ONEedge.io

A Software-defined Edge Computing Solution

D4.3. Infrastructure Report - c

Infrastructure Incremental Report

Version 1.0

3 November 2021

Abstract

This report, delivered at the end of the Third Innovation Cycle (M17-M23), describes in detail the new Continuous Integration infrastructure management process that has been totally redesigned during this cycle to be fully automated, ensuring that it can be deployed from scratch within hours in either local infrastructure or public cloud providers (e.g AWS or Equinix). This new dynamic and agile approach for the testing and packaging process of the ONEedge software uses the edge cloud reference infrastructure described in the second version of this deliverable (D4.2. "Infrastructure Report"). This report also includes a detailed list of the tests and extensions implemented to verify the functionality of the software developed during the cycle for each component and software requirement.

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:	WP4. Demo and Operational Infrastructure
Nature:	R: Report
Dissemination Level:	PU: Public
Version:	1.0
Contractual Date of Delivery:	30/9/2021
Actual Date of Delivery:	3/11/2021
Lead Authors:	Vlastimil Holer, Rubén S. Montero and Constantino Vázquez
Authors:	Sergio Betanzos, Ricardo Díaz, Jim Freeman, Christian González, Alejandro Huertas, Shivang Kapoor, Jorge M. Lobo, Jan Orel and Petr Ospaly
Status:	Submitted

Document History

Version	Issue Date	Status[1]	Content and changes
1.0	3/11/2021	Submitted	First final version of the D.4.3 report

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

Executive Summary

This report, delivered at the end of the Third Innovation Cycle (M17-M23), describes in detail the new Continuous Integration infrastructure management process that has been totally redesigned during this cycle to be fully automated, ensuring that it can be deployed from scratch within hours in either local infrastructure or public cloud providers (e.g AWS or Equinix). This new dynamic and agile approach applies the automation and multi-cloud features developed in ONEedge to the testing and packaging process of the ONEedge software itself.

The new Continuous Integration infrastructure management process uses the highly distributed cloud infrastructure for Edge Computing described in deliverable D4.2. "Infrastructure Report". This edge infrastructure is also being used in the actual demonstrations of the project's capabilities in real life situations and to deploy validation cases for different scenarios, described in D4.6. "Deployment of Validation Cases and Demonstrations". The infrastructure has been extended with nodes from the new providers supported during the cycle: Google Compute, Digital Ocean, Vultr, and multiple on-premises locations.

For each software requirement, this report also includes a detailed list of the extensions implemented to verify the functionality of the software developed in ONEedge during the Third Innovation Cycle (M17-M23). Also, for each requirement, we list the verification scenarios that have been addressed and a description of the functionality tested to fulfill the proposed scenarios.

During the Third Innovation Cycle (M17-M23), the project mostly focused on those software requirements needed to achieve the third milestone in M23, which is the base functionality required to meet networking & storage integration, and mostly its release as a standalone managed service (On-demand Edge Cloud Service). The work carried out during this Third Innovation Cycle involved software requirements from components CPNT1, CPNT3, CPNT4 and CPNT5, with a special focus on the completion and integration of all components to release a first version of the On-demand Edge Cloud Platform service (CPNT1) and the deployment and provision of edge infrastructures (CPNT4).

Table of Contents

1. Innovative Agile CI Infrastructure

1.1. Architectural Overview

The infrastructure management process has been totally redesigned during this cycle to be fully automated, ensuring that it can be deployed from scratch within hours in either local infrastructure or public cloud providers (e.g AWS or Equinix). The main components of the architecture are:

- **Jenkins VM**: the server where Jenkins server runs. The installation is fully automated by Ansible playbooks, and the Jenkins configuration is defined in a declarative way using JCasC.[2]
- **Services VM** (usually deployed along Jenkins): This VM provides multiple services that act as requirements of the different Jenkins Pipelines:
  - *Packages*: Once the software packages are built, they are published internally in the Services VM.
  - *HTTP*: An HTTP server is deployed to act as an image repository for deploying the different VMs required for building and testing.
  - *DNS*: A DNS server is deployed to resolve the local infrastructure domain names.
  - Many of the build dependencies are architecture and distribution independent and are built only once before building the packages; this process is also executed in the services VM.

