ONEedge.io

A Software-defined Edge Computing Solution

D3.2. Software Report - b

Software Report

Version 1.0

10 March 2021

Abstract

This report summarizes the design of the technology components that have been implemented as part of the Second Innovation Cycle (M10-M16), 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.

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

Deliverable Metadata

Project Title:	A Software-defined Edge Computing Solution
Project Acronym:	ONEedge
Call:	H2020-SMEInst-2018-2020-2
Grant Agreement:	880412
WP number and Title:	WP3. Product Innovation
Nature:	R: Report
Dissemination Level:	PU: Public
Version:	1.0
Contractual Date of Delivery:	28/2/2021
Actual Date of Delivery:	10/3/2021
Lead Authors:	Vlastimil Holer, Rubén S. Montero and Constantino Vázquez
Authors:	Sergio Betanzos, Pavel Czerny, Ricardo Díaz, Jim Freeman, Christian González, Alejandro Huertas, Shivang Kapoor and Jorge M. Lobo
Status:	Submitted

Document History

Version	Issue Date	Status[1]	Content and changes
1.0	10/3/2021	Submitted	First final version of the D3.2 report

[1] A deliverable can be in one of these stages: Draft, Peer-Reviewed, Submitted and Approved.

Executive Summary

The purpose of deliverable D3.2 is to offer a summary of the design of the technology components that have been implemented in the Second Innovation Cycle (M10-M16), 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 Second Innovation Cycle (M10-M16), the project mostly focused on those software requirements needed to achieve our second milestone in M16, which is the base functionality needed for a multi-host edge deployment. The work carried out during this Second Innovation Cycle involved software requirements from components CPNT1, CPNT2, CPNT3, CPNT4 and CPNT5, with a special focus on the edge instance management (CPNT1) and the deployment and provision of edge infrastructures (CPNT4). These are some of the main new features that have been implemented as part of this process:

- First version of OpenNebula deployment architecture based on application containers, with automatic upgrade and rollback of ONEedge instances, and their basic monitoring and control.
- New 3-tier replica storage datastore for edge clusters.
- First version of backup system for VM disks.
- New version of OneFlow engine with enhanced functionality, reliability and scalability.
- Improvements in Graphical User Interface to expose new functionality.
- First version of Edge Provider Catalog Service as part of the new edge provisioning interface.
- First version of tests to certify the provider drivers for AWS and Packet/Equinix.
- New Edge Catalog Web interface as part of new Graphical User Interface, Provision FireEdge.
- Provision tools redesigned to use Terraform and the cloud database and multi-tenancy.
- Development of provision templates to implement ONEedge reference infrastructure.
- Development of new drivers for host provision based on Terraform.
- Development of new drivers for IP address management in AWS and Packet/Equinix clouds.
- Development of new drivers for networking to provide private networking based on VXLAN and EVPN BGP extensions.
- New Graphical User Interface, Provision FireEdge, for Edge resource Provision.
- Support for OneFlow templates and importing of multiple images in the Marketplace.
- New Kubernetes appliance images and template in the Marketplace.
- Enhanced integration with Docker Hub allowing ONEedge to orchestrate containers on virtual machines.
- First prototype of a self-service portal on top of an ONEedge.

Table of Contents

1. Edge Instance Manager (CPNT1)

[SR1.1] Simple Product Deployment

Description

The OpenNebula Front-end components are installed the traditional way on a (physical or virtual) host via OS package management tools. Services are running natively on the host while sharing the rest of host installed libraries, services and resources. This is the most common way to deploy services. It's simple and transparent, but also features a few drawbacks.

- various **host libraries versions** (than certifies ones) can lead to service misbehaviour,
- a **security** problem in OpenNebula might have an impact on a host,
- **only one instance** of OpenNebula can run on a host or
- complex **rollback** to previous working version (packages, configurations, database).

To simplify even more the Front-end deployment and maintenance process and **eliminate all drawbacks** above, we have implemented a new installation model which leverages the application containers.

"*A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings*".[2]

As a complement to operating system packages, we have developed a **container image** which ships all Front-end services and their dependencies preinstalled in required versions. It's used to run all services on any host with a supported container runtime in an isolated environment, which is not affected by host services. Security issues with the services inside containers don't (usually) affect the host itself. Due to the isolation of containers from host and from other containers, multiple instances of OpenNebula Front-end services can run on the same host. Last but not least, rollbacks of failed upgrades are as easy as restarting the deployment from an older version of image (likely still available on a host).

There are two supported ways to deploy the Front-end services from a newly developed container image - a **single (all-in-one) container** with all services inside or as a **multiple-containers (composition)**, where each (or sometimes more) service is isolated also from the other Front-end services to provide even more secure deployment.

Requirements and Specifications

<u>Run-time Environments</u>

Target hosts need to comply with the basic requirements to run the containerized OpenNebula Front-end services. Following container run-time environments are going to be supported and a selected one must be preinstalled on a host

- Docker (18.06.0+ on CentOS/RHEL, Ubuntu and Debian)
- Podman (2.0.5+ on CentOS/RHEL only) in both root/rootless modes

[2] https://www.docker.com/resources/what-container

For multi-container deployment mode, an additional tool to manage services composed of multi-containers is needed. Following are supported:

- Docker Compose (1.27.4+)
- Podman Compose (0.1.7+)

While the deployment might run on different host operating systems and with different container run-time versions, the complete stack is being certified with the mentioned configuration and only those are covered by the user/customer support.

Deployment Types

There are two ways to run the containerized OpenNebula Front-end deployment, as a

- single container, where all services are running affined, in same the same container,
- multi-container, where services are anti-affined in their own containers.

The **single (all-in-one) container** mode targets more evaluation environments, where the speed and simplicity to run and maintain the Front-end deployment is important. It doesn't provide extra security features to the inner services, because all are running alongside in the same container and share the environment. This type of deployment is similar to how services run when installed traditionally on a host via packages. Below we can see an example of how to run such a deployment type via Docker. Note: on Podman, instead of calling docker command, the user is expected to run podman command (all arguments are the same).

```
$ docker run -d --privileged --name opennebula \
    -p 80:80 -p 443:443 -p 22:22 -p 29876:29876 -p 2633:2633 \
    -p 5030:5030 -p 2474:2474 -p 4124:4124 \
    -e OPENNEBULA_FRONTEND_HOST=${HOSTNAME} \
    -e OPENNEBULA_FRONTEND_SSH_HOST=${HOSTNAME} \
    -e ONEADMIN_PASSWORD=changeme123 \
    -e DIND_ENABLED=yes \
    -v opennebula_db:/var/lib/mysql \
    -v opennebula_datastores:/var/lib/one/datastores \
    -v opennebula_oneadmin_auth:/var/lib/one/.one \
    -v opennebula_oneadmin_ssh:/var/lib/one/.ssh \
    -v opennebula_srv:/srv/one \
    opennebula/opennebula:6.0
```

The **multi-container (composed)** deployment type splits the services into multiple containers to improve the security by isolation of logic as much as possible and reduce the chance that one service might break another. This deployment targets production deployments and requires an extensive description of deployment prepared beforehand. Such descriptor is going to be shipped alongside with the image and running the deployment is as easy as

- creating a site specific configuration file and
- running a simple command.

Examples of both steps:

```
# configure deployment
$ cat - >custom.env <<EOF
OPENNEBULA_FRONTEND_HOST=one.example.com
```

```
EOF

# run deployment
$ docker-compose up
```

See the Data Model, Multi-Container Deployment Descriptor section below for a descriptor example.

Managed Resources

Before we could implement the new deployment based on the application containers, we had to review all our services - the network ports they communicate over, required data volumes and shared directories they exchange the data over.

The reviewed and identified public network ports extended by the ones specific for the containerized deployment are presented in the next table.

PORT	DESCRIPTION
22	Embedded OpenSSH server
80	Apache HTTP server
443	Apache HTTP server TLS-secured (https)
2474	OpenNebula Flow
2475	OpenNebula Flow TLS-secured
2633	OpenNebula Server (oned)
2634	OpenNebula Server (oned) TLS-secured
4124	OpenNebula Monitor Daemon
5030	OpenNebula Gate
5031	OpenNebula Gate TLS-secured
29876	OpenNebula noVNC

Table 1.1. Exposed network ports from containerized deployment.

In the next table we show the reviewed persistent data volumes and shared directories (for services which exchange data through filesystem) required for the containerized deployment.

VOLUME	DIRECTORY	DESCRIPTION
opennebula_mysql	/var/lib/mysql	Data directory for MySQL DB.
	/var/lib/one/backups	Service db./cfg. backups

opennebula_datastores	/var/lib/one/datastores	OpenNebula datastores
opennebula_shared_vmrc	/var/lib/one/sunstone_vmrc_tokens	Shared directory for Sunstone/FireEdge
opennebula_oneadmin_auth	/var/lib/one/.one	OpenNebula auth. credentials
opennebula_oneadmin_ssh openenbula_oneadmin_ssh_provision	/var/lib/one/.ssh	User oneadmin's .ssh/
opennebula_oneadmin_ssh_copyback	/var/lib/one/.ssh-copyback /var/lib/one/.ssh (in sshd container)	User oneadmin's .ssh/ for sshd cont.
	/var/log	System logs
opennebula_logs	/var/log/one	OpenNebula logs
opennebula_shared_tmp	/var/tmp/sunstone	Shared directory for Sunstone/oned
	/srv/one	Various persistent data
opennebula_secret_tls	/srv/one/secret-tls	TLS certificates.
opennebula_secret_ssh_host_keys	/srv/one/secret-ssh-host-keys	SSH host keys.

Table 1.2. Persistent volumes and shared directories.

Container Image Build

Container is running from a container image, which has to be built. The common practice is to have images as small as possible with least services possible. Instead of building several images (for each service one) we have decided to build only one image with all services in specific versions preinstalled. Such image is

- (unfortunately) much bigger,
- with all services preinstalled (not all might be needed by end-user),
- but simpler to use,
- and allows to run complete Front-end in only a single container.

Container image is created based on the Dockerfile build descriptor, which specifies the base source image, all installation steps, configuration and filesystem changes, volumes, published network ports, or various metadata. Example of a vastly simplified image build descriptor:

```
FROM centos:8

# opennebula packages and dependencies
RUN dnf -y  --setopt=install_weak_deps=False --setopt=tsflags=nodocs install cronie crontabs
logrotate supervisor mariadb mariadb-server expect augeas docker-ce docker-ce-cli
containerd.io memcached httpd mod_ssl passenger mod_passenger stunnel fuse terraform-0.13.*
ansible-2.9.* git opennebula opennebula-tools opennebula-flow opennebula-gate
opennebula-sunstone opennebula-fireedge opennebula-guacd opennebula-provision file e2fsprogs
nc && dnf clean all && rm -rf /var/cache/dnf/*

# cleanup
```

```
RUN rm -rf /etc/yum.repos.d/opennebula.repo /var/log/dnf* /var/log/anaconda
/var/log/hawkey.log && find /tmp /var/tmp /run -mindepth 1 -maxdepth 1 -exec rm -rf '{}' \;
&& truncate -c -s 0 /var/log/wtmp /var/log/btmp /var/log/lastlog

# volumes
VOLUME ["/var/lib/mysql"]
VOLUME ["/var/lib/one/datastores"]

# published services
EXPOSE 22/tcp
EXPOSE 80/tcp
EXPOSE 443/tcp

ENTRYPOINT [ "/frontend-bootstrap.sh" ]
```

The build descriptor is passed to the container tools for build (via docker build, podman build or buildah build-using-dockerfile commands).

Run-time Container Configuration (Bootstrap)

New container started from the container image executes the entrypoint command preinstalled and specified during the container build (see example in previous subsection). The entrypoint usually takes care of configuring and starting the service inside.

In the case of the OpenNebula Front-end image, we call it a bootstrap (start) script (placed as /frontend-boostrap.sh) and is expected to be quite complex as it has to prepare the inner-container environment for more than 20 preinstalled services. Following significant steps takes place on container start:

- run user's custom pre-hook
- apply user's custom OpenNebula configuration
- common configurations (e.g., clean temp., fix perms. on volumes, create state dirs)
- configure service manager
- configure and enable each (selected) service
- run user's custom post-hook
- optionally - enter maintenance mode if requested
- pass control to service manager, which starts and manages the services

Container bootstrap can be customized by a set of parameters, see Table 1.3.

Improved Security

The multi-container deployment isolates services into their own containers, so that they can't negatively affect each-other in case of problems. The proposed container volumes and shared directories described in Table 1.2 and their limited explicit use in the containers ensure that services have access on the filesystem only to the minimum data they require for their work. Those practices vastly improve the deployment security.

For example: Data images for running virtual machines (and even the running virtual machines themselves) are stored in the OpenNebula datastores volume (/var/lib/one/datastores). In traditional deployment where all services are deployed on a single host (or single container), all

OpenNebula services have access to the datastores, despite they don't need to. The multi-container deployment allows to lower the attack vector, as each service is isolated in their own container and each container precisely specifies the minimum of required volumes. In the case of datastores, they are attached only to the OpenNebula server and SSH server containers. That means that e.g. the compromise of the web UI service doesn't give attackers direct access to the datastores.

Container Image Hosting

The container images are stored and served from the container registries, the specialized storage and download servers. To distribute the container image for the new type of deployment, we need to leverage different types of registries to serve different OpenNebula releases and editions:

- **public** - for Community Edition, the common one available by default in the container tools (e.g., Docker Hub). Fetching an image from the registry must be as easy as running a simple command:

```
$ docker pull opennebula/opennebula:6.0
```

- **private** - for Enterprise Edition for the customers, self-hosted and authenticated. The suitable chosen service for self-hosting the container images is the registry (https://hub.docker.com/_/registry/), which implements the Docker Registry 2.0 API. To fetch the private container image, users firstly need to login with credentials:

```
$ docker login enterprise.opennebula.io
Username: JohnDoe
Password:
Login Succeeded!

$ docker pull docker://enterprise.opennebula.io/opennebula/opennebula:6.0
```

Architecture and Components

The layered deployment schema of the single (all-in-one) container deployment is shown in the next figure and lists core services all running inside the single container.

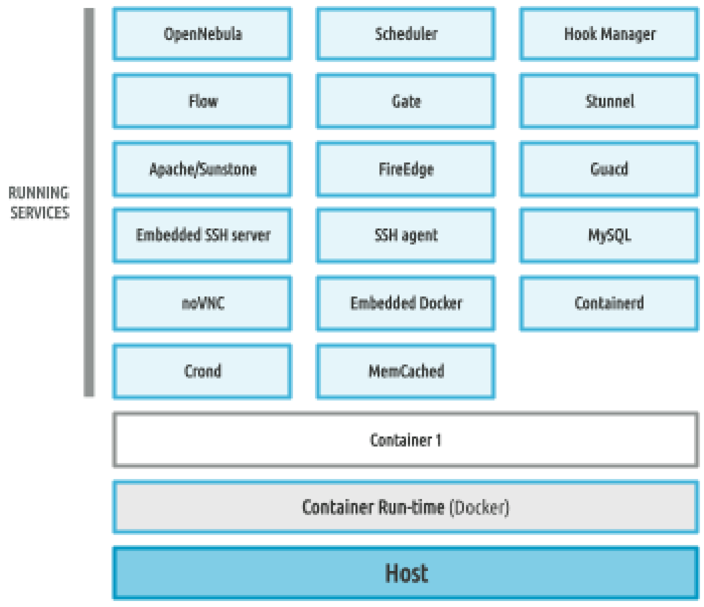


Figure 1.1. Single (all-in-one) container deployment schema.

The multi-container deployment in the next figure distributes the services across 12 isolated containers, which communicate over the network or shared directories. Some services are running redundantly in more containers (e.g., crond service is required for scheduled actions which happen everywhere).

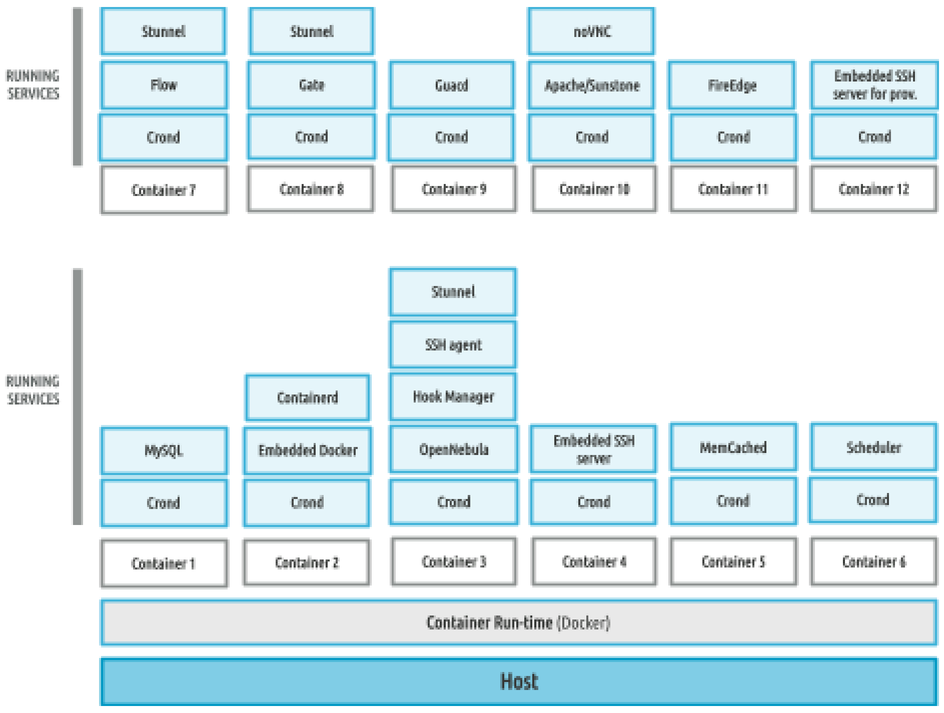


Figure 1.2. Composed multi-container deployment schema.

