```

<DISK>
  ...
  <DISK_ID><![CDATA[0]]></DISK_ID>
  ...
  <IMAGE_ID><![CDATA[0]]></IMAGE_ID>
  ...
  <READONLY><![CDATA[NO]]></READONLY>
  ...
</DISK>
...
<MEMORY><![CDATA[128]]></MEMORY>
<NIC>
  ...
  <MAC><![CDATA[02:00:c0:a8:96:64]]></MAC>
  ...
  <NIC_ID><![CDATA[0]]></NIC_ID>
  ...
  <TARGET><![CDATA[one-16-0]]></TARGET>
  ...
</NIC>
...
<OS>
  <KERNEL><![CDATA[/var/lib/one//datastores/0/16/kernel]]></KERNEL>
  <KERNEL_CMD><![CDATA[console=ttyS0 reboot=k panic=1 pci=off i8042.noaux i8042.nomux
i8042.nopnp i8042.dumbkbd]]></KERNEL_CMD>
  <KERNEL_DS><![CDATA[$FILE[IMAGE="kernel"]]]></KERNEL_DS>
  <KERNEL_DS_CLUSTER_ID><![CDATA[0]]></KERNEL_DS_CLUSTER_ID>
  <KERNEL_DS_DSID><![CDATA[2]]></KERNEL_DS_DSID>
  <KERNEL_DS_ID><![CDATA[1]]></KERNEL_DS_ID>

<KERNEL_DS_SOURCE><![CDATA[/var/lib/one//datastores/2/3255664bc145314981e863454b65319e]]></KE
RNL_DS_SOURCE>
  <KERNEL_DS_TM><![CDATA[ssh]]></KERNEL_DS_TM>
</OS>
...
<TEMPLATE_ID><![CDATA[0]]></TEMPLATE_ID>
<TM_MAD_SYSTEM><![CDATA[ssh]]></TM_MAD_SYSTEM>
<VMID><![CDATA[16]]></VMID>
</TEMPLATE>
...
</VM>

```

Figure 2.2. VM XML representation (some attributes have been omitted)

In order to deploy a microVM the XML document representing it must be converted to a deployment file or configuration file representing the microVM in a way the hypervisor can understand. Firecracker microVMs configuration file is a JSON document covering all the VM specifications.

The supported resources and their configuration attributes are defined in the API definition file which is publicly available at Firecracker GitHub repository:

[https://github.com/firecracker-microvm/firecracker/blob/master/src/api\\_server/swagger/firecracker.yaml](https://github.com/firecracker-microvm/firecracker/blob/master/src/api_server/swagger/firecracker.yaml)

```

{
  "boot-source": {
    "kernel_image_path": "kernel",
    "boot_args": "console=ttyS0 reboot=k panic=1 pci=off i8042.noaux i8042.nomux i8042.nopnp
i8042.dumbkbd"
  },

```

```
"drives": [
  {
    "drive_id": "disk.0",
    "path_on_host": "disk.0",
    "is_root_device": true,
    "is_read_only": false
  },
  {
    "drive_id": "disk.1",
    "path_on_host": "disk.1",
    "is_root_device": false,
    "is_read_only": true
  }
],
"machine-config": {
  "mem_size_mib": 128,
  "vcpu_count": 1,
  "ht_enabled": false
},
"network-interfaces": [
  {
    "iface_id": "eth0",
    "host_dev_name": "one-16-0",
    "guest_mac": "02:00:c0:a8:96:64",
    "allow_mmds_requests": true
  }
],
"logger": {
  "log_fifo": "logs.fifo",
  "metrics_fifo": "metrics.fifo"
}
}
```

Figure 2.3. Firecracker microVMs configuration file in JSON format

The virtualization driver takes care of the translation from the OpenNebula XML VM representation to the Firecracker JSON microVM representation when the deploy action is performed.

A high level image of the translation between the VM XML document to the deployment file can be achieved by comparing Figure 2.2 and Figure 2.3. Many of the VM attributes, like permission, context or other information related to how OpenNebula manage the resources have been omitted from Figure 2.2 as they are not relevant to the VMM. The VMM only requires the basic information to boot the microVM leaving the resource management tasks to OpenNebula drivers.

## API and Interfaces

### VMM Interaction

In order to integrate the hypervisor with OpenNebula 3 different APIs or interfaces are used at different levels:

1. **VMM Level interface:** this interface allows OpenNebula drivers to interact with Firecracker and the Jailer. The Jailer is interacted by using a command line interface while the VMM process is interacted via REST API. VMM Level interface is used by the virtualization and monitoring drivers.



2. **Driver interface:** this interface wraps the VMM Level interface. Each possible action is defined by a driver which is used by OpenNebula (oned) to carry out the different supported actions.
3. **XML-RPC interface:** this interface is exposed by OpenNebula and it allows users to interact with VM resources, among others. This is the interface used by the final user to interact with OpenNebula either directly or using higher level interfaces like CLI or Sunstone.

These different layers of abstraction allow us to easily add new hypervisors as only the VMM Level interface is hypervisor dependent.

### Configuration

A new configuration file, **firecrackerrc**, have been created to set up configuration attributes for tuning how Firecracker VMM interacts with the system and OpenNebula. The following can be configured:

- **VNC options:** the width, height and timeout can be configured for VNC connection.
- **Datastore location:** if datastores are not mounted in the default path it must be set to let the driver know where to find the datastores.
- **Jailer uid and gid:** these attributes are used by the Jailer to isolate the process using performing a chroot action.
- **Firecracker binary location.**
- **Shutdown timeout:** the time to wait for a VM to gracefully shutdown before killing the process.
- **Cgroups configuration:** the cgroup mount point, the delete timeout and the use of `cpu.shares` for limiting the cpu access can be configured in this section.
- **NUMA placement strategy:** it defines how the microVMs are going to be distributed over the available NUMA nodes.

## [SR2.3] Secure and Scalable Distributed Monitoring

### Description

The monitoring system was redesigned to enable better scalability and to reduce the CLI response times under heavy load.

The monitoring system was separated from the oned to Monitor Daemon. The Monitor Daemon controls the monitoring workflow: starts monitoring agents, processes network messages from agents and writes monitoring info to the database.

### Requirements and Specifications

- Allow better scalability of the monitoring system
- Reduce CLI response times under heavy load
- Use a separate component to process monitoring information
- Encrypt communication

### Architecture and Components

Main components of the monitoring system are:

- **Monitor Daemon:** The Monitor Daemon dirigates the whole Monitoring System. It's connected to the oned with the Mem Pipe. Starts or stops Monitor Agents through Monitor Driver according to Host state and receives monitoring information from the Monitor Agents and writes the monitoring information to OpenNebula database.
- **Monitor Driver:** Communication class between the Monitor Daemon and Monitor Agents
- **Monitor Agent:** Runs on host, periodically starts monitoring probes, receives the information from probes and sends it by UDP or TCP network to Monitor Daemon.
- **Monitor Probe:** Gathers information from the host.

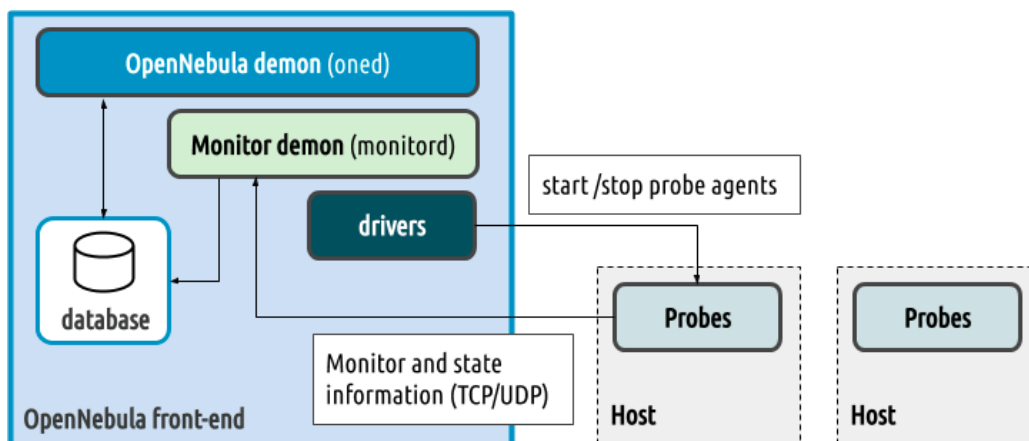


Figure 2.4. Monitor Daemon architecture



## Message Format

There is a message system for communication between Monitor Daemon, oned and Monitor Agents. The messages have following format:

MESSAGE TYPE	OBJECT ID	STATUS	TIMESTAMP	PAYLOAD
--------------	-----------	--------	-----------	---------

Figure 2.5. Monitor message format

- MESSAGE TYPE - String, as described in the tables below
- OBJECT ID - Integer, ID of the host the message refers to
- STATUS - String describing the status of the operation or host "SUCCESS", "FAILURE", "-", "ERROR", "ONLINE", ...
- TIMESTAMP - Unix epoch time
- PAYLOAD - Base64 encoded and zipped string with message data. The payload could be encrypted

## Interface between oned and Monitor Daemon

Used to send information from/to the monitor daemon to/from oned. Messages will include state changes and the information gathered by the system probes.

TYPE	ID	STATUS	PAYLOAD
SYSTEM_HOST	Host ID	success/fail	General information about the host, which does not change too often (e.g. total memory, disk capacity, datastores, pci devices, NUMA nodes, ...)
HOST_STATE	Host ID		Additional information associated to the state change
VM_STATE	Host ID		State of VMs running on the host

Table 2.1. Messages from Monitor Daemon to oned

TYPE	ID	PAYLOAD
UPDATE_HOST	Host ID	XML Host information
HOST_LIST	-	XML HostPool information
DEL_HOST	Host id	
RAFT_STATUS	-	Leader state in HA environment

Table 2.2. Messages from oned to Monitor Daemon



### Interface between Monitor Daemon and Monitor Driver

TYPE	ID	PAYLOAD
START_MONITOR	Host ID	template with host information + monitord configuration
STOP_MONITOR	Host ID	template with host information

**Table 2.3.** Messages from Monitor Daemon to Monitor Agent

TYPE	ID	STATUS	PAYLOAD
BEACON_HOST	Host ID	success/fail	No extra payload. Notification message, indicating that the agent is still alive
MONITOR_VM	Host ID	success/fail	VMs monitoring information: used memory, used CPUs, disk io, ...
MONITOR_HOST	Host ID	success/fail	Monitoring information: used memory, used cpu, network traffic, ...
SYSTEM_HOST	Host ID	success/fail	General information about the host, which does not change too often (e.g. total memory, disk capacity, datastores, pci devices, NUMA nodes, ...)
STATE_VM	Host ID	success/fail	VMs state: running, power-off, ...
LOG		E, I, W, D	Error/info message from agent, should be displayed in monitor log

**Table 2.4.** Messages from monitor agent to monitor daemon

### Probes

Probes are executed by Monitor Agent to collect specific metrics from the host. Probes are structured in different directories that determine the frequency in which they are executed, as well as the data sent back to the frontend. Each hypervisor has its own set of probes located at `remotes/im/<hypervisor>-probes.d`. Table 2.5 shows the purpose of each probe directory.

Directory	Purpose	Update frequency
host/beacon	Heartbeat & watchdog to collect rouge probe processes	BEACON_HOST (30s)
host/monitor	Monitor information (variable) (e.g. memory usage) stored in HOST/MONITORING	MONITOR_HOST (120s)
host/system	General quasi-static information about the host (e.g. NUMA nodes) stored in HOST/TEMPLATE and HOST/SHARE	SYSTEM_HOST (600s)
vm/monitor	Monitor information (variable) (e.g. used cpu, network usage) stored in VM/MONITORING	MONITOR_VM (30s)
vm/status	State change notification, only send when a change is detected	STATE_VM (30s)

**Table 2.5.** Monitor Probes



## Configuration

The configuration of the Monitor Daemon is located at `/etc/one/monitord.conf`. Table 2.6 describes basic configuration attributes.

