H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable 5.2

Runtime Environment 2

Version: 1.0

Publication Date: 31-October-2020

Deliverable Card

Deliverable	5.2
Title:	Runtime Environment 2
Editor(s):	Matija Cankar (XLAB)
Contributor(s):	Anestis Sidiropoulos (ATC), Hans Georg Næsheim (PRQ), Mainak Adhikari (UTR), Anže Luzar, Špela Dragan, Matija Cankar (XLAB), Vladimir Yussupov (UST)
Reviewers:	Mark Law (IMP), Michael Wurster (UST)
Type:	Report
Version:	1.0
Date:	31-October-2020
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/
Copyright:	The RADON project partners

The RADON project partners

IMP	IMPERIAL COLLEGE OF SCIENCE TECHNOLOGY AND MEDICINE
TJD	STICHTING KATHOLIEKE UNIVERSITEIT BRABANT
UTR	TARTU ULIKOOL
XLB	XLAB RAZVOJ PROGRAMSKE OPREME IN SVETOVANJE DOO
ATC	ATHENS TECHNOLOGY CENTER ANONYMI BIOMICHANIKI EMPORIKI KAI TECHNIKI ETAIREIA EFARMOGON YPSILIS TECHNOLOGIAS
ENG	ENGINEERING - INGEGNERIA INFORMATICA SPA
UST	UNIVERSITAET STUTTGART
PRQ	PRAQMA A/S

The RADON project (January 2019 - June 2021) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825040

Executive summary

This is the second and final Runtime Environment deliverable that provides a technical overview of the runtime environment and delivery toolchain tools integrated in the RADON framework.

The document provides a high-level integration diagram first, followed by the detailed description of each tool by means of functionality that provides the integration status, technology readiness level (TRL) and example or future work. Within the last sections of the deliverable, the complex workflow examples of the RADON delivery toolchain and Runtime environment usage are presented. The document concludes with the requirement status check and future work.

Glossary

Term	Meaning
API	Application Programming Interface
AWS	Amazon Web Services
CI / CD	Continuous Integration / Continuous Delivery
CLI	Command Line Interface
CSAR	Cloud Service Archive
CTT	Continuous Testing Tool
EC2	Elastic Compute Cloud
FaaS	Function as a Service
GMT	Graphical Modelling Tool
GUI	Graphical User Interface
IDE	Integrated Development Environment
REST	Representational State Transfer
SaaS	Software as a Service
VM	Virtual Machine
Yn	Year n of the project.

Table of contents

1 Introduction

This is the second and final Runtime environment deliverable. While in the first deliverable we put an emphasis on the presentation of each tool and its basic functionality, with this second one we put an emphasis on the way the tool is integrated in RADON and how it is used to provide an end-to-end experience.

In this sense, the deliverable will provide an overall presentation of the tools including the main purpose of the tool in RADON tool chain, links to the tool documentation and usage examples, and finally we will present the examples of different approaches of employing the tools for more complex deployments including CI/CD jobs and runtime scaling.

1.1 Deliverable objectives

The main objective of the deliverable is to present the progress of the delivery tools and runtime environment during the Y2. This forms the following sub-objectives:

- Present the overall workflow of the delivery toolchain
- For each delivery or runtime tool present:
    - The final functionality set of the tool included in the delivery toolchain or runtime. Any progress of the tool improvements is depicted.
    - The integration status of the tool and how the tool is integrated with IDE
    - The maturity level of each component.
    - Example(s) or links to the user manuals.
- Populate the WP5 requirements table with achieved levels of compliance.
- Present usage examples for complex tasks as CI/CD job or scaling.

1.2 Overview of all achievements

The main achievement of this deliverable is presenting the progress of the tools in Y2 and the ability to use them together to achieve the deployment goals of different roles or stakeholders, e.g., a developer, who needs fast development and continuous testing, a testing officer, who requires isolated deployment for testing purposes, a product owner, who approves the deployment of a new version. Among others, there is an important role of the DevOps engineer, where it should be possible that a DevOps team can overtake the procedures of application delivery and sustain an application lifecycle.

This deliverable contains the achievements of tool owners accomplished by integration of the tools under the same environment and provides a more user friendly experience for the multi-cloud users and FaaS enthusiasts.

1.3 Structure of the document

Section 2 updates the status of the Runtime environment and the high level architecture. At the end of the section, a workflow that includes RADON runtime environment tools integration is presented.

Section 3 gives a tool overview and presents each tool integrated in the Runtime environment. It explains how each tool is used, its final functionalities, its current integration status, the level of maturity that has been achieved, the technical skills required to manage the tool and how the tool can be tested in action on a particular given example.

Section 4 presents Runtime management operations. This section includes the proposed scaling operation overview with an example. It also thoroughly describes the Blue/Green deployment and Canary testing examples. Finally, the section explains how the security for the RADON tool is maintained and ensured.

Section 5 concludes the document and gives the perspectives and motivations for the work that is intended to be done in the future.

2 . Runtime environment - status of the Year 2

In this section, we highlight the updated status of the runtime environment of the RADON framework. A preliminary presentation of the runtime tools including their interfaces, installation, maturity level and the toy example demo has been described in D5.1 [RadD5.1]. The main focus of this section is to continue the description of the runtime tools by highlighting their basic principles and how the updated versions of the tools interact with each other in order to meet the objectives of the project.

2.2 High level Architecture

During the second year, the runtime environment and the delivery toolchain became an integral part of the RADON framework environment. From the ideas and plans composed in the WP2 and the functionalities available and planned from the D5.1, we managed to integrate all tools so that they work together. As Figure 1 presents, the central piece is the RADON IDE, an Eclipse Che instance, which uses plugins developed in RADON to connect to each tool and make it available from the central IDE.

RADON users employ the RADON IDE to develop an application and also develop Infrastructure as Code (IaC) to deploy the application. In this deliverable we focus only on the tools for IaC code management and execution. When a user has finished the application topology with the RADON GMT tool, they can take the following actions (the reader might want to follow the Figure 1):

- Export the application topology from GMT to the RADON environment, i.e., CSAR generation.
- Download the FaaS scripts from Function Hub when the CSAR is created.
- Publish the CSAR version to the Template Library (multiple options) - this step is required only if the user would like to share application CSAR with others.
- Deploy the CSAR immediately using the Che plugin
- Use the SaaS orchestrator to manage deployment credentials, workspaces and projects (CSAR, outputs,etc). This tool can be used for monitoring the deployment process.
- Connect to the monitoring and retrieve the runtime application data.
- Use CI/CD to enable more complex deployment and testing scenarios.

The integration is achieved in various ways. A subset of the tools run as standalone tools (as Prometheus, Orchestrator, TPS) and the RADON IDE accesses them through Eclipse Che plugins. Another subset of tools can be installed in Eclipse Che in the bash console using the pip command

(Function hub, TPS).Finally, some tools create their own container inside the IDE environment to expose their functionality to the user.

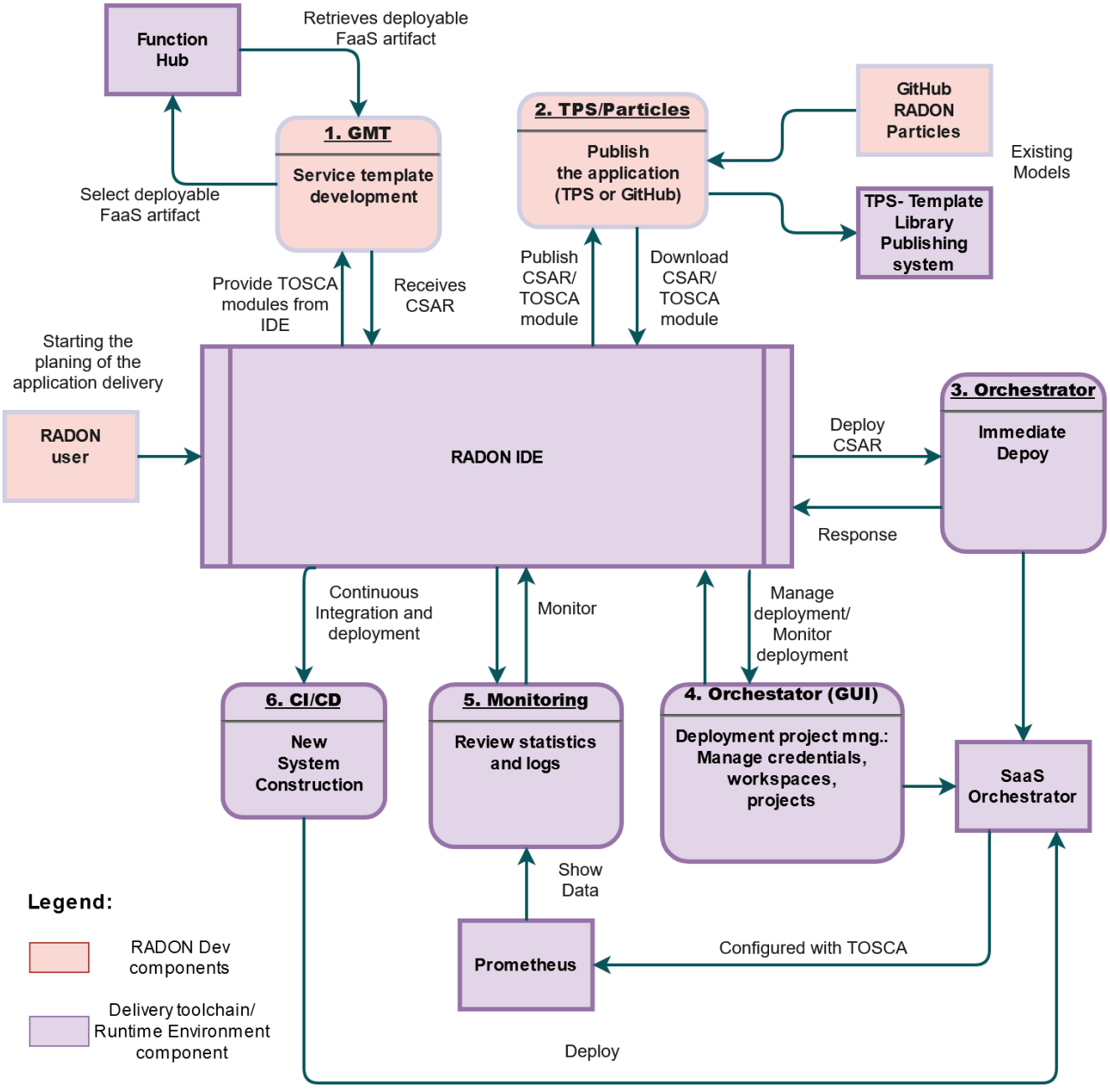


Figure 1. RADON runtime environment

The standalone tools (RADON Monitoring, SaaS orchestrator and TPS) share support for OpenID, meaning that the user can use RADON credentials to access them, which makes integration and usage seamless for the end user. Here, we conclude this short overview of runtime environment and

delivery toolchain and invite the reader to find more detailed descriptions of Y2 improvements of each tool in the following section.

3 Tools overview

The initial presentation of the tools was already done in D5.1 where the reader can find the initial tool descriptions, individual installation instructions, maturity level and the toy example demo, where this was possible. This deliverable continues the description with a focus on the integration point of each tool inside the RADON environment, a maturity level update and usage demo that is sometimes equipped with the screenshots and instructions or it invites the reader to explore the content already published on the web in tool manuals or in the tool GitHub pages.

The updates are mainly the result of day-to-day work with partners to achieve the best possible integration between the tools and desired functionality expressed by the use-case providers.

3.1 Monitoring

The Monitoring tool addresses the need for adding runtime monitoring capabilities to applications modeled in TOSCA. It supports monitoring on a FaaS level (cloud functions), on a container level as well as on an OS level. It provides feedback with regard to the resource usage and performance characteristics of the components that compose an application topology modeled as TOSCA blueprint.

3.1.1 Final tool functionality set

The final functionality set of the Monitoring tool includes:

- dynamic configuration of monitoring components (nodeExported, pushGateway) to a Prometheus server (auto-discovered monitoring endpoints using Consul service discovery)
- TOSCA node and relationship type definitions for monitoring components
- FaaS monitoring libraries to avoid manually injected code
- Integration with alert manager to support auto scaling

3.1.2 Integration status

Since the monitoring solution is based on the Prometheus ecosystem, an open-source monitoring and alerting toolkit, the idea is to wrap the deployment and configuration of the various Prometheus components (PushGateway, NodeExporter, cAdvisor) as TOSCA nodes. To this respect, the monitoring TOSCA nodetypes definitions have been integrated in the Radon particles and tested that are compatible with the xOpera orchestrator.

Monitoring a FaaS in the context of RADON implies that a Prometheus PushGateway component is attached to the FaaS to be monitored. This way, the FaaS pushes its metrics to the PushGateway and the PushGateway then exposes the metrics to the Prometheus server. The PushGateway target endpoints are dynamically discovered by the Prometheus server using a service-discovery mechanism based on Consul, an open-source service networking component. The dynamic

registration of a PushGateway node to a Consul instance is part of the TOSCA PushGateway nodetype definition.

To accomplish the workflow described, new TOSCA relationship type definitions have been implemented, specifically the AWSisMonitoredBy and the GCPisMonitoredBy in radon.relationships.monitoring. The former is used to allow monitoring the AWS implementation of FaaS namely the Lambda, and the latter to allow monitoring the GCP implementation of FaaS namely the CloudFunction. To this respect, the RADON TOSCA nodetype definitions of Lambdas (radon.nodes.aws.AwsLambdaFunctionFromS3 and radon.nodes.aws.AwsLambdaFunction) as well as the nodetype definitions of CloudFunctions (radon.nodes.google.GoogleCloudFunction and radon.nodes.google.GoogleCloudBucketTriggeredFunction) have been updated to allow the monitoring relationships. Based on the code snippet below, a new requirement is added to the cloud functions TOSCA definitions. This requirement implements the connection of a cloud function with the PushGateway monitoring node which has a newly introduced Monitor capability type. Hence a AWSIsMonitored/GCPIsMonitoredBy requirement is linked with a monitor capability type. The occurrences are unbounded, indicating that it is possible to monitor more than one cloud function in the same Pushgateway node. The monitoring stack is implemented in a way where the relationship alone is sufficient for it to be functional. No function name provision is needed as an extra step. If the relationship exists the function is monitored. In addition the Pushgateway node is agnostic of the cloud function vendor. Hence, the definition of the TOSCA monitoring nodes preserves the modularity of the RADON particles and thus can be applied in a similar way to any node of FaaS in the same cloud vendor family, as an example:

```
requirements:
  - monitor:
      capability: radon.capabilities.monitoring.Monitor
      node: radon.nodes.monitoring.PushGateway
      relationship: radon.relationships.monitoring.AWSIsMonitoredBy
      occurrences: [ 0, UNBOUNDED ]
```

A shared instance of the monitoring visualization dashboard is available from within the RADON IDE. The Grafana dashboard endpoint has been registered as a client in the RADON KeyCloak, the RADON IDE's access management component. The monitoring dashboard is configured with OAuth authentication (see figure 2) making the access delegation to the RADON KeyCloak possible. This way, a RADON user can access the monitoring dashboard through the RADON IDE with the same credentials, preserving a common user management.

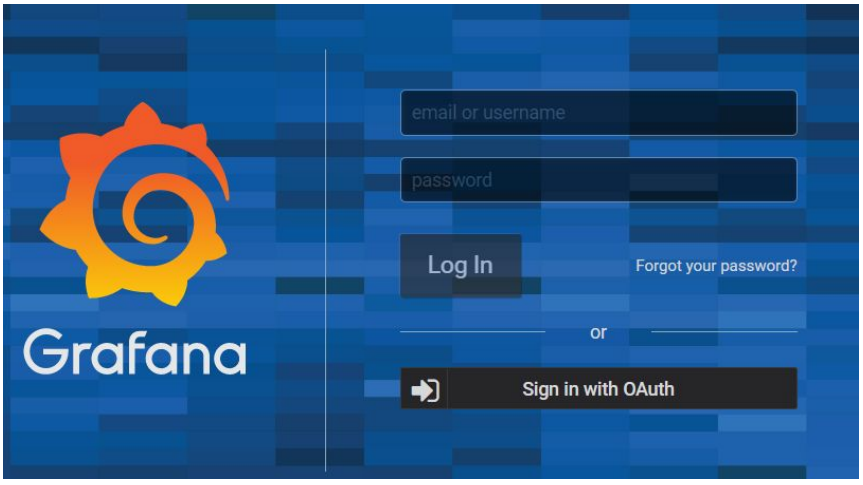


Figure 2. Grafana login view

The (AWS/GCP) isMonitoredBy TOSCA relationship type triggers the configuration of a monitoring dashboard in the shared Grafana instance. A RADON user is granted permissions to monitor his/her own resources only.