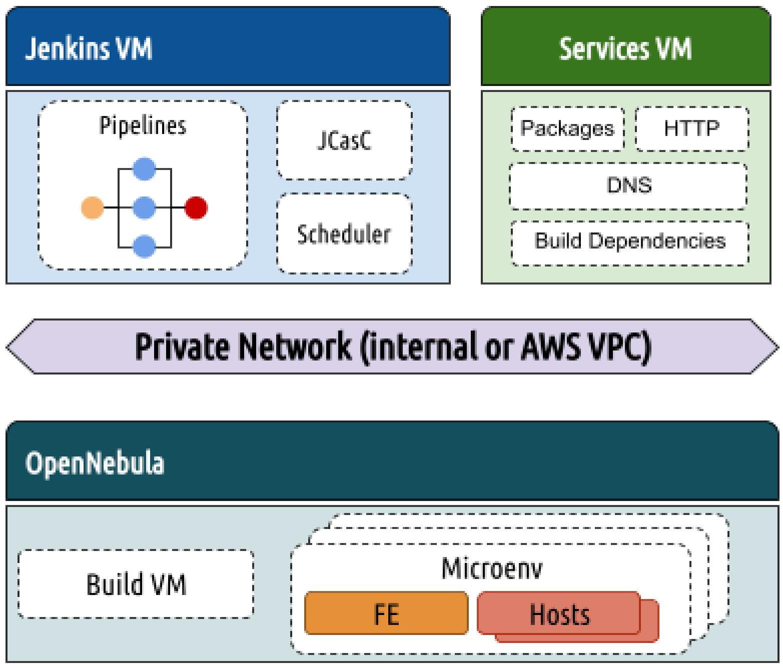

Figure 1.1.1 Overview of the CI infrastructure

- **OpenNebula**: The OpenNebula node is intended to be a self-contained installation (i.e containing both Frontend and hypervisor node) and serves as Jenkins executor provider. The number of concurrent executors (i.e. number of concurrent integration

[2] https://www.jenkins.io/projects/jcasc/

tests) is limited in the Jenkins configuration. Integration tests are structured on microenv, the unit integration unit.

In order to automate the process a set of tools has been developed:

- **Deploy tool**: Automates the deployment and configuration of the entire infrastructure.
- **Microenv deployment tool**: automates deployment of microenvs. A *microenv* is the basic unit for testing. It defines basic OpenNebula environments, typically containing one Frontend and two hypervisor nodes.
- **Build scripts and other supporting tools**: many other minor tools have been deployed and included in the different Pipelines to automate processes such dependencies building, packages building, or test execution.

1.2. Remote On-Demand Deployment

The deploy tool is written in Ruby and it allows the deployment of a new remote (or on-premises) infrastructure from scratch. There are three operations that can be performed, namely: deploy, reconfigure, and delete. In this section we provide a detailed description of each of them.

Deploy

This action will deploy a new infrastructure in the remote provider (AWS or Equinix) or locally using existing infrastructure. The signature of the command is:

```
./deploy <INFRA YAML>
```

Figure 1.2.1: Deploy operation signature

Input files

The deploy command uses two input files: an infrastructure description and a secrets file. Both files are written in YAML format and have the following structure:

- **Infra YAML**: describes the infrastructure that is going to be deployed, the physical servers and the services that are running on them. In this file, there is an entry for each server that is going to be deployed. This entry contains:
  - `Name` of the server (this is the entry itself).
  - `Public IP` of the server, for remote providers this is left blank and the tool will fill the information.
  - `Private IP` of the server, for remote providers this is left blank and the tool will fill the information.
  - `Playbooks`, list of Ansible playbooks that are going to be used to configure the server.
  - `VM type`, this is only used for remote providers and indicates the server specs that are going to be used.
  - `OS`, this is only used for remote providers and indicates the operating system that needs to be installed on the server.
  - `Executor` is a boolean variable that indicates that the server is going to be an executor in Jenkins.
  - `Capacity` works together with executor, so when executor is set to true, this variable indicates the number of available threads in Jenkins.

```
services:
  public: ''
  private: ''
  playbooks:
    - dns
    - http
    - jenkins
    - prebuild-vm
  vm_type: t2.large
  os: ami-0eb471e022a0d8fc6

epsilon:
  public: ''
  private: ''
  playbooks:
    - opennebula
  vm_type: i3.metal
  os: ami-0eb471e022a0d8fc6
  executor: true
  capacity: 50
```

Figure 1.2.2: Example of AWS deploy file

The above file will create two physical servers:

- Services contain DNS server, HTTP server, Jenkins and pre-build vm (this is used for compiling OpenNebula).
- Epsilon contains the OpenNebula that is going to deploy all the virtual machines used for testing.