Parameter	Attribute	Description
MANAGER_TIMER		Timer in seconds, monitord evaluates host timeouts
MONITORING_INTERVAL_HOST		Wait this time (seconds) without receiving any beacon before restarting the probes
HOST_MONITORING_EXPIRATION_TIME		Seconds to expire host monitoring information, 0 to disable monitoring recording.
NETWORK	ADDRESS	Network address to bind the UDP/TCP listener to
	MONITOR_ADDRESS	Agents will send updates to this monitor address. If "auto" is used, agents will detect the address from the ssh connection frontend -> host (\$SSH_CLIENT), "auto" is not usable for HA setup
	PORT	Listening port
	THREADS	Number of threads used to receive messages from monitor probes
	PUBKEY	Absolute path to public key. Empty for no encryption.
	PRIKEY	Absolute path to private key. Empty for no encryption.
PROBES_PERIOD	BEACON_HOST	Time in seconds to send heartbeat for the host
	SYSTEM_HOST	Time in seconds to send host static/configuration information
	MONITOR_HOST	Time in seconds to send host variable information
	STATE_VM	Time in seconds to send VM status (ie. running, error, stopped...)
	MONITOR_VM	Time in seconds to send VM resource usage metrics.
	SYNC_STATE_VM	Send a complete VM report every SYNC_STATE_VM seconds

**Table 2.6.** Monitor Daemon configuration attributes

## Data Model

All monitor data is stored in the OpenNebula database in xml format. Extract of the monitoring data from xsd schema:

```
<xs:element name="MONITORING">
  <xs:complexType>
    <xs:all>
      <!-- Percentage of 1 CPU consumed (two fully consumed cpu is 2.0) -->
```



```

<xs:element name="CPU" type="xs:decimal" minOccurs="0" maxOccurs="1"/>
<!-- Amount of bytes read from disk-->
<xs:element name="DISKRDBYTES" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<!-- Number of IO read operations -->
<xs:element name="DISKRDIOPS" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<!-- Amount of bytes written to disk -->
<xs:element name="DISKWRBYTES" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<!-- Number of IO write operations -->
<xs:element name="DISKWRIOPS" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<!-- ID of the VM -->
<xs:element name="ID" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<!-- Consumption in kilobytes -->
<xs:element name="MEMORY" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<!-- Sent bytes to the network -->
<xs:element name="NETTX" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<!-- Received bytes from the network -->
<xs:element name="NETRX" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<!-- Exact time when monitoring info were retrieved -->
<xs:element name="TIMESTAMP" type="xs:integer" minOccurs="0" maxOccurs="1"/>
</xs:all>
</xs:complexType>
</xs:element>

```

Figure 2.6. VM monitor information xsd schema

```

<xs:element name="MONITORING">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="TIMESTAMP" type="xs:integer" minOccurs="0"/>
      <xs:element name="ID" type="xs:integer" minOccurs="0"/>
      <xs:element name="CAPACITY" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="FREE_CPU" type="xs:integer"/>
            <!-- ^^ Percentage, Free CPU as returned by the probes -->
            <xs:element name="FREE_MEMORY" type="xs:integer"/>
            <!-- ^^ KB, Free MEMORY returned by the probes -->
            <xs:element name="USED_CPU" type="xs:integer"/>
            <!-- ^^ Percentage of CPU used by all host processes (including VMs) over a
total of # cores * 100 -->
            <xs:element name="USED_MEMORY" type="xs:integer"/>
            <!-- ^^ KB, Memory used by all host processes (including VMs) over a total of
MAX_MEM -->
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="SYSTEM" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="NETRX" type="xs:integer" minOccurs="0"/>
            <xs:element name="NETTX" type="xs:integer" minOccurs="0"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

```
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
```

**Figure 2.7.** Host monitor information xsd schema

## API and Interfaces

The API and CLI remain unchanged.



## [SR2.5] Integration with Remote VMware vCenter Service

### Description

The capabilities of ONEedge to interface with VMware services has been improved to extend its networking capabilities. The new drivers can define virtual networks and security groups leveraging the VMware native capabilities (NSX-t & NSX-v).

NSX is the Network and Security software from VMware that enables a virtual cloud network to connect and protect applications across data centers, multi-clouds, bare metal, and container infrastructures. VMware NSX Data Center delivers a complete L2-L7 networking and security virtualization platform—providing agility, automation, and dramatic cost savings that come with a software-only solution.

With this integration OpenNebula is able to manage NSX-V and NSX-T logical switches and Distributed Firewalls to enforce security groups.

### Requirements and Specifications

The requirements of the NSX manager to be able to interact with OpenNebula are stated in the following points:

- **NSX Manager.** The NSX appliance must be deployed with only one IP Address. OpenNebula installation must be able to connect to NSX Manager with the needed credentials.
- **Controller nodes.** At least one controller node must be deployed.
- **ESXi Hosts.** All ESXi of the cluster must be prepared for NSX.
- **Transport Zone.** At least one transport zone must be created.

Adding a NSXmanager into OpenNebula This is a semi-automatic process. When vCenter is connected to a NSX Manager, OpenNebula will detect it in the next monitoring cycle. As a result, a new tab called "NSX" will show in the UI allowing the configuration of the credentials (User and Password) needed to connect to NSX Manager. The same process is applied when importing a new vCenter cluster that is prepared to work with NSX-V or NSX-T.

Also, whenever a vCenter cluster is imported in OpenNebula two hooks are created to aid in the creation and destruction of virtual networks. These two hooks have been adapted to also support NSX.

After a vCenter cluster is imported and monitor cycle finalises, the NSX Manager registered for that cluster is detected.

### Architecture and Components

OpenNebula manages logical switches using NSX Driver and the hook subsystem. An action to create or delete a logical switch either from Sunstone, CLI or API, generates a specific event. If there is a hook subscribed to that event, it will execute an action or script.

In the NSX integration a hook will use the NSX Driver to send commands to the NSX Manager API and will wait for an answer. When NSX Manager finishes the action it will return a response and the hook, based on that response, will end up as success or error.

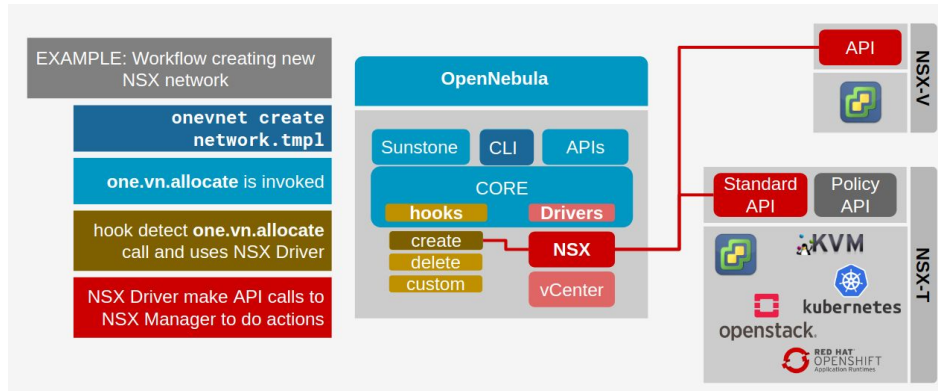


Figure 2.8. The process of creating a logical switch

OpenNebula security groups define rules, associated to a Virtual Network, that are applied into NSX Distributed Firewall over a specific VM logical port group. NSXDriver is in charge of translating OpenNebula security group rules into DFW rules, on both NSX-T and NSX-V.

**Data Model**

All NSX objects have a reference within the NSX Manager. Some NSX objects also have references into vCenter Server. This is the case of logical switches, that have an object representation in vCenter.

The following table describes the components and their reference formats in NSX and vCenter, and also the attributes used in OpenNebula to store that object:

Object	Type	OpenNebula Attr	NSX Managed Object Reference	vCenter Managed Object Reference
Logical Switch	NSX-T	NSX_ID	xxxxxxxx-yyyy-zzzz-aaaa-bbbbbb	network-oXXX
VirtualWire	NSX-V	NSX_ID	virtualwire-XXX	dvportgroup-XXX
Transport Zone	NSX-T	TZ_ID	xxxxxxxx-yyyy-zzzz-aaaa-bbbbbb	N/A
Transport Zone	NSX-V	TZ_ID	vdnscope-XX	N/A

Table 2.7. Mapping OpenNebula, NSX and vCenter resources

Here are the actions that affect the creation, modification or deletion of rules in the Distributed Firewall:

OpenNebula action	Net driver actions	NSX driver action
Instantiate	PRE & POST	Create rules
Terminate	CLEAN	Delete rules
PowerOn	PRE & POST	Create rules
PowerOff	CLEAN	Delete rules

Table 2.8. Actions over Distributed Firewall

The NSX driver library is a complex component, as depicted in the following figure.

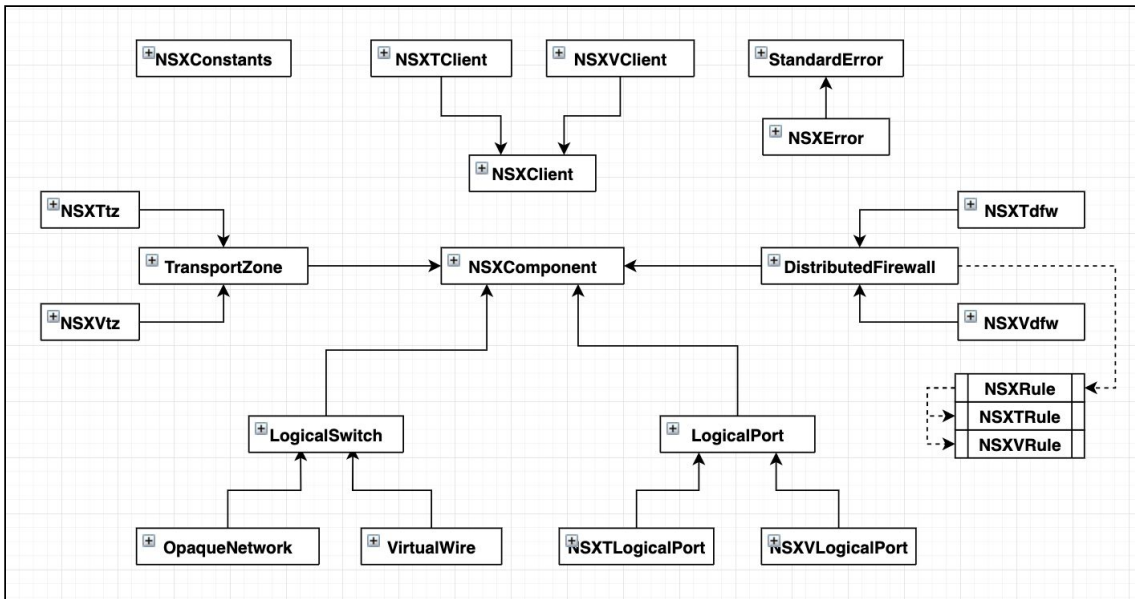


Figure 2.9. Class Diagram in the NSX driver

### API and Interfaces

The NSX integration reuses the Virtual Network and Security Groups already present in the OpenNebula Sunstone WebUI. However, the interface in Sunstone for NSX manager configuration has been developed for this functionality, and placed within the OpenNebula Host information panel.

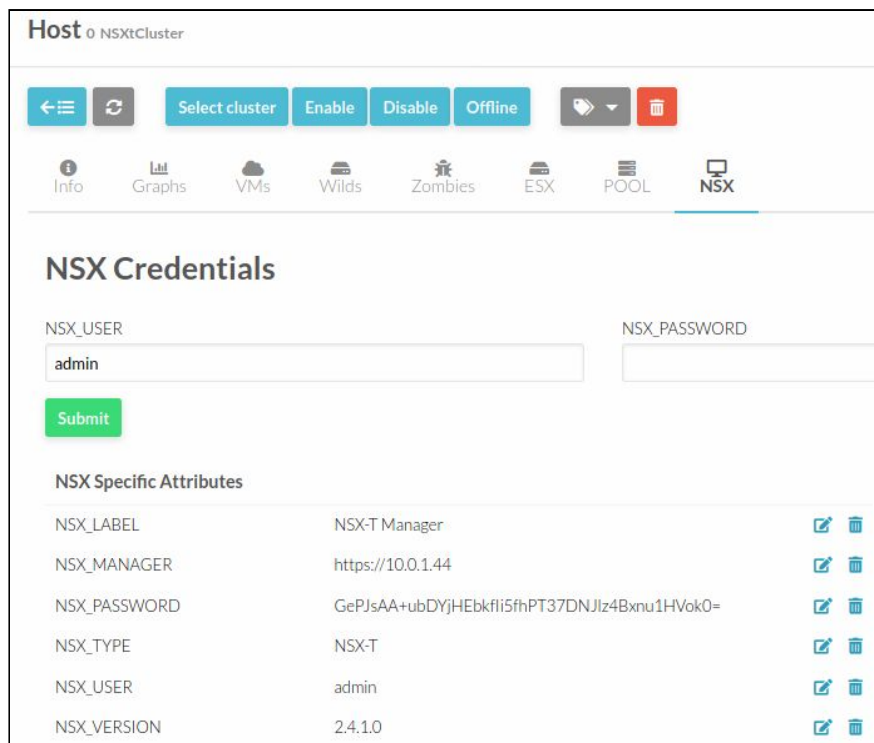


Figure 2.10. NSX details in Sunstone



## [SR2.6] VNF Support

### Description

Fine grain NUMA placement is needed to optimize the performance of some VM workloads, especially VNFs. We have extended the functionality of OpenNebula to control how VM resources are mapped onto the hypervisor ones. The implementation assumes the following model and server characteristics:

- **NUMA.** Multi-processor servers are usually arranged in nodes or cells. Each NUMA node holds a fraction of the overall system memory. In this configuration, a processor accesses memory and I/O ports local to its node faster than to the non-local ones.
- **Cores, Threads and Sockets.** A computer processor is connected to the motherboard through a socket. A processor can pack one or more cores, each one implements a separated processing unit that shares some cache levels, memory and I/O ports. CPU Cores performance can be improved by the use of hardware multi-threading (SMT) that permits multiple execution flows to run simultaneously on a single core.
- **Hugepages.** Systems with big physical memory use also a big number of virtual memory pages. This big number makes the use of virtual-to-physical translation caches inefficient. Hugepages reduces the number of virtual pages in the system and optimizes the virtual memory subsystem.

The NUMA placement of a VM in a hypervisor node consists of mapping (pinning) each virtual thread into a physical server thread, considering the overall usage of the server, the virtual topology of the VM as well as its memory and PCI passthrough requirements.

### Requirements and Specifications

#### Virtual CPU Topology Definition

The most basic configuration is just to define the number of vCPU (virtual CPU) and the amount of memory of the VM. In this case the guest OS will be configured with VCPU sockets of 1 core and 1 thread each. A more detailed topology can be defined by providing a custom number of sockets, cores and threads for a given number of vCPUs.

For example a VM with 2 sockets and 2 cores per sockets and 2 threads per core is defined by the VM template shown in Figure 2.11.