3.1.3 Achieved maturity level

The monitoring TOSCA nodes have been recently integrated into the RADON Particles and manual testing conducted by validating the compliance to the latest xOpera orchestrator runtime. As it has not been published and validated for a long enough period the estimated TRL is 3-4.

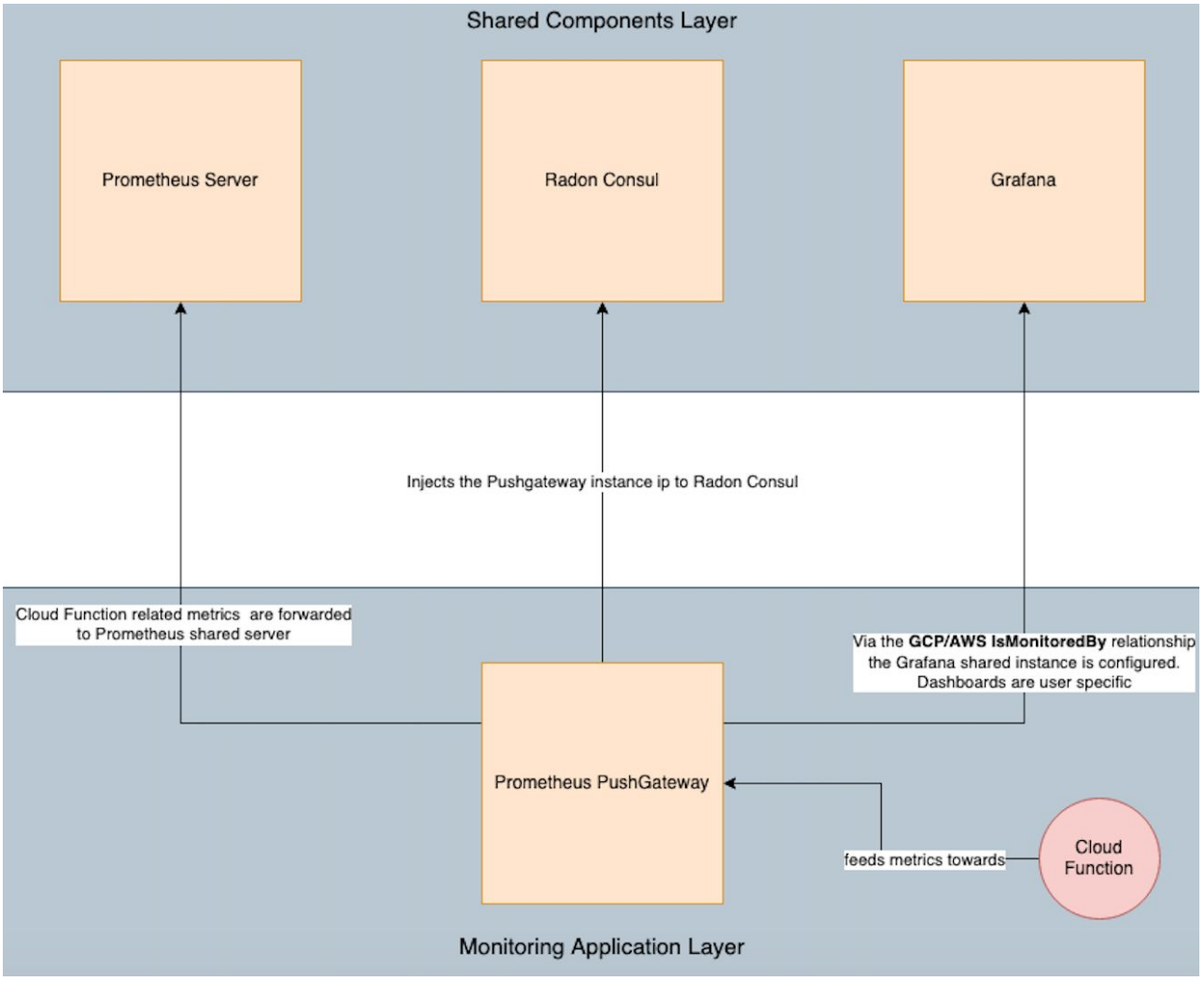
3.1.4 Usage example/demo

The following example implements the dynamic deployment of a Python based S3 triggered Lambda function and the whole monitoring stack of this function. The function has been code injected in order to expose specific metrics (RAM, CPU usage) The main nodes needed for this example are:

- EC2 VM
- Pushgateway
- AWSLambda
- AWSS3

After the successful deployment of this example the user can trigger the function by uploading a .json document to the bucket. When the document is added the Lambda function is triggered and immediately the code injected in the function exposes the collected metrics to the Prometheus Pushgateway node. Subsequently, Pushgateway propagates the metrics to Prometheus and later on on the relevant dashboards in Grafana. This process is taking place under the scenes. The user adds the file in the bucket and then collects the metrics in the Grafana account.

The interface between the monitoring nodes and the shared components of the mentoring stack as described below can be visualized as follows:



The steps for this example are:

1. Include the monitoring library in the FaaS deployment package https://github.com/radon-h2020/radon-monitoring-tool/blob/master/lambda/python-runtime/monitoring_lib.py, and annotate the function handler with the monitoring metrics, e.g.

   @monitor_ram

   @monitor_cpu

2. Use the GMT tool to model the topology along with the PushGateway stack (AWS EC2, DockerEngine and PushGateway TOSCA nodes).
   a. Connect AwsLambda to PushGateway node using the AWSisMonitoredBy relationship. Draw a requirement from AwsLambda node to the monitor capability of PushGateway node, as it is shown in figure 3. Connecting multiple FaaS instances to a single TOSCA PushGateway node is encouraged and allowed by design (the

relationship is unbounded). Nevertheless, according to Pushgateway official documentation, "*when monitoring multiple instances through a single Pushgateway, the Pushgateway becomes both a single point of failure and a potential bottleneck*". A way to avoid this situation when multiple functions are monitored through a single Pushgateway node is to deploy Pushgateway on a VM with sufficient resources.

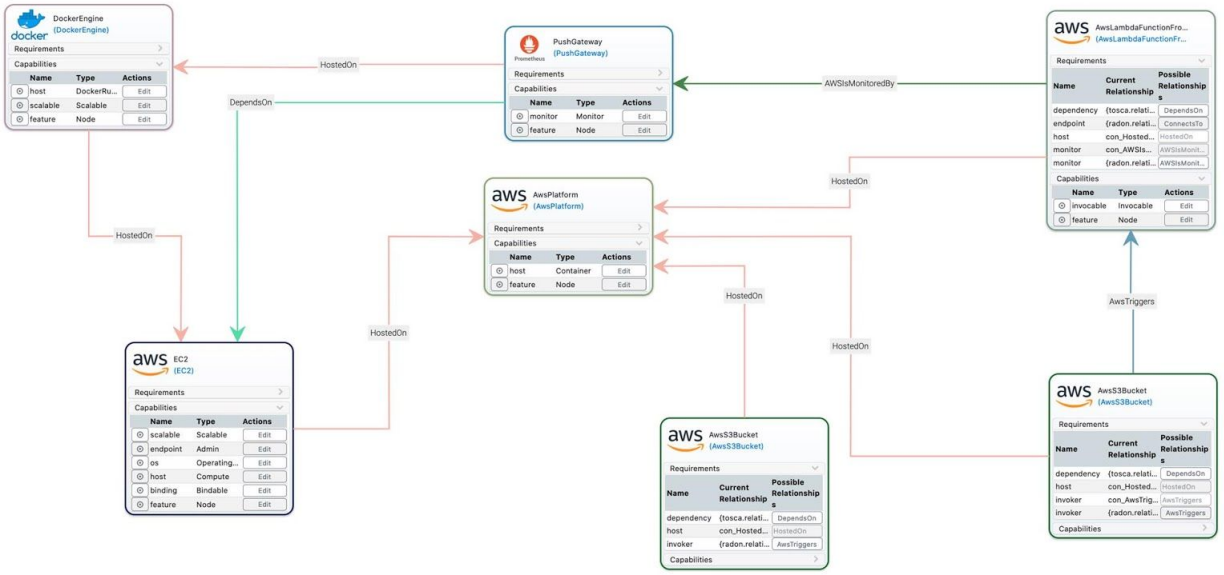


Figure 3. Topology model created with GMT

b. Fill in the PushGateway node properties. A Consul service IP and port is required. The Consul service allows new pushgateway endpoints to be dynamically discovered by a Prometheus server. RADON users can use the RADON Consul shared instance http://3.127.254.144:8500. The port where the push-gateway instance will be listening for connections is also required. The user_email property refers to the RADON user's email as it is registered in the RADON IDE KeyCloak. The grafana_api_ip property points to a Grafana API instance, an utility service of the monitoring solution to support multi-tenancy and restrict each RADON user in being able to monitor its own resources only. It preconfigures Grafana dashboards in the RADON shared Grafana instance. The RADON Grafana API shared instance is http://3.127.254.144:3100/.

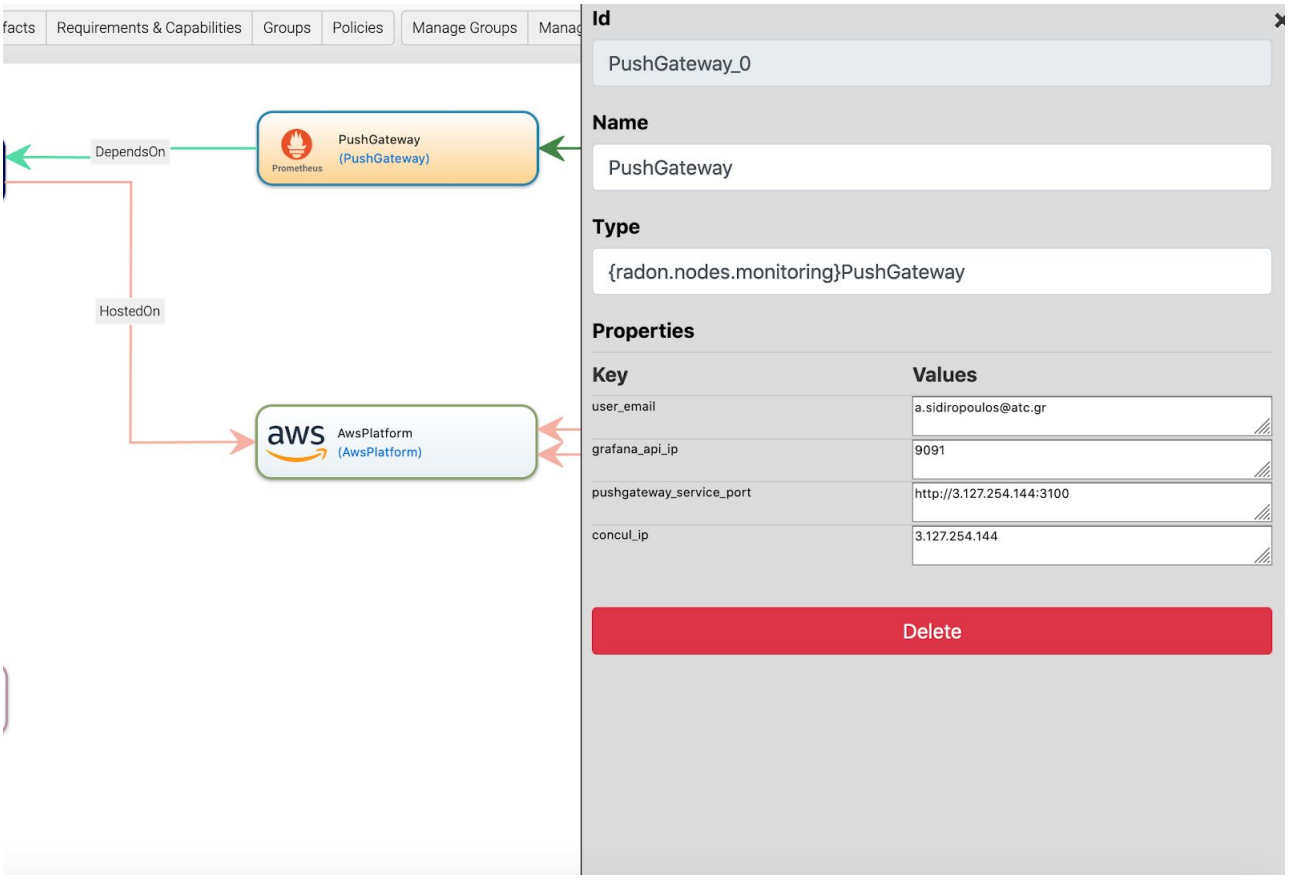


Figure 4. PushGateway node properties.

3. Deploy the generated service blueprint with xOpera orchestrator.
4. Access the monitoring dashboard through the RADON IDE. A preconfigured metrics dashboard should be available for each FaaS connected to the PushGateway node.

Figure 4. Monitoring dashboard

3.2 Template Library

The Template Library is the place to store the TOSCA particles, corresponding Ansible playbooks and TOSCA CSARs describing a particular application. This section will present the high level overview of the integration of the Template Library with current tools, while a more detailed description of the content will be in *D5.4: Technology Library II*. To understand the integrational approaches of the Template Library, note that Template Library was divided to support two different online repositories. One is community based on GitHub and we refer to it as RADON Particles. The second one is Template Library Publishing Service (TPS), which holds published TOSCA content that can be publicly available or closed. This division was introduced in *D5.3: Technology Library I* [RadD5.3].

3.2.1 Final tool functionality set

The purpose of the Template Library is to simplify and make use of the TOSCA particles, corresponding Ansible playbooks and TOSCA CSARs easier and more accessible. It enables sharing different versions of content which simplifies the complexity of managing multiple versions and shortens time required to deliver improvements to content. Templates can be joined by some criteria in template groups to make private templates accessible to members of user groups with access to corresponding template groups.

3.2.2 Integration status

Template Library in the form of public repository RADON Particles is integrated in RADON IDE as an Eclipse Che workspace project and is included via GitHub link in its dev file.

TPS is integrated in RADON IDE as a plugin for workspace. It can also be used as a CLI client in the IDE terminal. The RADON user should pay attention to use the same TPS REST API endpoint everywhere. Only then all the information will be stored to the same database.

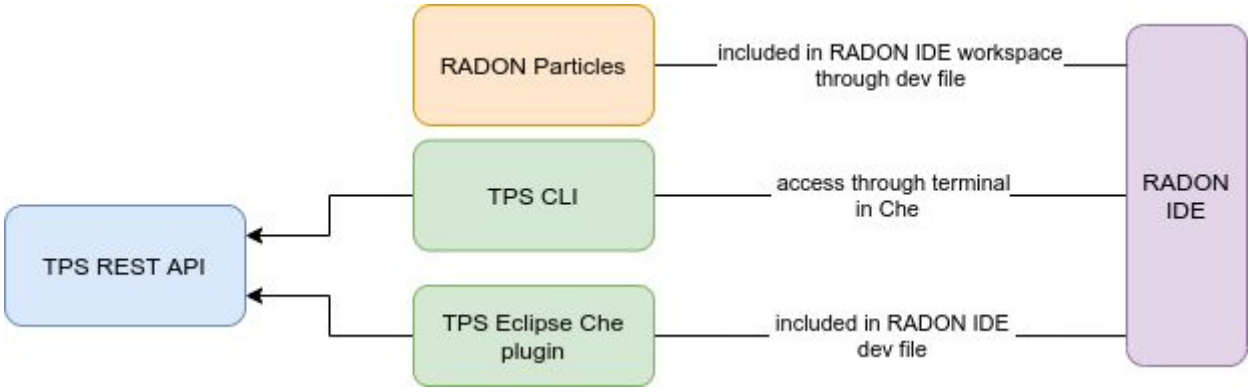


Figure 5. Integration of Template Library in RADON IDE

RADON particles

RADON Particles is a community based repository on GitHub. Consequently it is easily integrated to RADON IDE as it uses GitHub. As such, it was available in early stages of the project.