The sequence diagram in the next figure explains a flow of actions, which happen when a container is being created by a container engine. The created container executes the entrypoint, which is represented by the bootstrap script preinstalled in the OpenNebula Front-end image. Bootstrap script prepares the environment inside the container, configures and enables the services. At the very end, passes control to the service manager, which starts the services.

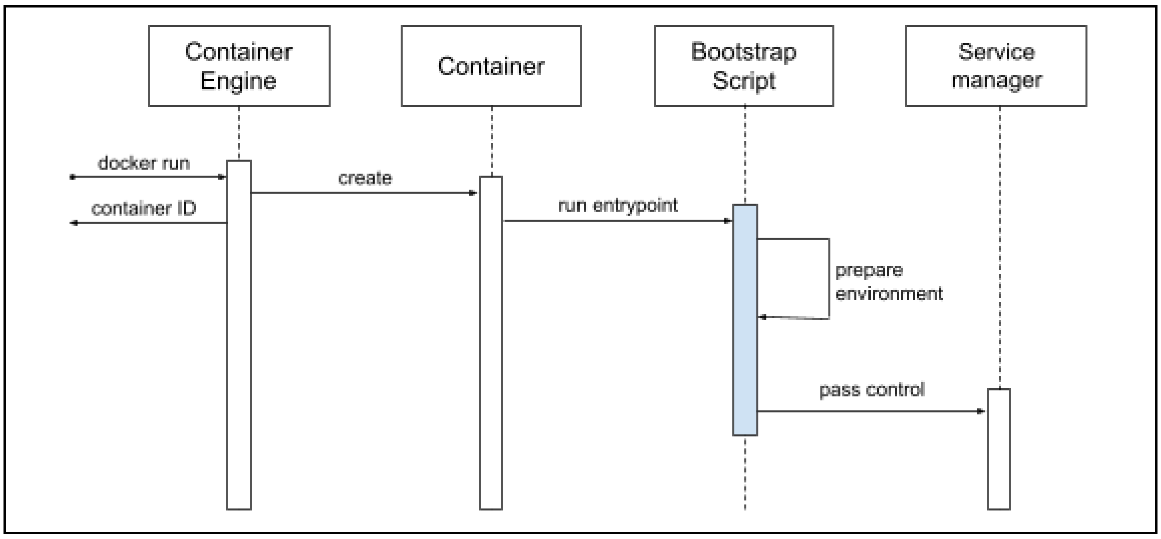


Figure 1.3. Sequence diagram of container bootstrap script

Data Model

Multi-Container Deployment Descriptor

The deployment composed of multiple containers is controlled by a descriptor in docker-compose.yaml format and managed by Docker Compose or Podman Compose tools. The descriptor lists all containers shown on Figure 1.2 as services, their image, volumes, network ports, dependencies or configuration parameters. The example below shows a snippet of a one selected container (service) description.

```
services:
  opennebula-oned:
    image:
"${DEPLOY_OPENNEBULA_REGISTRY}${DEPLOY_OPENNEBULA_IMAGE_NAME:-opennebula}:${DEPLOY_OPENNEBULA
_IMAGE_TAG:-latest}"
    init: true
    env_file:
      - "default.env"
      - "custom.env"
    cap_add:
      - SYS_ADMIN
    environment:
      OPENNEBULA_FRONTEND_SERVICE: "oned"
    depends_on:
      - opennebula-mysql
    ports:
      -
"${DEPLOY_BIND_ADDR:-0.0.0.0}:${DEPLOY_ONED_EXTERNAL_PORT:-2633}:${DEPLOY_ONED_INTERNAL_PORT:
-2633}"
      - "${DEPLOY_BIND_ADDR:-0.0.0.0}:${DEPLOY_MONITORD_EXTERNAL_PORT:-4124}:4124"
    volumes:
      - ./config:/config:ro,z
      - ./certs:/certs:z
      - ./ssh:/ssh:z
      - opennebula_datastores:/var/lib/one/datastores
      - opennebula_secret_tls:/srv/one/secret-tls
      - opennebula_oneadmin_auth:/var/lib/one/.one
      - opennebula_oneadmin_ssh:/var/lib/one/.ssh
      - opennebula_oneadmin_ssh_copyback:/var/lib/one/.ssh-copyback
      - opennebula_logs:/var/log/one
      - opennebula_shared_tmp:/var/tmp/sunstone
    devices:
      - /dev/fuse:/dev/fuse
    tmpfs:
      - /tmp
      - /run
      - /run/lock
    networks:
      - onenet
    stop_grace_period: "90s"
```

Container Configuration

Main logic of container services is configured via environment configuration parameters. The next table presents the basic ones.

PARAMETER	DESCRIPTION
MAINTENANCE_MODE	Starts containers in maintenance mode (YES/NO).
OPENNEBULA_FRONTEND_SERVICE	Front-end service to run inside the container.
OPENNEBULA_FRONTEND_HOST	Hostname/IP of Front-end public endpoint.
OPENNEBULA_FRONTEND_SSH_HOST	Hostname/IP of Front-end's SSH endpoint.
OPENNEBULA_FRONTEND_PREHOOK	Path to pre-hook custom script.
OPENNEBULA_FRONTEND_POSTHOOK	Path to post-hook custom script.
ONED_HOST	Container hostname/IP of OpenNebula service.
ONEFLOW_HOST	Container hostname/IP of OneFlow service.
ONEGATE_HOST	Container hostname/IP of OneGate service.
ONEGATE_PORT	Advertised port where OneGate is published.
MEMCACHED_HOST	Container hostname/IP of MemCached service.
GUACD_HOST	Container hostname/IP of Guacd service.
SUNSTONE_HTTPS_ENABLED	Enable Sunstone HTTPS (YES/NO).
SUNSTONE_PORT	Published Sunstone HTTP port.
SUNSTONE_TLS_PORT	Published Sunstone HTTPS port.
SUNSTONE_VNC_PORT	Published Sunstone VNC port.
FIREEDGE_HOST	Container host/IP of FireEdge service.
ONEPROVISION_HOST	Container host/IP for remote oneprovision runs.
TLS_PROXY_ENABLED	Enable TLS proxy to all OpenNebula APIs (YES/NO).

TLS_DOMAIN_LIST	TLS certificate generating - List of DNS names.
TLS_VALID_DAYS	TLS certificate generating - Validity period.
TLS_KEY_BASE64	Base64-encoded private key for custom TLS cert.
TLS_CERT_BASE64	Base64-encoded custom TLS certificate.
TLS_KEY	Path to private key for custom TLS cert.
TLS_CERT	Path to custom TLS certificate.
ONEADMIN_PASSWORD	Initial oneadmin password.
ONEADMIN_SSH_PRIVKEY_BASE64	Base64-encoded SSH private key for oneadmin.
ONEADMIN_SSH_PUBKEY_BASE64	Base64-encoded SSH public key for oneadmin.
ONEADMIN_SSH_PRIVKEY	Path to custom SSH private key for oneadmin.
ONEADMIN_SSH_PUBKEY	Path to custom SSH public key for oneadmin.
DIND_ENABLED	Enable Docker-in-Docker service (YES/NO).
DIND_TCP_ENABLED	Enable TCP access for Docker-in-Docker (YES/NO).
DIND_HOST	Container host/IP of Docker-in-Docker service.
DIND_SOCKET	Path for Docker socket for Docker-in-Docker service.
MYSQL_HOST	Container host/IP of MySQL service.
MYSQL_PORT	Listening port of MySQL service.
MYSQL_DATABASE	Name of OpenNebula database.
MYSQL_USER	User to connect to OpenNebula database.
MYSQL_PASSWORD	Password to connect to OpenNebula database.
MYSQL_ROOT_PASSWORD	Password for database root user.

Table 1.3. Containers configuration parameters

The custom configuration parameters needs to be set in one of the following configuration files (in a KEY=VALUE format), which come with the multi-container descriptor:

- custom.env
- .env

API and Interfaces

No API or CLI changes are required in OpenNebula.

[SR1.2] Automatic Product Upgrade

Description

The container image with OpenNebula Front-end components developed in SR1.1 vastly simplifies the deployment. While updating to a newer OpenNebula maintenance/patch release does not introduce any additional maintenance steps (except switching the package/image and restarting the services), the upgrade to new OpenNebula major/minor release requires the system administrators to update the database schema, configuration files, or do other version specific actions.

In traditional deployment types via operating system packages, the additional manual upgrade steps were required to be executed by the system administrator before starting the new version of services. In the containerized deployment, we have hidden all this complexity from the system administrator and automated all necessary upgrade steps as part of OpenNebula start scripts inside the container image.

Firstly, the container bootstrap (start) script is always current with the version of OpenNebula Front-end services bundled within the image and reconfigures the services to allow inter-service and end-user communications precisely as it's needed for the running version. Secondly, the database upgrade tool onedb had to be extended to report about the database state and the need of upgrade, to avoid unnecessary delays during the start.

The always-current services (re)configuration during the container bootstrap and smart database upgrades provides a fully automated maintenance-free upgrade process.

Requirements and Specifications

Database Status Check

There was no easy way in the OpenNebula CLI to check if the database schema is current or needs an upgrade. Tool onedb provides various database operations (consistency check, backup/restore, purge, …) and is required to be extended to provide a status of the database in the form of script exit codes. Should cover at least following cases when database

- is current
- needs upgrade
- is in unsupported version (too new)
- is not bootstrapped yet

Upgrades without Backups

The tool onedb manages the database upgrade and rollback in case of error. New command line option --no-backup is required to be added to optionally skip the (currently) mandatory backup. Users should be warned that the option is not suitable for general use.

Database Upgrade Automation

Before OpenNebula core services are started, the service scripts must detect based on newly implemented machine readable database status provided by onedb tool if the database must be firstly upgraded. The suitable place for the upgrade automation is opennebula.sh script which starts oned process. Proposed example automation below:

```
# to avoid script termination on non-zero code from command - we wrap the
# command in if-else construct
if onedb version -v ; then
    _status=0
else
    _status=$?
fi
case "$_status" in
    2)
        msg "Upgrading database..."
        if is_true "${ONED_DB_BACKUP_ENABLED:-yes}" ; then
            _mysqldump="/var/lib/one/backups/db/opennebula-$(date +%Y-%m-%d-%s).sql"
            onedb upgrade --backup "${_mysqldump}"
        else
            onedb upgrade --no-backup
        fi
        ;;
esac
```

Architecture and Components

On the system administrator's request, the container engine responsible for managing the run-time environment creates a container from the selected image. Container runs the OpenNebula services, but before it happens, the initial bootstrap (start, entrypoint) script is executed which prepares the environment inside the container. The natural thing about this image version specific bootstrap script is that it's always current with the versions of OpenNebula services inside the image. At the very end when everything is prepared and configured, the bootstrap script passes the control to the service manager.

Service manager starts all the configured services. The main service (and crucial for this requirement) is the opennebula, which executes the oned daemon responsible for cloud management, remote API and data persistence. As part of the start of this service, necessary automatic database upgrade steps are taking place. The next figure shows the basic actions and interactions among related components.

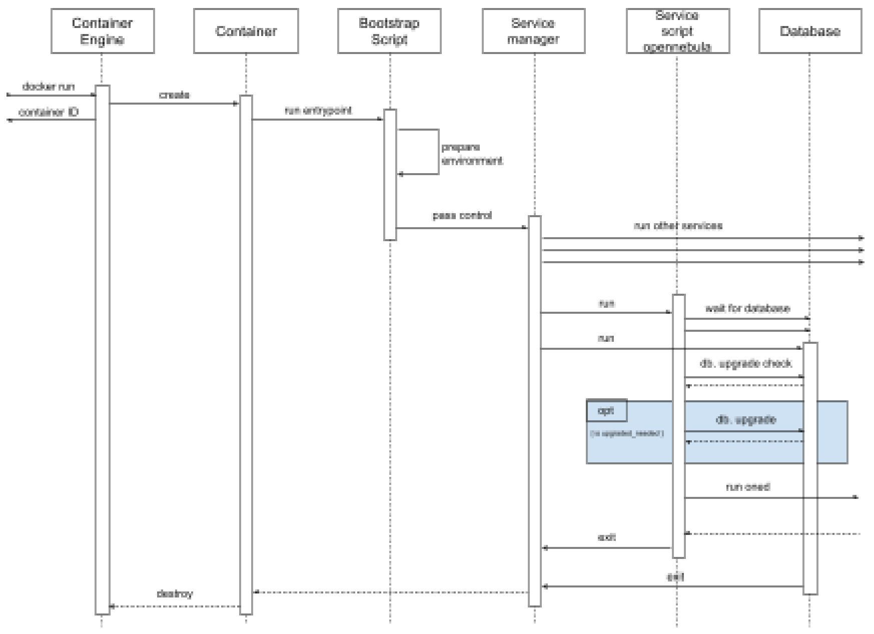


Figure 1.4. Sequence diagram of automatic database upgrade

On automatic upgrade failure, following scenarios might happen:

- Upgrade tool automatically restores the database to the pre-upgrade state. Start of the opennebula service fails, and the system administrator might try to start back again the older version without any traces of upgrade.
- Upgrade tool is running in a mode when backup before upgrade is not done automatically by the upgrade tool. In this case, the upgrade leaves the database in an inconsistent state and the start of the opennebula service fails. System administrator has to restore the database from his own backup before he can start the OpenNebula Front-end containers again (both same or older versions).

Data Model

Container Configuration

Main logic of container services is configured via environment configuration parameters. The table below presents newly introduced parameters.

PARAMETER	DESCRIPTION
ONED_DB_BACKUP_ENABLED	Disables backup before automatic db. upgrades (YES/NO).

Table 1.4. Containers configuration parameters

API and Interfaces

CLI

Database management tool onecfg was extended to provide information about the database status suitable for processing by automation script and triggering consecutive upgrade operations. Also, the database upgrade process now allows you to skip the emergency backup not to delay the upgrade to support cases when backup was already done before by other means. New functionality is summarized in the next table:

CLI PARAMETER	DESCRIPTION
onedb version	Prints the current DB version. The version command will return different exit code depending on the current state of the installation: 0 - The current version of the DB matches with the source version. 1 - The database has not been bootstrapped yet. 2 - The DB version is older than required (upgrade needed). 3 - The DB version is newer and not supported by this release. -1 - Any other problem (e.g connection issues)
onedb upgrade --no-backup [...]	Upgrades the DB to the latest version while skipping the emergency backup done before the upgrade so that in case of failure, rollback operation can restore the database.

Table 1.5. New onedb tool functionality

[SR1.3] Instance Management

Description

On operating systems which are supported for running the OpenNebula Front-end, the services are managed by the most common service manager systemd. The container image introduced in SR1.1 to run OpenNebula Front-end within container run-times required to run multiple services within the same container, but it was not possible to leverage on systemd inside the container as it's not generally supported in all container runt-time environments.

We had to evaluate the other options and selected a Supervisor (http://supervisord.org) as a tool to manage (start, stop, check) our services running in the containers. We had to review all the services we have and create their descriptors and start scripts for the Supervisor.

The services managed by the Supervisor are configured to restart automatically in case of failure. The multi-container deployment was updated to execute the health checks by periodically querying the Supervisor in each container and report an overall status of services back to the container engine.

Basic configuration of OpenNebula services running inside container(s) is done by a container bootstrap (start) script. It's expected that the deployment with a minimal customizations to defaults can't fit all, that's why we had to extend the functionality of onecfg tool to allow end-user customization of any OpenNebula configuration - in a single way and in a simple line format.

Requirements and Specifications

Service Manager

Running multiple services within a container requires use of a service manager. Unfortunately, the systemd is not the generally supported one across the container run-times. It's necessary to select a different one, which is

- able to run properly everywhere,
- packaged in the CentOS/RHEL 8 (i.e., in the base OS used for the container image).

The one which matches the criterias above is Supervisor. To be able to manage OpenNebula Front-end services by Supervisor, we need to:

1. review all current and required services,
2. create their configuration for Supervisor,
3. ensure the Supervisor is started inside the containers.