- Secrets YAML: contains all the secrets (passwords and SSH keys) that are needed to deploy all the services.

Deploy Steps

1. Read environment variables: all the information that the tool needs to connect to the remote provider is provided via ENV variables because this is more secure.
2. Deploy the resources in the remote provider. This is done using Terraform and the information from infra YAML and ENV.
3. Prepare all the information for Ansible: playbooks, inventory, variables, etc.
4. Trigger Ansible command to configure all the servers.

Output files

All the information about the deployment is stored in a folder. This is used to reconfigure the infrastructure, e.g.:

```
example-20211019-152534
├── ansible.cfg
├── deploy.tf
├── fetch
├── group_vars
├── inventory
├── output.16c6984899
├── requirements.yaml
├── roles
├── services.yaml
├── site.yaml
```

```
└── terraform.tfstate
```

Figure 1.2.3: Output directory

The tool also prints a report of the resources that have been deployed, e.g.:

```
===============
Services Report
===============
* VM services with dns,http,jenkins,prebuild-vm running on 147.75.85.17 and 10.80.122.129
* VM epsilon with opennebula running on 147.75.80.33 and 10.80.122.131

All the information about the deployment is in deployments/20211015-173516
```

Figure 1.2.4: Deploy action output report

Reconfigure

This operation reconfigures an existing infrastructure using the files generated by deploy action. The idea of this operation is to be able to add new changes to an infrastructure without deploying new servers. The signature of the command is:

```
./deploy deployments/<DEPLOY FOLDER>
```

Figure 1.2.5: Reconfigure operation signature

Steps

1. Read all the information for Ansible: playbooks, inventory, variables, etc.
2. Trigger Ansible command to configure all the servers.

Delete

This operation deletes an existing infrastructure using the files generated by deploy action. If the infrastructure is deployed in a remote provider, this will also delete the servers; if it is not, just the deployment information will be deleted. The signature of the command is:

```
./deploy -d deployments/<DEPLOY FOLDER>
```

Figure 1.2.6: Reconfigure operation signature

All the information about the deployments is committed into the private repository that stores the tool and all the Ansible information. This can be used by any member of the team to reconfigure the infrastructure with his/her changes.

1.3. Services

A service is a combination of the files needed by Ansible to configure it and specific variables that the service needs. These services are included in the `infra.yaml` file described in the previous section. The supported services are the following:

- **DNS**: installs a DNS server that is able to resolve the virtual machines' IPs. It also installs all the dependencies needed by OpenNebula to trigger the hook that updates the DNS record. The `site.yaml` is the following:

```
- hosts: dns
  vars_files:
    - '../../secrets.yaml'
  remote_user: root
  roles:
    - dnsmasq     # Install and configure dnsmasq
    - resolv.conf # Update resolv.conf with dnsmasq IP
    - ruby        # Install Ruby
```

Figure 1.3.1: DNS site.yaml

- **HTTP**: installs an Nginx server. This is used to store the resulting build when building OpenNebula and the images that are used for testing. The service, apart from installing the Nginx, downloads all the images for testing. The `site.yaml` is the following:

```
- hosts: http
  vars_files:
    - '../../secrets.yaml'
    - "{{   './group_vars/public.yaml' if on_premises|default('True') == 'True'
       else './group_vars/private.yaml' }}"
  remote_user: root
  roles:
    - resolv.conf  # Update resolv.conf with dnsmasq IP
    - role: nginx  # Install and configure Nginx
      download_images:
        - alpine-testing.qcow2
        - alpine-testing.raw
        - centos7.qcow2
        - fc_fs
        - fc_kernel
        - lxc/lxc-nginx
        - lxc/lxc-qcow2-ext4
        - lxc/lxc-raw-xfs
        - lxd.qcow2
        - lxd-xenial.qcow2
        - lxd_xfs.qcow2
        - ubuntu1604_docker.qcow2
        - windows2012.qcow2

        # context:
        - alma8.qcow2
        - alpine310.qcow2
        - alpine311.qcow2
        - alpine312.qcow2
        - alpine313.qcow2
        - alpine314.qcow2
        - alt9.qcow2
...
        - ubuntu2104.qcow2
        - windows2012.qcow2

    - createrepo_c # Build and install createrepo_c tool
    - iptables
```