```
VCPU = 8
MEMORY = 1024

TOPOLOGY = [ SOCKETS = 2, CORES = 2, THREADS = 2 ]
```

**Figure 2.11.** Definition of a VM with 2 sockets, 2 cores per socket and 2 threads per core. The total number of CPUs is 8 (2 x 2 x 2).

Additional details can be provided for the topology of your VM by defining the placement of the sockets (NUMA nodes) into the hypervisor NUMA nodes. In this scenario each VM socket will be exposed to the guest OS as a separated NUMA node with its own local memory. This behavior is configured by setting a specific PIN\_POLICY as explained below.



Some applications may need an asymmetric NUMA configuration, i.e. not distributing the VM resources evenly across the nodes. You can define each node configuration by manually setting the `NUMA_NODE` attribute. See the Data Model section below.

### CPU and NUMA Pinning

When a VM needs to expose the NUMA topology to the guest a pinning policy needs to be defined to map each virtual NUMA node resource (memory and vCPUs) onto the hypervisor nodes. OpenNebula can work with three different policies:

- **CORE:** each vCPU is assigned to a whole hypervisor core. No other threads in that core will be used. This policy can be useful to isolate the VM workload for security reasons.
- **THREAD:** each vCPU is assigned to a hypervisor CPU thread.
- **SHARED:** the VM is assigned a set of the hypervisor CPUS shared by all the VM vCPUs.

VM memory is assigned to the closest hypervisor NUMA node where the vCPUs are pinned to, trying to prioritize local memory accesses.

When using a pinning policy the scheduler can automatically pick the number of cores and threads of the virtual topology. OpenNebula will try to optimize the VM performance by selecting the threads per core according to the following algorithm:

- For the CORE pin policy the number of THREADS is set to 1.
- Prefer as close as possible to the hardware configuration of the host and so be power of 2.
- The threads per core will not exceed that of the hypervisor.

### PCI Passthrough

The scheduling process is slightly modified when a pinned VM includes PCI passthrough devices. In this case the NUMA nodes where the PCI devices are attached to, are prioritized to pin the VM vCPUs and memory to speed-up I/O operations.

### Open vSwitch with DPDK

Another important aspect to optimize the performance of NFVs is the ability to use optimized dataplanes, like DPDK. When using the DPDK backend, the OpenNebula drivers have been extended to automatically configure the bridges and ports accordingly. DPDK bridge type can be enabled in the regular OpenvSwitch network drivers in `oned.conf`.

## **Architecture and Components**

No new components were needed to implement the NUMA topology. However, the following existing modules have been extended:

- OpenNebula core daemon, to add support for the new attributes for VMs and hosts. OpenNebula core checks, and tracks the allocation of NUMA resources on the hosts.
- Scheduler. Implements the NUMA allocation policies described above. It also checks the capacity and availability of hugepages.
- Monitor Probes were added to gather the NUMA topology of the host.



## Data Model

The Virtual Machine templates can include the TOPOLOGY attribute to specify the VM NUMA topology. Table 2.9 shows the attributes that can be set to define it.

TOPOLOGY	Meaning
PIN_POLICY	vCPU pinning preference: CORE, THREAD, SHARED, NONE
SOCKETS	Number of sockets or NUMA nodes
CORES	Number of cores per node
THREADS	Number of threads per core
HUGEPAGE_SIZE	Size of the hugepages (MB). If not defined no hugepages will be used
MEMORY_ACCESS	Control if the memory is to be mapped shared or private

**Table 2.9.** Topology attributes and their meaning

The host data model has also been extended to support the NUMA attributes. See Table 2.10 for more details.

ATTRIBUTE	Meaning
PIN_POLICY	<ul style="list-style-type: none"> <li>• NONE Use numad for placement, VMs are not pinned to any CPU</li> <li>• PINNED Allocate NUMA nodes and pin vCPU and MEMORY</li> </ul>
HUGEPAGE	<ul style="list-style-type: none"> <li>• NODE_ID of the NUMA cell for the hugepage allocation</li> <li>• SIZE Kb of the pages</li> <li>• PAGES total pages available</li> <li>• FREE pages</li> </ul>
MEMORY_NODE	<ul style="list-style-type: none"> <li>• NODE_ID of the node</li> <li>• TOTAL total memory in the node</li> <li>• FREE free memory in the node</li> <li>• USED used memory in the node</li> <li>• DISTANCE distance vector to other nodes.</li> </ul>
CORE	<ul style="list-style-type: none"> <li>• NODE_ID of the NUMA cell where the CPU core lives</li> <li>• ID of the CPU core</li> <li>• CPUS IDs of thread siblings in the core</li> </ul>

**Table 2.10.** NUMA attributes in the host and their meaning

An example of the NUMA attributes for a host is shown in Figure 2.12. It corresponds to a Host with 1 socket, 4 cores and 2 threads per core. Hugepages of 2M and 1GB have been defined in the system.

```
HUGEPAGE = [ NODE_ID = "0", SIZE = "2048", PAGES = "0", FREE = "0" ]
HUGEPAGE = [ NODE_ID = "0", SIZE = "1048576", PAGES = "0", FREE = "0" ]
```





```

CORE = [ NODE_ID = "0", ID = "3", CPUS = "3,7" ]
CORE = [ NODE_ID = "0", ID = "1", CPUS = "1,5" ]
CORE = [ NODE_ID = "0", ID = "2", CPUS = "2,6" ]
CORE = [ NODE_ID = "0", ID = "0", CPUS = "0,4" ]
MEMORY_NODE = [ NODE_ID = "0", TOTAL = "7992892", FREE = "347944", USED = "7644948", DISTANCE
= "0" ]

```

Figure 2.12. Example of the NUMA attributes for a host

## API and Interfaces

System APIs were not needed to be extended, the same API calls to manage VMs and hosts can handle the NUMA extensions. In terms of the configuration interface the oned.conf file includes a new restricted attribute to restrict the access to the PIN\_POLICY attribute.

The command line tool shows the information about the NUMA usage and pinning information. The onehost command includes the NUMA characteristics of the host as well as the current usage, see Figure 2.13.

```

$ onehost show 0
...
NUMA NODES

  ID CORES                                USED FREE
  0 XX XX -- --                          4   4

NUMA MEMORY

  NODE_ID TOTAL    USED_REAL        USED_ALLOCATED    FREE
  0 7.6G    6.8G                1024M              845.1M

NUMA HUGEPAGES

  NODE_ID SIZE      TOTAL    FREE    USED
  0 2M      2048    1536    512
  0 1024M   0       0       0
...

```

Figure 2.13. NUMA information in the output of onehost command

Equivalently the onevm command shows the topology of the VM, see Figure 2.14.

```

$ onevm show 0
...
NUMA NODES

  ID  CPUS    MEMORY TOTAL_CPUS
  0   0,4    512M      2
  0   1,5    512M      2

TOPOLOGY

  NUMA_NODES CORES  SOCKETS  THREADS
  2          2      1        2

```

Figure 2.14. NUMA information in the output of onevm command

Finally the Sunstone interface has been also updated to include the NUMA topology and usage of hosts and Virtual Machines, see Figure 2.15 for an example.

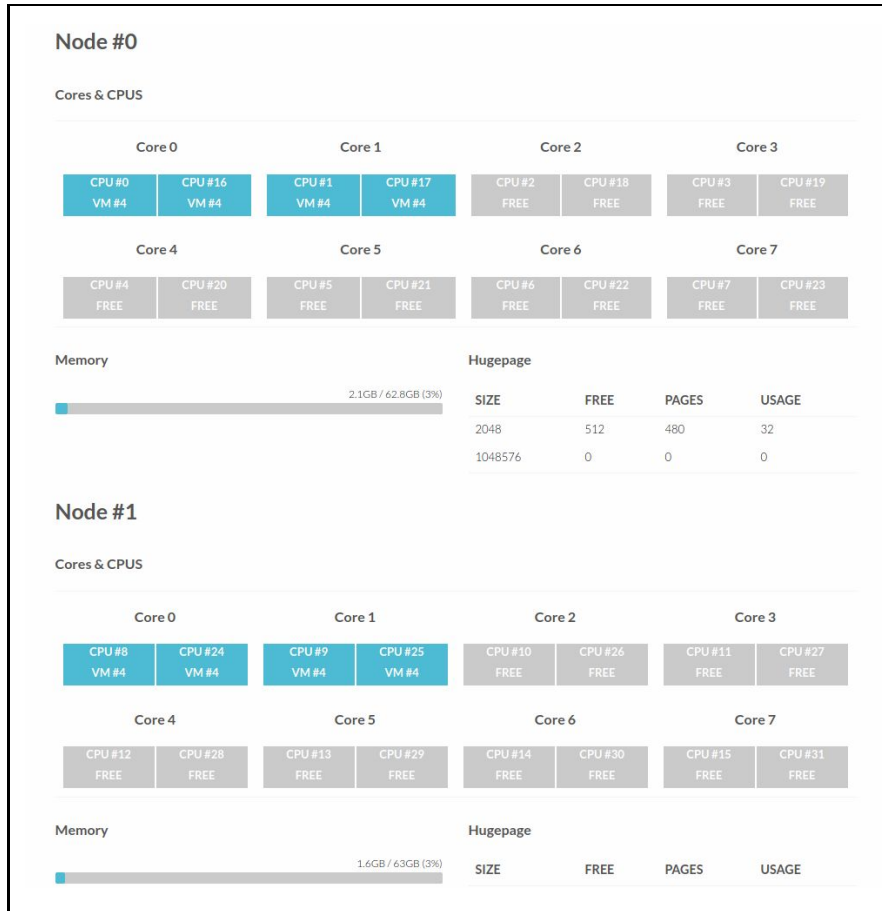


Figure 2.15. NUMA information panel as shown in Sunstone



## [SR2.8] Complete Service Flows

### Description

OneFlow core has been re-written to improve:

- **Scalability**, so now it supports a large number of running services at the same time.
- **Response time**, the response time has been hugely reduced, due to the use of asynchronous update mechanism to update the VM status.

Additionally, Flow description has been extended to include the ability to co-allocate networks together with the service VM. All the VMs that belong to the service can communicate with each other using these virtual networks that are automatically deployed. Also, when the service is destroyed, virtual networks are also destroyed, everything is managed together.

Finally, the ability to add custom attributes to the VMs that are deployed has been added. So now, users can choose some custom attributes that are passed to the VM when the service is deployed. These custom attributes can be used inside the VM to make some extra configuration steps.

### Requirements and Specifications

#### Network creation

A Service Template can define three different dynamic network modes, that determine how the networks will be used:

- An existing Virtual Network can be used, VMs in the role will just take a lease from that network. This method is used for networks with a predefined address set (e.g. public IPs).

```
{"networks_values": [{"Private":{"id":"0"}}]}
```

- Create a network reservation, in this case it will take the existing network and create a reservation for the service. The name of the reservation and the size in the input dialog must be specified. This method is used to allocate a pool of IPs for the service.

```
{"networks_values":[{"Private":{"reserve_from":"0", "extra": ""NAME=RESERVATION\nSIZE=5""}}]}
```

- Create a network instantiating a network template. In this case as an extra parameter the address range to create may need to be specified, depending on the selected template. This is useful for service private VLAN for internal service communication.

```
{"networks_values": [{"Private":{"template_id":"0", "extra":["AR=[ IP=192.168.122.10, SIZE=10, TYPE=IP4 ]"}]}]}
```



## Response time comparison

The response time has been hugely reduced. There is a comparison below, with times from 5.10.2 (before the improvement) and 5.12.0 (after the improvement):

### 1) OpenNebula 5.10.2

- Deploy (~1 minute)

```
11:15:26 [I]: [SER] New state: DEPLOYING
11:15:26 [I]: [ROL] Role Master new state: DEPLOYING
11:15:57 [I]: [ROL] Role Master new state: RUNNING
11:15:57 [I]: [ROL] Role Slave new state: DEPLOYING
11:16:27 [I]: [ROL] Role Slave new state: RUNNING
11:16:27 [I]: [SER] New state: RUNNING
```

- Scale (~1 minute 20 seconds)

```
11:24:10 [I]: [ROL] Role Master scaling down from 2 to 1 nodes
11:24:10 [I]: [ROL] Role Master new state: SCALING
11:24:10 [I]: [SER] New state: SCALING
11:25:03 [I]: [ROL] Role Master new state: COOLDOWN
11:25:03 [I]: [SER] New state: COOLDOWN
11:25:33 [I]: [ROL] Role Master new state: RUNNING
11:25:33 [I]: [SER] New state: RUNNING
```

Note: cooldown takes 10 seconds, so real time is 1 minute and 10 seconds.

- Warning (23 seconds)

```
12:30:18 [Z0][DiM][D]: Powering off VM 1
12:30:41 [I]: [ROL] Role Slave new state: WARNING
12:30:41 [I]: [SER] New state: WARNING
```

### 2) OpenNebula 5.12.0

- Deploy (11 seconds)

```
11:47:09 [I]: [SER] New state: DEPLOYING
11:47:09 [I]: [ROL] Role Master new state: DEPLOYING
11:47:15 [I]: [ROL] Role Master new state: RUNNING
11:47:15 [I]: [ROL] Role Slave new state: DEPLOYING
11:47:20 [I]: [ROL] Role Slave new state: RUNNING
11:47:20 [I]: [SER] New state: RUNNING
```

- Scale (15 seconds)

```
12:00:48 [I]: [ROL] Role Master scaling up from 1 to 2 nodes
12:00:48 [I]: [ROL] Role Master new state: SCALING
12:00:48 [I]: [SER] New state: SCALING
12:00:53 [I]: [SER] New state: COOLDOWN
12:00:53 [I]: [ROL] Role Master new state: COOLDOWN
12:01:03 [I]: [ROL] Role Master new state: RUNNING
12:01:03 [I]: [SER] New state: RUNNING
```

Note: cooldown takes 10 seconds, so real time is 5 seconds.

- Warning (1 second)