We have reviewed our and related operating system services and identified all which are necessary to be managed by the Supervisor. List of the services is presented in the next table:

SERVICE	DESCRIPTION
containerd	Embedded container run-time
crond	Daemon to execute scheduled commands
docker	Embedded Docker daemon

infinite-loop	Stub service (not doing anything)
memcached	High performance, distributed memory object cache
mysqld-configure	Service for initial MySQL configuration
mysqld-upgrade	Service for upgrading older MySQL metadata
mysqld	MySQL database
opennebula-fireedge	OpenNebula next-generation FireEdge GUI
opennebula-flow	OpenNebula Flow
opennebula-gate	OpenNebula Gate
opennebula-guacd	Embedded Guacamole daemon
opennebula-hem	OpenNebula Hook Manager
opennebula-httpd	Apache HTTP server
opennebula	OpenNebula Server (oned)
opennebula-novnc	OpenNebula noVNC
opennebula-scheduler	OpenNebula Scheduler
opennebula-showback	Periodic Showback recalculation
opennebula-ssh-add	Service to add SSH keys into OpenSSH auth. agent
opennebula-ssh-agent	Instance of OpenSSH auth. agent for OpenNebula
opennebula-ssh-socks-cleaner	Periodic cleaner of stale SSH connections
opennebula-sunstone	OpenNebula Sunstone GUI
sshd	Embedded OpenSSH server
stunnel	TLS-encrypting socket wrapper

Table 1.6. Services managed inside container by Supervisor

For most of the services shown above we need to create following pair of files to be able to run and manage them by the Supervisor service manager:

1. program descriptor - describes what and how to run, privileges, behaviour on failure
2. service script - executes the pre-service actions and core services at the end

Below we present an example of a program descriptor for OpenNebula Scheduler service for Supervisor service manager.

```
[program:opennebula-scheduler]
command=/usr/share/one/supervisor/scripts/opennebula-scheduler.sh
user=oneadmin
directory=/var/lib/one
priority=310
autorestart=true
```

```
startsecs=5
stopasgroup=true
killasgroup=true
redirect_stderr=true
stopwaitsecs=90
```

A simplified example of a service script written in a POSIX shell for the OpenNebula Scheduler is presented below. As can be seen, the service script must deal with waiting for dependencies or doing other pre-execute actions (like rotating logs) before we can start the final service at the very end.

```
#!/bin/sh

set -e

# give up after two minutes
TIMEOUT=120

#
# functions
#
. /usr/share/one/supervisor/scripts/lib/functions.sh

#
# run service
#
if [ -f /var/lib/one/.one/one_auth ] ; then
    msg "Found one_auth - we can start service"
else
    msg "No one_auth - wait for oned to create it..."
    if ! wait_for_file /var/lib/one/.one/one_auth ; then
        err "Timeout!"
        exit 1
    fi
    msg "File created - continue"
fi

msg "Rotate log to start with an empty one"
/usr/sbin/logrotate -s /var/lib/one/.logrotate.status \
    -f /etc/logrotate.d/opennebula-scheduler || true

msg "Starting service"
exec /usr/bin/mm_sched
```

Introducing support for Supervisor not only allows to run and manage multiple services within a single container easily, but (with some additional changes) would allow to run and manage OpenNebula services on platforms, which don't use the most common systemd service manager.

When the new container instance starts, the bootstrap (start) script bundled in the OpenNebula Front-end container image is executed to get all services inside configured and running. The logic has to be split so that bootstrap script only prepares, configures and enables the services, but doesn't run them. At the every end bootstrap passes control to the supervisor, which starts and manages the services.

```
msg "FRONTEND: Exec supervisord"
exec env -i PATH="${PATH}" /usr/bin/supervisord -n -c /etc/supervisord.conf
```

The proposed way to pass control to the Supervisor is using an exec function, which replaces the one process image with a new one. The example above shows such usage at the very end of bootstrap script.

Automatic Recovery and Health Checks

It's expected that all parts of the deployment are unreliable and can fail. It's required that failed services are restarted automatically and the inner-container service status is reported back to the system administrator via container management tools.

For automatic recovery, two levels of service restart are going to be used

- inner-container recovery by service manager,
- outer-container recovery by container engine.

Inside the container, the Supervisor not only starts and stops the services, but also proactively monitors if the started service is still running. It can also be configured to automatically restart the failed service. All our services inside the containers running via the Supervisor enables the automatic restarting for quick recovery to reduce the service unavailability

Outside the container, the container engine responsible for managing the full life-cycle of the container (from creation to destruction) can be configured to restart the container (and all the services inside) if the container spontaneously stops. This is usually a result of fatal failure inside the container. Automatic container restart is configured in different ways based on type of deployment. For:

- **single container** (OpenNebula Front-end all-in-one deployment) by setting the automatic restart condition as parameter --restart=policy (where policy is one of: “no”, “on-failure”, “always”, or “unless-stopped”). Example of setting restart policy below:

```
$ docker run -dit --name opennebula \
  -p 80:80 -p 443:443 -p 22:22 -p 29876:29876 -p 5030:5030 -p 2474:2474 \
  -e OPENNEBULA_FRONTEND_HOST=one.example.com \
  -e ONEADMIN_PASSWORD=changeme123 \
  --restart=always \
  opennebula/opennebula:6.0
```

- **multi-container** orchestrated by Docker Compose, the restart policy must be preset in the composition descriptor (docker-compose.yml) via restart parameter of the container (service). Below we can see how we preconfigure the policy to restart always in the composition of containers (services). While for a single container deployment system administrator deals with all the parameters and functionality on the low-level of command line and tons of arguments, the composed deployment comes with everything preconfigured to provide the best experience.

```
services:
  opennebula-mysql:
    env_file:
      - ".env"
    image: opennebula/opennebula:6.0
    restart: always
    environment:
      OPENNEBULA_FRONTEND_SERVICE: "mysqld"
    volumes:
      - opennebula_mysql:/var/lib/mysql:Z
    tmpfs:
      - /tmp
      - /run
      - /run/lock
    networks:
      - onenet
    healthcheck:
      test: ["CMD", "/frontend-healthcheck.sh"]
      interval: "10s"
      timeout: "10s"
      retries: 3
      start_period: "2m"
```

Status of the services running inside the container can be propagated to the container engine via a mechanism of **health checks**. The health checks are custom commands executed periodically inside the container, their success or failure exit status persisted in the container metadata, and easily fetched by the systems administrator by common container engine tools. It's necessary to

- develop health check script running inside container,
- configure container engine to run health checks.

Health check script is being preinstalled in the container image. It queries all services managed by Supervisor inside the container and returns back to the container engine a healthy state if all services are running or unhealthy state if any/all services are not running. Example of a similar health check script is below.

```
_status=$(LANG=C supervisorctl status 2>/dev/null | awk '
    {
        if ($2 != "RUNNING") {
            print "1";
            exit 1;
        }
    }
    END {
        print "0";
    }')

if [ "$_status" -eq 0 ] ; then
    exit 0
fi

exit 1
```

Periodic running of the health checks can be configured directly in some container image type (e.g., Docker Image v2), but not in all (e.g., OCI Image). For this reason, we didn't set the health and depend on configuration on container run. For:

- **single container**, the health checks must be set on the container engine command line via the parameters --health-cmd=cmd, --health-timeout=time, --health-retries=count, --health-interval=time, --health-start-period=time. Example of configuring the health checks below:

```
$ docker run -dit --name opennebula \
  -p 80:80 -p 443:443 -p 22:22 -p 29876:29876 -p 5030:5030 -p 2474:2474 \
  -e OPENNEBULA_FRONTEND_HOST=one.example.com \
  -e ONEADMIN_PASSWORD=changeme123 \
  --health-cmd=/frontend-healthcheck.sh \
  --health-interval=30s \
  --health-timeout=10s \
  --health-retries=3 \
  --health-start-period=2m \
  --restart=always \
  opennebula:latest
```

- **multi-container** orchestrated by Docker Compose can have the health checks preconfigured in the container (service) metadata in the healthcheck section. An example of such configuration has been presented previously in this subsection.

In initially supported containerized deployments of the OpenNebula Front-end, the health checks have only reporting nature back to the container engine and systems administrator. There are no automatic recovery actions when a particular container (service) becomes unhealthy.

OpenNebula Defaults

Default configuration of the OpenNebula must change to fit more production usage, so that it's not needed to customize the brand new deployments. Following defaults have been adjusted to match the todays' expectations:

- scheduling interval from 30 to 15 seconds
- for KVM:
    - GUEST_AGENT enabled
    - VIRTIO_SCSI_QUEUES set to 1
    - DISK/DISCARD to unmap blocks on trim command
    - shutdown timeout from 300 to 180
    - enabled VM destroy after shutdown timeout
    - enabled time synchronization on VM resume

Custom Services Configuration

The default OpenNebula configuration, even with small production-ready customizations, can't fit all users and requirements. OpenNebula is highly customizable, but comes with more than 80 configuration files to fine-tune the individual parts of CLI, server, drivers or web GUI behaviour. On top of that, the configuration files are not unified and use 3 file formats (oned.conf-like, YAML, shell assignments). The introduction of containers and seamless version upgrades, both makes the inside logic more a black-box - a thing which a regular user doesn't need to care how works inside, emphasis on simplifying the configuration management and abstracting the file formats.

OpenNebula already had a tool onecfg, which was able to upgrade configuration files to newer OpenNebula based on automatic detection of changes and applying them to the older configurations. The tool is able to manage all the configuration files in all required formats. It was required to extend the onecfg tool to be able to apply into the configuration files custom user changes, not only the automatically detected ones. To fulfill this requirement, it was improved with

- supporting new single line change format to describe the configuration changes,
- new subcommand patch to apply ad-hoc custom changes in the configuration files,
- improved subcommand diff to output in the single line change format,
- integration within container bootstrap to reconfigure services based on user req.

The container bootstrap (start) script was adapted to read (optionally) passed user's change descriptor file into the container and reconfigure OpenNebula services before starting them.

This improvement enables users to

- use easy to write and understand format to describe changes in configuration files,
- abstract from individual conf. files and formats and focus on change in data,
- have single tool to achieve change in any OpenNebula configuration,
- use the same mechanism for both containerized and even native deployments.

Architecture and Components

The container integrates new functionality

- **onecfg** - triggered during the container bootstrap to apply user requested changes in the configuration files based on the provided patch file in simple line format. This should happen at the very start of the bootstrap, so that user changes don't clash with necessary bootstrap reconfigurations to make the services working correctly.
- Service manager **Supervisor**, which starts and stops the inner-container services and monitors their state continuously. The services are configured to be restarted automatically by the Supervisor, if they fail.
- **Health checks** periodically executed by the container engine to query the status of services inside the container and report back.

Interaction among bootstrap (start) script, onecfg, Supervisor and related components inside the container is displayed in the figure below, where relevant parts are highlighted.

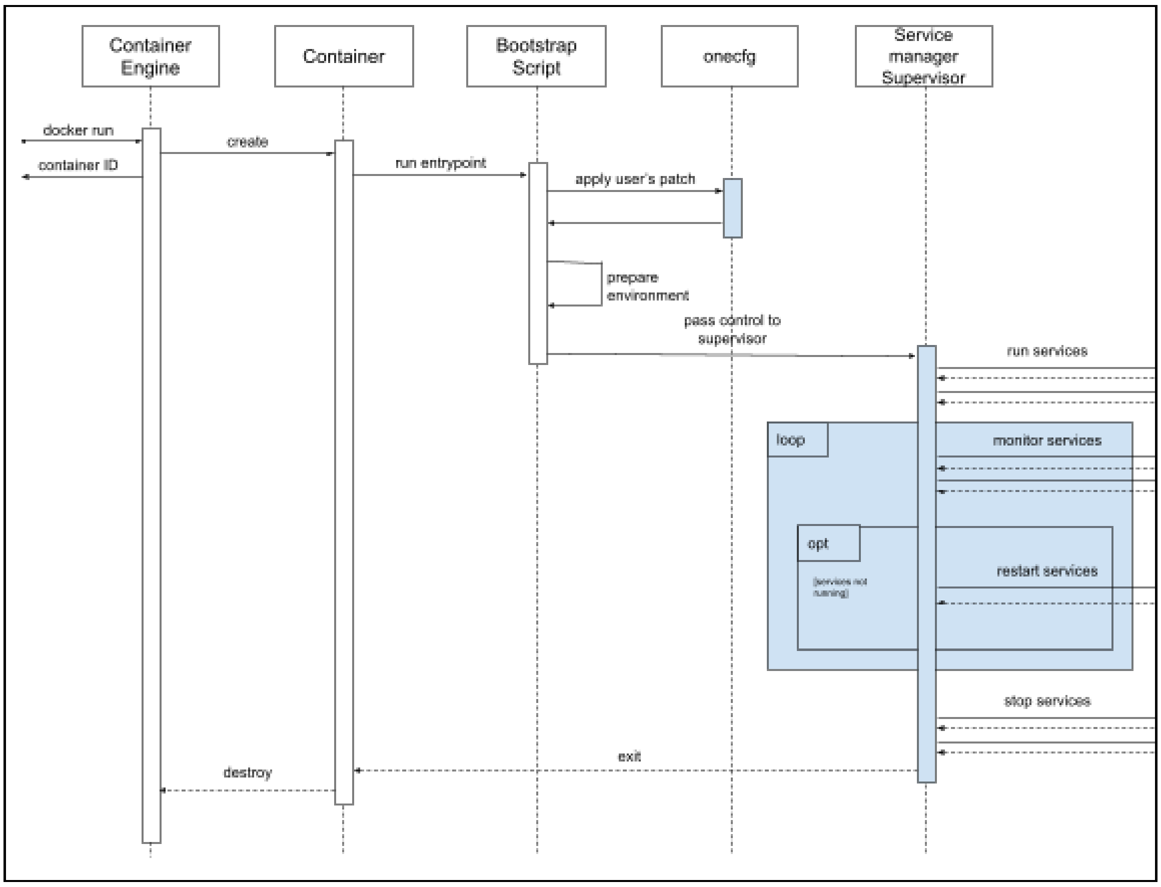


Figure 1.5. Sequence diagram of container bootstrap with onecfg and service monitoring

The simplified logic of health checks is explained by a sequence diagram below:

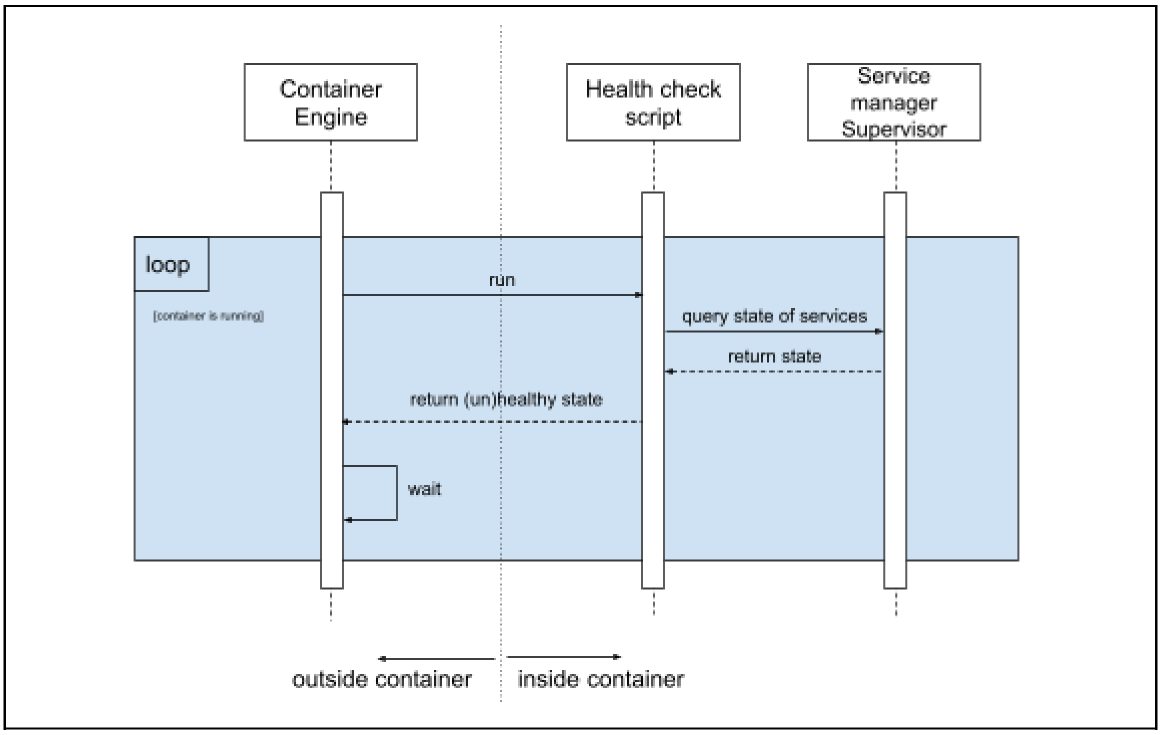


Figure 1.6. Sequence diagram of health checks.

Data Model

Change Line Format

Differences between configuration files or a change to apply in the required configuration file can be described by a newly introduced change line format, where one line represents one change operation and contains all needed information. The format is quite simple and powerful, but cannot achieve all conceivable changes.

```
<FILENAME> <COMMAND> <PATH> [<JSON VALUE>]
```

The change line format consists of one or more lines, where each line has several space separated parts and is shown above with description and examples in the next table.

PART	DESCRIPTION
FILENAME	Filename is an absolute path to the well-known configuration file of OpenNebula, which is going to be changed.
COMMAND	Type of operation to be executed on a configuration file, possible values ● ins - insert new configuration parameter ● set - change existing parameter ● rm - remove parameter

PATH	Path uniquely identifies the location of the changed parameter as a slash / separated XPath-like segment and key from document root and without leading slash (e.g., path/path/path/key). If any path element contains spaces, the whole path must be quoted with (and only with!) double quotes ("), internal double quotes must be escaped by backslash (\\).
	