TPS (CLI client)

Template Library CLI is a client tool that uses the TPS REST API for managing templates, blueprints and their versions. It simplifies the use of Template Library REST API from the command line. An OpenAPI specification to describe REST API endpoints is available here: https://template-library-radon.xlab.si/swagger/

CLI client is a Python pip package, which can be installed with `pip install xopera-template-library`. After installation, an endpoint must be configured by calling setup. Login is enabled for native users and KeyCloak users. Endpoint and user credentials are saved to `.config/template_library/`. User session for native users is limited to 10h and for Keycloak users a session is closed after 15min of the last call to the API.

Template Library CLI supports many actions available through API calls. Most important are publishing and retrieving different versions of templates and blueprints. Users can also create and list template groups and user groups, view templates in template groups and users in user groups. User groups can be granted access to template groups and private templates can be viewed by user group members.

Last published version to PyPI is 0.2.0.

TPS (Che - console)

Template Library CLI can also be used from Eclipse Che terminal. As Che is not meant for such usage, a few steps are required for CLI to work. They are explained in Section 3.2.4.

TPS (Che - plugin)

The Template Library plugin for RADON IDE (Eclipse Che) v0.0.2 uses an open-source cloud and desktop IDE framework called Eclipse Theia, which is very similar to VS Code, therefore VS Code extensions can also be used in Theia. The extension uses Template Library REST API and can invoke main Template Library actions.

Supported are:

- setting Template Library REST API endpoint
- creating and publishing TOSCA template or CSAR and its version
- downloading a specific template version files

The plugin is located on GitHub:
https://github.com/radon-h2020/radon-template-library-publishing-service-plugin

A RADON user can create a new workspace in Che using `devfile.yaml` in `/publishing-samples` available on the GitHub link above.

The plugin is invoked by right clicking on the file in the editor. First an endpoint needs to be configured, then a TOSCA template or CSAR and its version can be published or a specific template`s version files can be downloaded.

A JSON config file of exact structure to execute supported actions can also be used.

3.2.3 Achieved maturity level

The CLI version of the Template Library has been used and manually tested for a few months now and has been improved to be more intuitive and user friendly. The estimated maturity level of the CLI version of the Template Library is TRL 4.

The Template Library plugin for RADON IDE (Eclipse Che) has been just recently developed and integrated with the other tools. The Template Library API it uses is stable, tested and improved with the development of Template Library CLI. As it has not been published for long, the current estimation of TRL is 3-4.

3.2.4 Usage example/demo

The Template Library API enables use of the same data library in multiple ways.

a) **CLI client** (v0.2.0) uses Template Library API (v.0.5.0).

   After installing the library with `pip install xopera-template-library` the setup command to set the endpoint for API should be used.

   All the commands where data needs to be input can be executed in two ways. It can be passed in with the initial command or left out and input on prompt. This is most useful for entering user credentials. Otherwise, it comes in handy when a user doesn't know all the required parameters.

```
radon:~$ xopera-template-library login --username xlab --password ItStaysInTerminalHistory
Wrong password for user!
radon:~$ xopera-template-library login
Username: xlab
Password:
User 'xlab' logged in.
```

Figure 6. Two ways to input the data in CLI

   Currently, commands for template actions are separated to entity templates, those are TOSCA particles and Ansible playbooks, and service templates or TOSCA CSARs. Both template types have the same actions with different requirements e.g. for publishing a service template required format is a zip or a tar and for entity templates, there is a template file, implementation files and a readme file.

   Most important commands are for publishing and retrieving templates of different versions.

   A draft for a new template can be initialized with `xopera-template-library entity-template create`. A template name and a template type must be provided.

When the files are edited, a new template can be published using the `save` command. For publishing a template the following data must be provided: template name, description, path where the template is located, template type, version and if the template is to be published publicly a `--public` argument. If it is not set, the template is considered private.

```
radon:~$ xopera-template-library entity-template save
Path to template directory: ExampleTemplate
Valid template types:data, artifact, capability, requirement, relationship, interface, node, group, policy, other
Could not extract valid template type. Please provide template type: node
Template name: ExampleTemplate
Description of the template: Example of a template
Template has been added successfully!
Version name of the template to upload (e.g. 1.3.0): 0.0.1
Version insertion with implementation was successful!
```

Figure 7. Publishing a template in CLI

For publishing a new version, save the template as if it was a new template. Template name must be the same, description and type are not important. This will be improved in future versions.

For downloading a template, its name must be provided. By default the last version is retrieved. Another version can be retrieved when specifying `--version` argument.

```
radon:~$ xopera-template-library entity-template list
+List of entity templates---------+-------------------------+------+--------+------------+---------------------+
| ID | Name                       | Description             | Type | Public | Created by | Created at          |
+----+----------------------------+-------------------------+------+--------+------------+---------------------+
| 2  | AwsBucket                  | radon.nodes             | node | True   | xlab       | 2020-05-29 11:51:12 |
| 3  | AwsLambda                  | radon.nodes.function    | node | True   | xlab       | 2020-05-29 11:55:50 |
| 4  | AwsBucketNotification      | radon.nodes.triggers    | node | True   | xlab       | 2020-05-29 11:59:03 |
| 5  | AwsRole                    | radon.policies          | node | True   | xlab       | 2020-05-29 12:02:35 |
| 6  | AwsApiGateway              | radon.nodes.triggers    | node | True   | xlab       | 2020-05-29 12:05:32 |
| 7  | AzureContainer             | radon.nodes             | node | True   | xlab       | 2020-06-01 06:36:30 |
| 8  | AzureContainerNotification | radon.nodes             | node | True   | xlab       | 2020-06-01 06:42:14 |
| 9  | AzureFunction              | radon.nodes             | node | True   | xlab       | 2020-06-01 06:51:52 |
| 10 | MinIOBucket                | radon.nodes             | node | True   | xlab       | 2020-06-01 07:44:56 |
| 11 | OpenFaaSFunction           | radon.nodes             | node | True   | xlab       | 2020-06-01 08:01:33 |
| 12 | OpenFaaSFunctionBuild      | radon.nodes             | node | True   | xlab       | 2020-06-01 08:37:16 |
| 25 | GcpBucket                  | radon.nodes.GcpBucket   | node | True   | xlab       | 2020-07-21 10:04:22 |
| 26 | GcpFunction                | radon.nodes.GcpFunction | node | True   | xlab       | 2020-07-21 10:04:22 |
| 40 | vs_code_test               | VS code test            | node | True   | xlab       | 2020-10-08 11:41:20 |
| 41 | vs_code_test1              | VS code test            | node | True   | xlab       | 2020-10-08 11:48:55 |
| 42 | my_vs_code_test            | VS code test            | node | True   | xlab       | 2020-10-08 11:54:15 |
| 43 | my_vscode_super_test       | This is a test          | node | True   | xlab       | 2020-10-08 11:55:56 |
| 1  | TestModel                  | radon.nodes.aws         | node | False  | xlab       | 2020-05-29 11:38:24 |
| 50 | Test                       | Testing                 | node | False  | xlab       | 2020-10-15 20:58:35 |
| 52 | ExampleTemplate            | Example of a template   | node | False  | xlab       | 2020-10-16 07:24:37 |
+----+----------------------------+-------------------------+------+--------+------------+---------------------+
radon:~$ xopera-template-library entity-template version
Template name: TestModel
+List of versions for TestModel--------+---------------------+
| Version name | Template file name    | Created at          |
+--------------+-----------------------+---------------------+
| 0.0.0        | NodeType.tosca        | 2020-05-29 11:38:24 |
| 0.0.1        | NodeType.tosca        | 2020-05-29 11:51:12 |
| 0.1.1        | NodeType.tosca        | 2020-06-03 10:11:16 |
| 0.1.2        | NodeType.tosca        | 2020-06-03 11:48:35 |
| 0.1.3        | NodeType.tosca        | 2020-06-04 09:12:57 |
| 0.0.7        | AwsLambdaFunction.zip | 2020-06-05 07:50:26 |
| 0.0.11       | NodeType.tosca        | 2020-07-21 10:01:17 |
| 0.0.12       | NodeType.tosca        | 2020-07-21 10:04:22 |
+--------------+-----------------------+---------------------+
radon:~$ xopera-template-library entity-template get --name TestModel --version 0.1.3
Path to save template: TestModel
Downloaded TestModel version 0.1.3 to 'TestModel'.
```

Figure 8. Downloading a template of specific version

Other actions, as the usage of template groups and user groups is documented here: https://template-library-radon.xlab.si/examples.html

b) **CLI client from Eclipse Che terminal**. As Che is not meant for such usage, a few steps are required for CLI to work.

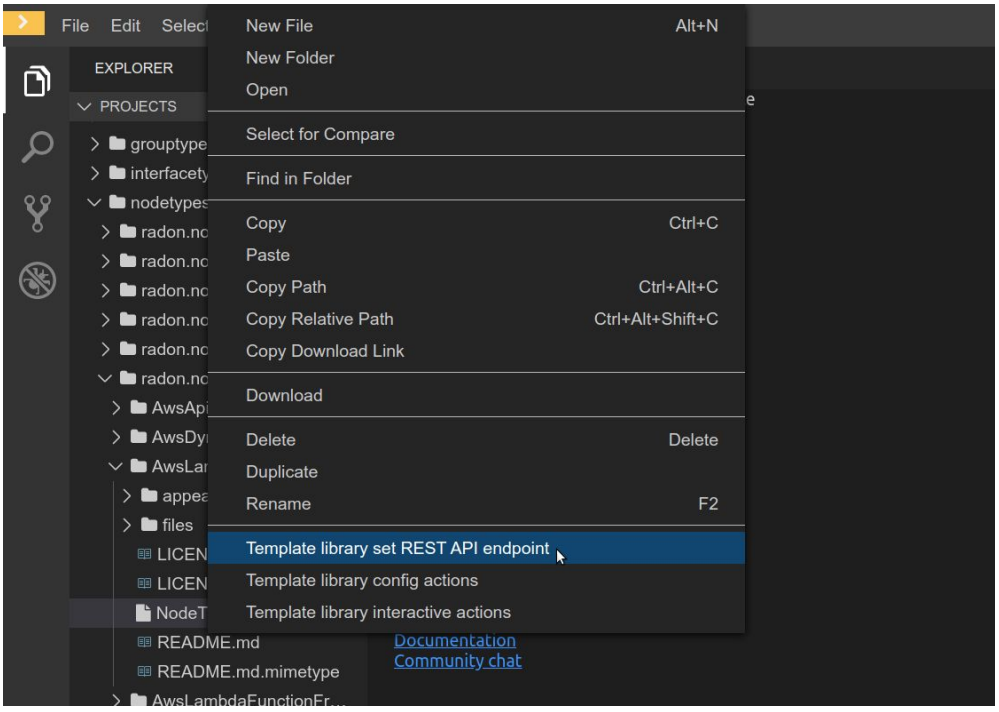
- First, creating a new custom workspace.
- After the workspace is ready, the package needs to be installed with `pip install xopera-template-library --user`.
- Exporting of PATH `/home/theia/.local/bin`.
- Creating a directory `.config/template_library` so endpoint and user credentials can be saved.
- Last step is moving to a directory with user permission e.g. `/projects`.

After these steps, the package should work as from the command line.

a) Using the **Template Library plugin for RADON IDE** (v0.0.2) is a bit more simple than using CLI client in Eclipse Che terminal, as it does not require special steps to prepare a workspace.

For using the plugin in Che there are two options. One is creating a custom workspace and using devfile for the plugin[1], another is using RADON IDE as the plugin is integrated and included in its dev file. In both ways, usage is the same.

When the workspace is ready, right click on any file opens dropdown options. There are three commands that can be selected.



[1] https://raw.githubusercontent.com/radon-h2020/radon-template-library-publishing-service-plugin/master/publishing-samples/devfile.yaml

Figure 9. Plugin's dropdown options with the Template Library commands

First, the REST API endpoint needs to be configured. To publish and retrieve templates user credentials need to be provided.

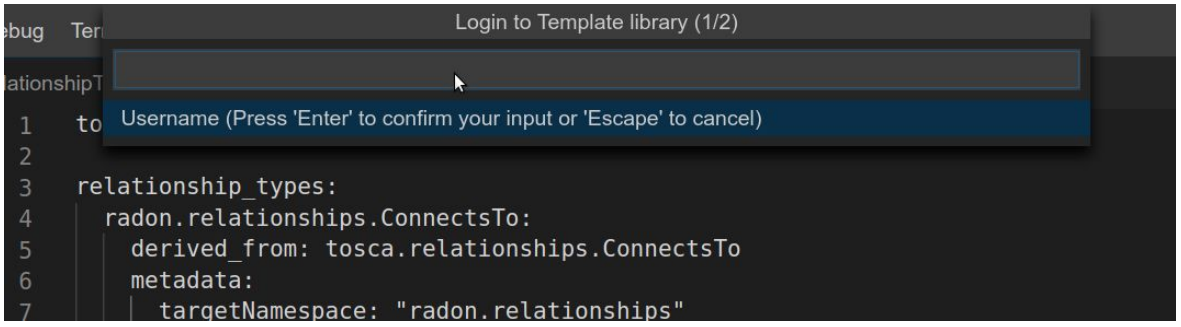


Figure 10. Providing user credentials for Template Library API

Using interactive actions, options to choose from are shown in another dropdown menu. Uploading or downloading templates of different versions is possible.

A JSON config file of the exact structure may also be used to perform upload or download of the templates, if choosing the Template Library config actions in plugin's dropdown options.

Status of the plugin's executed actions is shown in the popups.

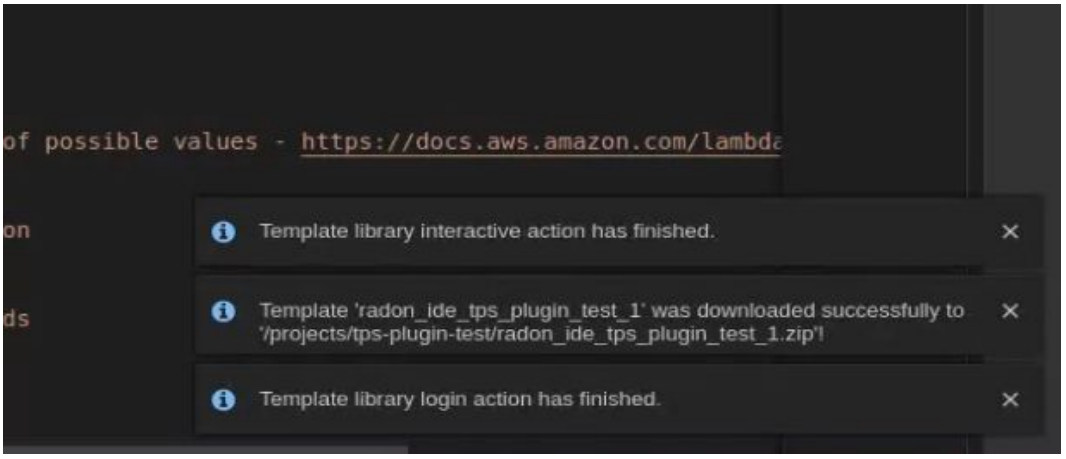


Figure 11. Status of the plugin's executed actions is shown in the popups

3.4 Orchestrator

The RADON Orchestrator is one of the most improved tools during the Y2 results. The orchestrator has received many updates on the ability to manipulate the deployment through the TOSCA

templates and the ability to run as a SaaS component. This allows setting up the orchestrator as a separate component holding the state of all IaC projects in progress which eases the possibilities of sharing the delivery tools among teams. This multitenancy component is crucial for establishing 24/7 support of the continuous delivery of online applications and services.