```
12:07:01 [Z0][DiM][D]: Powering off VM 10
12:07:02 [I]: [SER] New state: WARNING
12:07:02 [I]: [ROL] Role Master new state: WARNING
```

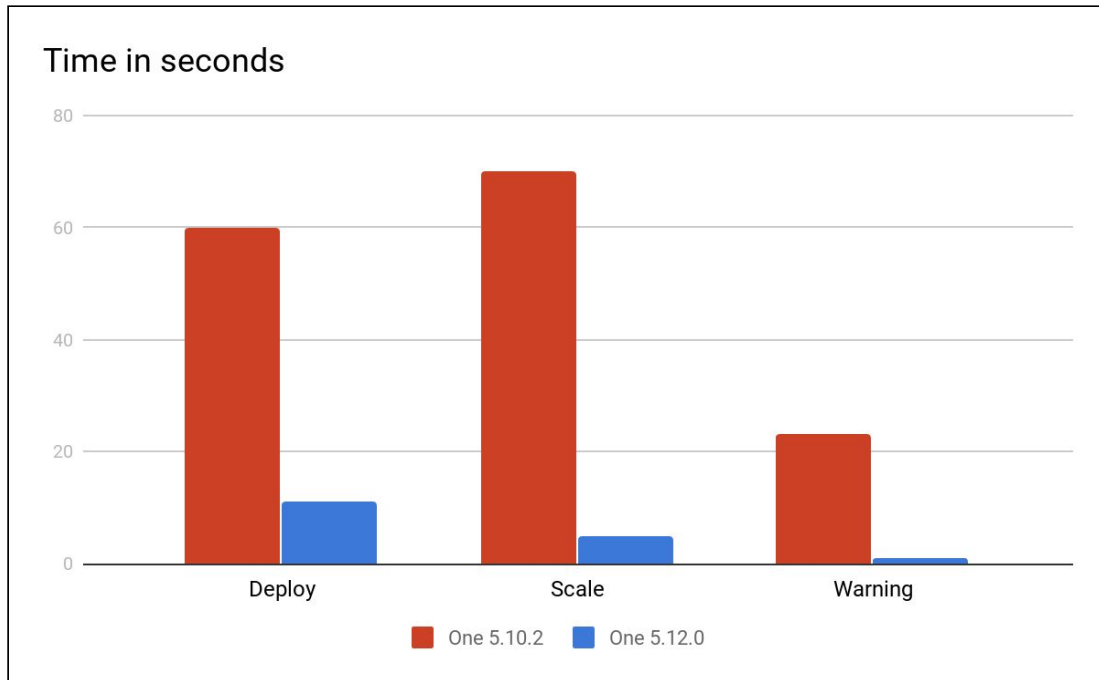


Figure 2.16. Times comparison

## Architecture and Components

Some of the components have been re-written and new ones have been created:

- **Life cycle manager:** this component has been completely re-written. Instead of iterating over the service pool, it triggers asynchronous actions to perform the operations.
- **Event manager:** this component is new. It is in charge of waiting for state changes, when the state changes it notifies it to the LCM.
- **Service watchdog:** this component is new. It is in charge of checking the state of the VM and notifies when the VM is not running. In this situation the service and role will be in a warning state. When the situation is recovered, it will also notify so the service and role can change their states.
- **Service auto scaler:** this component is new. It is in charge of checking the elasticity and scheduled rules to apply them.

## Architecture

The OneFlow is an API rest. The OneFlow server executes an initial check of the data received and if everything is correct it calls the LCM. The LCM will read the service information and then is going to send the actions to the event manager. This component will talk with OpenNebula

and wait until states change. After that, it will notify the LCM. These operations will be performed until all the roles are in the desired states. Figure 2.17 shows the deploy operation diagram.

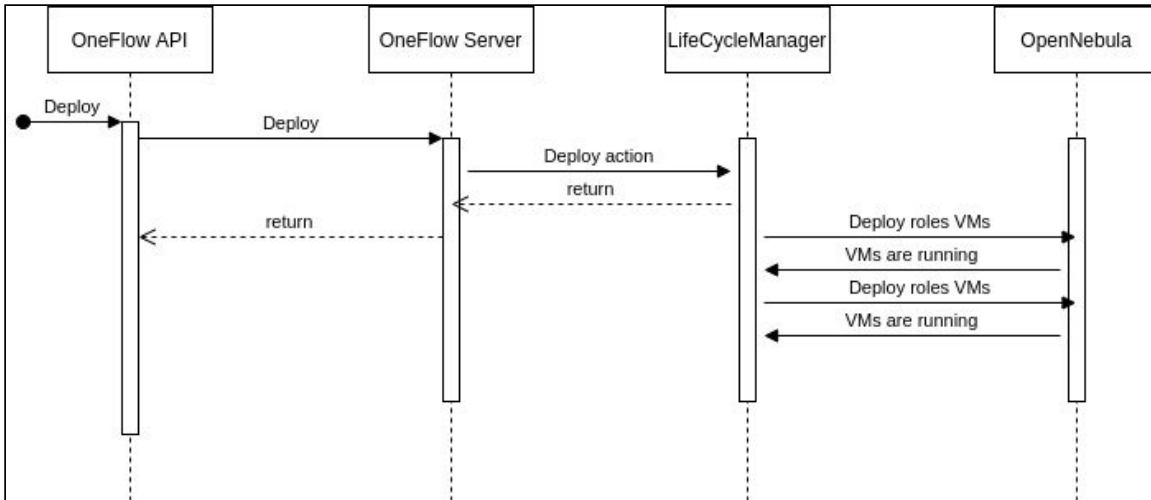


Figure 2.17. Deploy operation diagram

### Data Model

The information related with services is stored in JSON format. Table 2.11 shows attributes that can be set in a Service template and Table 2.12 shows attributes that are available in a service.

Attribute	Meaning
Name	Service template name
Deployment	<ul style="list-style-type: none"> <li>• <b>None:</b> all roles are deployed at once</li> <li>• <b>Straight:</b> roles are deployed following parent relationships</li> </ul>
Description	Description of what kind of service is going to be deployed
Roles	Array of roles that are going to be deployed in the service, each role has: <ul style="list-style-type: none"> <li>• <b>name:</b> name of the role</li> <li>• <b>cardinality:</b> number of VMs that are going to be deployed</li> <li>• <b>vm_template:</b> virtual machine template ID to deploy</li> <li>• <b>min_vms:</b> minimum number of VMs that the roles must have</li> <li>• <b>max_vms:</b> maximum number of VMs that the roles can have</li> <li>• <b>shutdown_action:</b> action to perform when undeploying the role                             <ul style="list-style-type: none"> <li>○ terminate</li> <li>○ terminate-hard</li> </ul> </li> <li>• <b>parents:</b> array of parents roles</li> <li>• <b>elasticity_policies:</b> rules to change service cardinality</li> <li>• <b>scheduled_policies:</b> scheduled rules to change service cardinality</li> </ul>
Shutdown_action	Action to perform when undeploying the role: <ul style="list-style-type: none"> <li>• terminate</li> <li>• terminate-hard</li> </ul>



Ready_status_gate	If this is set to true, the roles are not considered as running until VMs communicate to OneGate that they are ready
Networks	Networks that are going to be created
Custom_attrs	Custom attributes to pass to the VM when the template is instantiated

**Table 2.11.** Service template attributes and their meaning

Attribute	Meaning
Name	Service name
Deployment	<ul style="list-style-type: none"> <li>• <b>None:</b> all roles are deployed at once</li> <li>• <b>Straight:</b> roles are deployed following parent relationships</li> </ul>
Description	Service description
Roles	Array of service roles, each role has: <ul style="list-style-type: none"> <li>• <b>name:</b> name of the role</li> <li>• <b>cardinality:</b> current number of VMs</li> <li>• <b>vm_template:</b> virtual machine template ID</li> <li>• <b>min_vms:</b> minimum number of VMs that the roles must have</li> <li>• <b>max_vms:</b> maximum number of VMs that the roles can have</li> <li>• <b>shutdown_action:</b> action to perform when undeploying the role               <ul style="list-style-type: none"> <li>○ terminate</li> <li>○ terminate-hard</li> </ul> </li> <li>• <b>parents:</b> array of parents roles</li> <li>• <b>elasticity_policies:</b> rules to change service cardinality</li> <li>• <b>scheduled_policies:</b> scheduled rules to change service cardinality</li> <li>• <b>vm_template_contents:</b> information passed to VM</li> <li>• <b>state:</b> role state, see Table 2.13 for more details</li> <li>• <b>cooldown:</b> seconds to wait after scale operation</li> <li>• <b>nodes:</b> array containing information about role VMs:               <ul style="list-style-type: none"> <li>○ <b>deploy_id:</b> VM identifier</li> <li>○ <b>vm_info:</b> hash containing basic information about the VM</li> </ul> </li> </ul>
Shutdown_action	Action to perform when undeploying the role: <ul style="list-style-type: none"> <li>• terminate</li> <li>• terminate-hard</li> </ul>
Ready_status_gate	If this is set to true, the roles are not considered as running until VMs communicate to OneGate that they are ready
Networks	networks that are going to be created
Networks_values	chosen networks
Custom_attrs	custom attributes to pass to the VM when the template is instantiated
Custom_attrs_values	custom attributes values passed to the VM
State	service state, see Table 2.13 for more details
Log	array of service logging events

**Table 2.12.** Service attributes and their meaning



State	Name	Meaning
0	PENDING	Service/role is waiting to be deployed by the server
1	DEPLOYING	Service/role is being deployed, this means, VM are being deployed by OpenNebula
2	RUNNING	Service/role is running, this means all the VMs are in running state
3	UNDEPLOYING	Service/role is being undeployed, so VMs and virtual networks are going to be destroyed
4	WARNING	Some of the VMs are in a not running state
5	DONE	Service and roles are deleted
6	FAILED_UNDEPLOYING	Service/role failed to undeploy
7	FAILED_DEPLOYING	Service/role failed to deploy
8	SCALING	Service/role is scaling UP/DOWN, this means, cardinality is being adjusted
9	FAILED_SCALING	Service/role failed to scale
10	COOLDOWN	Service/role is in waiting cooldown seconds after scale operation

**Table 2.13.** Service and role states

An example of service template can be checked below:

```
{
  "name": "TestLab",
  "deployment": "straight",
  "description": "",
  "roles": [
    {
      "name": "frontend",
      "cardinality": 1,
      "vm_template": 127,
      "elasticity_policies": [
      ],
      "scheduled_policies": [
      ]
    },
    {
      "name": "worker",
      "cardinality": 1,
      "vm_template": 120,
      "elasticity_policies": [
      ],
      "scheduled_policies": [
      ]
    }
  ],
  "ready_status_gate": false,
  "networks": {
  },
  "custom_attrs": {
  }
}
```

**Figure 2.18.** Service template example





## API and Interfaces

### API

The API remains untouched.

### CLI

The CLI has been re-written, to follow command/helper pattern. The command oneflow-template has been adapted to read network and custom attributes information. The oneflow list command has been updated to show colored state and also to fit the terminal width.

### Sunstone

The tab related with service template definition has been redesigned to be able to choose the networks and attributes, Figure 2.19 shows a screenshot of the tab:



^ Network configuration

Mandatory	Name	Description	Type	Network	Extra
<input checked="" type="checkbox"/>	test	test	Create		

+ Network

^ Custom Attributes Values Configuration

Name	Type	Description	Mandatory
Test	text	Test	<input checked="" type="checkbox"/>

Default value: Test

+ Network

^ Advanced service parameters

### Roles

Test

Role name:  Number of VMs:

VM template to create this role's VMs

You selected the following Template: Ttylinux - KVM

ID	Name	Owner	Group	Registration time
1	Ttylinux - KVM	oneadmin	oneadmin	08/07/2020 11:43:15
0	testing	oneadmin	oneadmin	08/07/2020 11:41:39

10 Showing 1 to 2 of 2 entries Previous 1 Next

^ Role network

Network Interfaces  test  As nic alias

RDP:

Figure 2.19. Service template network and custom attributes configuration

Template instantiation dialog has also been redesigned to be able to choose virtual networks and custom attributes, Figure 2.20 shows a screenshot of the tab:

←
Instantiate

Service name ?

Number of instances


### Test

Network

test

Create

You selected the following network template: ■ ↻

ID	Name	Owner	Group	Labels
 There is no data available				

10
Showing 0 to 0 of 0 entries
Previous Next

Extra

Custom Attributes

Test

Test

Add Charters Values Configuration

?

Figure 2.20. Service template instantiation dialog

### Configuration file

The main configuration file is located `/etc/one/oneflow-server.conf`.

The following new attributes have been added:

- **wait\_timeout**: default timeout in seconds to wait VMs to report different states
- **autoscaler\_interval**: time in seconds between each time scale rules are evaluated

The following attributes have been removed from:

- **lcm\_interval**: time in seconds between Life Cycle Manager steps.



## [SR2.9] Web UI extensions

The web extension task is a horizontal development task. Each SR includes in section API and Interfaces a description of the modification to the web UI to accommodate the new functionality. Note that these extensions do not apply to all SR.



## 3. Edge Provider Selection (CPNT3)

### [SR3.4] Driver Maintenance Process

#### Description

The Amazon EC2 and Packet drivers from the Edge Infrastructure Provision component drivers lay the basis to build an acceptance and certification process as well as the needed testing framework and documentation.

#### Requirements and Specification

The provision driver communicates with the remote infrastructure provider to allocate and control new resources. These resources are later added into the OpenNebula as virtualization hosts. There are several benefits of this approach over the traditional, more decoupled hybrid solution that involves using the provider cloud API. However, one of them stands out among the rest; it is the ability to move offline workloads between your local and rented resources.

In this first cycle the development of the Amazon EC2 and Packet drivers allows to partially address the following requirements:

- Improve integration guides for host, network and IPAM drivers.
- Develop driver architecture and ready-to-use driver skeletons that eases development.
- Define an acceptance and certification process for each driver type.
- Create development and process guides to foster development within the ecosystem.

#### Architecture and Components

The OneProvision infrastructure drivers allow the deployment of a fully operational OpenNebula cluster in a remote provider. Each new provision is described by the [provision template](#), a YAML document specifying the OpenNebula resources to add (cluster, hosts, datastores, virtual networks), physical resources to provision from the remote infrastructure provider, the connection parameters for SSH and configuration steps (playbook) with tunables. At the end of the process, there is a new cluster available in OpenNebula.

The sequence diagram below represents a full lifecycle of a remote resource, depicting the flow between the OneProvision component and the remote infrastructure provider (Amazon EC2 in this case) using the Infrastructure Provision drivers, and the OneProvision component and OpenNebula core using XMLRPC client wrappers.

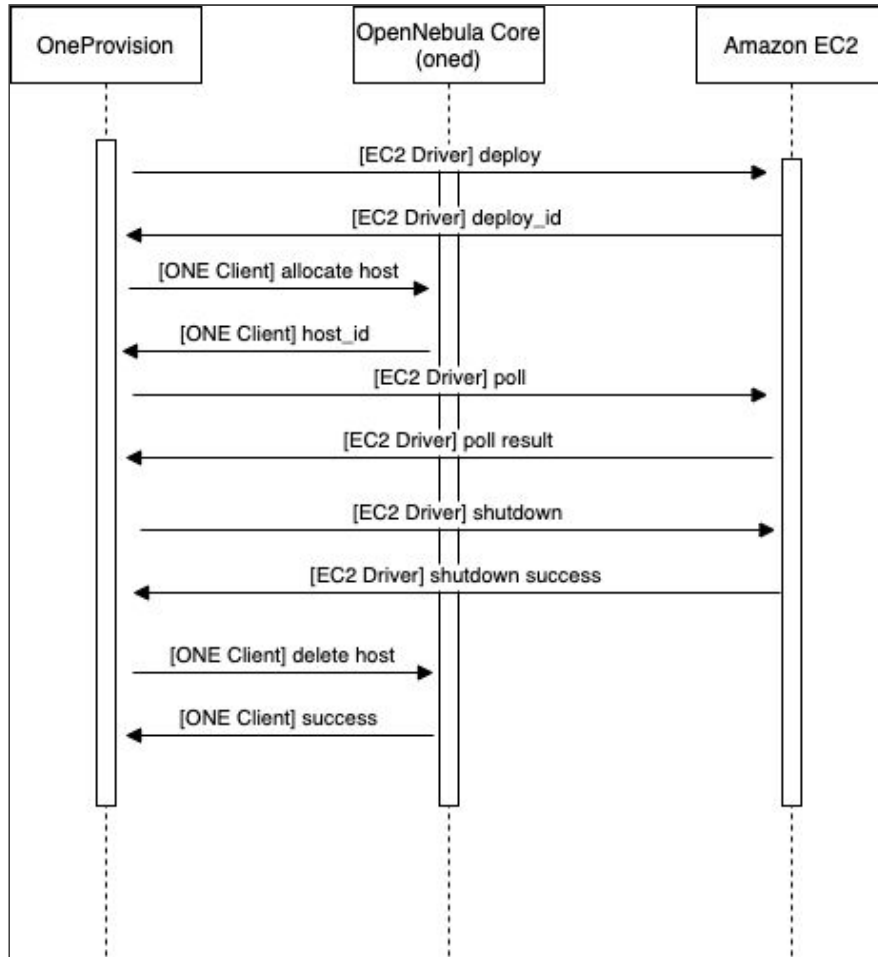


Figure 3.1. Sequence diagram for OneProvision drivers

### Data Model

The remote host representation in OpenNebula holds special attributes in the resource template storing the values needed to correctly manage the remote host.

An XML example of the representation can be found in Figure 3.2.