	For YAML configuration files, the path elements and values starting with colon (:) are transformed into Ruby symbols.
	
	Example:
	● for following /etc/one/oned.conf snippet:
	`PORT = 2633                                # path 1`
	`DB = [ BACKEND = "sqlite",                 # path 2`
	`       TIMEOUT = 2500 ]`
	`IM_MAD = [`
	`      NAME       = "monitord",`
	`      EXECUTABLE = "onemonitord",`
	`      ARGUMENTS  = "-c monitord.conf",`
	`      THREADS    = 8 ]                      # path 3`
	
	these paths are valid to address the emphasized parameters:
	1. PORT
	2. DB/BACKEND or "DB/BACKEND"
	3. IM_MAD/"monitord"/THREADS or "IM_MAD/\\"monitord\\"/THREADS"

JSON VALUE	The value encoded in JSON, mandatory only for ins and set commands. Can contain simple values or complex structures, string values must be quoted with (and only with!) double quotes (").

	Examples: ● no text is an uninitialized value ● null - uninitialized value ● 11 - Integer value 11 ● "11" - String value 11 ● '11' - invalid valid JSON value! ● "'11'" - String value '11' ● "\"11\"" - String value "11" ● true - Boolean value true ● "true" - String value true ● ["apple", "orange"] - Array with 2 String values apple and orange ● ['apple', 'orange'] - invalid JSON value ● {"fruit": "apple"} - Hash with key fruit with value apple ● {'fruit': 'apple'} - invalid JSON value

Table 1.7. Line change format parts

Examples of several changes in a line format are below.

```
/etc/one/cli/oneimage.yaml ins :ID/:adjust false
/etc/one/cli/oneimage.yaml set :USER/:size 15
/etc/one/cli/oneimage.yaml set :GROUP/:size 15
/etc/one/cli/oneimage.yaml ins :NAME/:expand false
/etc/one/oned.conf set DEFAULT_DEVICE_PREFIX "\"sd\""
/etc/one/oned.conf set VM_MAD/"vcenter"/ARGUMENTS "\"-p -t 15 -r 0 -s sh vcenter\""
/etc/one/oned.conf rm  VM_MAD/"vcenter"/DEFAULT
/etc/one/oned.conf ins HM_MAD/ARGUMENTS "\"-p 2101 -l 2102 -b 127.0.0.1\""
/etc/one/oned.conf ins VM_RESTRICTED_ATTR "\"NIC/FILTER\""
```

The output of few changes can be understood as, for example:

- `/etc/one/cli/oneimage.yaml ins :ID/:adjust false`
  - `in /etc/one/cli/oneimage.yaml file`
  - `insert new`
  - `key :adjust into top Hash structure :ID`
  - `with Boolean value false`
- `/etc/one/cli/oneimage.yaml set :USER/:size 15`
  - `in /etc/one/cli/oneimage.yaml file`
  - `change value for existing`
  - `key :size inside top Hash structure :NAME`
  - `to value 15`
- `/etc/one/oned.conf rm VM_MAD/"vcenter"/DEFAULT`
  - `in /etc/one/oned.conf`
  - `remove`
  - `key DEFAULT from VM_MAD section for vcenter`

Container Configuration

Main logic of container services is configured via environment configuration parameters. The next table presents newly introduced parameters.

PARAMETER	DESCRIPTION
OPENNEBULA_FRONTEND_ONECFG_PATCH	Path to custom onecfg patch file in container.

Table 1.8. Containers configuration parameters

API and Interfaces

CLI

Existing tool onecfg was extended with a new subcommand and selectable output/input change descriptor format. New features are summarized in the next table:

CLI PARAMETER	DESCRIPTION

onecfg patch	Patch applies diffs, change descriptors, generated by diff subcommand or created manually (as line or yaml formats) and provided on standard input or as filename passed as an argument. Changes are applied in replace mode and any user customizations on addressed places are overwritten.
	
	`# onecfg patch --verbose --format line <<EOF`
	`/etc/one/oned.conf set PORT 2633`
	`/etc/one/oned.conf set LISTEN_ADDRESS "\"127.0.0.1\""`
	`/etc/one/oned.conf set DB/BACKEND "\"mysql\""`
	`/etc/one/oned.conf ins DB/SERVER "\"localhost\""`
	`/etc/one/oned.conf ins DB/USER "\"oneadmin\""`
	`/etc/one/oned.conf ins DB/PASSWD "\"secret_password\""`
	`/etc/one/oned.conf ins DB/NAME "\"opennebula\""`
	`EOF`
	`INFO  : Applying patch to 1 files`
	`ANY   : Backup stored in`
	`'/var/lib/one/backups/config/2020-12-22_01:20:40_2878523'`
	`INFO  : Patched '/etc/one/oned.conf' with 6/7 changes`
	`INFO  : Applied 7/7 changes`
	
	Exit codes to better evaluate application state:
	● 0 - All diff changes were applied
	● 1 - Only some changes were applied
	● 255 - Error during application, nothing to apply or other error

onecfg diff --format=...	The subcommand diff was extended to provide diffs, change descriptors, in selectable format. Format of patch data on output: ● text (default) ● line ● yaml

Table 1.9. New onecfg tool functionality

2. Edge Workload Orchestration and Management (CPNT2)

[SR2.2] Specialized Cache Datastore

Description

The scalability and performance of the SSH transfer mode can be greatly improved by using the replication mode. In this mode the images are cached in each cluster and so available close to the hypervisors. This effectively reduces the bandwidth pressure of the image datastore servers and reduces deployment times. This is especially important for edge-like deployments where copying images from the frontend to the hypervisor for each VM could be slow.

This replication mode implements a three-level storage hierarchy: cloud (image datastore), cluster (replica cache) and hypervisor (system datastore). Note that replication occurs at cluster level and a system datastore needs to be configured for each cluster. The schema of this mode is depicted below.

Requirements and Specifications

- Datastore data model needs to include cached locations/servers
- Drivers must use file-based, storage efficient disk formats to optimize transfer operations.
- Storage system must present some fault tolerance by recovering VMs from a snapshot
- Users can define custom snapshot frequencies for each VM disk.
- Recovery system should not impact on VM I/O performance

Architecture and Components

The storage system leverages OpenNebula modular approach. The main component is a specialized SSH driver with support for replication. A node in the cluster is designated as Replica Server (RS), and it mirrors the Image datastores in the front-end. The RS also stores disk snapshots. When a node fails, a VM is restarted from the last snapshot available in the RS.

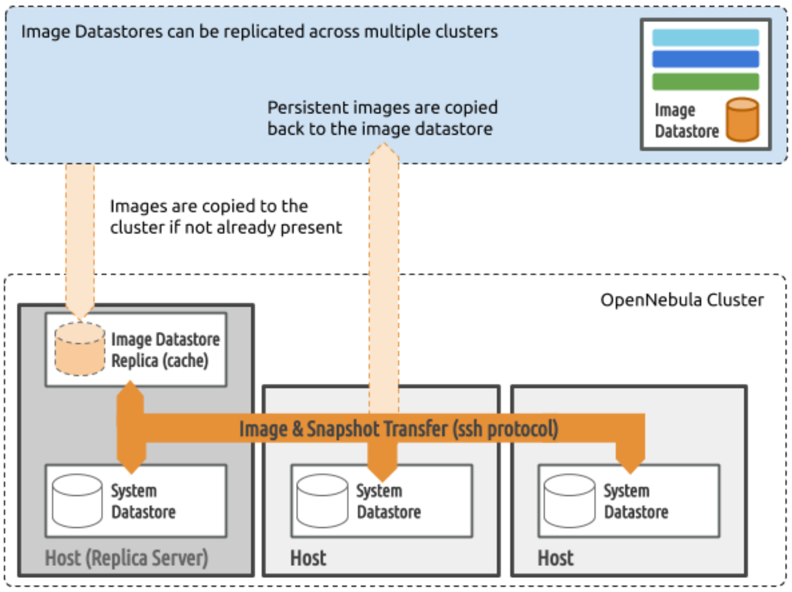


Image Datastores can be replicated across multiple clusters
Persistent images are copied back to the image datastore
Image Datastore
Images are copied to the cluster if not already present
OpenNebula Cluster
Image Datastore Replica (cache)
Image & Snapshot Transfer (ssh protocol)
System Datastore
System Datastore
System Datastore
Host (Replica Server)
Host
Host

Figure 2.1. Architecture for the Cache Datastore

Finally, the host agent includes a module that scans for VM disks to be snapshotted, takes snapshots of the relevant disks and transfers them to the RS.

Data Model

The datastore structure is replicated at the RS. When the VM disk is configured to be snapshotted an additional structure is created for the this to support this snapshots, see below:

```
$VM_DIR
|-- disk.0                //symlink -> disk.0.snap/base.1
|   `-- disk.0.snap
|       |-- disk.0.snap   //symlink -> . for relative referencing
|       |-- base          //base image (cp/mv from datastore)
|       `-- base.1        //qcow2 overlay (backing file = base)
|-- ...
```

API and Interfaces

The replica and recovery configuration interface follows the standard OpenNebula datastore configuration. A System Datastore must be added to each cluster in the cloud. To enable replication, the hostname of the designated RS (REPLICA_HOST) in the cluster must be added to the template of the System Datastore.

When using replication the following attributes can be tuned in sshrc file in /var/lib/one/remotes/etc/tm/ssh:

ATTRIBUTE	DESCRIPTION
REPLICA_COPY_LOCK_TIMEOUT	Timeout to expire lock operations should be adjusted to the maximum image transfer time between Image Datastores and clusters.
REPLICA_RECOVERY_SNAPS_DIR	Default directory to store the recovery snapshots. These snapshots are used to recover VMs in case of host failure in a cluster
REPLICA_SSH_OPTS	ssh options when copying from the replica to the hypervisor speed. Prefer weaker ciphers on secure networks
REPLICA_SSH_FE_OPTS	ssh options when copying from the frontend to the replica. Prefer stronger ciphers on public networks

Table 2.1. Configuration interface for disk replicas

Additionally, in replica mode you can enable recovery snapshots for particular VM disks by adding an option RECOVERY_SNAPSHOT_FREQ to DISK in the VM template.

[SR2.4] Virtual Machine Management Operations: Backups

Description

The aim of this requirement is to provide an integrated backup solution within OpenNebula's feature set. Traditionally OpenNebula users need to resort to custom solutions (like community addons provided by third parties) or out of band solutions (like Veeam in VMware environments).

As it follows, this complicates the ability to provide an end to end support to the complete stack.

Requirements and Specifications

Be able to schedule backup actions for VM disks automatically from OpenNebula Sunstone with the possibility to define and override defaults. At the very least, a frequency or backup periodicity can be defined per VM, as well as the backup destination storage backend.

As much hypervisors and storage backends to be supported as possible, but as a bare minimum those specified in the next table.

HYPERVISOR	STORAGE
KVM/FC/LXC	Shared filesystem (NFS, GlusterFS)
KVM/FC/LXC	Ceph
KVM/FC/LXC and SSH	SSH

Table 2.2. Supported matrix

Architecture and Components

Backups are made to a specific private marketplace. This allows to distribute backups to different storage backends, such a HTTP server which can be potentially hosted in another datacenter to comply with backups best practices, or even an S3 services backed by for instance a Ceph cluster or even the original AWS S3 service.

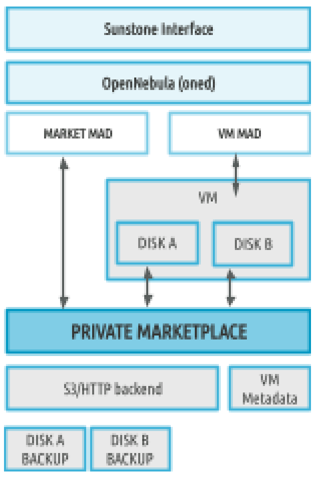


Figure 2.2. Architecture of OpenNebula Backup Solution

Data Model

In order to identify which VMs will be backed up, the following VM attributes can be defined in their templates (which can be seen as OpenNebula's key value store of VM metadata):

```
BACKUP_LAST_COPY_TIME={0|epoc}
BACKUP_FREQUENCY_SECONDS=[int]
BACKUP_MARKETPLACE_ID=[int]
BACKUP_MARKETPLACE_APP_ID=[int]
```

This functionality leverages the developments made as part of SR5.1, in particular the ability to export and import a full VM (ie, metadata plus images) from a public or private marketplace.

The backup mechanism is implemented as a cron task that periodically loops the OpenNEbula VM pool and applies the following algorithm:

```
if (NOW - BACKUP_LAST_COPY_TIME) > BACKUP_FREQUENCY_SECONDS
   ID=make_backup
   MPID=onemarket import ID
   PREVIOUS_BACKUP_MARKETPLACE_APP_ID=BACKUP_MARKETPLACE_APP_ID
   update BACKUP_MARKETPLACE_APP_ID with MPID
   delete PREVIOUS_BACKUP_MARKETPLACE_APP_ID if exists
end

make_backup
  * onevm poweroff
  * ID = save_as_template
  * onevm resume
```

API and Interfaces

This functionality is available either through the CLI and/or through Sunstone.

The CLI leverages the "onevm update" command, where the attributes defined in the "Data Model" section can be defined and later on interpreted by the backup subsystem.

Sunstone presents new options in the VM infotab to define the content of said attributes.

[SR2.8] Complete Service Flows

Description

OpenNebula Flows provides orchestration capabilities to multi-component applications (flows). The goal of this software requirement is to extend the current capabilities of the flow management engine to support additional flow operations as well as to better scale.

Requirements and Specifications

- Manage (create and destroy) virtual networks associated with a flow.
- Include update operations to flow roles and auto-scaling rules.
- Replace the current polling mechanism with an event driven approach to update flow states.

Architecture and Components

OpenNebula Flow is implemented as a separated control module that creates and controls service VMs through OpenNebula API . OpenNebula core also provides persistence and access control policies of Flow definitions.

Flow uses an event driven notification mechanism to update VM states and the associated flow states. VM changes also triggers elasticity rules evaluation and VM cardinality updates (add or removal of flow VMs). The new architecture includes:

- **OneFlow Server**: Manage user requests and pass them to LCM if a state change is needed.
- **Life Cycle Manager (LCM)**: Manage the state transitions of the services, delegates in EM if any wait is needed.
- **Event Manager (EM)**: Waits for specific events (VM state changes) and notify the LCM.

An overview of the components is depicted in the next figure.

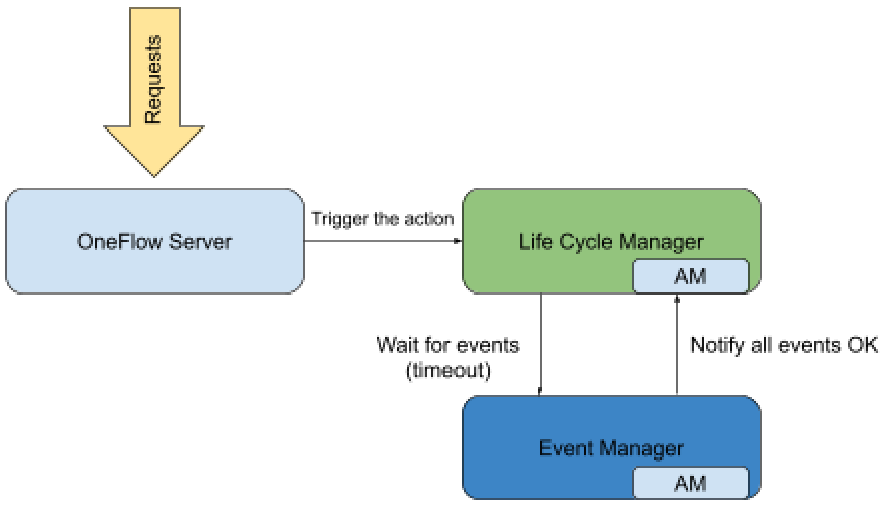


Figure 2.3. OpenNebula Flow new components

Data Model

In order to specify either a virtual network, a virtual network template or a virtual network reservation to be used when the service template is instantiated, the structure of the service template has to be changed. The flow document includes a network section to specify the networking information, see the example below.

```
{"name"=>"TEST",
 "deployment"=>"straight",
 "description"=>"",
 "roles"=>
  [{"name"=>"Master",
    "cardinality"=>1,
    "vm_template"=>0,
    "vm_template_contents"=>"NIC = [\n  NETWORK_ID = \"$Public\" ]\nNIC = [\n  NETWORK_ID =
\"$Private\" ]\n",
    "elasticity_policies"=>[],
    "scheduled_policies"=>[]},
   {"name"=>"Slave",
    "cardinality"=>1,
    "vm_template"=>0,
    "vm_template_contents"=>"NIC = [\n  NETWORK_ID = \"$Private\" ]\n",
    "elasticity_policies"=>[],
    "scheduled_policies"=>[]}],
 "custom_attrs"=>{"A"=>"M/O|description|<default_value>"},
 "networks"=>{"Public"=>"M|vnet_id|", "Private"=>"M|vnet_id|", "Reserve": "M|vnet_id|"},
 "ready_status_gate"=>false}
```

API and Interfaces

The improvement in the flow engine does not require any API or interface changes. The network extension does not require the addition of any new API call either.