3.4.1 Final tool functionality set

The orchestrator is a key component that deploys application onto the infrastructure. Due to its importance, a lot of attention and effort was put into this tool. The first important release of the xOpera orchestrator -- version 0.1.0, based on TOSCA v1.0 standard -- and after that 20 new releases were published to the Pypi service[2] and they provide a vast improvement of TOSCA standard coverage and following new features:

- Support for TOSCA Simple Profile in YAML Version 1.3
- Support to initialize and execute compressed TOSCA CSARs
- Special switches to manage the deployment based on existing or clean state
- First integration tests, nightly CI/CD testing and semi-automated PyPI releases
- Concurrent task execution (with multiple possible workers)
- Complete parsing of TOSCA policy type targets and triggers alongside with activity, condition clause and event filter definitions
- Private ssh key file support with environment variables
- Updates for opera CLI commands (validate, init, deploy, undeploy and outputs)
- Decouple a Python library API from CLI interaction to allow using opera as a Python library in other projects
- Debug CLI mode for higher verbosity and to print Ansible playbook outputs
- Establish first documentation (available at https://xlab-si.github.io/xopera-opera/)
- Initial support for get_artifact TOSCA function

Orchestrator architecture and integration capabilities

During the improvement of the orchestrator and adding features we followed a minimum viable product (MVP) approach of the development. In the first cycle, an Opera library together with the CLI interface was developed and improved, as it is presented with green in Figure 13. Some users and developers preferably use this approach as it can be easily installed through `pip install opera` call and you are ready to deploy. The CLI version was ready very early in the project and has powerful capabilities. The approach is the most convenient if you need to install a specific version of xOpera orchestrator or you are bound to the use of command-line shell commands like some (high-performance computing) HPC users.

The CLI approach follows the idea of UNIX programs and is very powerful, but it lacks the possibility to integrate it with other services or frameworks in a user-friendly way. As a result, we developed an OperaAPI which is a parallel component with the CLI client and provides the ability

[2] https://pypi.org/project/opera/#history

to manage the Opera engine with the power of HTTP/REST calls. The OperaAPI is published on PyPI and users can install it with `pip install opera-api` command. The xOpera API is a component that is used directly from the xOperaSaaS service which also provides a SaaS API itself for managing SaaS projects and workspaces, which then gets connected to the Opera API in order to be able to activate endpoints that are similar to the CLI commands (eg. validate, init, deploy, undeploy, outputs, etc.). There is another component parallel to the xOpera SaaS service, which represents other possible Opera API integrations such as manual curl testing commands. Eventually, the xOperaSaaS (equipped with Opera API) is used from xOpera web user interface and is also integrated into an Eclipse Che Theia/Visual Studio Code plugin.

The next step forward to the orchestrator usability was the development of xOperaSaaS setup that employs OperaAPI and includes xOperaSaaS GUI and xOperaSaaS API. This SaaS setup provides an important list of functionalities:

- Running orchestrator as a service.
- Management of multiple workspaces and in each multiple deployment projects that can run concurrently.
- Managing provider credentials and assigning them to the workspaces.
- KeyCloak user management, possibility to integrate with other OpenID providers
- Multitenancy
- Integration through API or GUI

As a result, xOperaSaaS is an optimal solution based on integration abilities and provided functionalities.

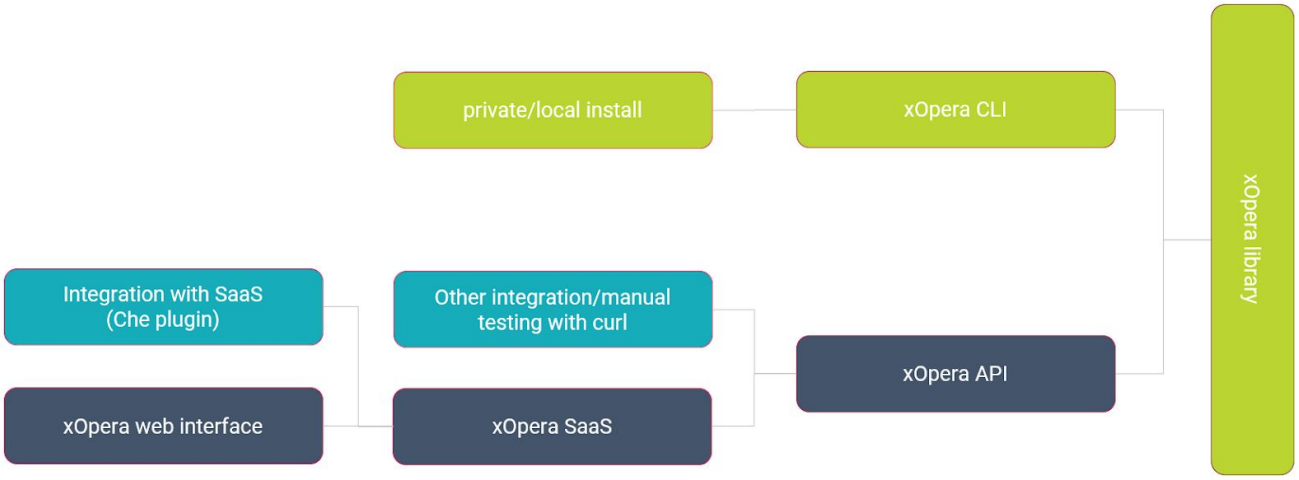


Figure 13. xOpera components, the only green were available before Y2.

3.4.2 Integration status

The orchestrator integration is twofold, as it is possible to integrate the CLI version or SaaS version of the orchestrator.

Orchestrator - CLI version

The CLI version of the orchestrator is available to be downloaded from the Python Package Index (PyPI). The CLI orchestrator can be managed with UNIX like commands, used in pipes and is very useful for developing and one-time job invocations, e.g., CI jobs where you can download and configure the orchestrator first and then submit the jobs.

Orchestrator - SaaS version

The SaaS orchestrator has been developed with the integration in mind and includes the API allowing managing all orchestrator functions and the integration with KeyCloak identity manager that allows easy integration with other identity management services. The SaaS API is a base for other integrations and has been exploited to integrate the orchestrator in the RADON environment. In this section we briefly integrate the integrations:

- Identity Manager (KeyCloak): orchestrator has user and group management, which means it also needs an identity management tool to take over the initial user and password management. Currently, orchestrator uses its own appliance of KeyCloak to get the possibility to run as an isolated service and can be integrated in multiple frameworks and services simultaneously.
- Orchestrator web GUI: The orchestrator has also the GUI based on the xOpera SaaS API, an additional option that enables managing orchestration. This way all actions made through the GUI can be made also with the individual API call. From a user perspective, it means that there is a seamless transition between management from GUI or Che and CI tools. Therefore if one way is not applicable, the user still has the possibility to use the other.

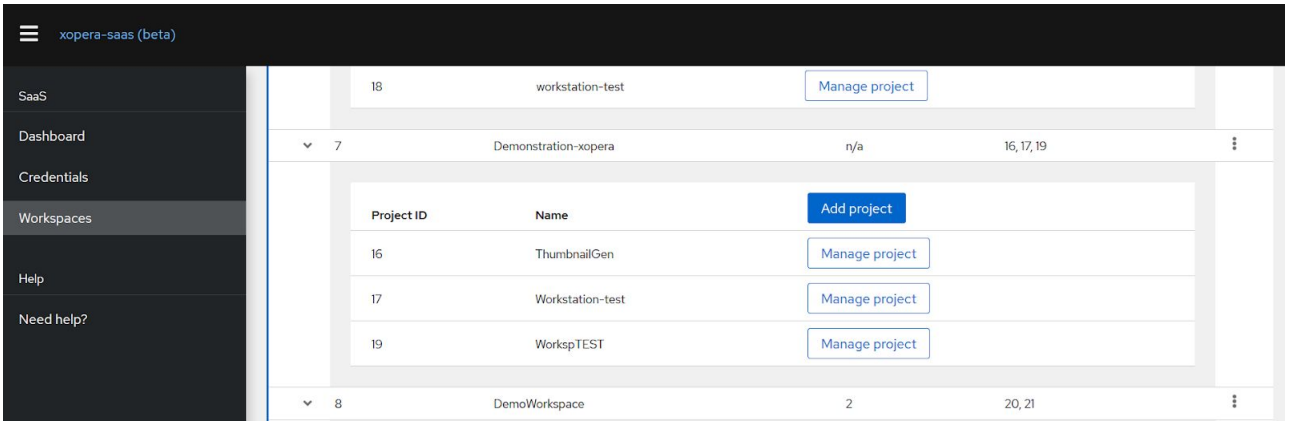


Figure 14. Orchestrator web GUI

- Che and Visual Studio Code plugin: There is an xOpera SaaS plugin for Che and Visual Studio Code available. This plugin provides the ability to create and run deployment projects created inside the SaaS orchestrator.

The RADON framework relies on Che plugins to interact with the SaaS orchestrator, while the more complex and delicate management actions (e.g. creating credentials and assigning them to the appropriate workspaces) is still done directly through the SaaS orchestrator GUI. The GUI is accessible from Che in a way that a new browser tab is opened with the appropriate orchestrator dashboard. Thanks to the KeyCloak integration there is no need to retype user credentials.

3.4.3 Achieved maturity level

The CLI version of the orchestrator has been used and tested for a while now and has been improved to fulfil the designed needs for the users. The estimated maturity level of the CLI version of the orchestrator is TRL 4-5. The xOpera SaaS orchestrator has been just recently developed and integrated with the other tools. However, underneath it shares the stable code library with CLI orchestrator, but due to its freshness, the current estimation of achievement is TRL 3-4.

3.4.4 Usage example/demo

The orchestrator can be used in multiple ways:

a) **CLI commands** has been already elaborated in previous deliverable D5.1 and described with the online published user manual[3]. The orchestrator interface was improved since the last version, mostly with introducing new commands for easier CSAR manipulation.

b) **Using a SaaS orchestrator through a web interface** is the most convenient way to gain full control over the SaaS orchestrator. The usual steps to create a deployment project environment are:

   i) *Credential management*: In the Credentials tab a user defines new provider credentials that can be used in the workspaces created by this user. When a new credential is added, the user defines a path, where it is stored and the content.

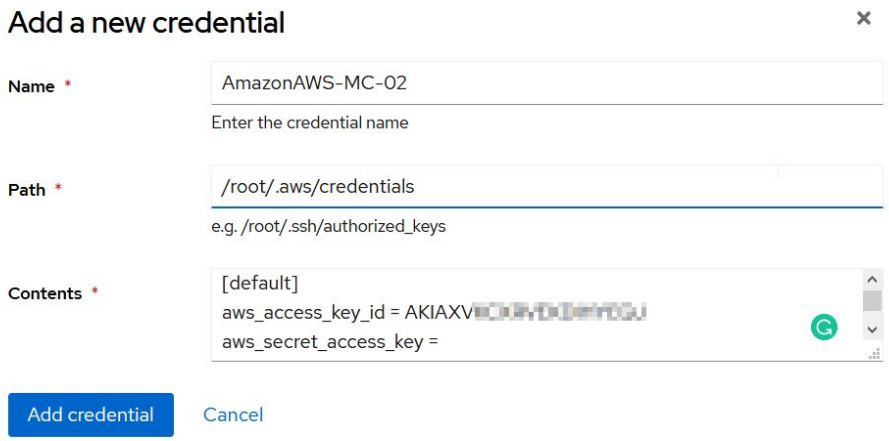


Figure 15. Credential management for SaaS orchestrator in web interface

[3] https://xlab-si.github.io/xopera-docs/

ii) Workspace management: In workspace tab a user can create new workspaces and assign the credentials to the workspaces. If the credential is assigned to the workspace, the credential will be accessible to all projects created in this workspace. The created project can be deployed and managed.

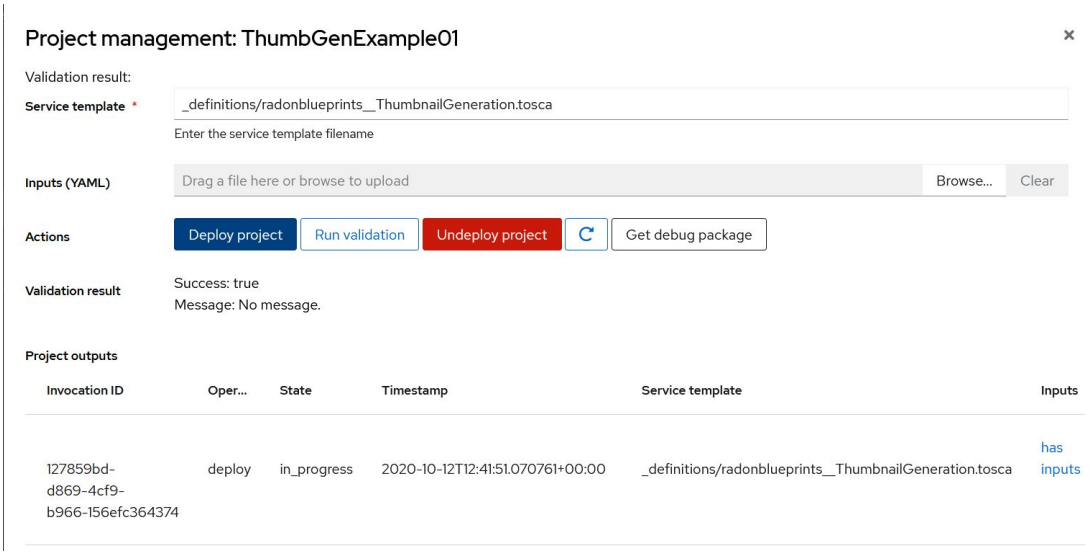


Figure 16. Project management for SaaS orchestrator in a web interface

iii)

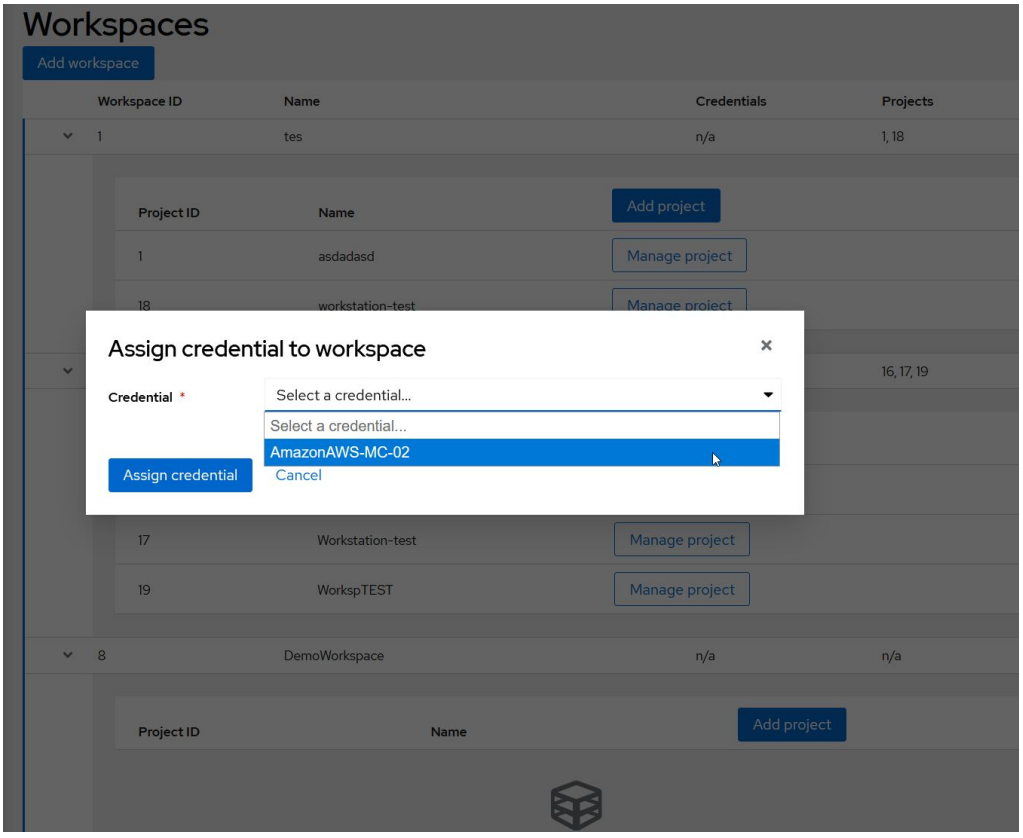


Figure 17. Assigning credentials to a workspace in the web interface

iv) Project management: In workspaces, new deployment projects can be created. One project is one deployment process, e.g. a project can be described by CSAR or other TOSCA compliant structure uploaded to the orchestrator. Users can attach new inputs to the blueprint and define the initial service yaml file. When the project is created, users can validate the project, deploy project, review responses and download all project's content from the SaaS if required for debugging purposes.

c) **Using the SaaS orchestrator through plugin.** The Opera SaaS can be accessed directly from Che using the Orchestrator SaaS plugin. The plugin allows users to a) create a new workspace; b) create a new project in a new or existing workspace; c) deploy a project. The functions through the plugin are a bit limited, as it is more convenient than other actions are performed through the Opera SaaS web interface.