Figure 1.3.2: HTTP site.yaml

- **Jenkins:** installs Jenkins and configures everything needed to be able to run OpenNebula tests. It also configures all the pipelines. The `site.yaml` is the following:

```
- hosts: jenkins
  vars_files:
    - '../../secrets.yaml'
  remote_user: root
  roles:
    - resolv.conf # Update resolv.conf with dnsmasq IP
    - jenkins     # Install and configure Jenkins
```

Figure 1.3.3: Jenkins site.yaml

- **OpenNebula**: installs the OpenNebula that is used to deploy all the virtual machines to run the CI tests. It also configures the networking so virtual machines can communicate with each other. Apart from installing the Frontend it also installs the node, so the same server is used for running the VMs. The `site.yaml` is the following:

```
- hosts: opennebula
  vars_files:
    - '../../secrets.yaml'
    - "{{   './group_vars/public.yaml' if on_premises|default('True') == 'True'
       else './group_vars/private.yaml' }}"

  remote_user: root
  roles:
    - resolv.conf                       # Update resolv.conf with dnsmasq IP
    - ansible                           # Install Ansible
    - opennebula-repository             # Add OpenNebula repository
    - role: opennebula-server           # Install OpenNebula packages
      opennebula_server_vmm_kvmm_cpu_model: 'host-passthrough'
      opennebula_server_one_auth_users:
        - { user: oneadmin, home: /var/lib/one }
        - { user: one,      home: /home/one }
    - role: opennebula-node-kvm         # Install OpenNebula node KVM
      opennebula_node_kvm_param_nested: True
    - networking                        # Install ifup/down
    - tuntap                            # Create tap0
    - bridge                            # Create brpub bridge
    - iptables                          # Nat + firewall
    - one-tools                         # Install One Tools
    - bootstrap-opennebula-server       # Create OpenNebula objects
    - infra.one-ssh-keys      # Copy oneadmin key to services one user

- hosts: dns
  remote_user: root
  roles:
    - infra.one-ssh-keys # Copy oneadmin key to services one user
```

Figure 1.3.4: OpenNebula site.yaml

- **Postfix**: installs and configures postfix to be able to send emails from all the servers; this is installed in all the machines, as all of them must be able to send emails. The `site.yaml` is the following:

```
- hosts: postfix
  vars_files:
    - '../../secrets.yaml'
  remote_user: root
  roles:
    - resolv.conf   # Update resolv.conf with dnsmasq IP
    - infra.postfix # Install and configure postfix
```

Figure 1.3.5: Postfix site.yaml

- **Pre-build-VM:** configures the virtual machine that is used to build OpenNebula packages. The `site.yaml` is the following:

```
- hosts: prebuild-vm
  vars_files:
    - '../../secrets.yaml'
  remote_user: root
  roles:
    - resolv.conf # Update resolv.conf with dnsmasq IP
    - init-build  # Install and configures everything for building VM
```

Figure 1.3.6: Pre-build-VM site.yaml

1.4. Microenvs

A microenv is the basic unit for defining a test environment. Its deployment is fully automated and its definition can be fully customized by defining the corresponding set of YAML files. The aim of this flexible definition is to allow the deployment of microenvs with any of the various supported combinations, including configuration of different sets of drivers (e.g. storage, networking, authentication) or different Frontend deployment scenarios (e.g. HA, Federation).

Each microenv definition consists of a folder with the following structure:

```
kvm-ssh
├── bootstrap.yaml
├── defaults.yaml
├── extra.yaml
├── inventory
├── postpare.sh
├── site.yaml
└── tests.yaml
```

Figure 1.4.1: Microenv folder tree

The content of the files listed above is described in the following list:

- `bootstrap.yaml`: Contains the definitions of the resources that are going to be automatically created within the microenv once the deployment is complete.

- `defaults.yaml`: Variables' default values definition.

- `extra.yaml`: defines extra modification to the microenv VMs (e.g. different value of CPU or Memory).

- `inventory`: Similar to an Ansible inventory file, it defines the hosts that are going to be allocated for the microenv, typically one Frontend node and two hypervisor nodes.

- `pre/postpare.sh`: optional scripts that are executed automatically before and after the microenv is configured.
- `site.yaml`: Ansible playbook containing the set of configurations that are required for that microenv.
- `tests.yaml`: The list of tests that are going to be executed within the microenv.