[SR2.9] Web UI extensions

Description

The functionality developed in SR2.1 - SR2.7 required adaptations to current interfaces to include new options or improve its layout based on the new data model. This is separate from new interfaces such as a FireEdge/OneProvision GUI defined in other SRs in this document.

The web extension task is a horizontal development task, and it affects the existing OpenNebula GUI component, Sunstone.

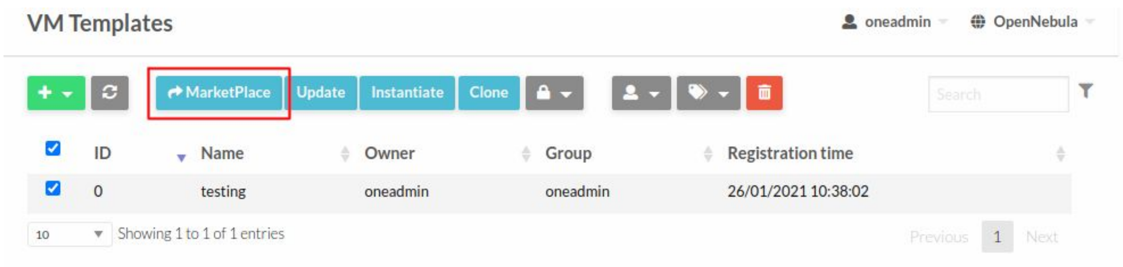
Requirements and Specifications

This second cycle focused on meeting the following requirements:

- MarketPlace tab to support multi-VM applications.
- VirtualMachine tab to display disk backups and policies.

API and Interfaces

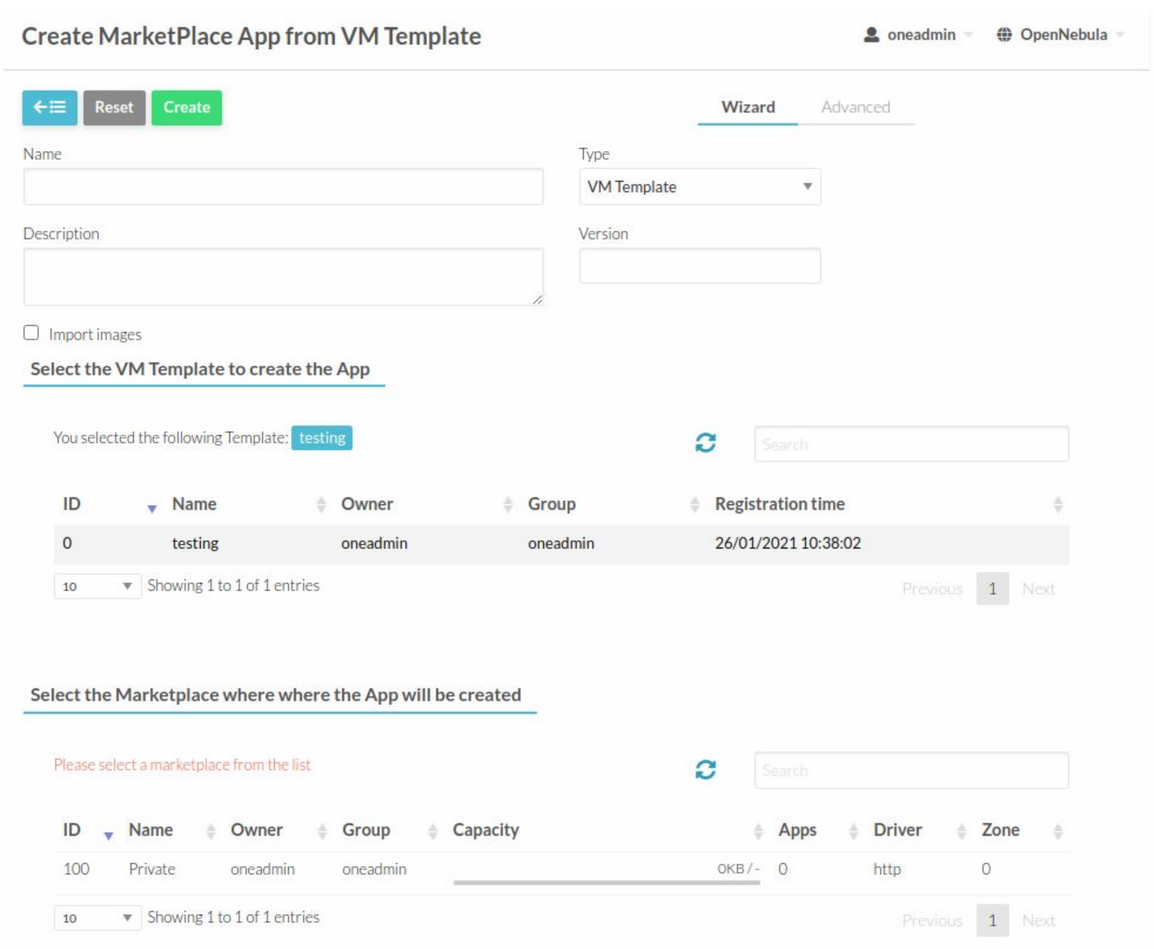
For multi-VM application support, there is a new button to import the resources into the marketplace:



	ID	Name	Owner	Group	Registration time
✓	0	testing	oneadmin	oneadmin	26/01/2021 10:38:02

10 Showing 1 to 1 of 1 entries Previous 1 Next

Figure 2.4. VM template import operation



Create MarketPlace App from VM Template

Name	
Type	VM Template
Description	
Version	

☐ Import images

Select the VM Template to create the App

You selected the following Template: testing

ID	Name	Owner	Group	Registration time
0	testing	oneadmin	oneadmin	26/01/2021 10:38:02

10 Showing 1 to 1 of 1 entries

Previous 1 Next

Select the Marketplace where where the App will be created

Please select a marketplace from the list

ID	Name	Owner	Group	Capacity	Apps	Driver	Zone
100	Private	oneadmin	oneadmin	0KB / -	0	http	0

10 Showing 1 to 1 of 1 entries

Previous 1 Next

Figure 2.5. Import operation in Sunstone GUI

In the screen displayed above, the user can select the different options about the import operation, like if he wants to import images or not and can select the marketplace to import the template.

VirtualMachine information subtab has been extended to present and manage backup frequency and target marketplace attributes as described in SR2.4.

[SR2.10] LXC virtualization drivers for OpenNebula

Description

OpenNebula features a driver-based modular architecture. This design enables it to interact with multiple virtualization platforms. This requirement aims to develop LXC specific drivers to create and manage system containers

Requirements and Specifications

- Manage (create and destroy) systems containers based solely on the LXC engine
- Develop de storage mappers
- Support Linux Bridge networking drivers
- Develop LXC monitor probes

Architecture and Components

LXC drivers interact directly with the LXC engine through its command line. The drivers are responsible for setting up all the associated infrastructure, including the container file systems and network connections.

The next figure shows a high level overview of the driver components. The storage mappers are in charge of mapping the corresponding device into a container filesystem. This first version supports file based and rbd mappings.

Additionally the LXC drivers interact with the network stack to create the needed bridges and tap devices.

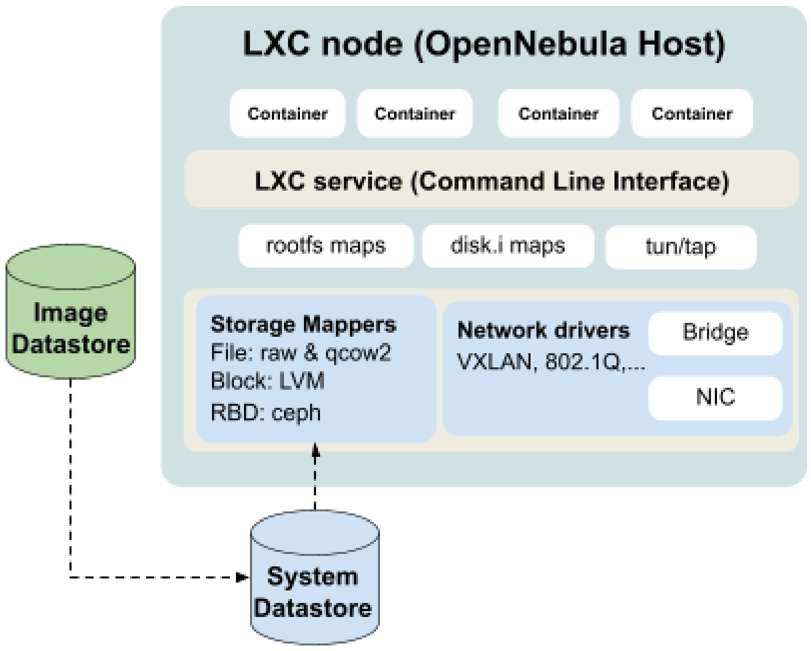


Figure 2.6. LXC main driver components

Data Model

No changes are needed in the VM Template.

API and Interfaces

The new driver does not require any API changes; the Sunstone GUI has been adapted.

3. Edge Provider Selection (CPNT3)

[SR3.1] Edge Provider Catalog Service

Description

This component is implemented in the OneProvision FireEdge service, built in Node.js and React/Redux. This is a brand new component (described as part of SR4.8) developed for ONEedge. It leverages a set of templates that define the providers that are part of the catalog.

Requirements and Specifications

The following requirements are implemented in the second cycle:

- Implement the data model persistence of an edge provider
- Implement an API to manage edge provider entries.
- Add Amazon EC2 and Packet entries

The following requirements are left for the third cycle:

- Implement automatic driver install and configuration in local ONEedge instance from the public Edge Catalog.

Architecture and Components

The FireEdge server implements a REST wrapper around the DOCUMENT POOL OpenNebula subsystem, which persists the information of the provider instances in the OpenNebula DB.

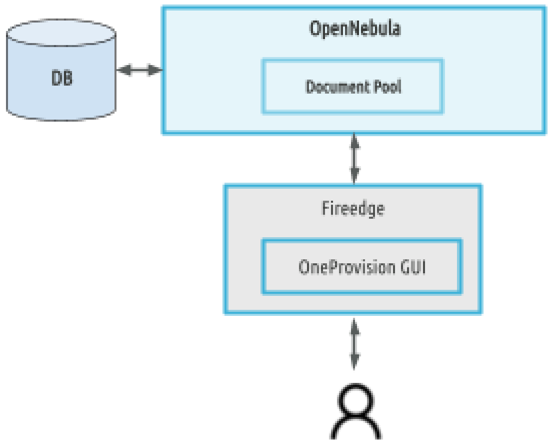


Figure 3.1. Architecture of OpenNebula Backup Solution

Data Model

Provider templates are represented in YAML in the filesystem, to allow git versioning. There is one template per provider per location. The most relevant entries to date are the Packet/Equinix provider template (example below)

```
name: 'packet-amsterdam'

description: 'Elastic cluster on Packet in Amsterdam'
provider: 'packet'

plain:
  provision_type: 'hybrid+'
  image: 'EQUINIX.webp'

location_key: 'facility'
connection:
  token: 'Packet token'
  project: 'Packet project'
  facility: 'ams1'

inputs:
    - name: 'packet_os'
      type: 'list'
      options:
        - 'centos_8'
    - name: 'packet_plan'
      type: 'list'
      options:
        - 'baremetal_0'
```

and the one for the AWS (example below):

```
name: 'aws-north-virginia'

description: 'Elastic cluster on AWS in North Virginia'
provider: 'aws'

plain:
  provision_type: 'hybrid+'
  image: 'AWS.webp'

location_key: 'region'
connection:
  access_key: 'AWS access key'
  secret_key: 'AWS secret key'
  region: 'us-east-1'

inputs:
    - name: 'aws_ami_image'
      type: 'list'
      options:
        - 'ami-0d6e9a57f6259ba3a'
    - name: 'aws_instance_type'
      type: 'list'
      options:
        - 'i3.metal'
```

API and Interfaces

OneProvision GUI providers are managed through the FireEdge server REST API implemented (described in the next table) in Node.js It implements the same functionality as the oneprovider CLI command.

URI	HTTP VERB	DESCRIPTION
/provision/providers/create	POST	Creates a new provider instance using a JSON representation of the provider YAML template.
/provision/providers/edit/<ID>	POST	Updates an existing provider instance (ID) using a JSON representation
/provision/providers/delete/<ID>	PUT	Deletes an existing provider instance (ID)
/provision/providers/list	GET	Get a JSON representation of the set of existing provider instances.

Table 3.1. FireEdge server provider API

[SR3.4] Driver Maintenance Process

Description

This process is used to certify third party companies that want to contribute into provision drivers and also the IPAM drivers. The process is fully automated and can test all the operations involved in each step, from the provider creation until the provision configuration to see if virtual machines can be run and reached through the public interface.

Requirements and Specifications

Each test needs specific information in order to be run:

- Credentials: these are used to connect to the remote provider. Each provider needs specific credentials, these need to be supplied by the third party companies and they are stored in a safe way.
- Terraform ERBs: these files contain all the information about how to create each resource that the provider needs using Terraform. This is not part of the tests itself, but the code needs to be there in order to run the tests.
- IPAM driver scripts: these files are in charge of configuring the networking to the hosts and VMs running on them. They need to be in the code in order to run the tests if not, VMs will not have public access.
- Ansible recipes: they are used to configure the host. OpenNebula uses their own, but in case the provider needs some special configuration, they need to be there in order to run the tests.

Architecture and Components

Tests are structured in three different parts:

- Create provider: in this part the provider is created into OpenNebula database using the valid credentials. After all information is checked, the provider is ready to be used by any provision and the test can continue, if not, the test will fail.
- Create provision: in this part a provision is created using the provision templates published by OpenNebula in the specific remote provider. When everything is configured, tests check that all the information is correct and all the resources are correctly monitored by OpenNebula. After that, a VM is created and run in one of the provision hosts to check that public connectivity works and SSH to it also works.
- Delete provision: if all above tests have passed, the provision is deleted and the test checks that elements have been correctly deleted from OpenNebula database and from the remote provider.

All these tests are executed using rspec tools. This allows us to run tests in an automated way and all together. Tests are written in Ruby and as said before they test the OpenNebula part and the remote provider part.

Data Model

There is no specific data model related to this feature. The important part here is how tests are organised:

```
.
├── aws
│   ├── aws_provision.rb
│   └── defaults.yaml
├── cleanup.rb
├── packet
│   ├── defaults.yaml
│   └── packet_provision.rb
├── provider
│   ├── provider.rb
│   ├── provider.yaml
│   └── update.json
└── provision.rb
```

As can be seen, for each provider there is a specific folder that contains the type_provision.rb which is the file in charge of executing the tests and the defaults.yaml which is the file that contains the URL to the credentials.

[SR3.5] Edge Catalog Web Interface

Description

The different edge providers that are part of the ONEedge catalog need to be presented in a comprehensive manner to the user to choose the optimal provider for their use cases. This is achieved through the OneProvision interface served by the FireEdge component described in SR4.8.

Requirements and Specifications

ONEedge users need to select the optimal provider depending on the type of provision they want to achieve. For instance, if they want to deploy a hybrid+ OpenNebula cluster in the edge, they should be able to choose only among those providers that are certified for that particular cluster.

This relationship is specified as part of the provider and provisions metadata developed in SR3.1, and the GUI developed in SR4.8 implements this Catalog as part of OpenNebula 6.0 release.

Architecture and Components

This catalog is implemented as part of the OneProvision GUI developed to meet SR4.8. The relationship between the different components is depicted in the next figure:

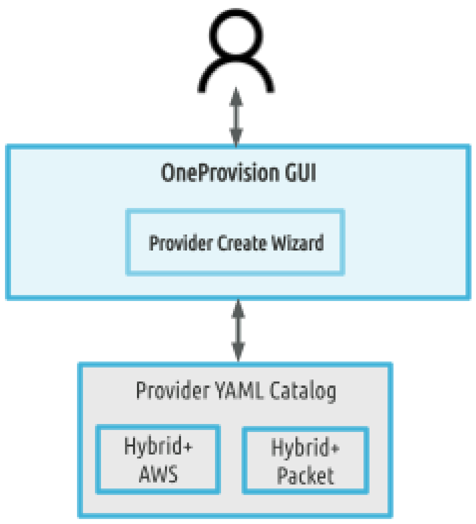
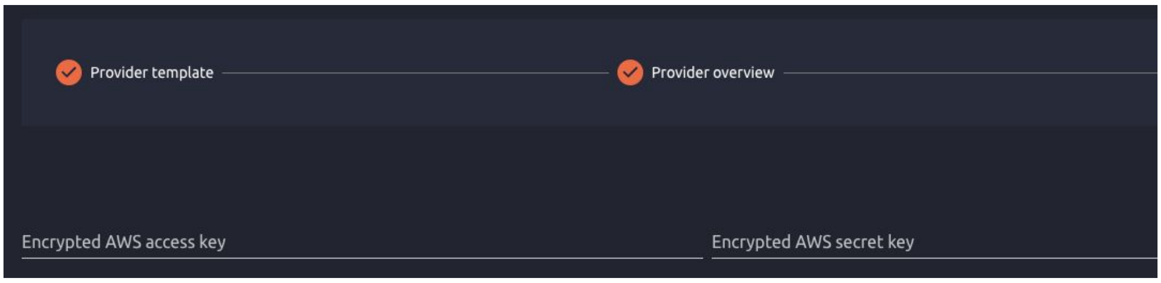


Figure 3.2. Architecture of OpenNebula Backup Solution

Once the user chooses the type of provider they want to add, the OneProvision GUI creates dynamic dialogs depending on the YAML metadata to present HTML forms with the required input needed to create and subsequently use the provider accordingly.