```

<PROVISION>
  <AMI><![CDATA[ami-66a7871c]]></AMI>
  <DEPLOY_ID><![CDATA[i-0275bfc0bc1d3787f]]></DEPLOY_ID>
  <EC2_ACCESS><![CDATA[chnAtXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXQc=]]></EC2_ACCESS>
  <EC2_SECRET><![CDATA[YWep/0QwXXXXXXXXXXXXXXXXXXXXXXXXXXXXMxopZ]]></EC2_SECRET>
  <HOSTNAME><![CDATA[centos-host]]></HOSTNAME>
  <INSTANCETYPE><![CDATA[t2.nano]]></INSTANCETYPE>
  <NAME><![CDATA[testing_provision]]></NAME>
  <PROVISION_ID><![CDATA[6436cfbc-d266-49ce-9d4b-2fc4db1297a3]]></PROVISION_ID>
  <REGION_NAME><![CDATA[us-east-1]]></REGION_NAME>
  <SECURITYGROUPSIDS><![CDATA[sg-0651cef55568e8f55]]></SECURITYGROUPSIDS>
  <SUBNETID><![CDATA[subnet-093c9f349a22f2571]]></SUBNETID>
</PROVISION>
<PROVISION_CONFIGURATION_BASE64><![CDATA[LS0tCm9wZW5uZWJ1bGFfbm9kZV9rdm1fcGFyYW1fbmVzdGVkOiBmYWxzZQo=]]></PROVISION_CONFIGURATION_BASE64>
    
```

```

<PROVISION_CONFIGURATION_STATUS><![CDATA[configured]]></PROVISION_CONFIGURATION_STATUS>
<PROVISION_CONNECTION>
  <PRIVATE_KEY><![CDATA[/var/lib/one/.ssh/id_rsa]]></PRIVATE_KEY>
  <PUBLIC_KEY><![CDATA[/var/lib/one/.ssh/id_rsa.pub]]></PUBLIC_KEY>
  <REMOTE_PORT><![CDATA[22]]></REMOTE_PORT>
  <REMOTE_USER><![CDATA[root]]></REMOTE_USER>
</PROVISION_CONNECTION>

```

**Figure 3.2.** XML Representation of a remote node

An exhaustive list of the attributes that describes an EC2 provision are described in Table 3.1.

Attribute	Meaning
AMI	Identifier of the Amazon EC2 bare metal template
DEPLOY_ID	Identifier of the Amazon EC2 bare metal instance
EC2_ACCESS	Access key for EC2 account
EC2_SECRET	Secret key for EC2 account
HOSTNAME	Name of the host in OpenNebula
INSTANCE_TYPE	Type of the EC2 bare metal instance
NAME	Name of the provision
PROVISION_ID	Identifier of the provision
REGION_NAME	Name of the EC2 region where the bare metal instance is located
SECURITYGROUPSIDS	Identifiers of the Security Groups in Amazon EC2 that applies to this bare metal instance provision
SUBNETID	Identifier of the subnet that contains this Amazon EC2 bare metal instance
PROVISION_CONFIGURATION_BASE64	Information needed to configure the Amazon EC2 bare metal instance as part of an OpenNebula cluster, encoded in base64.
PROVISION_CONFIGURATION_STATUS	The state of the configuration task over the Amazon EC2 bare metal instance
PRIVATE_KEY	Path to private key to connect to the remote server using the SSH protocol
PUBLIC_KEY	Path to the SSH public key to authorize in the remote server
REMOTE_PORT	Remote server's SSH port
REMOTE_USER	Remote server's user account name

**Table 3.1.** Provision attributes semantics



## API and Interfaces

Each new edge infrastructure provider needs a set of provision drivers to claim compatibility with OneProvision. These drivers are responsible for the creation and lifecycle management of the bare metal instances that sustain the edge platforms.

The following table describes the API that each set of drivers needs to implement.

Driver Name	Description	Arguments	Response
cancel	Destroy a provision	<ul style="list-style-type: none"> <li>DEPLOY_ID: Provision deployment ID</li> <li>HOST: Name of OpenNebula host</li> </ul>	<ul style="list-style-type: none"> <li>Success: -</li> <li>Failure: Error message</li> </ul>
deploy	Identifier of the Amazon EC2 bare metal instance	DEPLOYMENT_FILE: where to write the deployment file, which contents come through stdin	<ul style="list-style-type: none"> <li>Success: Deploy ID, unique identification from provider</li> <li>Failure: Error message</li> </ul>
poll	Get information about a provisioned host	<ul style="list-style-type: none"> <li>DEPLOY_ID: Provision deployment ID</li> <li>HOST: Name of OpenNebula host</li> </ul>	<ul style="list-style-type: none"> <li>Success: Output as for poll action in the Virtualization Driver</li> <li>Failure: Error message</li> </ul>
reboot	Orderly reboots a provisioned host	<ul style="list-style-type: none"> <li>DEPLOY_ID: Provision deployment ID</li> <li>HOST: Name of OpenNebula host</li> </ul>	<ul style="list-style-type: none"> <li>Success: -</li> <li>Failure: Error message</li> </ul>
reset	Hard reboots a provisioned host	<ul style="list-style-type: none"> <li>DEPLOY_ID: Provision deployment ID</li> <li>HOST: Name of OpenNebula host</li> </ul>	<ul style="list-style-type: none"> <li>Success: -</li> <li>Failure: Error message</li> </ul>
shutdown	Orderly shutdown a provisioned host	<ul style="list-style-type: none"> <li>DEPLOY_ID: Provision deployment ID</li> <li>HOST: Name of OpenNebula host</li> <li>LCM_STATE: Emulated LCM_STATE string</li> </ul>	<ul style="list-style-type: none"> <li>Success: -</li> <li>Failure: Error message</li> </ul>

**Table 3.2.** Provision drivers API





## 4. Edge Infrastructure Provision and Deployment (CPNT4)

### [SR4.1] Reliable Edge Resource Provision

#### Description

Provisioning tools were improved for better resilience to the errors by multi-staged handling of error situations (called failover combinations). With this implementation users can combine multiple types of options in case anything fails. This is very powerful as it allows users to deal with more complex error situations. The error handling options were available in the previous version of the provisioning tools, but now users can combine them. For example, users are able to retry several times the failing configuration step of deployed hosts and in case of recurrent failure, delete the provision and release the allocated hosts back to Edge provider as a next failover step. All within a single command run.

For deployments which did not succeed and resources were left running on Edge location outside the inventory of OpenNebula provisioning tools, the prototype of background cleaner was developed and is still under testing and evaluation.

#### Requirements and Specifications

##### Failover Combinations

The new failover combinations for multi-staged error handling are available only in the CLI. This improvement is implemented in a backward compatible way, so users of previous versions can still use the same failover modes without combining it with others. All the fail modes can be combined with each other, there is no restriction on that and also all the options related to each fail mode are supported. Combination of failover modes is available both in interactive and non-interactive runs.

##### Background Cleaner

Cleaner tool lists the running machines on Edge directly over Edge Provider's API and terminates those left running. As a prototype it does not (have to) distinguish between resources in the OpenNebula evidence and orphaned ones and can clean everything. This is going to be extended in next iteration to:

- identify resources without relation to the provisioned deployments (true orphaned ones).
- cleanup also other resources than just virtual machines (e.g., IP ranges, persistent disks).
- run automatically and periodically.

Providers supported by Edge provisioning tools must and are supported by the cleaner, i.e. Packet and Amazon EC2. The tool also supports Azure for future use.

#### Architecture and Components

##### Failover Combinations

This improvement does not create any new component, the existing CLI tools (oneprovision) and provision backend have been modified to support this feature.

Users need to choose the failover options before running the command oneprovision to deploy the infrastructure on Edge. In case of error, the provision backend will follow (from one to another) the failover modes in the order they were specified by the user. These failover modes will be processed until there are no more modes available or the issue is resolved. If the error is persistent and there are no more failover modes available, the provisioning tools will terminate with error and leave the infrastructure state as is. This can be used by users to check and troubleshoot the problem.

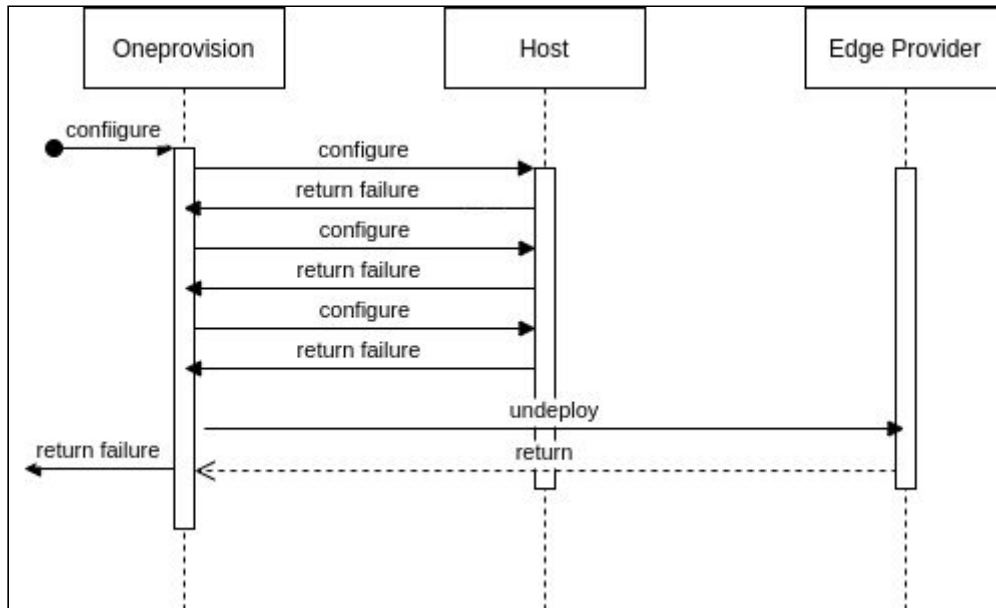


Figure 4.1. Sequence diagram of configuration process with recurrent failure

Figure 4.1 shows the sequence diagram of provision with failover combinations of *retry* and *cleanup*. The configuration process of the host if failing, and provision tools chooses to *retry* operation for 3 times. After 3 failures, the tooling continues with next *cleanup* failover mode. I.e., it contacts the Edge Provider, un-deploys the host and terminates.

Background Cleaner

This functionality does not come as a fully featured component, but only as a stand-alone script.

The Figure 4.2 shows the sequence diagram of interaction among cleaner tools and Edge Providers.

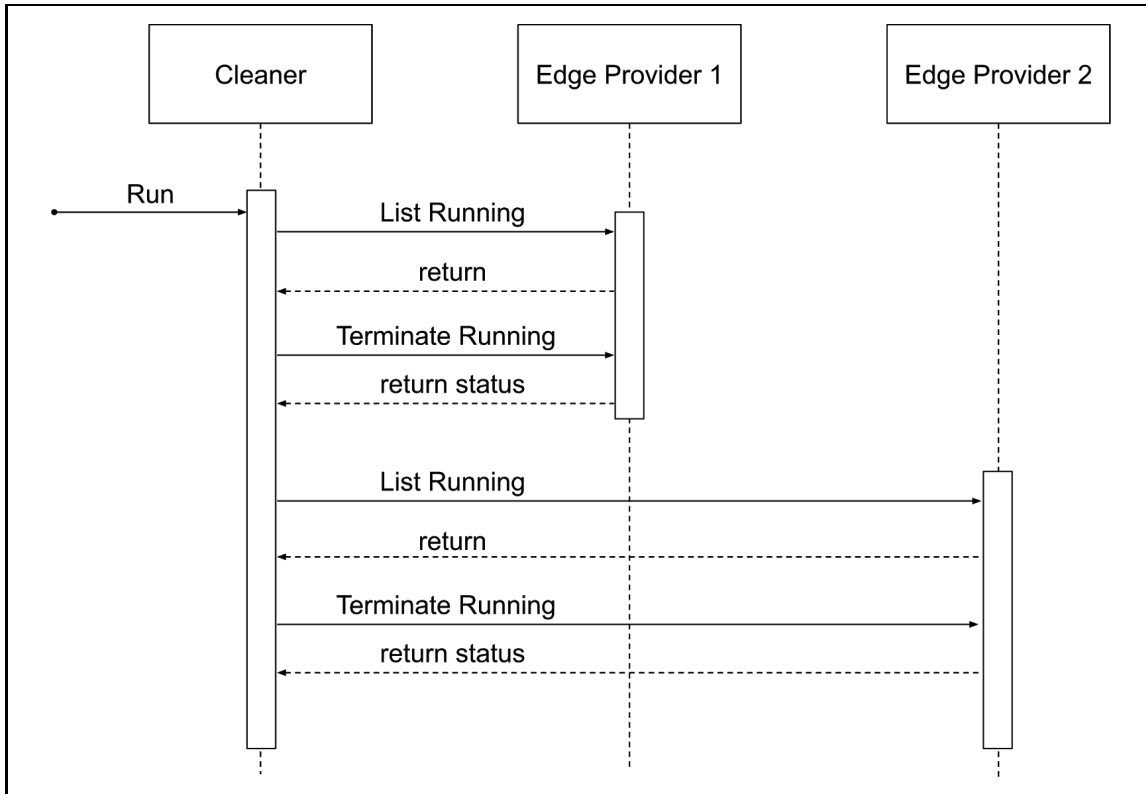


Figure 4.2. Sequence diagram of cleanup process

**Data Model**

There is no special data model related to this improvement.

**API and Interfaces**

Failover Combinations

The main change has been done in the CLI tool oneprovision and its subcommand create. It provides a new parameter to specify a group of failover modes as summarized in Table 4.1.

CLI PARAMETER	DESCRIPTION
oneprovision create --fail-modes [...]	Allows to specify comma separated list of multiple failover modes from: <ul style="list-style-type: none"> <li>• <b>retry</b> - retry the last failing operation</li> <li>• <b>cleanup</b> - cleanup all changes (on Edge Provider and ONE) and exit</li> <li>• <b>Skip</b> - skip failing operation and continue with next one</li> <li>• <b>Fail</b> - exit provision with error on failing operation</li> </ul>

Table 4.1. New command line options

Following Figure 4.3 shows a real example of provisioning CLI with multi-staged error handling via failover modes. There are combined modes to *retry* and *cleanup*. This makes the provision backend first retry the failing operation several times and at the end since the operation still fails, it will clean up the infrastructure and delete all entities created so far.