Microenvs are deployed by a specific deployment tool. This tool follows the stages listed below for deploying and configuring the microenvs:

- **Initialization**: the microenv information is automatically gathered from the files defined above and every dependency is installed.
- **Cleaning**: in order to avoid collision and duplicates, the cleaning phase is triggered before deploying the microenv VMs.
- **Deployment**: the microenv resources are allocated. This includes Virtual Network, VM Groups, and VMs.
- **Verification**: once the microenv has been deployed, the verification phase ensures that the microenv VMs are accessible via SSH using the corresponding domain name.
- **Configuration**: as soon as all the resources are available the microenv configuration is triggered. During this phase, the microenv will be configured as defined in (`site.yaml`) and every resource defined in `bootstrap.yaml` will be allocated.
- **Testing** (optional): if the corresponding option is set, once the microenv is ready, the tests defined in `tests.yaml` will be automatically triggered.
- **Cleaning**: depending on the configuration, the microenv will be automatically destroyed right after the tests are finished, or the destroy will be scheduled for a specific time in the future (24 hours later by default).

1.5. Jenkins Pipeline and Configuration as Code

In order to improve the integration tests, a set of Jenkins Pipelines has been developed to automate the building and testing of different OpenNebula products. The main requirements for these Pipelines are:

- **Flexibility**: they should provide enough flexibility to be run for a different set of resources and facilitate the development workflow.
- **Performance**: this should include the minimum requirements to improve performance for both building and microenv deployment.
- **Persistence**: The Pipelines' definition must persist inside the control version software (Git+GitHub) and must be able to be automatically recreated.

Every Pipeline has been developed using a Jenkinsfile and the corresponding code is properly committed and managed by the version control software. These files are automatically taken during the infrastructure deployment and used to automatically create the required Pipelines.

1.5.1 Testing Pipeline

This is the main CI Pipeline. It takes care of running the test for the specified set of microenvs. It will either build the software packages using the specified version of the source code, or use pre-existing packages.

The Pipeline has the following stages:

1. **Checkout**

   During this stage the code with the different tools for automation building and microenv deployment is gathered from the corresponding repository branches (both repositories and branches can be customized).

2. **Pre-build**

   When existing packages are not provided, the build of the source code is triggered automatically. The pre-build stage is triggered before the build stage in order to build the platform and architecture-independent components just once, and then to pass them as artifacts to the build stage.

3. **Build**

   After the pre-build stage, the build phase is triggered. During the build the software is compiled in a compatible environment. This means an environment that has a compatible architecture and platform. Apart from building the code, the software packages are generated for the specified platforms (it can be fine tuned using the Pipeline parameters). As a result of this phase the packages are published within the HTTP server of the Services VM.

4. **Readiness**

   Right after building the packages the specified set of microenvs is deployed within the OpenNebula node. If the corresponding option is selected, tests will be triggered automatically inside the microenvs. If not (depending on the Pipeline parameters) the microenv will be left in the OpenNebula node to be used manually.

5. **Summary**

   During this stage the Pipeline execution results are gathered. These contain:

   - Tests results (if triggered)
   - Build parameters used
   - Microenv status (in case the deployment failed)

   This information is formatted and sent via email to the email specified as a Pipeline parameter. Also, when configured, a notification will reach a predefined Slack channel referencing the testing results.

1.5.2 Context Pipeline

One of the main components of OpenNebula is the contextualization packages. These packages are meant to be installed in the images used for deploying OpenNebula VMs and they provide a wide range of configuration parameters for configuring the VMs on boot time. As contextualization packages provide a large number of features and are supported for a wide range of OS/Distributions and different versions of the same, the tests for the contextualization packages are executed by a specific Pipeline that adds a set of performance improvements for each individual case.