For instance, for the Hybrid+ AWS provider the dialog presented to the end user can be seen in the next figure:



Encrypted AWS access key

Encrypted AWS secret key

Figure 3.3. Provider create dynamic form

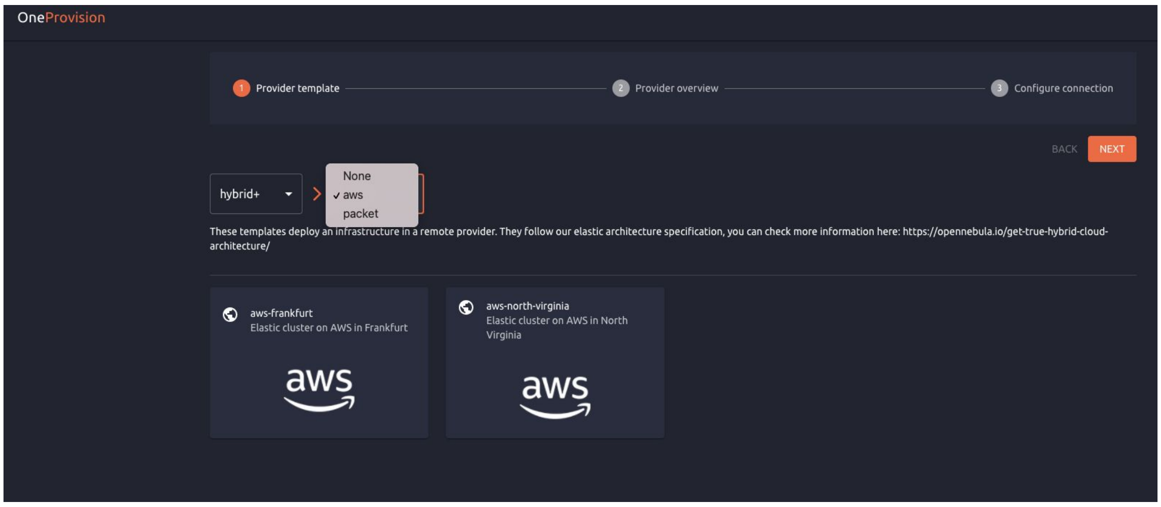
Data Model

The OneProvision GUI leverages the providers YAML metadata defined as templates in SR3.1, using a JSON representation.

API and Interfaces

OneProvision GUI providers are managed through the FireEdge server REST API implemented in Node.js described in SR3.1.

The catalog is presented to the user through a Material Card Interface as captured in the next figure, in combination with several HTML dropdowns to better filter and pinpoint the desired provider.



These templates deploy an infrastructure in a remote provider. They follow our elastic architecture specification, you can check more information here: https://opennebula.io/get-true-hybrid-cloud-architecture/

aws-frankfurt
Elastic cluster on AWS in Frankfurt

aws-north-virginia
Elastic cluster on AWS in North Virginia

Figure 3.4. Edge Catalog GUI dialog

4. Edge Infrastructure Provision and Deployment (CPNT4)

[SR4.1] Reliable Edge Resource Provision

Description

Edge provisions comprises the allocation of multiple resources, like servers, IP allocations or network filter rules. Once the resources are provisioned they need to be configured. The overall process involves multiple components, API calls and network operations that may fail. This SR improves the reliability of the provision component in this scenario.

Requirements and Specifications

- Automatic retry and clean-up operations.
- Background detection and clean-up of orphaned hosts.
- Progress state detection and reporting.
- Individual trigger of provision and configuration phases

Architecture and Components

OneProvision is now based on terraform drivers to interact with the underlying cloud infrastructures. Terraform state is used to track the provision state of Edge resources and perform clean-up and retry operations in a reliable way. All CI/CD tests have been updated to incorporate reliability use cases.

Data Model

Terraform is based on state files that contain the current status of Terraform resources. It is managed automatically by Terraform. OpenNebula stores the file as part of the provision document in the database. The status file is updated every time an operation is performed.

Each Terraform provider resource is mapped with an OpenNebula one in the provider definition file. For example, the AWS mapping includes the following information:

```
TYPES = {
    :cluster   => 'aws_vpc',
    :datastore => 'aws_ebs_volume',
    :host      => 'aws_instance',
    :network   => 'aws_subnet'
}
```

Then each resource is defined by an ERB template. The resource template includes all the information for Terraform to provision the associated OpenNebula resource in the specific provider. For example, the information to provision a cluster in AWS is included in cluster.erb as:

```
resource "aws_instance" "device_<%= obj['ID'] %>" {
    ami             = "<%= provision['AMI'] %>"
    instance_type   = "<%= provision['INSTANCETYPE'] %>"

    security_group_ids = [aws_security_group.device_<%= c['ID'] %>_ssh.id]

    subnet_id    = aws_subnet.device_<%= c['ID'] %>.id

    <% ecidr = c['TEMPLATE']['PROVISION']['CIDR'] || "10.0.0.0/16"
       pref = ecidr.split('/').first.rpartition(".")[0]
       ip = pref << '.' << ( provision['INDEX'].to_i + 4 ).to_s
    %>

    private_ip = "<%= ip %>"

    user_data  = "<%= obj['user_data'] %>"
    tags       = {
        Name = "<%= provision['HOSTNAME'] %>"
    }
}
```

API and Interfaces

A driver is defined by:

- An interface class for the provision engine. The class includes the provider to OpenNebula mapping.
- A set of templates with the terraform definition for each resource type.

Once defined terraform drivers expose an homogenous interface to the basic provision operations, namely: apply, destroy and init.

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

Description

This software requirement improves the usability of the provision core by extending its functionality and ease the provision of a complete infrastructure.

Requirements and Specifications

- Provision needs to be managed as first-class entities, so they are fully managed by OpenNebula
- Provisions should implement access policies to share resources across user groups
- Providers need to be decoupled from the provision definition so they can be reused. A provider includes location definition and characteristics.
- A Provision must include virtual objects like images or service definitions.

Architecture and Components

Provision and provider information are stored in OpenNebula. The core logic of the module is implemented in a stateless library that can be accessed by any other component, e.g. Web UI server, command line tools, etc.

Data Model

Provider and provisions are stored in OpenNebula using the generic Document abstraction. In particular the following types were used:

- Document_type 102: provider.
- Document_type 103: provision.
- Document_type 104: provision template.

Provider

A Provider includes the connection details and zone definition. It also includes the location characteristics like the available instance types or images. The following file shows the basic attributes for an AWS EC2 zone:

```
{
    "id": "Provider ID",
    "provider": "ec2",
    "connection": {
        "ec2_access": "Encrypted AWS access key",
        "ec2_secret": "Encrypted AWS secret key",
        "region_name": "AWS region name",
    }
}
```

Provision

A provision consists of two parts:

- The Terraform state and configuration to manage the provisioned resources in the provider
- The OpenNebula representation of the infrastructure and its associated resources.

The following file outlines the contents of the provision (note that each object includes a name and its Opennebula ID):

```
{
    "id": "Provision ID",
    "name": "Provision name",
    "provider": "packet|ec2",
    "description": "Provision description",
    "state": "pending|deploying|configuring|running|error|done",
    "start_time": "Provision started time",
    "tf": {
        "configuration": "Terraform configuration in base 64",
        "state": "Terraform state in base 64"
    },
    "provision": {
        "infrastructure": {
            "clusters": [
                {
                   "name": "Cluster name",
                   "id": "Cluster ID"
                },
                ...
            ],
            "hosts": [
                …
            ],
            "datastores": [
                …
            ],
            "networks": [
                …
            ]
        },
        "resources": {
            "images": [
                …

            ],
          "templates": [
                …
            ],
          "vnet_templates": [
                …
            ],
          "service_templates": [
                …
            ]
        }
    }
}
```

API and Interfaces

Provision and providers are managed through a set of CLI tools that enable the common operation on the objects (creation, update, deletion and control). The provision core is structured in a stateless library that can be invoked by any other component to interface with existing providers and provisions.

[SR4.3] Provision Template for Reference Architectures

Description

From the operational perspective, the implementation of a reference architecture comprises three components, namely:

- Definition of a set of providers to deploy the architecture, including supported OS, instance types and locations
- Definition of provision files with the resources that implements the architecture, e.g. hosts, networks or datastores.
- A set of configuration recipes that setups the associated resources.

This software requirement provides the above set of files for the ONEedge reference architecture on two reference providers.

Requirements and Specifications

Deploy reference architecture based on different hypervisor flavors on reference providers. This cycle includes KVM, Firecracker based architectures on Amazon AWS and Equinix.

Architecture and Components

No additional components are included in this software requirement.

Data Model

Each provision, together with the supported providers are structured in their own folders ready to be deployed by users (either through CLI or Web UI). As an example the KVM implementation of the reference architecture includes the following providers and provision files:

```
|-- providers
|   |-- aws
|   |   |-- aws-eu-central-1.yml
|   |   |-- aws-eu-west-2.yml
|   |   |-- aws-us-east-1.yml
|   |   `-- aws-us-west-1.yml
|   `-- packet
|       |-- packet-ams1.yml
|       |-- packet-ewr1.yml
|       |-- packet-nrt1.yml
|       `-- packet-sjc1.yml
`-- provisions
    |-- aws.d
    |   |-- datastores.yml
    |   |-- defaults.yml
    |   |-- inputs.yml
    |   `-- networks.yml
    |-- aws.yml
    |-- common.d
    |   |-- defaults.yml
    |   |-- kvm_hosts.yml
    |   `-- resources.yml
    |-- packet.d
    |   |-- datastores.yml
    |   |-- defaults.yml
    |   |-- inputs.yml
    |   `-- networks.yml
    `-- packet.yml
```

Additionally, in this software requirement we have developed the associated ansible roles and playbooks to automatically configure the provisioned resources. The list of roles and playbooks are shown below:

```
|-- aws.yml      //AWS playbook configuration file
|-- packet.yml   //Equinix-Packet playbook configuration file
`-- roles
    |-- ddc
    |-- frr
    |-- iptables
    |-- opennebula-node-kvm
    |-- opennebula-repository
    |-- opennebula-ssh
    |-- python
    `-- update-replica-server
```

API and Interfaces

Not applicable.

[SR4.5] Drivers for Host Provision

Description

Oneprovision uses an uniform Terraform driver to provision cloud resources, from servers to security group rules or elastic IPs. The use of Terraform enables OneProvision to leverage its ecosystem that includes most of the existing providers.

Requirements and Specifications

- Seamlessly integrate with terraform
- Support advanced operations to interface the full range of the provider capabilities.

Architecture and Components

The terraform driver is implemented as a runtime component in oneprovision. It is used to the main terraform operations:

- Apply: this operation applies all the changes that have been made in the terraform internals files, adding or removing resources as needed. Corresponds to:

  ```
  terraform apply -auto-approve
  ```

- Destroy: this operation destroys a resource in the remote provider. It can destroy all the resources or just one if the ID is used:

  ```
  terraform destroy #{target} -auto-approve
  ```

- Init: this operation validates the configuration to check that everything is correct. It also validates the version of the configuration files:

  ```
  terraform init
  ```

Terraform use two different files to manage a provision:

- Configuration: the file stores all the resources that are going to be created in the remote provider.
- State: this file is created after the apply operation is performed. It contains the current status of Terraform resources.

These files are stored as part of the provision in order the driver to operate.

Data Model

The Terraform driver includes separate provider files to express the unique characteristics and features of each one. The implementation has the following structure:

```
src/oneprovision/lib/terraform/providers
```

```
.
├── aws.rb
├── dummy.rb
├── packet.rb
└── templates
    ├── aws
    │   ├── cluster.erb
    │   ├── datastore.erb
    │   ├── host.erb
    │   ├── network.erb
    │   └── provider.erb
    └── packet
        ├── cluster.erb
        ├── datastore.erb
        ├── host.erb
        ├── network.erb
        └── provider.erb
```

- <PROVIDER>.rb: the implementation of the driver itself.

Each provider includes specific Terraform templates for each resource class:

- cluster.erb: This is like a general section to create objects in the remote provider. It includes shared resources for the cluster.
- datastore.erb: it contains all the information to create a data volume in the remote provider.
- host.erb: it contains the server definitions.
- network.erb: it contains all the information needed to set up networking in the remote provider.
- provider.erb: template file to create and define the provider. It includes the attributes needed to interface the provider.

API and Interfaces

Not applicable.

[SR4.6] Drivers for IP Address Management

IPAM drivers for Edge providers have been rewritten from scratch to not expose the hypervisor to the virtual machines as well as to not require network address translations. The features planned to improve the original drivers have been superseded by the new features implemented as part of SR4.7. All associated features and requirements have been moved to SR4.7.

[SR4.7] Drivers for Network Drivers and Helpers

Description

This software requirement group all network related improvements. It includes some of the requirements previously defined by SR4.6. However, the design of SR4.7 has deprecated most of the requirements in SR4.6.

Requirements and Specifications

- Allocate public IPs to provisions
- Configure servers to route public traffic to the corresponding VM
- Setup private networking for intra VM communication
- Provide a generic implementation for private networking compatible with any cloud provider

Architecture and Components

The network stack of oneprovision includes the following components:

- IPAM drivers. It uses the standard OpenNebula interface and it is responsible for allocating and releasing elastic IPs on the remote provider.
- Elastic network driver. It setups routing in the hypervisor so public traffic associated to the elastic IPs are forwarded to the corresponding VM. It also instructs the provider network backbone to route the elastic IP to the target hypervisor.
- VXLAN EVPN routing. Private networking is based on VXLAN overlay to encapsulate VM traffic on hypervisor IPs. This traffic, generated by the servers, is correctly routed by the provider. A set of BGP servers and Route Reflectors are automatically configured as part of the provision.

Data Model

Network drivers access the provider information as defined in SR4.2 to interface with the cloud provider to allocate public IP addresses. No additional data is needed.

VM network interface definition has been extended to include the public IP when it is attached to a NAT'ed network, like that used by AWS. In this case the interface include an EXTERNAL attribute with the public facing IP:

```
"NIC": [
       {
         "AR_ID": "0",
         "AWS_ALLOCATION_ID": "eipalloc-06f87b0e6f030d761",
         "BRIDGE": "br0",
         "BRIDGE_TYPE": "linux",
         "CLUSTER_ID": "105",
         "EXTERNAL": "18.213.66.111",  // public IP address
         "GATEWAY": "10.0.66.110",
         "IP": "10.0.66.111",  // VM IP, as seen by the guest
         "MAC": "02:00:0a:00:42:6f",
         "MODEL": "virtio",
         "NAME": "NIC1",
         "NETWORK": "aws-cluster-public",
         "NETWORK_ID": "3",
         "NETWORK_UNAME": "oneadmin",
         "NIC_ID": "1",
         "SECURITY_GROUPS": "0",
         "TARGET": "one-15-1",
         "VN_MAD": "elastic"  // provision network driver
       }
]
```

API and Interfaces

Not applicable.

[SR4.8] GUI for Edge Resource Provision

Description

The full list of operations that are available through the set of oneprovision and oneprovider commands can be performed as well through the OneProvision GUI served by FireEdge.

FireEdge is a Node.js server that delivers the OneProvision GUI written in React/Redux that has been developed in the context of ONEedge and will be included in OpenNebula 6.0

Requirements and Specifications

The OneProvision GUI seeks to meet the following requirements:

- New OneProvision GUI extension.
- Update OpenNebula host interface for state control operations (power-off/on).
- Troubleshoot remote host configuration.
- Asynchronous background jobs runner.

Architecture and Components

FireEdge server components and their high-level relationship with OpenNebula components are depicted in the next figure.