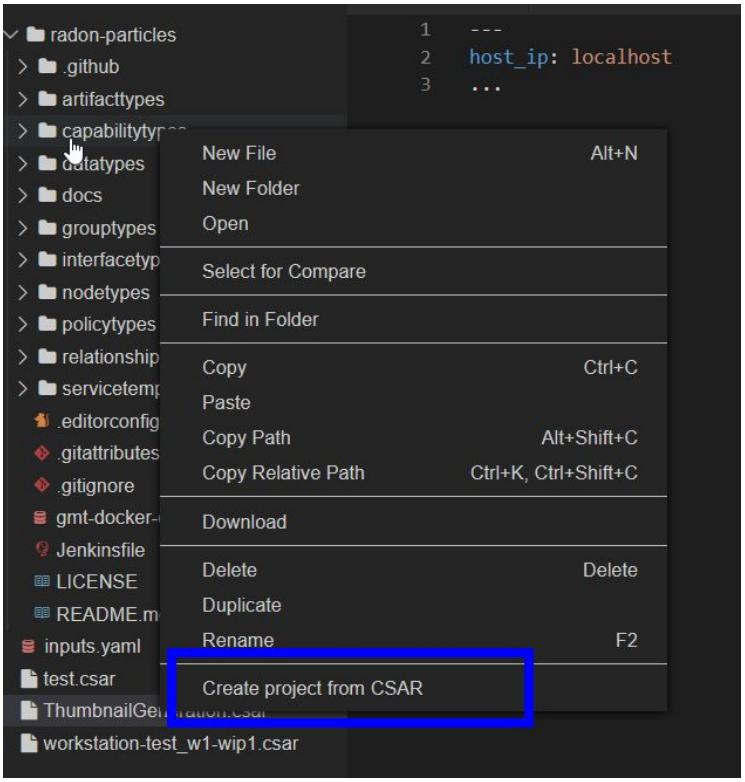


Figure 18. Plugin's dropdown option to create a project from CSAR

d) *Using the SaaS orchestrator through API calls.* The Opera SaaS can be used through API call. This is the most convenient way for CI/CD software to execute jobs on the orchestrator without the requirement of passing credentials through CI/CD. The credentials can be already in the project or workspace where CI/CD user is added.

3.5 Continuous integration

3.5.1 Final tool functionality set

From the D5.1 section, 3.5.5 *Future Plans*, we stated the future ambitions for the CI workflow. Generally it deals with improved usability, IDE interaction and additional features. Three topics were included in the mentioned section above:

1. Facilitate integration for each tool into a CI environment. We will achieve this by delivering a best practice approach and guidance for the relevant tool owners.

Most of the effort since the previous deliverable has been the alignment of the RADON toolstack for an unified approach to tool automation. This is crucial in an end-to-end workflow where each tool needs to handle the identical version of architecture from GMT. Table 2 depicts each tool and the format it is available as. Whether the tool is available as a CLI or container is usually based on the complexity of the required runtime environment of the tool.

Table 2. List of tools and their formats

Tool	Command-line execution
GMT	Not necessary
Opera	API / pip package
Template Library	API
CTT	Docker
DPT	Docker / pip package
VT	Docker container
DT	API
FunctionHub	pip package
Pipelines	Not necessary

Having each tool as a command-line executable enables the ability for automating tools through script configurations, which in turn is the central part of continuous integration. In the context of RADON we are focusing on the three tools; VT for verification of custom constraints, CTT for test

cases and dn DPT for quality analysis. Introducing these tools in a CI pipeline sets the foundation of automated quality assurance.

2. The next stage will be to look into how we can automate security scans and code review with FaaS, and include these into the CI pipeline.

Continuing with this ambition, PRQ has created functions for this purpose. It is also a central part of PRQ's KPI of populating the Function Hub with reusable functions. Examples of these functions are:

1. A function implementing SNYK[4] that checks the functions in Function Hub for dependencies or third party libraries for security risks. It checks the runtime and looks into the requirements.txt for python functions and package.json for JavaScript or Maven for Java dependencies.
2. A function that goes through the architecture of a RADON model searching for hard-coded keys, password or tokens.
3. A function that traverses a provided list of DNS addresses checking the expiry date of certificates.
4. A function that searches through policies of a RADON model looking for too loosely provided permissions.
5. A function checking the size of FaaS functions used in a RADON model. The size of the FaaS is commonly related to slower response times.

These functions will be available for end-users to include in their self defined CI pipeline.

3. Create templates and pseudo code config files for easier adoption from an end user perspective. Each CI provider has its own specific config syntax, making a pseudo config prevents tool lock-in with a specific CI provider.

Where Section 1 focused on aligning the tool and setting a standard for inputs and outputs, this section takes this further and creates concrete pipeline scripts showcasing its intended functionality. Examples can be found the CI template repository[5]. Here we are supporting configs and suggestions to CI setups. This is done to reduce the time bootstrapping a CI/CD environment. Figure 19 shows an example of how a RADON user will include VT in the CI pipeline.

3.5.2 Integration status

The RADON Runtime environment supports Jenkins as the default CI provider through the RADON IDE. The CI integration goes through a CHE plugin where the user can configure a Jenkins endpoint and trigger a specified pipeline. The CI pipeline needs to be defined separately in

[4] https://snyk.io
[5] https://github.com/radon-h2020/radon-cicd-templates

Jenkins based on the specific need of the end user. The Jenkins job can either be created in the Jenkins server provided by RADON or through a self hosted instance. The above mentioned CI template and the FaaS in Function Hub is supposed to work as resources for setting up the preferred CI workflow.

3.5.3 Usage example

Generally, what we want to achieve through CI, is an automated validation of our last contribution to the source code. Given this and the tools available, an example CI flow would look something like this:

Preliminary step: From the Jenkins instance in your Company workspace, create a Jenkins job.

1. In the IDE, after making a new feature to your application, the RADON user wants to receive feedback on the quality of the latest contribution. With the GMT extension, generate a CSAR file and execute the CI process.
2. The CI process will:
    a. First publish this CSAR to a test workspace in the Template Library.
    b. Execute the defined Jenkins job. The RADON user will be asked for URL, username and password.
3. In the Jenkins pipeline script we picture this workflow:
    a. Run Template Library client and fetch the CSAR version just uploaded to the test workspace.
    b. Run VT on the CSAR and constraint definition file as input. This verifies if your changes follow the constraints put upon your application architecture.
    c. Run DPT with CSAR and the trained prediction model as input. This gives you an analysis of the quality of your architecture as a whole and the latest contribution through historic comparison.
    d. Run CTT with the CSAR containing the test description. CTT will deploy application architecture in a test environment and perform the associated tests. Finally undeploy the application.
    e. Last job is dependent on the outcome of all previous jobs. If validations match the expected results, the Template Library client is executed and the CSAR is pushed to a stable workspace.
4. Through the user interface of Jenkins, RADON user can see the output of all performed tests and validations included in the pipeline.

```
pipeline {
    agent any
    environment {
        AWS_ACCESS_KEY_ID     = credentials('aws-id')
        AWS_SECRET_ACCESS_KEY = credentials('aws-secret')
    }
    stages {
       stage('Run VT') {
            environment {
                DEPLOY_FILE = 'todolist-dev.csar'
                VT_DOCKER_NAME = 'RadonVT'
                VT_FILES_PATH = '{"path":"/tmp/radon/container/main.cdl"}'
            }
            steps {
                sh 'echo Start VT container and perform verify test...'
                sh 'unzip -o ${DEPLOY_FILE}'
                sh 'mkdir -p $PWD/tmp/radon && cp -r _definitions $PWD/tmp/radon/_definitions && cp main.cdl $PWD/tmp/radon'
                sh 'docker run --name "${VT_DOCKER_NAME}" --rm -d -p 5000:5000 -v \
                $PWD/tmp/radon:/tmp/radon/container marklawimperial/verification-tool'
                sh 'sleep 5'
                sh 'docker exec ${VT_DOCKER_NAME} sh -c "cd /tmp/radon/container && pwd && ls -al && cat main.cdl"'
                sh 'curl -X POST -H "Content-type: application/json" http://localhost:5000/solve/ -d ${VT_FILES_PATH}'
                sh 'docker stop ${VT_DOCKER_NAME}'
            }
        }
    }
}
```

Figure 19. Verification tool example

3.6 Continuous deployment

3.6.1 Final tool functionality set

Continuous deployment often picks up where continuous integration ends; put differently, CI works as an enabler for CD. Through CI the user ensures the quality of the latest version and clear it as a ‘release candidate’, ready for deployment. From here, CD picks any of these versions for small, incremental improvements to the application. Similar to any CI process, CD also relies on scriptable commands and the potential of automation. Whether the flow of CD is autonomous or a manual step, the benefit of scriptable deployments is considerable either we focus on reproducibility, visibility, increased speed or mitigation of human errors.

3.6.2 Implementation

In consistency with the CI implementation, Jenkins is also the de facto tool in the RADON Runtime Environment when it comes to CD workflow. The CD integration also goes through a CHE plugin where you configure a Jenkins endpoint and trigger a specified pipeline. The pipeline is custom defined based on the specific need of the end user. Similar to the CI pipeline it’s also created with best practice suggestions in the CI Template repository.

3.6.3 Usage example

Continuous Deployment is a state achieved when you trust that a) generally, your code is good enough for production, and b) the deployment process is good enough for easy roll-back of previous stable versions. The first part is usually covered through code validation in the CI pipeline. The second part is controlled by the CD process in combination with monitoring. Basically, the more you can monitor the easier you can create responses to events, and the more resilient your deployment process becomes. One example of a CD flow is:

1. In the IDE, select the deployment of a CSAR, either a newly generated or ideally the version just passing the checks in the CI pipeline. This deployment will trigger the configured Jenkins job. From our example:
    a. Run Template Library client and fetch the CSAR version just validated and uploaded to the stable workspace.
    b. Run Orchestrator and deploy the application defined in the CSAR. (An example of this is shown in Figure 20)
    c. Monitor the behaviour of your application through the RADON monitoring functionality for a certain period of time. The CD pipeline is on hold during this period
    d. If any errors are detected within the time window, the previous version of application is deployed.

```
pipeline {
    agent any
    environment {
        AWS_ACCESS_KEY_ID     = credentials('aws-id')
        AWS_SECRET_ACCESS_KEY = credentials('aws-secret')
    }
    stages {
        stage('Install dependencies') {
            steps {
                withEnv(["HOME=${env.WORKSPACE}"]) {
                    sh 'echo install any dependencies..'
                }
            }
        }
        stage('Opera Deploy') {
            environment {
                DEPLOY_FILE = 'todolist-dev.csar'
                OPERA_DOCKER_NAME = 'prqCont'
            }
            steps {
                withEnv(["HOME=${env.WORKSPACE}"]) {
                    sh 'docker build --tag opera-deploy .'
                    sh 'mkdir -p $PWD/tmp/radon && cp -r todolist-dev.csar $PWD/tmp/radon'
                    sh 'docker run --name ${OPERA_DOCKER_NAME} --rm -d -p 18080:18080 -v $PWD/tmp/radon:/tmp/radon/container -e \
                    "AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}" -e "AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}" -e "CTT_FAAS_ENABLED=1" opera-deploy'
                    sh 'sleep 5'
                    sh 'docker exec ${OPERA_DOCKER_NAME} sh -c "cd /tmp/radon/container && opera init todolist-dev.csar && opera deploy "'
                    sh 'docker stop $OPERA_DOCKER_NAME'
                }

            }
        }
    }
```

Figure 20. Automation of deployment

The Jenkins user interface provides log of historic deploys for traceability.

3.7 Function Hub

3.7.1 Final tool functionality

Following up on *3.7.5 Future Plans* from D5.1, the remaining work was divided into three areas;

a) Common function format.

A common format of a FaaS package is necessary for both contributors and users. Both parties should know the amount of information necessary for using a function without access to the internal content. The solution to this is a config description containing the properties of a function.

```
[REPOSITORY]
org = # name of functionhub organization
repository = # name of your repository

[FUNCTION]
name = # name of the function
version = # version of the function
description = # text describing the content of the function

[RUNTIME]
provider = # which cloud provider [aws,gcloud,azure]
runtime = # which programming runtime
handler = # name of file and name of landing function eg. helloworld.hello
```

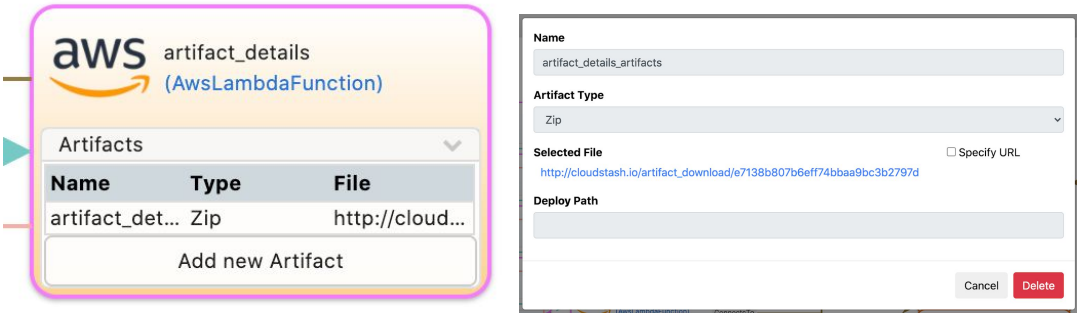
Figure 21: example config.ini

Figure 20 describes the input of FunctionHub client[6]. Based on the subsection of Function and Runtime, the end user is able to use the FaaS as a blackbox.

b) Integration with GMT and xOpera.

The power of RADON lies in its integration between the tools. Being able to create FaaS objects from GMT using Function Hub was an important step of securing this goal. This is implemented such that when a user is creating a FaaS object in GMT, he can either select a file in the local environment or an external url. The user can then provide a Function Hub link as the external file section. When the CSAR is generated, the necessary artifacts are downloaded and put into the file.

[6] https://pypi.org/project/functionhub/



c) General functionality. A common CLI client, support for private repositories, etc.

The development of the FunctionHub CLI was the main priority in this section of work. The client was essential for an intuitive user experience when interacting with FunctionHub, enforcing the function format and handling authentication. The client is wrapped as a pypi package available through pip install.

```
[~]$ fuhub --help
Usage: fuhub [OPTIONS] COMMAND [ARGS]...

Options:
  -e, --endpoint TEXT  Endpoint URL for your FunctionHub
  -d, --debug TEXT     Enable debug output
  -at, --token TEXT    Set access token for publishing to private repositories
  --version            Show the version and exit.
  --help               Show this message and exit.

Commands:
  create
  package
  upload
[~]$
```

The client supports three main commands; create, package and upload.

Within the RADON environment, Function Hub plays the role of providing a common collection of reusable functions. With the provided client, the user can share her functions publicly or stored in private repositories. Through the user interface at cloudstash.io the user can:

- create user / log in
- create repositories
- browse / download functions

Any function can either be downloaded directly from the website or through the integration with GMT.

3.3 Delivery toolchain

The delivery toolchain includes a set of tools that are tightly connected to provide the DevOps functionality for the end user. As already presented in the previous deliverable, the delivery toolchain combines CI/CD tools, orchestrator and monitoring. During the development of the RADON environment improved the Template Library, which can be used to store or publish a service template blueprint for the deployment. This change made the Template Library also a tool included in the delivery toolchain.

3.3.1 Final tool functionality set

The idea of the delivery toolchain is to manage the whole deployment process from the IDE and be able to monitor it. The deployment workflow is presented in Figure 12, where it is presented that an application service blueprint can be deployed directly by creating deployment projects on the orchestrator and initiating the deployment or, using the CI/CD jobs where the deployment flow is managed by the CI/CD jobs. Current delivery toolchain functionality allows:

- Managing, including storing/versioning, templates through IDE
- Setting the deployment project environment in SaaS orchestrator through IDE
- Deploying the service directly through IDE

3.3.2 Integration status

The Delivery toolchain is integrated in the IDE by the integration and collaboration of crucial components, which are CI/CD tool, orchestrator and Template Library. The IDE can use the SaaS version of the orchestrator directly, which means that it is possible to set up a deployment job for specific CSAR directly in a SaaS orchestrator version. The CI/CD jobs can also benefit from using the SaaS orchestrator with manipulation of the deployment jobs over the REST API calls.