Specifically, instead of deploying the microenv and running the test sequentially (as done with the test executed by the Testing Pipeline), just one microenv is deployed for each hypervisor, and multiple images are tested concurrently within the same microenv:

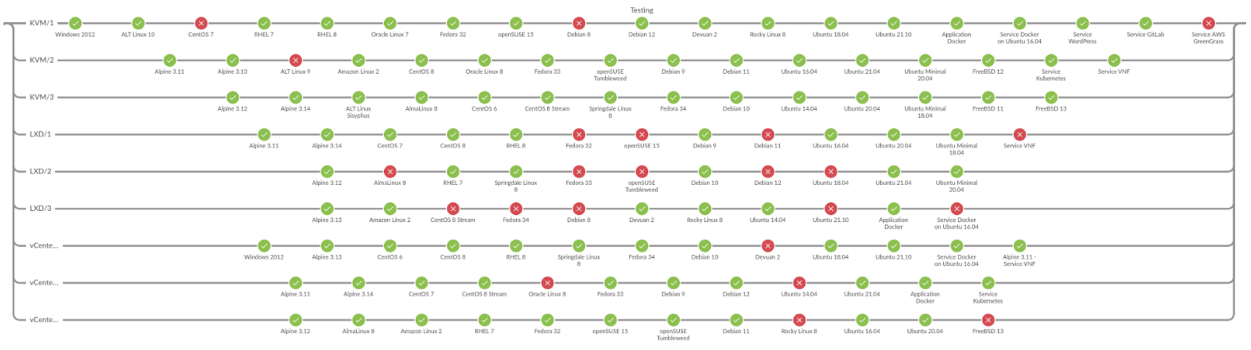

Figure 1.5.1: Context Pipeline concurrent images testing

As opposed to the Testing Pipeline, the Context Pipeline requires pre-existing packages to be passed as parameters. The Context Pipeline has the following stages:

1. **Checkout**

   During this stage the code with the different tools for automation building and microenv deployment is gathered from the corresponding repository branches (both repositories and branches can be customized).

2. **Deploy**

   At this stage, one microenv for each of the supported hypervisors is deployed using the same automation tool as Testing Pipeline.

3. **Testing**

   Once the microenvs are deployed, three images are tested concurrently inside each of the microenvs (three different threads can be found for each hypervisor in Figure 1.5.1). The Pipeline logic controls the number of concurrent tests and uses a thread-safe queue for storing the pending tasks.

4. **Summary**

   After every test is finished, the results for each of them are gathered and a report is generated. This report is sent via email to the email passed as parameter when triggering the Pipeline.

2. Software Requirements Verification

This section includes a detailed list of the extensions implemented to verify the functionality of the software developed in ONEedge during the Third Innovation Cycle (M17-M23). Tests and extensions of the verification framework are detailed for each component and grouped for each software requirement implemented during the cycle. For each requirement we include a summary of the extensions performed in the testing and certification infrastructure. Also, for each requirement, we list the verification scenarios that have been addressed and a description of the functionality tested to fulfill the proposed scenarios.

2.1. Edge Instance Manager (CPNT1)

SR1.1. Simple Product Deployment

Status: DONE

Description: We have extended the previous tests that verify the containerized installation to ONEedge. The process has been integrated in a new test suite that verifies the automatic installation and configuration of a ONEedge hosted environment using the *production* playbooks.

Verification Scenarios:

- [VS1.1.3] Automatically configure and install a Virtual Machine with the ONEedge hosted bundle. Performs basic functionality tests to verify that the environment has been properly installed and it is fully operational.

SR1.3. Instance Management

Status: DONE

Description: Testing has been extended to verify the life cycle of ONEedge hosted instances using AWS cloud.

Verification Scenarios:

- [VS1.3.2] Creates a new hosted environment on AWS cloud. Check that integration with ancillary services is correct (DNS).
- [VS1.3.3] Destroy a running hosted environment. Check that no AWS resources are left in the cloud (including security groups, vpc and gateways).

SR1.4. Subscription Management

Status: IN PROGRESS

Description: Tests include several checks to verify the updates on the customer portal.

Verification Scenarios:

- [VS1.4.1] Test to verify that customers are automatically created in the support portal when a new environment is created. Tests also check that labels are correctly set.

- [VS1.4.2] Test to verify that users are removed from Zendesk when the ONEedge hosted environment is destroyed.

SR1.5. Web Control Interface (GUI)
Status: DONE
Description: The main interface to access the hosted environment is GitHub web portal. All the ONEedge hosted operations can be triggered and queried through Github using common operations like issue creation.
Verification Scenarios: • [VS1.5.1] All the verification scenarios described above (especially VS1.3.2 and VS1.3.3) use the Github API that resembles the GUI interaction offered by the GitHub interface. In this way the interface used by the team is regularly tested.

2.2. Edge Workload Orchestration and Management (CPNT2)

No activity done during the cycle.