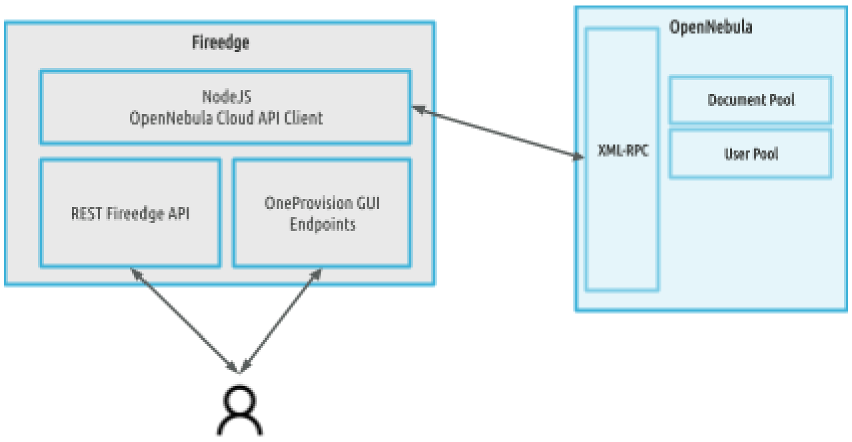


Figure 4.1. FireEdge server high-level architecture

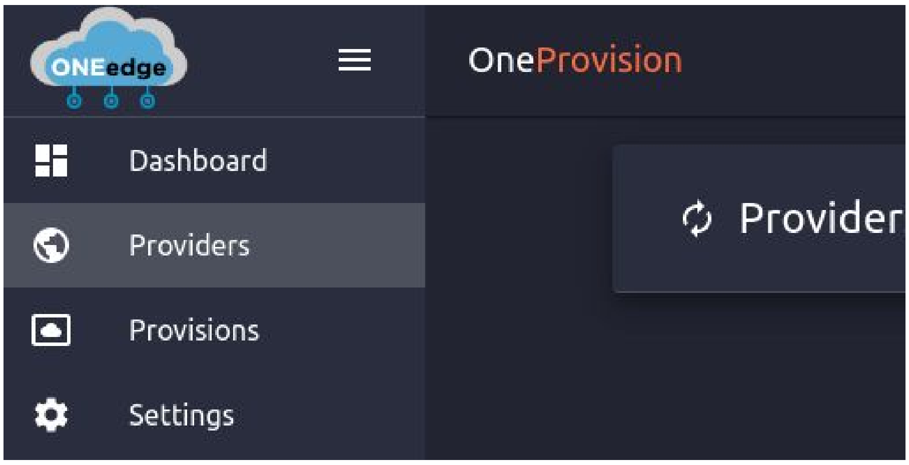


Figure 4.2. OneProvision GUI Wireframe

The OneProvision GUI (depicted in the figure above) implements a wireframe consisting of the elements described in the next table:

MENU ITEM	DESCRIPTION
Dashboard	Aggregates information about total number of providers and provisions, as well as current states and types.
Providers	Displays a card interface with the current existing providers. Implements a wizard like stepper to ease the creation of new providers .using a set of templates. Allows for the update of providers
Provisions	Serves the same functionality as providers as well as: ● Allows for live stream of oneprivison logs. Ansible playbooks can be inspected and configuration relaunched if anything fails ● Allows for the detail view of OpenNebula hosts, virtual networks and datastores as well as clusters that are part of the provision

Table 4.1. OneProvision GUI Wireframe element description

Since the provisions representation in the OpenNebula Document Pool also include the information about the spawned hosts in the remote edge locations, comprising both the IP endpoint as well as the SSH private key authorized in said remote hosts, this information is used by the FireEdge server to deliver a Guacamole gateway SSH session embedded in the user browser. This leverages the “oneprovision host ssh” functionality.

FireEdge server hence depends on the “guacd” daemon provided by the Apache Guacamole project, and implements a new API route that handles:

- Guacamole authorization
- Remote host ssh authorization
- Web socket creation to bridge the guacd gateway and the guacamole lite[3] HTML component (chosen due to its lightness over the default official guacamole HTML client).

[3] https://www.npmjs.com/package/guacamole-lite

Data Model

FireEdge server provides a JSON implementation of the Provision (described in SR4.3) and Provider (described in SR3.1) templates.

Additionally, it provides a complete wrapper as a REST interface of the OpenNebula XMLRPC API, which provides full functionality for the interaction with OpenNebula. This is mainly used for access to the document pool[4] to store the instances of provisions and providers; as well as the user pool.[5]

API and Interfaces

OneProvision GUI providers are managed through the FireEdge server REST API implemented (described in the table below) in Node.js. It implements the same functionality as the oneprovision CLI command.[6] Provider API is described in SR3.1.

URI	HTTP VERBS	DESCRIPTION
/provision/{ cluster, datastore, host, image, template network}/{ list, delete}	GET, PUT	List and delete OpenNebula resources in the Edge as part of a provision. Host object admits additional actions: ssh, reconfigure and top.
/provision/provision/edit/<ID>	POST	Updates an existing provision instance (ID) using a JSON representation
/provision/provision/delete/<ID>	PUT	Deletes an existing provision instance (ID)
/provision/provision/list	GET	Get a JSON representation of the set of existing provision instances.

Table 4.2. FireEdge server provision API

[4] http://docs.opennebula.io/5.13/development_guide/system_interfaces/api.html#documents
[5] http://docs.opennebula.io/5.13/development_guide/system_interfaces/api.html#oneuser
[6] http://docs.opennebula.io/5.13/operations_guide/automatic_provisioning_management/provider.html

5. Edge Apps Marketplace (CPNT5)

[SR5.1] Edge Applications and Services in Marketplace

Description

Extends the current OpenNebula VM and containers marketplace to deal with OneFlow services. The marketplace supports mainly three different types of applications:

- IMAGE: this contains just an image, it can be used by a VM or a container. It contains the image itself and also the attributes associated with it. When the user downloads it, an image is created in OpenNebula. This image is ready to be used and needs to be associated with a VM template.

Important note about this type is that in order to support old OpenNebula versions, some of the IMAGE applications also have a VM template associated with it, so when the user downloads it, a VM template is created along it:

- VMTEMPLATE: this contains a collection of IMAGE applications. When the user downloads it, all the IMAGE associated with it, will be also downloaded in a recursive way. At the end, the user will have a VM template and multiple images available.
- SERVICE_TEMPLATE: this contains a collection of VMTEMPLATE applications. When the ser downloads it, all the VMTEMPLATE associated with it, will be also downloaded in a recursive way. At the end, the user will have a service template, all the VM templates and the images.

Requirements and Specifications

<u>Export</u>

This operation is in charge of downloading an application from the marketplace into OpenNebula. The operation is completely transparent to the user. The user just needs to click on the export button, or use the CLI operation, and all the information will be downloaded in the selected datastore.

All the objects are created using the metadata information stored in each application and the operation returns the ID of each object that has been just created.

<u>Import</u>

This operation is in charge of adding a new object to the marketplace. The user can take any image, VM template, service template or VM and upload it to the marketplace. The upload operation is transparent to the user, but the user needs to specify some information. This information is mainly the marketplace to upload everything and specify if all the information needs to be uploaded or not.

For example, in the VM templates case, it can upload the VM template and the images, or just the VM template. The same as a service template, it can upload everything or just the service template information.

OpenNebula Public Marketplace

These enhancements can be used in private marketplaces, but have also been included in the OpenNebula public marketplace. New service templates have been prepared so any user can download and use it. For example, the Kubernetes service has been added as a SERVICE_TEMPLATE, so when the user downloads it, a service template with a master and worker is created.

Architecture and Components

Monitor

The script retrieves all the applications available in the marketplace. It has been adapted in order to support new applications types. It creates all the needed sections in each application template, so the rest of the components understand it. OpenNebula has one of these scripts for each market driver it supports, but the only one that has been modified is the OpenNebula market driver. The other types of marketplace drivers do not need these modifications as they are not able to have these new application types.

Marketplace

The marketplace code has been adapted to support new applications. These applications are YAML files that define all the information; they can be found on the OpenNebula public repository https://github.com/OpenNebula/marketplace

New functionality has been added to the marketplace code in order to give our users some SERVICE_TEMPLATES in the OpenNebula public marketplace.

Export

This operation was previously available in OpenNebula, it has been adapted to support new applications types. The main code has been added to the Ruby API, so it can be used in the CLI and in Sunstone. The code is divided into three parts depending on the type of application. The function is called recursively to export all the associated applications in case there are any.

As part of this adaptation, API extensions have been added. These extensions are code that is not an API itself, but it helps in the API functions. So the main functionality is in the API and these helper functions are in the extension files, so is a way of having the code more organised.

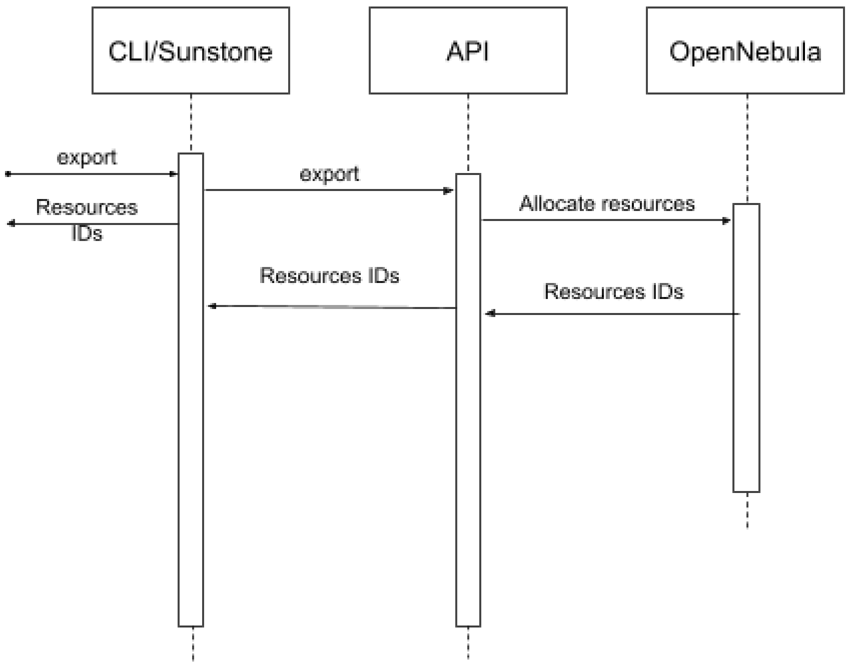


Figure 5.1. Export operation diagram

<u>Import</u>

This operation is completely new. The idea behind this is that the user is able to add the VM templates, service templates and VMs into the private marketplace. Import operation checks all the resources that the template or VM has and then creates an application in the marketplace for each of them. This import operation is only supported in HTTP and S3 private marketplace.

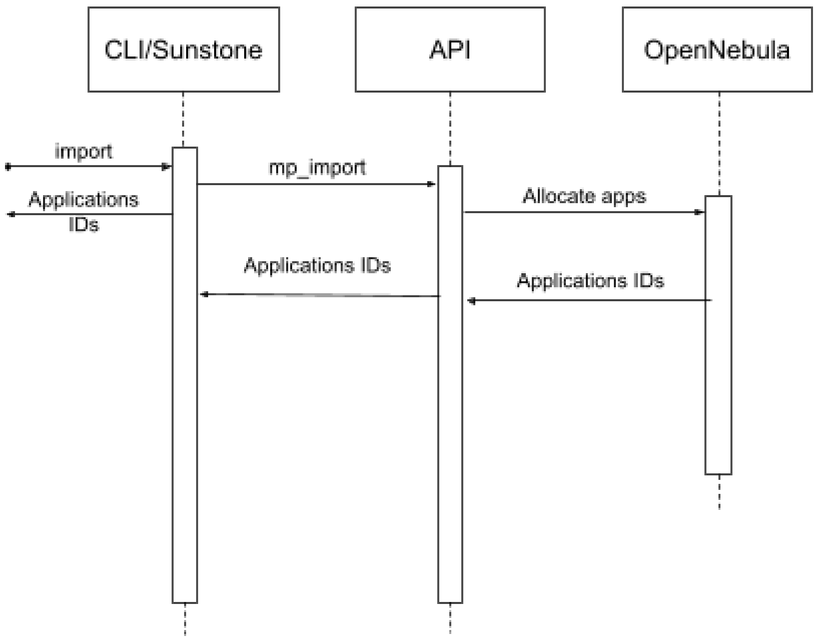


Figure 5.2. Import operation diagram

Data Model

Common attributes for all applications types:

Attribute	Description	Mandatory
NAME	Application name	YES
ORIGIN_ID	The ID of the source image. It must reference an available image and it must be in one of the supported datastores.	YES
TYPE	Application type: ● IMAGE ● VMTEMPLATE ● SERVICE_TEMPLATE	YES
MARKETPLACE_ID	The target marketplace ID. Alternatively you can specify the MARKETPLACE name.	YES
MARKETPLACE	The target marketplace name. Alternatively you can specify the MARKETPLACE_ID name.	YES
DESCRIPTION	Text description of the MarketPlaceApp.	NO
PUBLISHER	If not provided, the username will be used.	NO
VERSION	A string indicating the MarketPlaceApp version.	NO

Table 5.1. Common information to all application types

VM Template

This is a collection of single IMAGE in marketplace, so each VM template has DISK attribute to link other applications in the marketplace:

```
DISK = [
   APP  = APP_IN_MARKETPLACE,
   NAME = JUST_A_NAME,
   TYPE = DISK_TYPE
]
```

Type can have three possible values:

- IMAGE: this is a normal disk and should be added as DISK in the final VM template.
- KERNEL: this is a kernel and should be added under OS/KERNEL_DS
- CONTEXT: this are files and should be added under CONTEXT/FILES_DS

Attribute	Description
DISK	Application disks information: ● NAME: disk name to reference it ● APP: app name to import from marketplace

VMTEMPLATE64	VM template (in base64) to create in OpenNebula with disks section to reference applications to export

Table 5.2. VMTEMPLATE type specific attributes

See this VMTEMPLATE example:

```
MARKETPLACE APP 432 INFORMATION
ID             : 432
NAME           : MyTTY - template
TYPE           : VMTEMPLATE
USER           : oneadmin
GROUP          : oneadmin
MARKETPLACE    : Private HTTP
STATE          : rdy
LOCK           : None

PERMISSIONS
OWNER          : um-
GROUP          : ---
OTHER          : ---

DETAILS
SOURCE         :
PUBLISHER      :
REGISTER TIME  : Wed Jul 22 18:11:05 2020
VERSION        : 0.0
DESCRIPTION    :
SIZE           : 0M
ORIGIN_ID      : -1
FORMAT         :

IMPORT TEMPLATE


MARKETPLACE APP TEMPLATE
DISK=[
  APP="MyTTY-image",
  NAME="image_0",
  TYPE="IMAGE" ]
DISK=[
  APP="MyKernel-kernel",
  NAME="kernel_0",
  TYPE="KERNEL" ]
DISK=[
  APP="File1-context",
  NAME="context_0",
  TYPE="CONTEXT" ]
DISK=[
  APP="File2-context",
  NAME="context_0",
  TYPE="CONTEXT" ]
TYPE="VMTEMPLATE"
VMTEMPLATE64="Q09OVEVYVD1bCiAgTkVUV09SSz0iWUVTIiwKICBTU0hfUFVCTElDX0tFWT0i
JFVTRVJbU1NIX1BVQkxJQ19LRVldIiBdCkNQVT0iMC4xIgpHUkFQSElDUz1b
CiAgTElTVEVOPSIwLjAuMC4wIiwKICBUWVBFPSJWTkMiIF0KSU5QVVRTX09S
REVSPSIiCkxPR089ImltYWdlcy9sb2dvcy9saW51eC5wbmciCk1FTU9SWT0i
MTI4IgpNRU1PUllfVU5JVF9DT1NUPSJNQiIKT1M9WwogIEJPT1Q9IiIgXQ==
"
```

Service Template

This resource represent a collection of VMTEMPLATE apps in marketplace, so each service template has ROLE attribute to link other applications in the marketplace:

```
ROLE = [
   APP  = APP_IN_MARKETPLACE,
   NAME = ROLE_NAME
]
```

Attribute	Description
ROLE	Application information for roles: ● NAME: role name to reference it ● APP: app name to import from marketplace
VMTEMPLATE64	VM template (in base64) to create in OpenNebula with roles section to reference applications to export. It is in JSON format

Table 5.3. SERVICE_TEMPLATE type specific attributes

See this SERVICE_TEMPLATE example:

```
MARKETPLACE APP 433 INFORMATION
ID              : 433
NAME            : ServiceTemplate
TYPE            : SERVICE_TEMPLATE
USER            : oneadmin
GROUP           : oneadmin
MARKETPLACE     : Private HTTP
STATE           : rdy
LOCK            : None

PERMISSIONS
OWNER           : um-
GROUP           : ---
OTHER           : ---

DETAILS
SOURCE          :
PUBLISHER       :
REGISTER TIME   : Wed Jul 22 18:11:05 2020
VERSION         : 0.0
DESCRIPTION     :
SIZE            : 0M
ORIGIN_ID       : -1
FORMAT          :

IMPORT TEMPLATE


MARKETPLACE APP TEMPLATE
ROLE=[
  APP="MyTTY - template",
  NAME="Router" ]
ROLE=[
  APP="MyTTY - template",
  NAME="Router2" ]
TYPE="SERVICE_TEMPLATE"
VMTEMPLATE64="eyJuYW1lIjoiU2VydmljZVRlbXBsYXRlIiwiZGVwbG95bWVudCI6InN0cmFp
Z2h0IiwiZGVzY3JpcHRpb24iOiJUaGlzIGlzIGEgdGVzdCIsInJvbGVzIjpb
eyJuYW1lIjoiUm91dGVyIiwiY2FyZGluYWxpdHkiOjEsInZtX3RlbXBsYXRl
IjowLCJlbGFzdGljaXR5X3BvbGljaWVzIjpbXSwic2NoZWR1bGVkX3BvbGlj
aWVzIjpbXX0seyJuYW1lIjoiUm91dGVyMiIsImNhcmRpbmFsaXR5IjoxLCJ2
bV90ZW1wbGF0ZSI6MCwiZWxhc3RpY2l0eV9wb2xpY2llcyI6W10sInNjaGVk
dWxlZF9wb2xpY2llcyI6W119XSwicmVhZHlfc3RhdHVzX2dhdGUiOmZhbHNl
fQ==
"
```

API and Interfaces

API

The export operation has the same API function and options, the only thing changed is the functionality itself to adapt it to new applications.