In addition to the SaaS approach, the orchestrator can be installed separately and used with a CLI option, where the CI job could install a specific opera version with `pip install opera` command if needed and start the CI process. The drawback of this approach is in passing the credentials required to access the provider. As the credentials cannot be pre-set in an orchestrator they need to be passed together with the application itself or retrieved with the job in some way.

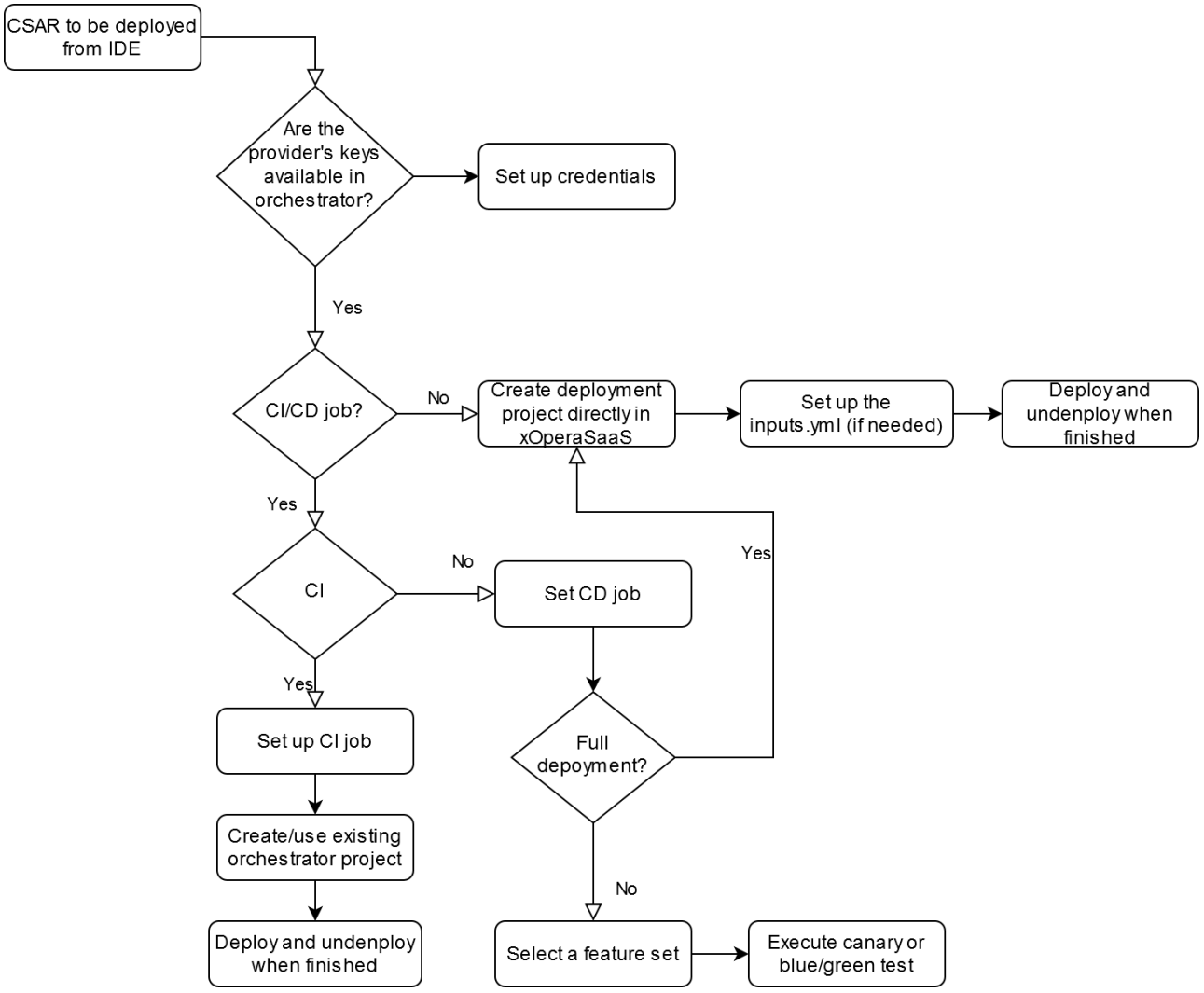


Figure 12. Delivery toolchain flow

4. Runtime management operations

Orchestration of the application is more than just deploying it and often requires a continuous runtime management of the deployed application.

4.1 Scaling management

To ensure that desired non-functional requirements of the deployed application are upheld, it is often required to scale up the computing resources allocated to a service after it is deployed based on its current load. While cloud providers manage FaaS platforms having built-in automatic scaling, microservices and functions, deployed on open source FaaS platforms and modified the allocated computing resources as needed.

How to achieve dynamic scaling for applications with increasing demand and diverse workloads is a critical issue. Managing the elasticity of applications deployed in cloud infrastructure requires modification of the computing resources (how many resources will be added or removed, i.e. scale-up/scale-down) at runtime. In the RADON approach, the structure of the deployed services and computing resources allocated to them are described in TOSCA language and the RADON orchestrator can be used to change the structure of a running service or its computing resources at runtime. Furthermore, the RADON framework is enriched with a monitoring system (designed with Prometheus server), which can be used to detect situations (e.g. CPU load higher than the threshold value, i.e. greater than 80% or CPU load lower than the threshold value, i.e. less than 20%), where the service should be scaled (e.g. up or down) and to generate respective events (e.g. using Prometheus Alertmanager), which in turn can be used to trigger the modification of the deployed system using the RADON orchestrator.

The TOSCA standard currently supports deployment, operation, configuration, and termination of applications on the top of the private/public cloud instances by standardizing the life-cycle management and creating a relationship between the instances. However, it does not specifically support life-cycle operations related to scaling. To appropriately investigate how to solve this issue, two approaches of scaling were established, each trying to achieve TOSCA scaling possibilities with current RADON abilities and TOSCA limitations.

The first approach is to design a policy-based auto-scaling model, which generates a modified TOSCA service template (either adding a set of TOSCA nodes to scale up the service or removing a set of nodes to scale the service down) after receiving an alert from the monitoring server and uses RADON orchestrator to modify the running service. The illustration of defining different scaling policies in a service template is depicted in **Figure 19**. Here, the *ScaleIn* policy represents that if the CPU utilization of the TOSCA nodes is less than or equal to 20% (i.e. *cpu_upper_bound: constraints: - less_or_equal: 20.0*), then an alert is generated and a TOSCA node is removed from the service template (i.e. *adjustment: constraints: - less_or_equal: -1*). Similarly, in the *ScaleOut* policy, if the CPU utilization of the TOSCA nodes is greater than or equal to 80% (i.e. *cpu_upper_bound: constraints: - greater_or_equal: 80.0*), then an alert is generated and a TOSCA node is added to the service template (i.e. *adjustment: constraints: - greater_or_equal: 1*). Finally, the *AutoScale* policy represents that the service template started with *X* number of TOSCA nodes (i.e. *min_size: constraints:- greater_or_equal: 1*), which is represented as 1 in the following example and scale up to *Y* number of TOSCA nodes (i.e. *max_size: constraints:- greater_or_equal: 10*), which is represented as 10 in the following example (illustrated in **Figure 19**).

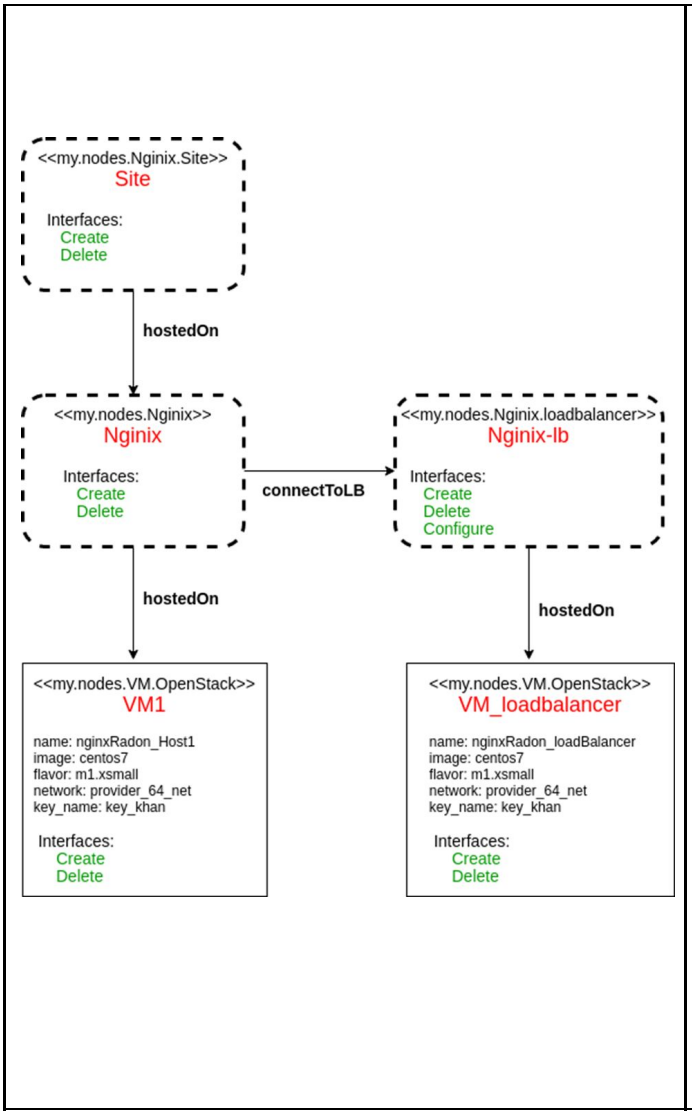
```
policy_types:
  radon.policies.scaling.ScaleIn:
    derived_from: tosca.policies.Scaling
    properties:
      cpu_upper_bound:
        description: The upper bound for the CPU
        type: float
        required: false
        constraints:
          - less_or_equal: 20.0
      adjustment:
        description: The amount by which to scale
        type: integer
        required: false
        constraints:
          - less_or_equal: -1

  radon.policies.scaling.ScaleOut:
    derived_from: tosca.policies.Scaling
    properties:
      cpu_upper_bound:
        description: The upper bound for the CPU
        type: float
        required: false
        constraints:
          - greater_or_equal: 80.0
      adjustment:
        description: The amount by which to scale
        type: integer
        required: false
        constraints:
          - greater_or_equal: 1

  radon.policies.scaling.AutoScale:
    derived_from: tosca.policies.Scaling
    metadata:
      abstract: "false"
      final: "false"
      targetNamespace: "radon.policies.scaling"
    properties:
      min_size:
        type: integer
        description: The minimum number of instances
        required: true
        status: supported
        constraints:
          - greater_or_equal: 1
      max_size:
        type: integer
        description: The maximum number of instances
        required: true
        status: supported
        constraints:
          - greater_or_equal: 10
```

Figure 19. Illustrative example of different scaling policies in a service template

The second one is trying to understand which TOSCA modifications would be required to equip all TOSCA orchestrators with the scaling functionality. The latter one would define scaling more in detail and would not alter the TOSCA blueprint definitions each time when the scaling notification would be accepted. This approach was elaborated in an article [Can20] and is still under ongoing research and in discussion with the standardisation group in TOSCA.

Autoscaling based on the separate scale application

To automatically scale the services by modifying their TOSCA service template, an autoscaler agent service is developed, which utilizes RADON monitoring to receive scaling related events, modifies the service template to scale it up or down as required. We illustrate how the automatic scaling process works with an example service template (Illustrated on **Figure 20**), consisting of an Nginx load balancer and an application server, serving a simple HTML webpage. The Nginx load balancer distributes the incoming user requests between the application servers.




```
topology_template:
  node_templates:
      vm_loadBalancer:
          type: my.nodes.VM.OpenStack
          properties:
          name: nginxRadon_loadBalancer_0
          image: centos7
          flavor: m2.xsmall
          network: provider_64_net
          key_name: key_adhikari
      vm_xx:
          type: my.nodes.VM.OpenStack
          properties:
          name: nginxRadon_Host_xx
          image: centos7
          flavor: m2.xsmall
          network: provider_64_net
          key_name: key_adhikari
      nginx_xx:
          type: my.nodes.Nginx
          requirements:
          - host: vm_xx
          - connectToLB: nginx-lb
      site_xx:
          type: my.nodes.Nginx.Site
          properties:
          hostname: site1
          requirements:
          - host: nginx_xx
      nginx-lb:
          type: my.nodes.Nginx.loadBalancer
          requirements:
          - host: vm_loadBalancer
```

(a) (b)

Figure 20. Illustrative example: (a) Initial service model; (b) Initial service template

A Node Exporter service is deployed on each of the application servers for exporting performance metrics of the underlying TOSCA nodes into the RADON monitoring system (i.e. Prometheus

server), and the VM metrics (e.g CPU utilization) are used to decide (based on the TOSCA autoscaling policy) when the number of application servers should be scaled up or down to handle the current amount of user requests.

Initially, the orchestrator is used as usual to deploy the TOSCA service template. Monitoring is set up to trigger events based on the scale-up and scale-down thresholds in the autoscaling policy and send the events to the autoscaler service through webhooks. The autoscaler agent waits for events and enacts the scaling actions if required. **Figure 21** illustrates the scaling-up action, where a copy of the scalable TOSCA node (and all its dependencies) are added to the service template. **Figure 22** illustrates a respective scale-down action.

During the scale-up process, a new set of TOSCA nodes are dynamically added to the service template, including a VM instance to be deployed (VM), application server (Nginx), metric exporter service (Node Exporter) and website (Site). In addition, the application server node is connected with the load balancer using the ConnectToLB relationship, which initiates the load balancing configuration. Similarly, during the scale-down process, an existing set of TOSCA nodes are dynamically removed from the service template.