2.3. Edge Provider Selection (CPNT3)

SR3.1. Edge Provider Catalog Service
Status: DONE
Description: Q&A tests that covered the addition/modification and removal of providers have been extended to exercise the dynamic load of providers by the FireEdge server, relevant for the OneProvision GUI and framework.
Verification Scenarios: ● [VS3.1.2] Dynamic load of new providers is exercised through specific tests that inject a new provider type in an existing OneProvision installation and execute the existing test battery over the newly added provider. One of the existing providers (Amazon AWS) is removed and added afterwards to cover this functionality.

SR3.4 Driver Maintenance Process
Status: DONE
Description: A dummy driver made following the new provider Development Guide is added to the Q&A procedures, in which a set of tests used to exercise the whole automatic provision procedure in ONEedge is applied to it in each iteration.
Verification Scenarios: ● [VS3.4.4] A dummy driver crafted following the guidelines on the Development Guide passes the mentioned driver maintenance process tests.

2.4. Edge Infrastructure Provision and Deployment (CPNT4)

SR4.4. Inter-edge Networking Deployment Scenario

Status: DONE

Description: Communication between application or application components across edge locations is performed through public IPs. The test suite verifies the proper use and allocation of these IPs or the sharing of node IPs using port-forwarding.

Verification Scenarios:

- [VS4.4.1] A new public IP can be attached to a running VM in an edge node. The VM automatically configures the IP to make use of it. The test checks L3 and L4 connectivity with basic tools (ping and netcat).
- [VS4.4.2] A port range is successfully allocated and forwarded from the node. The tests check that the VM can be accessed through the port range (L4 connectivity) using basic tools (netcat).
- [VS4.4.3] A VM deployed in an edge location can access external services. The test tries to access well known services (public DNS servers).

SR4.5. Drivers for Host Provision

Status: DONE

Description: ONEedge supports multiple edge/cloud providers. The tests in SR4.5 exercise the APIs of these providers to verify the correct integration into ONEedge product.

Verification Scenarios:

- [VS4.5.1.] The existing tests in this verification scenario have been extended to consider the new providers. The new tests verify the proper allocation of the provider resources and the cleanup once the provision has been destroyed.

SR4.9. Support on-premises far-edge for resource provisioning

Status: DONE

Description: The ability to integrate data center and edge resources has been a recurrent request from early adopters. In this cycle this functionality has been incorporated into ONEedge. The test suite verifies the creation and configuration of *on-premises* provisions.

Verification Scenarios:

- [VS4.9.1.] Create a provision out of a pre-configured (minimal OS installation and SSH access) pool of hosts. The test verifies the correct installation of the provision elements and its operational status by creating simple VM workloads.

SR4.10. Support ARM for resource provisioning
Status: DONE
Description: The support for ARM based provisions has been included in this cycle (aarch64). This enables us to create provisions using ARM instances (usually cheaper). The verification scenario validates the ARM packages and their dependencies.
Verification Scenarios: • [VS4.10.1.] Create a provision using ARM instances. The tests then perform basic operations (create a VM, verify its network connectivity) to validate the ARM distribution of FireEdge. • [VS4.10.2] Perform and verify a Frontend installation using ARM architecture. This test includes a subset of the full suite run on x86_64 architecture. The goal is to perform a verification of the Frontend AMD packages.

2.5. Edge Apps Marketplace (CPNT5)

SR5.2 Built-in Management of Application Containers Engine
Status: IN PROGRESS
Description: The existing set of tests used to cover the Kubernetes and K3s appliances has been extended to cover the new MetalLB.
Verification Scenarios: ● [VS5.2.4] A new set of tests to exercise the MetalLB (Load Balancer) functionality in Kubernetes has been added to the Q&A process, where it is ensured that outside requests are forwarded to different nodes sequentially.

SR5.5. Edge Market GUI Developments
Status: IN PROGRESS
Description: The new Sunstone is being rewritten but the same functionality as the existing OpenNebula GUI needs to be maintained. The current set of tests that covers all the interface functionality therefore needs to be rewritten from RSpec to Cypress, in order to use a more up to date web testing framework.
Verification Scenarios: ● [VS5.5.3] The subset of RSpec tests—part of the current OpenNebula Q&A process—that covers the Virtual Machine and Virtual Machine functionality has been rewritten using the Cypress web interface testing framework and added to the regular end-to-end testing for the new Sunstone Beta component.