```

$ oneprovision create simple.yaml -d --batch --fail-modes retry,cleanup
2020-06-17 10:30:54 INFO : Creating provision objects
ERROR: Failed to create some resources
[one.vn.allocate] VN_MAD named "alias_sdnat" is not defined in oned.conf
ERROR: Failed to create some resources
[one.vn.allocate] VN_MAD named "alias_sdnat" is not defined in oned.conf
ERROR: Failed to create some resources
[one.vn.allocate] VN_MAD named "alias_sdnat" is not defined in oned.conf
ERROR: Failed to create some resources
[one.vn.allocate] VN_MAD named "alias_sdnat" is not defined in oned.conf
ERROR: Failed to create some resources
[one.vn.allocate] VN_MAD named "alias_sdnat" is not defined in oned.conf
2020-06-17 10:30:54 INFO : Deleting provision 5949d07f-eb06-4b0b-8e8c-60c29ff30bb1
2020-06-17 10:30:54 INFO : Undeploying hosts
2020-06-17 10:30:54 INFO : Deleting provision virtual objects
2020-06-17 10:30:54 INFO : Deleting provision objects

```

Figure 4.3. Provision fail modes retry and cleanup

### Background Cleaner

Cleaner script does not provide many options to adjust the execution (see Figure 4.4). Without any provided parameter, it checks the running machines on Edge providers (as described in the accompanying configuration file) and only shows a list of candidates for termination. If executed with argument `-f`, it also terminates those candidates.

```

$ ./cloud_cleanup.rb --help
Cloud cleanup

Usage:
  ./cloud_cleanup.rb [--config=<FILE>]
  ./cloud_cleanup.rb -f | --force
  ./cloud_cleanup.rb -h | --help

Options:
  -h --help          Show this screen
  -c --config=<FILE> Config file location [default: cloud_cleanup.yaml]
  -f --force         Really clean, dry-run otherwise

```

Figure 4.4. Cleaner script help with available command line options

Configuration of Edge providers to check and clean up is a static file with providers list, the credentials for their API and types of entities to terminate (see Figure 4.5). This needs to be improved in upcoming development iterations to use the information stored in the OpenNebula (for existing provisions) now and in the past.

```

---
provider1:
  secret_url: http://localhost/secrets/p1.yaml
  delete:
    - instances

provider2:
  secret_url: http://localhost/secrets/p2.yaml
  delete:
    - instances

```

Figure 4.5. Cloud providers configuration



## [SR4.2] Usability, Functionality and Scalability of Provision

### Description

Provisioning tools to deploy virtualization clusters in the Edge datacenters were extended to create more complete deployments in the EdgeNebula with new entities for end-users. Such deployment can be created based on a combination of multiple provision descriptors within a single deployment process. Also, can contain entities created for direct use by end users - e.g., virtual machine images and templates, multi-VM deployments descriptors.

This improvement allows users to deploy a ready-to-use infrastructure with just a single deployment. Provision backend will create all the infrastructure resources in the remote provider and then will create all the objects in OpenNebula. At the end, the user will be able to instantiate a VM template with all the resources that have been deployed.

### Requirements and Specifications

#### New Object Types

The provision descriptor and tooling is extended in order to create following new object types:

- Images
- Marketplace Appliances
- Virtual Machine Templates
- Virtual Network Templates
- Multi-VM (OneFlow) Service Templates

#### Object Owners

By default the objects are created under the identity (user and group owner) of the user which triggers the provision. New options were added to specify different custom owners of each entity. So the provision objects can be shared with other users in the cloud.

#### Asynchronous Provision

Some of the objects may take a while until they are created (e.g., download of a big image over a slow network might significantly slow down the provision). The provision can create some objects in newly introduced asynchronous mode. The provision will not wait for such objects to be ready. All of these customizations are done in the deployment file in a user friendly way.

### Architecture and Components

The provision backend has been extended to support these new objects. They are called virtual objects, as they are created only in the OpenNebula (and are not the infrastructure objects like hosts or datastores) and not on any remote provider. The objects are specified in the provision template, which is a YAML formatted file with all the resources that have to be deployed during provision.

The following flow is executed to deploy the provision:

1. The deployment file is read and validated.

2. All the infrastructure objects are created in OpenNebula and in the remote provider, so the backend will wait until the resources are ready in the provider.
3. Hosts will be configured using Ansible recipes.
4. When all the infrastructure is ready, the backend will start creating all the virtual objects and will wait for them to be ready.

## Data Model

### Provision Template

Provision template has been extended to support the new objects. For each new object there is a new section in the provision template, so the provision tool can distinguish and deploy them. Following Figures 4.6 - 4.10 show the examples of snippets of provision templates for different object types:

```
templates:  
- name: "test_template"  
  memory: 1  
  cpu: 1
```

Figure 4.6. Provision template for Virtual Machine Template

```
vntemplates:  
- name: "test_vntemplate"  
  vn_mad: "bridge"  
  ar:  
  - ip: "10.0.0.1"  
    size: 10  
    type: "IP4"
```

Figure 4.7. Provision template for Virtual Network Template

```
images:  
- name: "test_image"  
  ds_id: 1  
  size: 2048
```

Figure 4.8. Provision template for Image

```
marketplaceapps:  
- appid: 238  
  name: "test_image_2"  
  dsid: 1
```

Figure 4.9. Provision template for Marketplace Appliance

```
flowtemplates:  
- name: "test_service"  
  deployment: "straight"  
  roles:  
  - name: "frontend"  
    vm_template: 0  
  - name: "backend"  
    vm_template: 1
```

Figure 4.10. Provision template for Multi-VM Service Template

### Enriched OpenNebula XML Objects Metadata

All objects created in the OpenNebula are represented as XML documents, with main fixed structure and free form extensible TEMPLATE section which contains various essential metadata about each object. Provision process injects into each created OpenNebula object a unique identification of provision run during which the objects were created. This is used to manage the provision as one entity, so when the provision is being destroyed, all the related virtual objects are destroyed as well. Figure 4.11 shows a partial example of the Image XML object with TEMPLATE section, which contains reference ID to the provision.

```
<TEMPLATE>
  <DEV_PREFIX><![CDATA[sd]]></DEV_PREFIX>
  <DS_ID><![CDATA[1]]></DS_ID>
  <PROVISION>
    <PROVISION_ID><![CDATA[953d632b-8c66-49b4-bd90-ffc45765bd60]]></PROVISION_ID>
    <WAIT><![CDATA[false]]></WAIT>
  </PROVISION>
</TEMPLATE>
```

**Figure 4.11.** OpenNebula object template

### **API and Interfaces**

New functionality needed to be implemented in provisioning tools to track and deal with newly created objects, injected reference metadata, and mechanism to skip specific parts of provision.

### Provision XML Object

Provisioning tools returns a complete description of provision as an XML document. This document needed to be extended to contain a list of newly created object types. Figure 4.12 shows an example of existing provision covering also Images.

```
<PROVISION>
  <ID>953d632b-8c66-49b4-bd90-ffc45765bd60</ID>
  <NAME>myprovision</NAME>
  <STATUS>pending</STATUS>
  <CLUSTERS>
    <ID>101</ID>
  </CLUSTERS>
  <DATASTORES>
    <ID>103</ID>
    <ID>102</ID>
  </DATASTORES>
  <HOSTS>
    <ID>2</ID>
  </HOSTS>
  <NETWORKS>
    <ID>1</ID>
  </NETWORKS>
  <IMAGES>
    <ID>1</ID>
  </IMAGES>
</PROVISION>
```

**Figure 4.12.** Provision in XML format



### New CLI parameters

New command line arguments for provisioning tools to skip infrastructure provision and/or configuration were introduced. This is very useful in case a user only wants to deploy virtual objects, not a real physical infrastructure. Table 4.2 describes the new CLI options.

CLI PARAMETER	DESCRIPTION
<b>oneprovision create --skip-provision</b>	Skips provision on remote Edge provider and configuration phase. Only creates objects in OpenNebula.
<b>oneprovision create --skip-configuration</b>	Provision hosts on remote Edge provider, but skips configuration phase. Leaves hosts unconfigured (e.g., without KVM hypervisor).

**Table 4.2.** New command line options





## [SR4.3] Provision Template for Reference Architectures

### Description

Provision templates were extended with examples of complete deployment specifications of fully usable clusters. The example specifications can be used by experienced cloud administrators to easily create ready-to-use clusters with a single run of provisioning tools.

These templates are available for Packet and Amazon EC2 which are the only supported Edge providers by OpenNebula now. Templates contain complete cluster specification with everything needed, the only change that must be performed by the end administrator is to put his own Edge provider's account with credentials and uncomment and update hosts he wants to deploy.

### Requirements and Specifications

#### Complete Cluster Provision Templates

These templates are ready-to-use complete cluster templates with (commented) hosts, datastores and virtual networks specific for each provider. They are expected to be run on OpenNebula version 5.12 or higher, packaged and installed with OpenNebula into `/usr/share/one/oneprovision/examples` on the front-end.

They have basic provision configuration with:

- **Hosts** - these hosts are commented in the template, and left up to the user to select the required counts and sizing. The commented hosts contain examples for selecting CentOS 7 and Ubuntu 18.04 LTS as base operating systems installed on the host.
- **Datastores** - necessary system and image datastores (to store image and running VMs state).
- **Virtual Networks** - necessary virtual networks for various types of communications:
  - **private-host-only-nat** - for inter-VM host-only networking and NATed access to public
  - **private-host-only** - for inter-VM host-only networking (only Packet)
  - **private** - for inter-VM private networking across physical hosts
  - **public** - for public networking over dedicated range of public IP addresses (only Packet)

Places the cloud admin needs to update before using are clearly labeled and commented.

### Architecture and Components

There is no relevant information in this section, the component itself remains untouched.

### Data Model

The only introduced changes are the new provision template descriptors, which are the YAML formatted documents. This section documents these templates at the state current for OpenNebula 5.12.0 release.



## Packet Provision Template

Figure 4.13 shows the whole provision template for complete cluster on Packet Edge provider with commented hosts, several datastores and virtual networks.