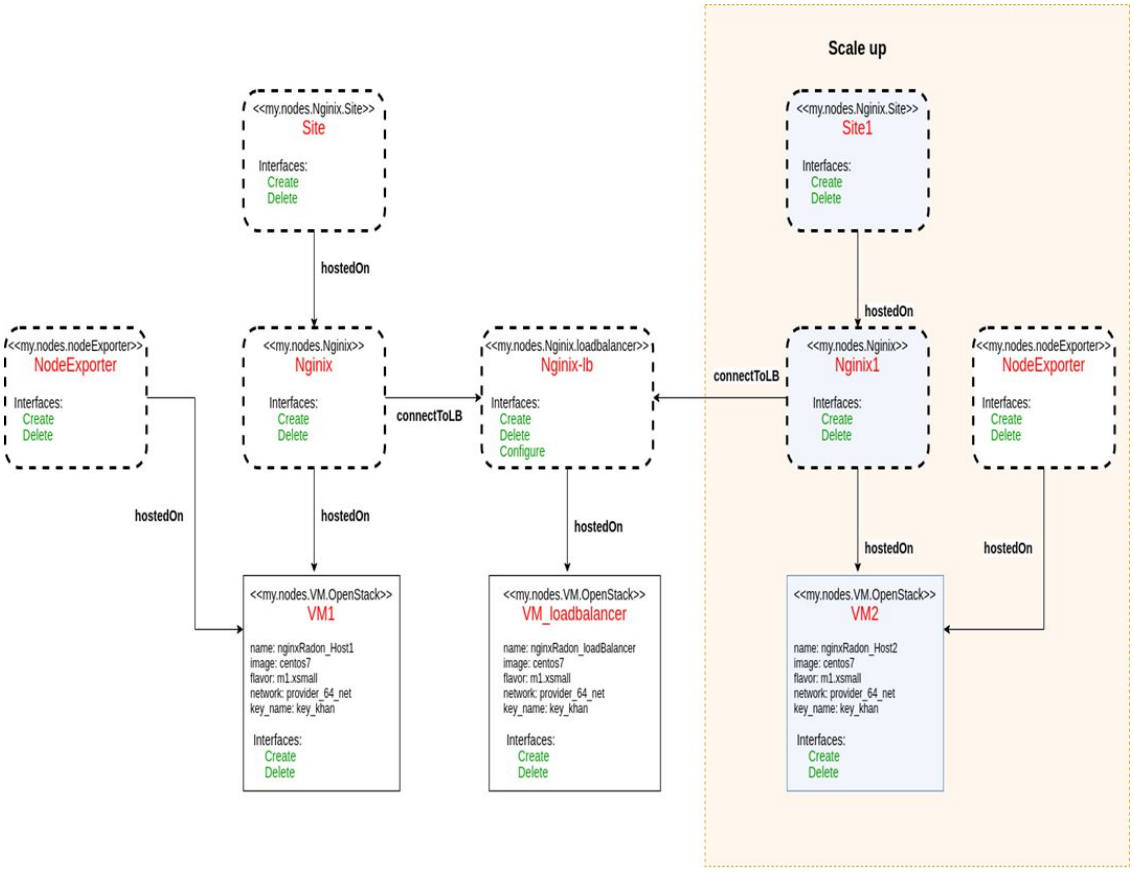


Figure 21. Overview of scaling-up model with TOSCA template

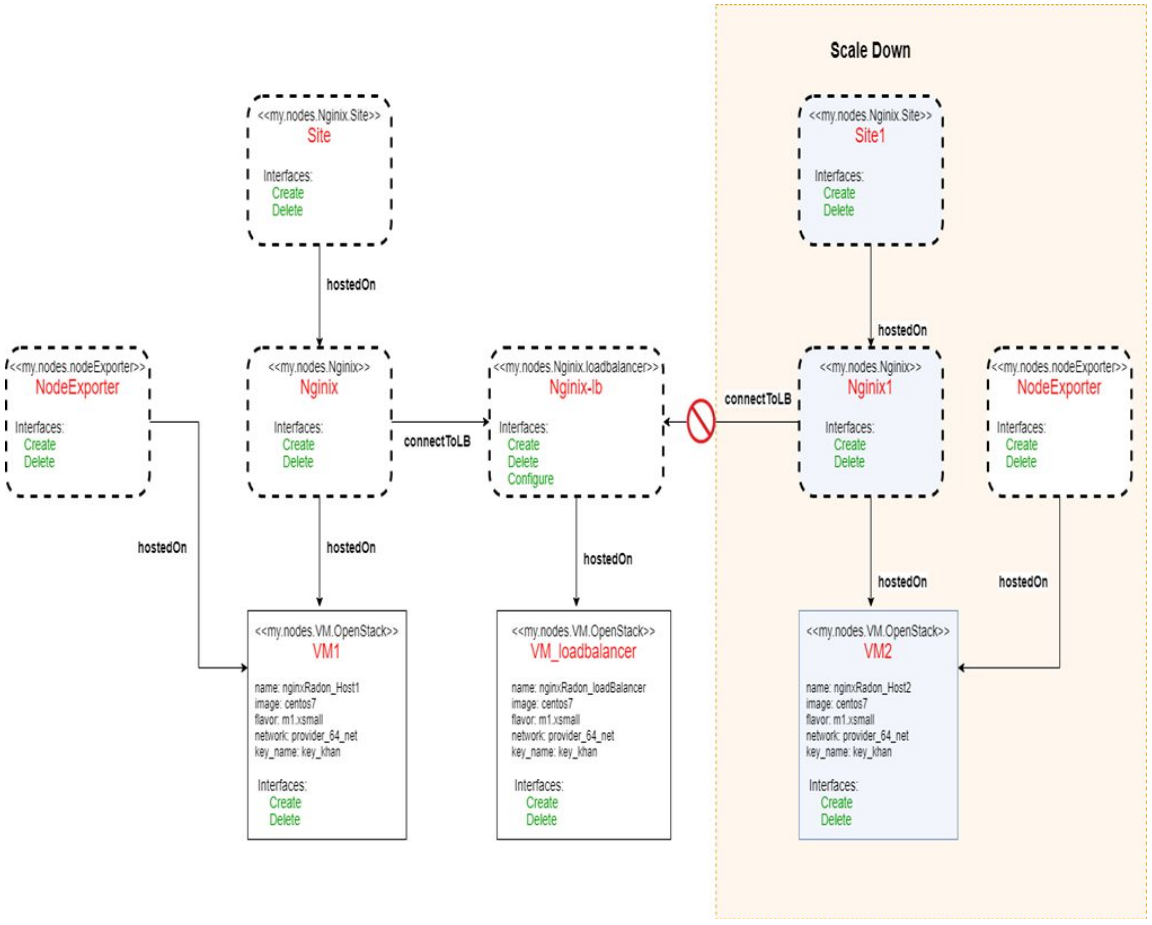


Figure 22. Overview of scaling-down model with TOSCA template

In order to develop an auto-scaling agent, which is able to change the TOSCA service template at runtime, we need to utilize both RADON orchestrator and the monitoring system. The orchestrator is used to deploy an application on the run-time environment by enforcing the state described by the application blueprint (CSAR) onto the target cloud provider. The monitoring server is composed of two different components, namely Prometheus server and Alertmanager. The Prometheus server fetches the run-time metrics from the running cloud instances to measure their current states and the Alertmanager generates an alert to the auto-scaler agent endpoint based on the rules defined in the Prometheus server.

The auto-scaler agent receives the notifications from the Alertmanager, takes a scaling-up/scaling-down decision and deploy/undeploy an instance in the cloud infrastructure using the help of the orchestrator. Thus, the main purposes of the auto-scaler agent are to i) receive the monitoring notification from the monitoring server; ii) create a new TOSCA blueprint for scaling-up or scaling-down an instance; and iii) send the blueprint to the orchestrator for further deploying or undeploying an instance from cloud infrastructure. The developed auto-scaling model

of the running instances in a cloud infrastructure is shown in Figure 23. A repository of the proposed auto-scaling model is provided online and published in RADON GitHub[7].

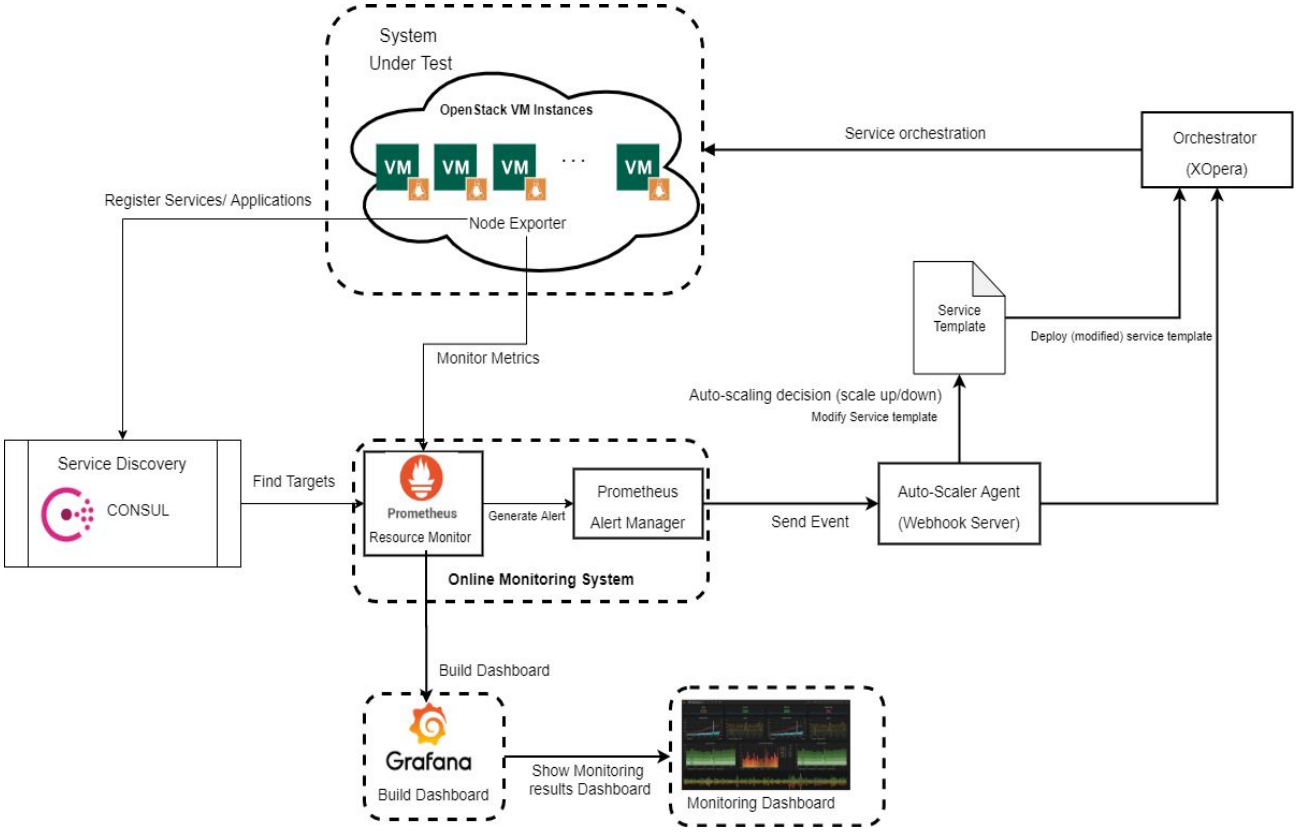


Figure 23. Auto-scaling model of the running instances in a cloud infrastructure

After the initial orchestration of the application, the monitoring server starts to collect the run-time metrics (CPU utilization) and stores the collected metrics in the time-series database. In order to configure the monitoring server (Prometheus and Alertmanager) for generating an alert, the first step is to configure the prometheus.yml file. In this configuration file, the scrape_interval and monitoring activity of the active set of instances are explicitly defined. The next step is to define the alerting rules (shown in Table 3) in the Prometheus server for further configuration to generate an alert message after reaching a predefined threshold value of the active cloud instances.

Table 3. Descriptions of alert rules defined in Prometheus server

Name of the Alerts	Configurations of the Alerts	Description
HighCPULoad	*alert: HighCpuLoad* *expr: 100* *- (avg by(instance)* *(irate(node_cpu_seconds_total{mode="idle"}[1m])) * 100)* *> 80*	This rule determines whether the average CPU usage of all the running instances is more than 80% for more than 5 minute or not.

[7] Policy-based auto-scaling model: https://github.com/radon-h2020/demo-policy-based-autoscaling

	for: 5m	
LowCPULoad	*alert: LowCpuLoad* *expr: 100* *- (avg by(instance)* *(irate(node_cpu_seconds_total{mode="idle"}[1m])) * 100)* *<20* *for: 5m*	This rule determines whether the average CPU usage of all the running instances is less than 20% for more than 5 minute or not.

It is also important to make sure that the alerts are not triggered too frequently in situations where only a slight peak of increased user activity is detected. For example, the Prometheus server can be configured to wait for 5 minutes if the overall CPU capacity of the active cloud instances is greater than 80% or less than 20%. After 5 minutes, if the CPU capacity of the active cloud instances still is greater than 80% or less than 20%, then the Prometheus server sends an alert message to the Alertmanager for taking further action, as shown in **Figure 24**.

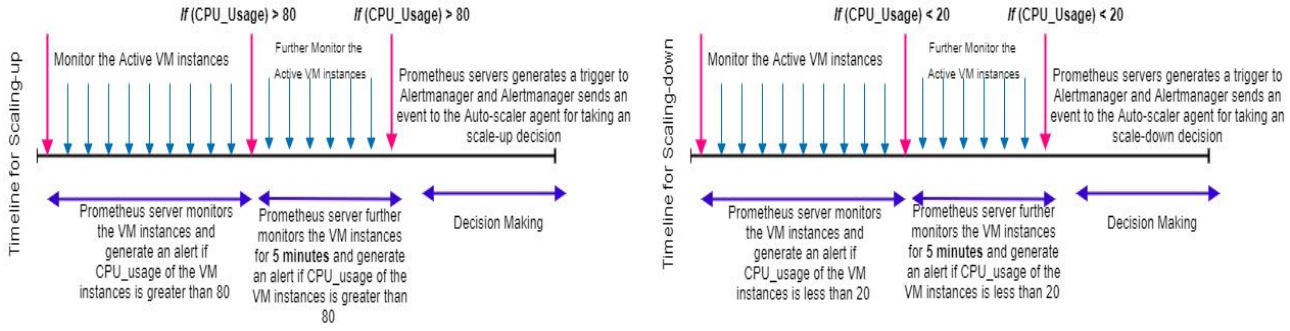


Figure 24. Methodology for resource monitoring and scaling-up/down decision making

The Alertmanager is responsible to handle the alerts, sent by the Prometheus server and sends a POST request to the auto-scaler agent (through webhook integration) through an URL with the monitoring metrics (i.e. current CPU usage) in JSON format. The autoscaler agent is developed as a Python application and uses the Flask framework to serve a REST endpoint for receiving alerts from the Alertmanager, which are triggered by the autoscaling rules configured in Prometheus. When an alert is triggered, the autoscaler agent receives a POST request from the alertmanager, where the content of the alert is provided as a JSON format.

The agent parses the JSON file to extract the alert name and the IP address of the running instances for which the alert was triggered. Based on the name of the alert and the instance, the autoscaler agent decides whether the system is to be scale-up or scale-down in the first place. Finally, the autoscaler agent generates a modified TOSCA service template as per the scaling decision and deploys a new instance or undeploy an existing instance through the orchestrator. An illustration of the scaling-up decision with initial TOSCA service template and the modified TOSCA service template is depicted in **Figure 25** (a) and **Figure 25** (b), respectively.

```
topology_template:
  node_templates:
      vm_loadBalancer:
          type: my.nodes.VM.OpenStack
          properties:
          name: nginxRadon_loadBalancer_0
          image: centos7
          flavor: m2.xsmall
          network: provider_64_net
          key_name: key_adhikari
      vm_xx:
          type: my.nodes.VM.OpenStack
          properties:
          name: nginxRadon_Host_xx
          image: centos7
          flavor: m2.xsmall
          network: provider_64_net
          key_name: key_adhikari
      node_xx:
          type: my.nodes.NodeExporter
          requirements:
          - host: vm_xx
      nginx_xx:
          type: my.nodes.Nginx
          requirements:
          - host: vm_xx
          - connectToLB: nginx-lb
      site_xx:
          type: my.nodes.Nginx.Site
          properties:
          hostname: site1
          requirements:
          - host: nginx_xx
      nginx-lb:
          type: my.nodes.Nginx.loadBalancer
          requirements:
          - host: vm_loadBalancer
```

```
topology_template:
  node_templates:
      site_1:
          type: my.nodes.Nginx.Site
          properties:
          hostname: site1
          requirements:
          - host: nginx_1
       nginx_1:
          type: my.nodes.Nginx
          requirements:
          - host: vm_1
          - connectToLB: nginx-lb
       vm_1:
          type: my.nodes.VM.OpenStack
          properties:
          name: nginxRadon_Host_1
          image: centos7
          flavor: m2.xsmall
          network: provider_64_net
          key_name: key_adhikari
       node_1:
          type: my.nodes.NodeExporter
          requirements:
          - host: vm_1
      vm_loadBalancer:
          type: my.nodes.VM.OpenStack
          properties:
          name: nginxRadon_loadBalancer_0
          image: centos7
          flavor: m2.xsmall
          network: provider_64_net
          key_name: key_adhikari
      vm_xx:
          type: my.nodes.VM.OpenStack
          properties:
          name: nginxRadon_Host_xx
          image: centos7
          flavor: m2.xsmall
          network: provider_64_net
          key_name: key_adhikari
      node_xx:
          type: my.nodes.NodeExporter
          requirements:
          - host: vm_xx
      nginx_xx:
          type: my.nodes.Nginx
          requirements:
          - host: vm_xx
          - connectToLB: nginx-lb
       site_xx:
          type: my.nodes.Nginx.Site
          properties:
          hostname: site1
```

	requirements: *- host: nginx_xx* *nginx-lb:* *type: my.nodes.Nginx.loadBalancer* *requirements:* *- host: vm_loadBalancer*
(a)	(b)

Figure 25. Illustrative example of service templates for scaling-up decision: (a) Initial service template with a Nginx load balancer and a VM instance; (b) Modified service template with scaling-up decision

In such an elastic environment, the newly added instances in the cloud infrastructure also need to be monitored. To dynamically update the instances that Prometheus needs to monitor, a Consul service is set up to keep track of the currently deployed instances and a Consul agent is deployed on each of the instances. The Consul agent registers itself in the Consul service and the Monitoring platform retrieves the IP addresses of the current instances in the auto-scaling group from the Consul service and monitors them for generating further events.