Import operation is completely new. It has been added in the corresponding API extension:

- template_ext: the function mp_import is in charge of getting the template and check all the information associated with it. Then, it will remove all the network information, as this information is related with the OpenNebula installation itself. Finally, will create all the needed applications in the marketplace.
- service_template_ext: the function mp_import checks all the VM templates associated with each role and calls the template mp_import in order to import them. Finally, it creates an application in the marketplace with all the service template information.

CLI

VM import:

```
$ onemarketapp vm import

Command vm import requires one parameter to run
## USAGE
vm import <vm_id>
        Imports a VM into the marketplace
        valid options: market, no, vmname, yes
```

VM template import:

```
$ onemarketapp vm-template import

Command vm-template import requires one parameter to run
## USAGE
vm-template import <vm_template_id>
        Imports a VM template into the marketplace
        valid options: market, no, vmname, yes

## OPTIONS
     --market market_id        Market to import all objects
     --no                      Do not import/export associated VM
```

```
                                  templates/images
     --vmname name                Selects the name for the new VM Template, if the
                                  App contains one
     --yes                        Import associated VM templates/images
```

Service template import

```
$ onemarketapp service-template import

Command service-template import requires one parameter to run
## USAGE
service-template import <service_template_id>
        Imports a service template into the marketplace
        valid options: market, no, vmname, yes

## OPTIONS
     --market market_id           Market to import all objects
     --no                         Do not import/export associated VM
                                  templates/images
     --vmname name                Selects the name for the new VM Template, if the
                                  App contains one
     --yes                        Import associated VM templates/images
```

Check SR2.9 for a description of the changes affecting the Sunstone GUI.

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

Description

OpenNebula's Kubernetes Virtual 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.

The goal of this SR is to improve this OpenNebula and Kubernetes integration, enabling elasticity of the OpenNebula managed Kubernetes cluster. It is complementary to SR5.3, which provides an alternative way to run existing container appliances in ONEedge.

Additionally, a lightweight flavour of Kubernetes by Rancher Labs, called k3s, has been tested and integrated to be consumed by ONEedge users.

Requirements and Specifications

The requirements for the application containers engine are:

- Kubernetes managed controller functionality with automatic upgrades and credential provisioning.
- Implement easy worker node addition/subtraction to the Kubernetes cluster, based optionally on elasticity rules.

The objective of this second cycle relates to the second requirement, in particular the ability to deploy different Kubernetes architectures, as well as a better network support using a preconfigured with Canal CNI networking (Calico and Flannel). Moreover, as part of the second cycle, the Kubernetes appliance was certified as part of Certified Kubernetes Conformance Program.[7]

Architecture and Components

A new appliance with preinstalled Kubernetes has been produced that is intended to work as a single-node cluster, manually managed multi-node cluster as in the first cycle, or automatically managed multi-node cluster as OpenNebula OneFlow Service, and as such OneFlow elasticity rules[8]. These rules can be acting over hypervisor gathered metrics (through the OpenNebula monitoring probes) or through service defined metrics (pushed from the VMs to the OneGate service).

[7] https://opennebula.io/certified-kubernetes-appliance/

[8] http://docs.opennebula.io/5.13/configuration_guide/oneflow_configuration/appflow_elasticity.html

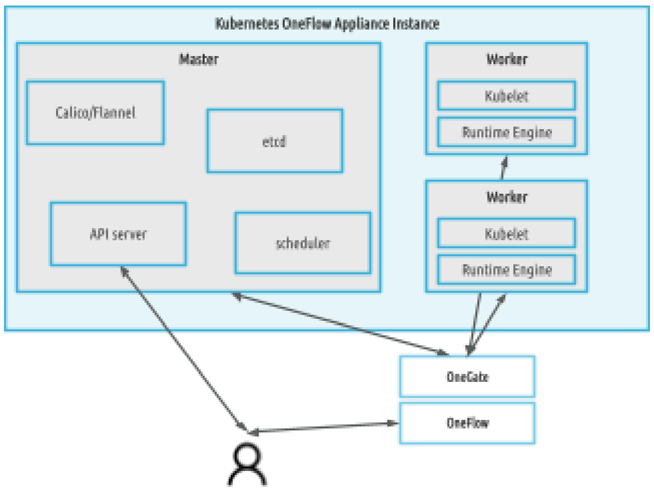


Figure 5.3. Kubernetes appliance as a multi-node deployment with OneFlow/OneGate

The appliance comes with an optional Dashboard component[9] that eases the management of container applications deployed in the particular Kubernetes clusters deployed in the edge.

Additionally, means to automatically launch k3s clusters in the edge are provided leveraging the Dockerfile functionality described in SR5.3. Using the Dockerfile as the base information source, ONEedge performs the following steps to launch a k3s cluster:

- Constructs a docker container from the Dockerfile.
- Creates a VM with either KVM or Firecracker (more suitable for edge platforms).
- Configures the Virtual Machine so it launches the docker container as its sole process.

Using this mechanism, even if we are using a docker container and not a Virtual Machine, all the functionality from OpenNebula can be leveraged, including any existing storage and networking backed support.

Data Model

Only one addition in this second cycle to the Data Model with respect to the one defined in D3.1. For multi-node deployment managed by OneFlow, it's necessary to set the parameter in the table below to enable an exchange of the connection parameters among the nodes via OneGate, and report when the nodes are ready to use. This ensures that the enrollment in the Kubernetes cluster is performed smoothly.

ATTRIBUTE	VALUE
ONEGATE_ENABLE	Needs to be set to YES for multi-node deployment, NO otherwise

Table 5.4. Additional contextualization parameter.

API and Interfaces

Not applicable.

[9] https://docs.opennebula.io/appliances/service/kubernetes.html#k8s-dashboard

[SR5.3] Integration with Application Containers Marketplace

Description

Improve existing integration with application containers marketplace (i.e DockerHub) to make it more customizable and straightforward to the user.

Requirements and Specifications

The integration with DockerHub allows end users not only to download existing images from DockerHub but also build their own images compatible with OpenNebula defining a Dockerfile.

In order to improve the number of supported images, busybox based distributions should be supported by OpenNebula context packages. This simple distribution is a very common distribution for building simple and lightweight applications images.

Architecture and Components

The architecture remains the same, a new URL format has been added to the existing components (see Data Model section below).

The context packages have been updated to support Busybox based images without any changes to its architecture.

Data Model

Currently in order to download an image from DockerHub an special URL needs to be used pointing to the image that needs to be downloaded:

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

This has been extended by adding a new URL format in order to allow, not only to download existing images from DockerHub, but also to build custom images using a Dockerfile.

URLs starting with dockerfile:// will be used to generate an image from the referenced Dockerfile file:

```
dockerfile://<file_path>?fileb64=<file_content_b64>?size=<image_size>&filesystem=<fs_type>
&format=raw
```

ATTRIBUTE	DESCRIPTION
<file_path>	Path to the Dockerfile.
fileb64	Dockerfile content encoded in base64. If this argument is used <file_path> will be ignored.

size	The size of the new image.
filesystem	The image underlying file system type (e.g ext4, xfs, ...).
context	NO to avoid adding context dependencies to the image.

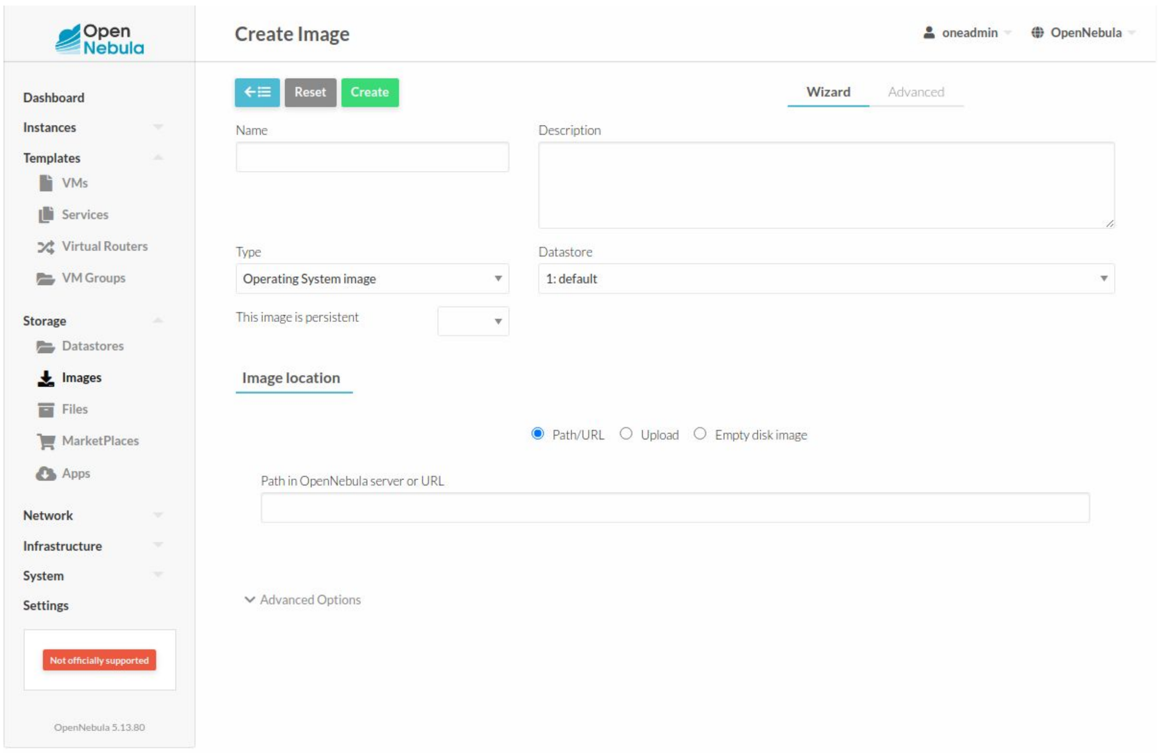
Table 5.5. Available attributes

API and Interfaces

Support for Dockerfiles is available in the main user interfaces. The CLI interface adds a new flag interpretation to the oneimage command. The --path argument should be used with a different URL.

```
oneimage create --name <name> --path  dockerfile://<path_to_file> -d 1
```

In Sunstone web interface (see figure below), the path can be input with the same syntax and the backend will interpret it correctly.



Type: Operating System image

Datastore: 1: default

[✓ Path/URL] Upload Empty disk image

Figure 5.4. Sunstone create image dialog

[SR5.5] Edge Market GUI Developments

Description

This component intends to enable end users to be able to deploy both Edge Applications based on VMs and system containers as well as application containers in the Distributed Edge Cloud using a simple point and click web interface.

A technology preview of FireEdge Flow, a self service portal to deploy and manage applications on top of an OpenNebula cloud, will be included. It is not ready for production but it showcases the ability to create a full application (using a graph interface) based on VM Templates and containers (which can be mixed) and deploy it on-prem or over a provision created with the OneProvision GUI.

Requirements and Specifications

This component meets the following requirements, although it is still a technology preview:

- Aggregate OpenNebula marketplace for VMs in EdgeMarket.
- Aggregate System Container Linux Containers marketplace.
- Aggregate System Container TurnKey Linux.
- Add functionality to ONEedge interface for Application Containers enabling deployment in selected Edge Provider locations.

Architecture and Components

FireEdge Flow leverages the same server, FireEdge, used to implement the OneProvision GUI defined in the following architecture:

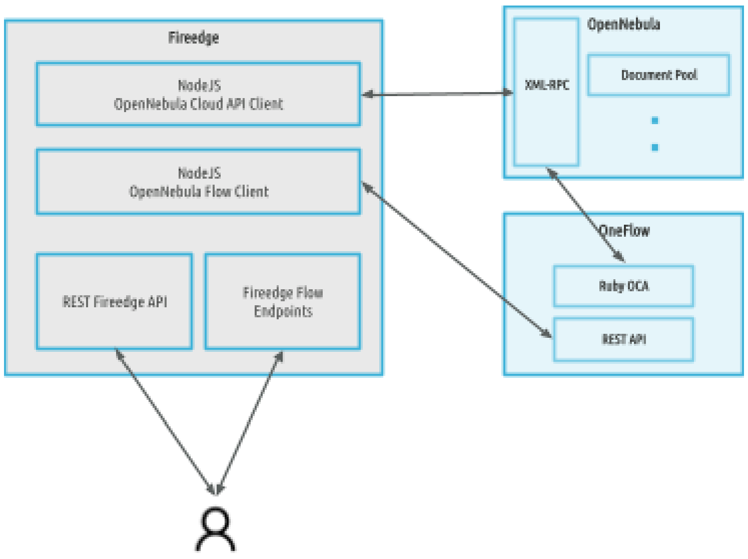


Figure 5.5. FireEdge Flow high-level architecture

Data Model

Besides the Node.js wrapper as a REST interface of the OpenNebula XMLRPC API mentioned in SR4.8, FireEdge Flow needs a Node.js wrapper of the OneFlow server component, which is a separated component of OpenNebula and as such is not accessible through the XMLRPC API.

The OneFlow component implements a REST interface in turn (described in the official OpenNebula documentation).[10] As FireEdge Flow provides an interface over OneFlow, this endpoint is used for Flow creation and management, and the OpenNebula API is mostly used for authentication.

Since OneFlow internal representation of services is structured in JSON, the conversion is more straightforward than with the XMLRPC API of OpenNebula.

API and Interfaces

The main components that can be managed through FireEdge related to service management are presented in the table below. In the description, managing means to create, update and delete a resource.

URI	HTTP VERB	DESCRIPTION
/provision/flow/service/[<ID>]	POST, GET	Manages a Service representation in JSON
/provision/flow/service/<ID>/roles/[<ID>]	POST, GET	Manages roles within a Service ID
/provision/flow/service_templates/[<ID>]	POST, GET	Manages a Service Template representation in JSON
/provision/flow/service_pool	GET	Retrieve all Service instances
/provision/flow/service_templates	GET	Retrieve all Service templates

Table 5.6. FireEdge Flow REST API

A screenshot of a technology preview FireEdge Flow can be seen in the figure below. The card interface is showing a list of entries in the DockerHub registry that can be used to compose Applications (called Service in FireEdge Flow), potentially with a mix of Virtual Machines and containers.

[10] http://docs.opennebula.io/5.13/development_guide/system_interfaces/appflow_api.html

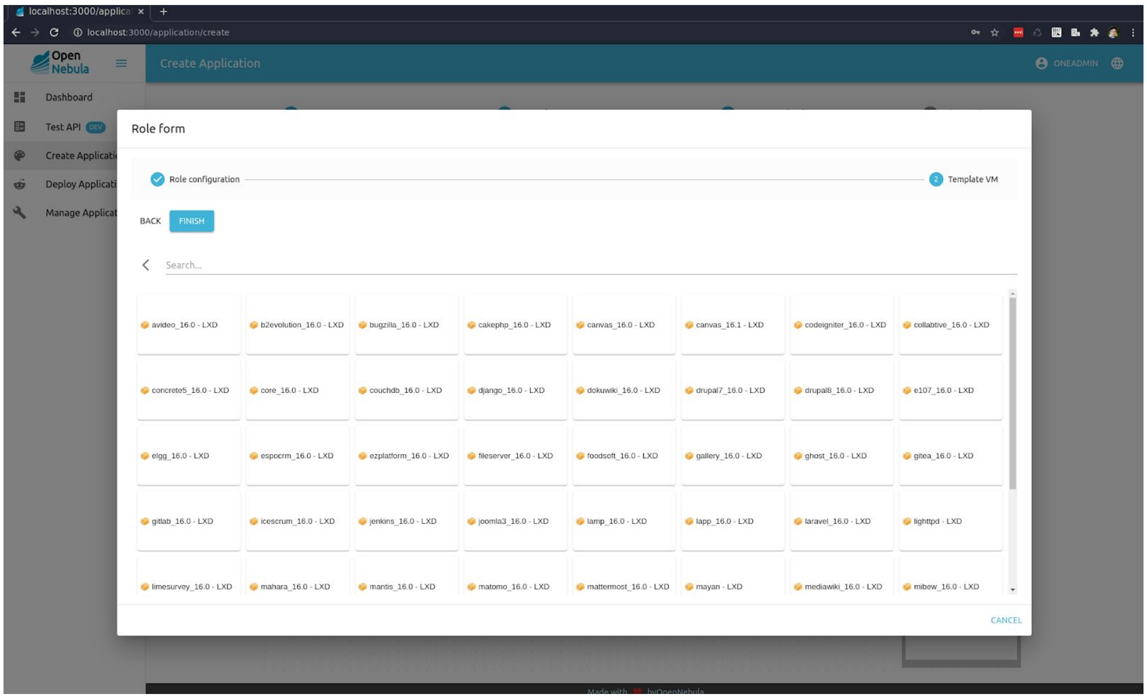


Figure 5.6. FireEdge Flow service definition dialog