```

---
#####
# WARNING: You need to replace ***** values with your
# own credentials for the particular provider. You need to
# uncomment and update list of hosts to deploy based
# on your requirements.
#####

# Ansible playbook to configure hosts
playbook: "static_vxlan"

# Provision name to use in all resources
name: "PacketCluster"

# Defaults sections with information related with Packet
defaults:
  provision:
    driver: "packet"
    packet_token: "*****"
    packet_project: "*****"
    facility: "ams1"
    plan: "baremetal_0"
    os: "centos_7"
  configuration:
    iptables_masquerade_enabled: false # NAT breaks public networking

# Hosts to be deployed in Packet and created in OpenNebula
hosts:
# - reserved_cpu: "100"
#   im_mad: "kvm"
#   vm_mad: "kvm"
#   provision:
#     hostname: "centos-host"
#     os: "centos_7"

# - reserved_cpu: "100"
#   im_mad: "kvm"
#   vm_mad: "kvm"
#   provision:
#     hostname: "ubuntu-host"
#     os: "ubuntu_18_04"

# Datastores to be created in OpenNebula
datastores:
  - name: "<%= @name %>-image"
    ds_mad: "fs"
    tm_mad: "ssh"

```

```

- name: "<%= @name %>-system"
  type: "system_ds"
  tm_mad: "ssh"

# Network to be created in OpenNebula
networks:
- name: "<%= @name %>-private-host-only"
  vn_mad: "dummy"
  bridge: "br0"
  description: "Host-only private network"
  gateway: "192.168.150.1"
  ar:
    - ip: "192.168.150.2"
      size: "253"
      type: "IP4"

- name: "<%= @name %>-private"
  vn_mad: "dummy"
  bridge: "vxbr100"
  mtu: "1450"
  description: "Private networking"
  ar:
    - ip: "192.168.160.2"
      size: "253"
      type: "IP4"

- name: "<%= @name %>-public"
  vn_mad: "alias_sdnat"
  external: "yes"
  description: "Public networking"
  ar:
    - size: "4" # select number of public IPs
      type: "IP4"
      ipam_mad: "packet"
      packet_ip_type: "public_ipv4"
      facility: "ams1"
      packet_token: "*****"
      packet_project: "*****"

```

Figure 4.13. Provision template for Packet

Figure 4.14 shows a deployment diagram on Packet provider that can be achieved using the deployment template in Figure 4.13. This is a deployment with two different CentOS 7 hosts running, private connectivity between VMs on different hosts and public connectivity to those VMs using the virtual networks that the provision creates.

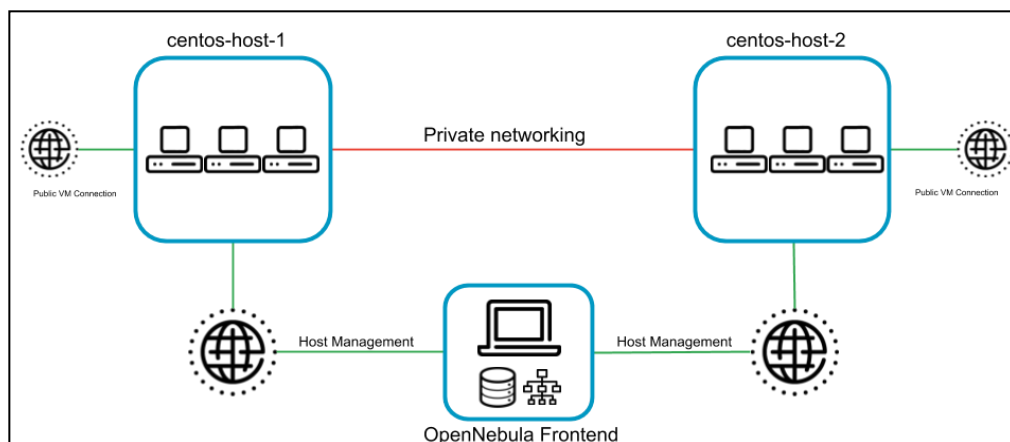


Figure 4.14. Deployment diagram of cluster provisioned on Packet



## EC2 Provision Template

Figure 4.15 shows the whole provision template for complete cluster on Amazon EC2 provider with commented hosts, several datastores and virtual networks.

```

---
#####
# WARNING: You need to replace ***** values with your
# own credentials for the particular provider. You need to
# uncomment and update list of hosts to deploy based
# on your requirements.
#####

# Ansible playbook to configure hosts
playbook: "static_vxlan"

# Provision name to use in all resources
name: "EC2Cluster"

# Defaults sections with information related with Packet
defaults:
  provision:
    driver: "ec2"
    instancetype: "i3.metal"
    ec2_access: "*****"
    ec2_secret: "*****"
    region_name: "us-east-1"
    cloud_init: true

# Hosts to be deployed in Packet and created in OpenNebula
hosts:
# - reserved_cpu: "100"
#   im_mad: "kvm"
#   vm_mad: "kvm"
#   provision:
#     hostname: "centos-host"
#     ami: "ami-66a7871c"

# - reserved_cpu: "100"
#   im_mad: "kvm"
#   vm_mad: "kvm"
#   provision:
#     hostname: "ubuntu-host"
#     ami: "ami-759bc50a" # (Ubuntu 16.04)

# Datastores to be created in OpenNebula
datastores:
- name: "<%= @name %>-image"
  ds_mad: "fs"
  tm_mad: "ssh"

- name: "<%= @name %>-system"
  type: "system_ds"
  tm_mad: "ssh"

# Network to be created in OpenNebula
networks:
- name: "<%= @name %>-private-host-only-nat"
  vn_mad: "dummy"
  bridge: "br0"
  dns: "8.8.8.8 8.8.4.4"
  gateway: "192.168.150.1"
  description: "Host-only private network with NAT"

```

```

ar:
  - ip: "192.168.150.2"
    size: "253"
    type: "IP4"

- name: "<%= @name %>-private"
  vn_mad: "dummy"
  bridge: "vxbr100"
  mtu: "1450"
  description: "Private networking"
  ar:
    - ip: "192.168.160.2"
      size: "253"
      type: "IP4"

```

Figure 4.15. Provision template for EC2

Figure 4.16 shows a deployment diagram on Amazon EC2 provider that can be achieved using the deployment template in Figure 4.15. This is a deployment with two different CentOS 7 hosts running on EC2, private connectivity between VMs on different hosts and connectivity from private to public places over NAT. This is the full deployment created by provisioning tools.

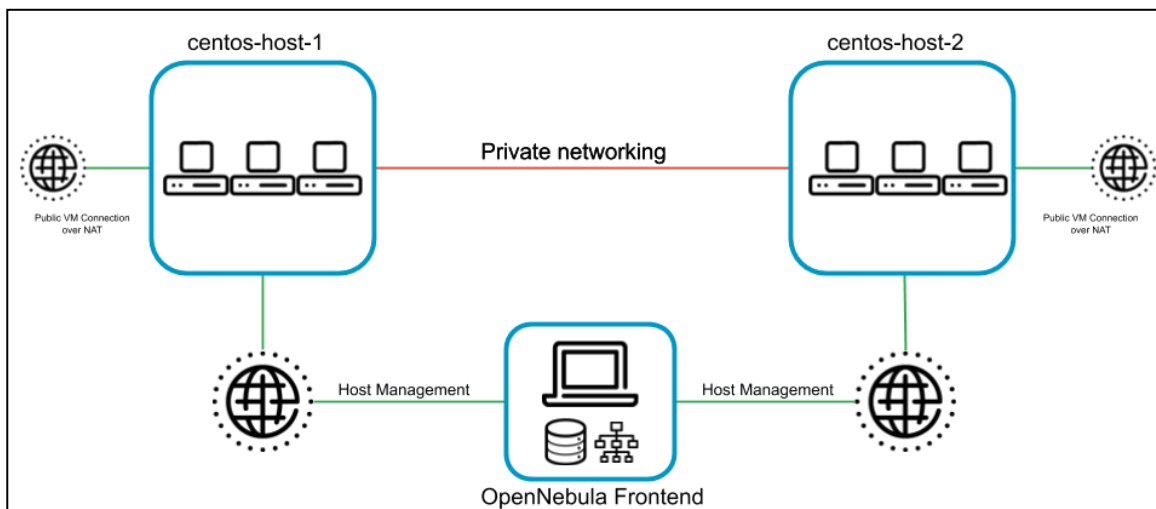


Figure 4.16. Deployment diagram of cluster provisioned on Amazon EC2

## API and Interfaces

There are no changes to report.



## 5. Edge Apps Marketplace (CPNT5)

### [SR5.2] Built-in Management of Application Containers Engine

#### Description

OpenNebula Kubernetes appliance provides an easy way to deploy a Kubernetes cluster. It utilizes the already present functionality of VM contextualization but it is also able to leverage OpenNebula's OneFlow feature to dynamically scale the cluster's nodes. Thanks to the VM and Service templates it is possible to spawn new multi-node Kubernetes cluster instances on demand with the simplicity of a click-of-the-button.

Kubernetes is the most used industry-standard orchestrator of application containers and thus by virtue of this appliance OpenNebula now also has means to deploy such containers next to the other virtualization offering.

The goal of this SR is to improve this OpenNebula and Kubernetes integration, enabling elasticity of the OpenNebula managed Kubernetes cluster.

#### Requirements and Specifications

There is currently no special requirement for this appliance which would make it to deviate from any other VM image. Although, the optional OneFlow and OneGate services of OpenNebula must be configured and running to fully utilize the features of Kubernetes appliance.

#### Architecture and Components

Currently Kubernetes appliance supports only a single master setup with zero to many worker nodes. The master node is also always a worker node and so by deploying only one node this node automatically becomes a single-node Kubernetes cluster still able to run containers. Single node cluster can be always any time later extended by an arbitrary number of the worker nodes and therefore grow the cluster without a need to redeploy it. Shrinking also works but it is not graceful and so containers running on an affected node will be terminated.

Node scaling can be achieved either via instantiating a new VM with the correct set of contextualization parameters or better by utilizing the OneFlow service.

OneFlow integration is a much more powerful construct which will be able to not only dynamically shrink and grow the cluster but also ensure that the required number of nodes is always present. Service must be created with these two roles:

- master
- worker

A master role must be configured as the parent role of the worker role and for the best outcome the VMs should report READY via OneGate to signal they are fully functional and ready.

A master node must be always present while worker nodes are dependent on it (parent relationship) and will not spawn until master is up and ready (report READY). OneFlow service will also provide all the credentials needed for worker nodes to be able to join the already existing cluster. As a result, one has to only click on the button to deploy or to scale the cluster. With the OneFlow integration there is no more a need for the tedious management of nodes by hand and providing them with the correct contextualization.

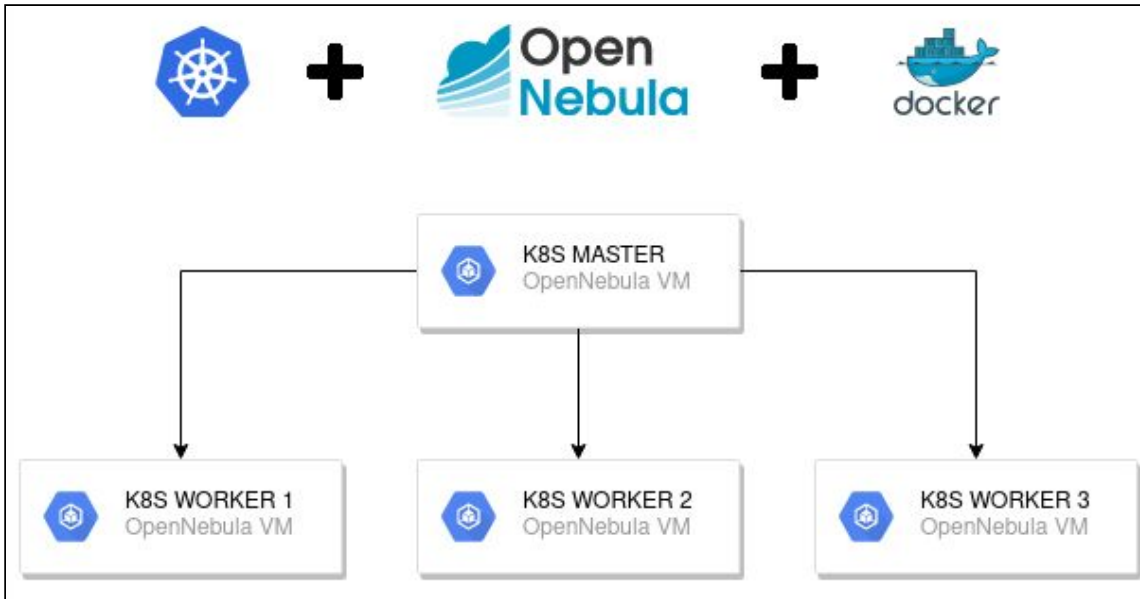


Figure 5.1. OpenNebula Kubernetes Service appliance

**Data Model**

Kubernetes appliance uses a concept called contextualization. Each instance is provisioned with the set of environmental variables which are passed into the VM where one-context scripts will take care of the actual setup and configuration based on the provided values. On top of the usual context parameters shared by all VMs - Kubernetes appliance defines also these:

App Attribute	Value and Meaning
ONEAPP_K8S_ADDRESS	Kubernetes master node address or network (in CIDR format)
ONEAPP_K8S_TOKEN	Kubernetes token - to join worker node to the cluster
ONEAPP_K8S_HASH	Kubernetes hash - to join worker node to the cluster
ONEAPP_K8S_NODENAME	Kubernetes master node name
ONEAPP_K8S_PORT	Kubernetes API port on which nodes communicate (default 6443)
ONEAPP_K8S_PODS_NETWORK	Kubernetes pods network - pods will have IP from this range (default 10.244.0.0/16)
ONEAPP_K8S_ADMIN_USERNAME	UI dashboard admin account - Kubernetes secret's token is prefixed with this name (default admin-user)

Table 5.1. Summary of contextualization from Marketplace documentation



## [SR5.3] Integration with Application Containers Marketplace

### Description

OpenNebula Docker Hub integration provides access to Docker Hub official images.<sup>2</sup> This integration allows to easily import these Docker Hub images into an OpenNebula cloud. The OpenNebula context packages are installed during the import process so once an image is imported it's fully prepared to be used.

The Docker Hub marketplace will also create a new VM template associated with the imported image. This template can be customized by the user, e.g adding a kernel, tune parameter, etc...

### Requirements and Specifications

In order to provide a smooth experience and with as little user intervention as possible the imported images should be fully functional for several hypervisors and provide the same interface as other OpenNebula images. This imposes the following requirements:

- Contextualization, container images should auto-configure using the standard context procedure. This requires to automatically install context packages when importing a Docker Hub image. Context allows container images to automatically configure its networking, execute custom scripts upon boot or set up SSH keys.
- Boot process, container runs with the same kernel as the host. The resulting image should be able to boot in any hypervisor so a functional init system and associated services should be installed.
- The build process should be as compatible as possible with the standard container management procedures preferably using docker tools.

The previous requirements ensure an integrated experience for the user as well as the ability to use Docker Hub images in LXD, Qemu/KVM and Firecracker. The latter two VMM require to provide a separate kernel image to boot the container, and register it in the Kernels & Files Datastore in OpenNebula. The VM Templates must include this kernel file. As part of ONEedge we provide pre-compiled kernels and configuration files as a reference for the users.

### Architecture and Components

The integration of Docker Hub comprises the development of two different components:

- Marketplace drivers. The drivers are responsible for talking to Docker Hub API and provide OpenNebula core daemon a list of available images.
- Datastore downloader, which is responsible for preparing a contextualized image based on the docker container.

Once the container image is downloaded and built it can be deployed in any compatible hypervisor. This process is outlined in Figure 5.2:

---

<sup>2</sup> <https://hub.docker.com>



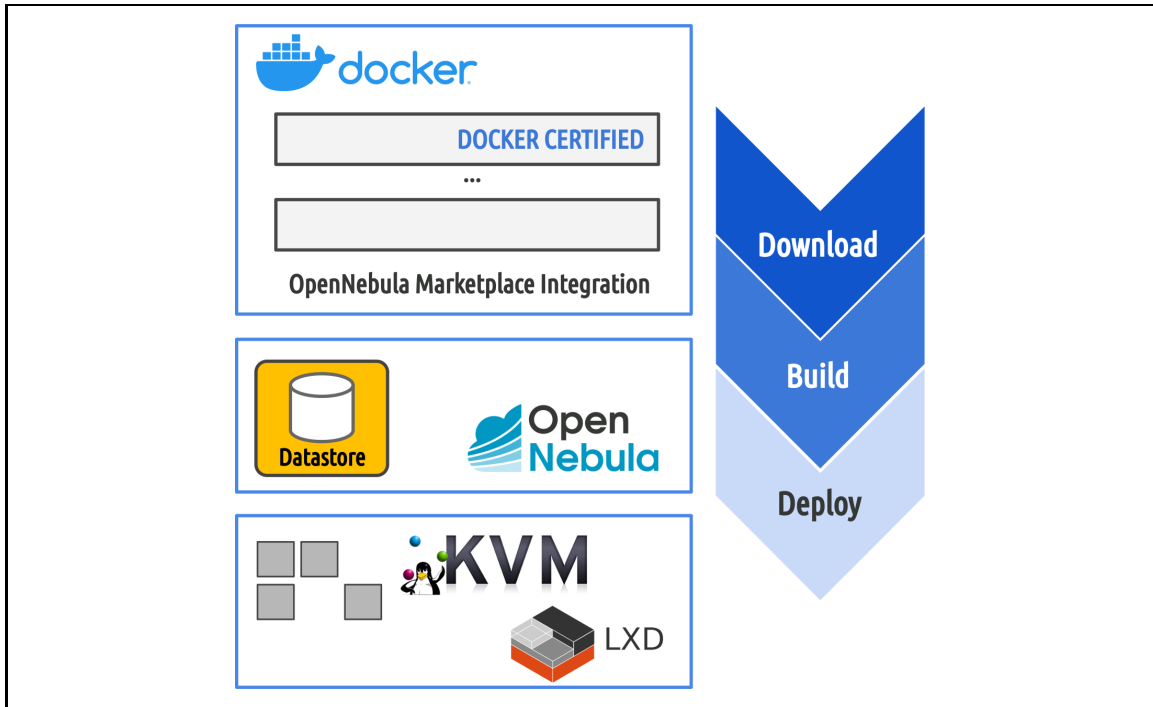


Figure 5.2. Overview of the deployment process of images downloaded from Docker Hub

### Marketplace Drivers

A datastore driver includes three actions:

- import - not implemented. Docker Hub is a public marketplace. An user cannot create Marketplace Apps through OpenNebula.
- delete - not implemented. An user cannot delete apps from public Marketplaces.
- monitor - The monitor action generates a list of available images in Docker Hub. In order to reduce the potential list of images the monitor script only gets official and certified images. The list of images is obtained directly from the public API endpoint and the driver generates a Marketplace App list based on this information. See the Data Model section below for a detailed description of the generated data.

### Datastore Downloader

The downloader is part of the Datastore drivers and is responsible for downloading images for specific protocols. Container images uses a custom protocol that identifies them:

```
docker://<image>?size=<image_size>&filesystem=<fs_type>&format=raw&tag=<tag>&distro=<distro>
```

The different arguments for the custom docker URL are explained below:

- <image> - Docker Hub image name
- <image\_size> - Resulting image size. (It must be greater than actual image size)
- <fs\_type> - Filesystem type (ext4, ext3, ext2 or xfs)



- `<tag>` - Image tag name (default latest)
- `<distro>` - Image distribution (Optional). OpenNebula finds out the image distribution automatically by running the container and checking `/etc/os-release` file. If this information is not available inside the container the `distro` argument has to be used.

The docker downloader performs the following actions:

1. Creates a docker file based on the selected container image. The docker file includes the build instructions for the target image. It includes the installation of specific distribution packages to include a functional init system (including basic services) as well as the OpenNebula contextualization packages.
2. Builds a container based on the docker file using docker build command.
3. Export the resulting image as a tarball
4. Create an image file with the selected size and format and dump the contents of the tarball in it.

The resulting image is ready and fully functional in an OpenNebula cloud. Note that using this URL, a user can download non-official Docker Hub images.

## Data Model

The Docker Hub integration uses the existing Marketplace Application data model. The specific information for a container obtained is detailed in Table 5.2. The information is gathered directly from the official API endpoint, <https://hub.docker.com/v2/repositories/library/>. The rest of the App attributes are adapted or generated to fit with the serverless model used by the integration with Firecracker (see SR2.1).

App Attribute	Value and Meaning
NAME	Name of the container as published in Docker Hub registry. This is the main key for searching for an image.
MD5	MD5 is used to detect new versions or updates of a given image. In this case, as the actual bits of the image are not available, the registration time is used to detect any update.
REGTIME	As provided by the API.
DESCRIPTION	As provided by the API
FORMAT	Images are stored in raw format
VERSION	As there is no versioning in Docker Hub it is fixed to 1.0. Note that tags are directly handled by the SOURCE attribute.
APPTEMPLATE	Sets virtio as the default bus and configures a default command line for the Kernel.

**Table 5.2.** Information obtained from Docker Hub API

As an example Figure 5.3 shows the relevant data for the memcached container:

```

MARKETPLACE APP 184 INFORMATION
ID           : 184
NAME        : memcached
TYPE        : IMAGE
USER        : oneadmin
GROUP       : oneadmin
MARKETPLACE : DockerHub
STATE       : rdy
LOCK        : None

DETAILS
SOURCE      : docker://memcached?size=2048&filesystem=ext4&format=raw
MD5         : 4f33d8eea858298427cfb4db3d760154
PUBLISHER   :
REGISTER TIME : Fri Jun 12 00:00:00 2020
VERSION     : 1.0
DESCRIPTION  : Free & open source, high-performance, distributed memory object caching
system.
SIZE        : 2G
ORIGIN_ID   : -1
FORMAT      : raw

IMPORT TEMPLATE
DRIVER="raw"
DEV_PREFIX="vd"

MARKETPLACE APP TEMPLATE
APPTEMPLATE64="RFJJVkvSPSjyYXciCkRFVL9QUkVGSVg9InZkIgo="
DESCRIPTION="Free & open source, high-performance, distributed memory object caching system."
IMPORT_ID="-1"
LINK="https://hub.docker.com/_/memcached"
PUBLISHER="hub.docker.com"
VERSION="1.0"
VMTEMPLATE64="...5pYz0xIgpD"

```

Figure 5.3. Example of the data for a MarketplaceApp from Docker Hub

## API and Interfaces

No API additions were needed to integrate Docker Hub, as the existing APIs to interface other MarketPlaces were general enough and simple adaptation to the data model where needed. The configuration interface is also the same as other MarketPlaces, i.e. based on specific attributes in its template definition. The specific configuration values available are described in Table 5.3.

App Attribute	Value and Meaning
ENDPOINT	For the Docker Hub API
IMAGE_SIZE_MB	Default size for the images created from Docker Hub
FILESYSTEM	Default FS (xfs, ext4...) for the images created from Docker Hub
FORMAT	Default image format (raw, qcow2...)
FORMAT	Images are stored in raw format

Table 5.3. Configuration attributes for Docker Hub marketplace

Finally the Sunstone interface has been extended in the case of Docker Hub when importing a container to present the user the tags available to that particular container, see Figure 5.4.

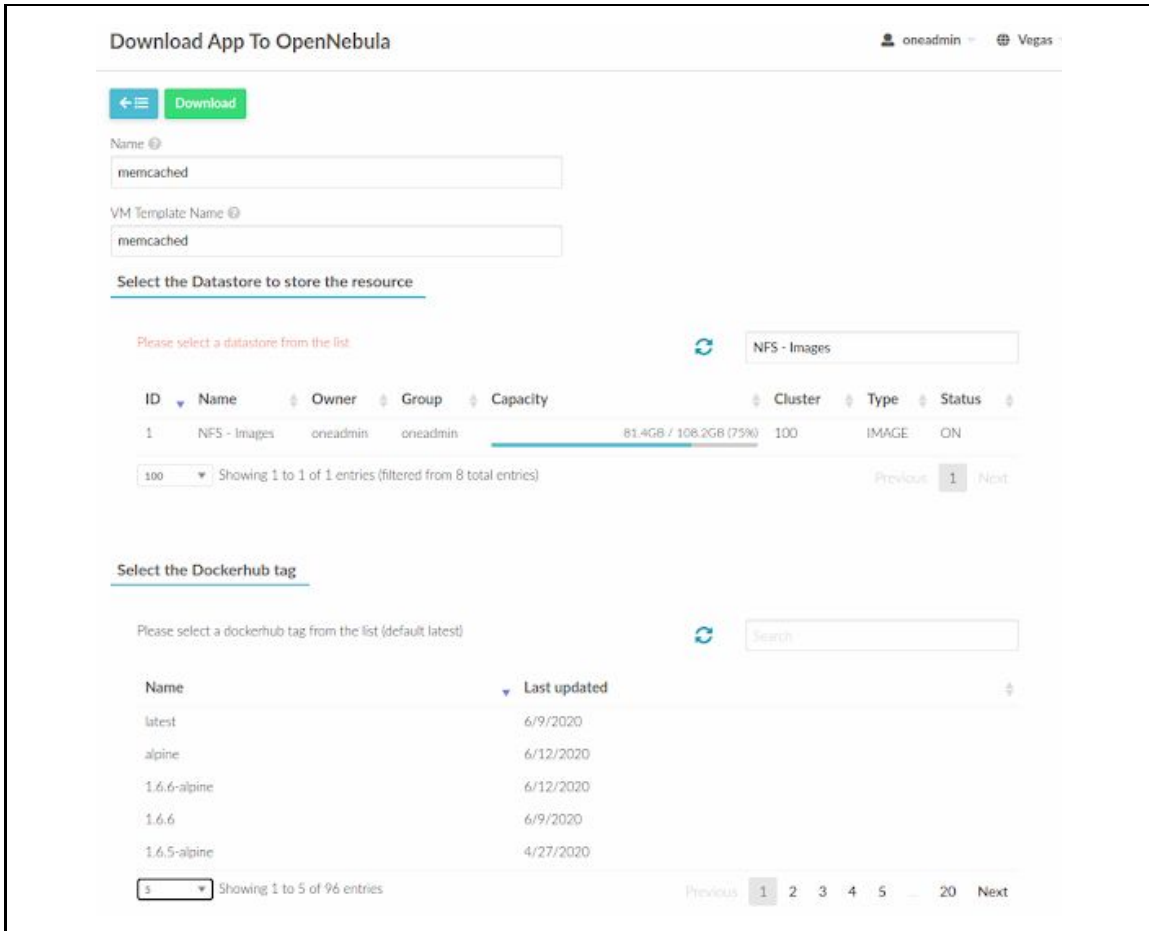


Figure 5.4. Tag selection in Sunstone while importing a container image



## **[SR5.4] New Edge Applications Marketplace Entries**

Kubernetes appliance in the OpenNebula marketplace has been updated (including minor enhancements) to version 1.18.3. The updated appliance can add more nodes to the cluster at any time using the OpenNebula contextualization process.

This allows for the deployment of helm charts thanks to Kubernetes. This, coupled with the Docker Hub integration in SR5.3, renders software requirement SR5.4 as fully met. Please see SR5.2 for the design of the Kubernetes appliance.