Autoscaling based on a scaler integrated in an orchestrator.

The second approach is based on the idea running the scaling operations inside the orchestrator. In comparison to the first approach the monitoring part stays the same, while the operational part is different. The orchestrator does not act as a black box, which only receives instructions by the scaler, but it receives the notification, processes the trigger and runs an appropriate scaling script.

An example of the proposed TOSCA scaling approach is shown in Figure 26, where the scaling policy and interface are presented. The example is simplified[8] and lacks some parts to emphasise only the crucial parts of an approach required to fulfill scaling within the TOSCA orchestrator by configuring scaling in TOSCA YAML service template. In the first part of the listing in Figure 22 -- lines 1-22 -- the initial application state is defined using an OpenStack VM. The next TOSCA block -- lines 24-40 -- define TOSCA scaling policy with limiting the property of CPU. This threshold is globally limited with additional lines not accepting values lower than 80. This CPU threshold should be configured by the orchestrator in a particular monitoring, which configuration is omitted from our example for brevity. The next part -- lines 41-51 -- defines the TOSCA policy triggers that initiate scaling procedures and is related to the OpenStack node. Trigger will call the scaling interface block from line 53. The scaling block definition comes next -- lines 53-61 -- where the scaling operation (from the defined TOSCA interface type) is called by the trigger and can be discovered with the implementation script e.g., a specific Ansible playbook that the orchestrator would process at the deployment phase. This *scale_out.yaml* script would perform scaling on an OpenStack VM e.g. deploy one additional instance to balance the load on the application. The rest of the TOSCA template -- lines 63-79 -- defines a topology template that initializes the first VM and sets scaling policy properties.

[8]More complex example: https://github.com/xlab-si/xopera-opera/blob/master/examples/policy_triggers/service.yaml

The example of this approach is described in the article [Can20] and will be discussed with the TOSCA standardisation group. To demonstrate the applicability of the approach through a working example, a trigger endpoint is being developed on the SaaS Orchestrator and will be used to demonstrate this approach to the TOSCA committee.

```yaml
tosca_definitions_version: tosca_simple_yaml_1_3

node_types:
  radon.nodes.OpenStack.VM:
    derived_from: tosca.nodes.Compute
    properties:
      name:
        type: string
      image:
        type: string
      flavor:
        type: string
      network:
        type: string
      key_name:
        type: string
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        operations:
          create:
            implementation: playbooks/create.yaml

policy_types:
  radon.policies.scaling.ScaleOut:
    derived_from: tosca.policies.Scaling
    properties:
      cpu_upper_bound:
        description: The upper bound for the CPU
        type: float
        required: false
        constraints:
          - greater_or_equal: 80.0
      adjustment:
        description: The amount by which to scale
        type: integer
        required: false
        constraints:
          - greater_or_equal: 1
    targets: [ radon.nodes.openstack.VM ]
    triggers:
      radon.triggers.scaling:
        description: A trigger for scaling
        event: trigger
        target_filter:
          node: radon.nodes.openstack.VM
        action:
          - call_operation:
              operation: radon.interfaces.scaling.scale
              inputs:
                adjustment: { get_property: [ SELF, adjustment ] }

interface_types:
  radon.interfaces.scaling:
    derived_from: tosca.interfaces.Root
    operations:
      scale:
        inputs:
          adjustment: { default: { get_property: [ SELF, name ] } }
        description: Operation for scaling.
        implementation: playbooks/scale_out.yaml

topology_template:
  node_templates:
    vm1:
      type: radon.nodes.OpenStack.VM
      properties:
        name: HostVM
        image: centos7
        flavor: m1.xsmall
        network: provider_64_net
        key_name: my_key

  policies:
    test:
      type: radon.policies.scaling.ScaleOut
      properties:
        cpu_upper_bound: 90
        adjustment: 1
```

Figure 26. An example of the proposed TOSCA scaling approach

4.2 Blue/Green deployment and Canary testing

Nowadays applications and services are running continuously, and cannot be stopped even when the updates are rolled out. It is known that updating the applications is a critical operation due to the possibility of new bugs or other technical or non-technical issues connected with the update. For example, the update can provide functionality that users do not like or remove some functionalities that users used more frequently. To limit the surface of impact, software vendors create partial rollouts that affect a subset of the users and if the feedback is successful, they create a full rollout of the application. The techniques associated with those principles are called *canary testing, dark launch and Blue/Green deployment.*

One Blue/Green deployment testing example for the SaaS orchestrator component would be to run the two versions of the prepared TOSCA CSARs at the same time. This means that we will have two versions of the same applications. To realize the test we can use the CI/CD job that would run the two deployments in parallel. Figure 27 shows an example of the Jenkinsfile. In the environment, we define the SaaS API endpoint, already created workspace name and the path to the two blueprints. The username and password for the SaaS API would be retrieved from the Jenkins credentials. In the example we were using the todolist CSAR. The CI/CD job first prepares the base64 versions of the two CSARs and then uses the environment variables to login to the SaaS API. Then the two projects, each for the different version of the CSAR, are created. The last thing are the two deployment processes which need to be deployed simultaneously. For that, we used two background shell processes here.

```
pipeline {
  agent none

  environment {
    SAAS_API_ENDPOINT = "https://xopera-radon.xlab.si/api"
    USERNAME = credentials('saas-username')
    PASSWORD = credentials('saas-password')
    WORKSPACE = "test"
    TODOLIST_CSAR_V1 = "ServerlessToDoListAPI_v1.csar"
    TODOLIST_CSAR_V2 = "ServerlessToDoListAPI_v2.csar"
  }

  stages {
    stage('Deploy two CSARs in parallel with the SaaS orchestrator') {
      steps {
        sh '''
          # prepare todolist CSAR v1
          TODOLIST_CSAR_V1_BASE64="$(base64 --wrap 0 ${TODOLIST_CSAR_V1})"

          # prepare todolist CSAR v2
          TODOLIST_CSAR_V2_BASE64="$(base64 --wrap 0 ${TODOLIST_CSAR_V2})"

          # login with SaaS credentials
          alias cookiecurl="curl -sSL --cookie-jar cookiejar.txt --cookie cookiejar.txt"
          response="$(cookiecurl ${SAAS_API_ENDPOINT}/credential)"
          redirect_url="$(echo $response | xmllint --html --xpath "string(//form[@id='kc-form-login']/@action)" - 2>/dev/null)"
          cookiecurl "$redirect_url" -d "username=$USERNAME" -d "password=$PASSWORD" -d credentialId=""

          # create projects
          cookiecurl "${SAAS_API_ENDPOINT}/${WORKSPACE}/1/project" -XPOST -d "{\"name\": \"todolist_project_v1\", \"csar\": \"${TODOLIST_CSAR_V1_BASE64}\"}"
          cookiecurl "${SAAS_API_ENDPOINT}/${WORKSPACE}/1/project" -XPOST -d "{\"name\": \"todolist_project_v2\", \"csar\": \"${TODOLIST_CSAR_V2_BASE64}\"}"

          # deploy projects
          cookiecurl "${SAAS_API_ENDPOINT}/${WORKSPACE}/1/project/1/deploy" -XPOST -H "Content-Type: application/json" -d @inputs-request.json &
          cookiecurl "${SAAS_API_ENDPOINT}/${WORKSPACE}/1/project/2/deploy" -XPOST -H "Content-Type: application/json" -d @inputs-request.json &
          '''
      }
    }
  }
}
```

Figure 27. An example of the proposed Jenkinsfile for the parallel CSAR deployment

4.3 Security tools for RADON applications

Applications and services provide high quality services to the end users only if they can be secured from malicious attacks that could harm users or limit the application availability. This means that applications need to be monitored and the suspicious behavior requires special attention with manual or automatic response. The application creators are frequently aware of their potential security breaches and how to handle the mitigations procedure, what they lack is tools for fast response. Within the RADON environment we put a focus on this matter and try to understand what needs to be done with applications under attack to limit the potential damage. The usual steps making application more secure are:

- Preventive actions:
  - resilience
    - forcing security updates
    - using up-to-date applications
    - forcing appropriate configurations on networking and applications
    - confirm that all resilience actions are successfully applied.
  - redundancy
    - running services in parallel in different zones, etc.
  - stress testing
    - load test and penetration tests
  - Monitoring/detection
    - active monitoring and analysis tools, deep packet inspection
- Curative actions:
  - Identification:
    - detecting and confirming the DDoS attack
  - Diversion and load balancing
    - Traffic mitigation, moving traffic to other points using DNS.
  - Filtering load
    - Filtering the real requests from the malicious.

Inside RADON we provide tools that contribute to the above list of actions. For example, the whole set of preventive actions RADON users can achieve through IaC, CTT, CI/CD and monitoring tools. Using Ansible playbooks in modules allows re-running the deployment with the switch "--clean-state" in a way to re-check and enforce the desired state of the applications again. This makes applications more up-to-date and resilient. Monitoring tools and CTT cover stress testing and detection while CI/CD can be configured in a way to provide the redundancy of the application. All those actions can be addressed in an application design and preparation step.

RADON tools that can be exploited for achieving curative actions are monitoring for detecting and confirming an attack. The diversion, load balancing and filtering of the load can be also achieved

with the RADON Monitoring and IaC tools. Diversion, load balancing and filtering is done by the application (web) router configuration, which can be another IaC task changing the configuration of the application.

Hello world example

Let's imagine that we have up and running the thumbnail generator example, which was deployed to AWS cloud with the TOSCA orchestrator and it uses AWS cloud resources like Lambda for resizing the images, S3 bucket for storage and so on. The application and its parts are exposed to the other world through the proxy server so that users can upload their images and receive back generated thumbnails of different sizes. This cloud application is running and serving well to the needs of users until the DDoS attack occurs - a large number of pictures were uploaded from one connected service under attack, which causes that all of the services get unavailable for the users.

When the application owner realized that his services are unreachable, he thought that AWS function service may be too busy, so he first tried to expand the application with scaling the FaaS image-resize function allowing the more RAM and extended time of execution. As this did not help, the owner then decided to create another application in parallel and move users (or other services that rely on this one) to new one with new buckets, etc. He was still unable to solve the issues. Finally, he used the AWS Shield resource that detected DDoS attack and he was able to overcome it. To prevent these events in the future, he wanted to protect the application blueprint with the proper security, so that he will be warned in case of abuses automatically and that the services will be able to respond quickly when the attack takes place.

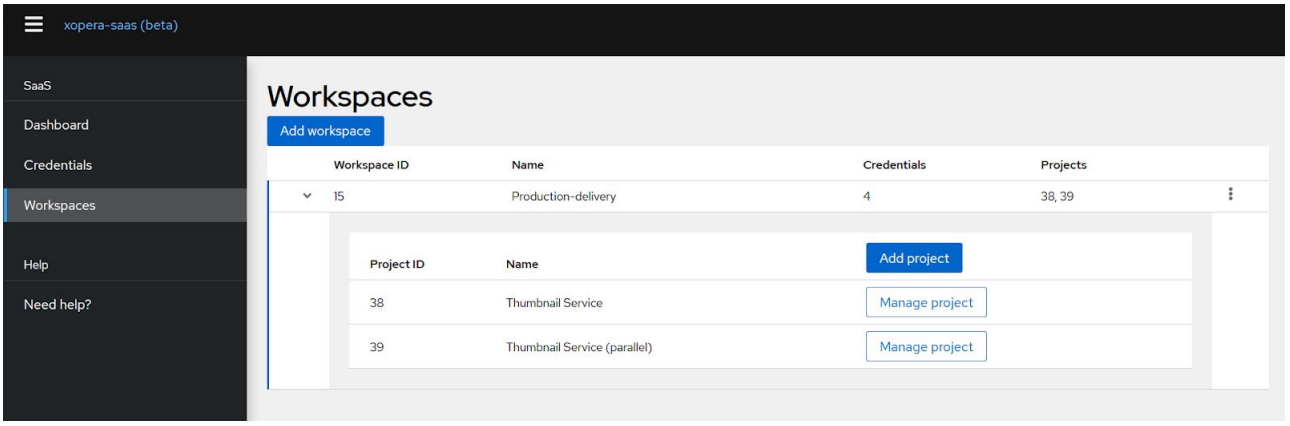


Figure 28. A list of user's workspaces and projects in xOpera SaaS UI

A good measure of security can be achieved by combining different RADON tools such as the SaaS orchestrator and monitoring. When the orchestrator deploys the application blueprint it can configure monitoring based on the TOSCA policies that were supplied within the TOSCA service template. Users can, for example, configure multiple scaling policies (e.g. scaling up, scaling down, auto-scaling) and other security policies (e.g. DDoS policy) and connect them with the proper TOSCA triggers that will invoke the security scripts (i.e. Ansible playbooks) which can take actions to minimize the attack damage by taking actions in the cloud. Triggers can be set to invoke

only when certain thresholds are reached and these can be obtained from monitoring. These policies can be then configured in a way that the SaaS TOSCA orchestrator will call their trigger operations when the monitoring notices that there might be an attack going on. So after the deployment of the application CSAR, the Prometheus monitoring would be configured through the SaaS orchestrator based on the defined policies. This means that the application owner could for instance constantly monitor his services (e.g AWS lambda and S3 bucket) by retrieving the CPU load and other parameters. The orchestrator would then have to have a notify command which would receive a JSON from Prometheus monitoring tool with the measured params and would then check if the thresholds for the policies are reached. And if this happens the linked scripts can be called so that the application and its resources can be scaled up if the attack occurs.

Moreover, in case of DDoS it would be even possible to measure the TOSCA trigger invocations and if we had to scale up the application too many times, this can be a signal that an attack is going on an we can then call another trigger that would protect our resources on the cloud even further. So, for our user when scaling of AWS Lambda and S3 bucket won't be enough, we could also configure AWS Shield DDoS protection to stop the attacks. To notify the user what is going on, each one of these scripts can send him an e-mail or a message so that he can later manually check if everything is fine.

5 Conclusions and Future Work

This deliverable presents an overview of the current status of the Runtime environment and all the tools that are integrated in the RADON framework. The document presents all advances and improvements of each tool in detail. However, to have a compliance overview on one place, Table 4 shows an overview of the level of fulfillment for each of the agreed requirements, some are discussed in detail after the table. The labels specifying the "Level of fulfillment" are defined as follows:

(i) ✗ (unsupported): the requirement is not fulfilled by the current version

(ii) ✔ (partially-low supported): a few of the aspects of the requirement is fulfilled by the current version

(iii) ✔✔ (partially-high supported): most of the aspects of the requirement is fulfilled by the current version

(iv) ✔✔✔ (fully supported): the requirement is fulfilled by the current version.

Table 4. Achieved level of compliance to RADON requirements

Id	Requirement Title	Priority	Resolution	Level of compliance
R-T5.1-1	The orchestrator tasks must be executable through CLI	SHOULD_HAVE	Achieved at start.	✔✔✔
R-T5.1-2	A user describes the application architecture and dependencies at least in TOSCA YAML 1.2	MUST_HAVE	Orchestrator supports TOSCA v1.3	✔✔✔
R-T5.1-3	The runtime toolchain need to provide a status on each stage of deployment of the application on the underlying architecture	MUST_HAVE	"opera info" provides the status	✔✔✔
R-T5.1-4	At the end of deployment the deployment of services needs to be verified and its dependencies	MUST_HAVE	Verification of TOSCA + orchestrator finishes successfully.	✔✔
R-T5.1-5	The orchestrator should be able to calculate the diff between the current state and the desired state expressed by a new model version and redeploy only the difference.	SHOULD_HAVE	Simple diff is available in GMT, but not in the orchestrator - this will be addressed in the next period.	✔
R-T5.1-6	Support of FaaS deployment to OpenFaas	MUST_HAVE	Example available.	✔✔✔
R-T5.1-7	Support of FaaS deployment to AWS cloud platform	MUST_HAVE	Example available.	✔✔✔

R-T5.1-8	The xOpera command line interface needs to have a dry run mode to verify changes without asking for input in execution	COULD_HAVE	Validate command checks the CSAR	✔✔✔
FR-T5.1-9	xOpera orchestrator should be available as a service/RADON SaaS component	COULD_HAVE	SaaS version is available.	✔✔✔
FR-T5.1-10	TOSCA orchestrator could provide a packaging command to prepare CSAR files	COULD_HAVE	Currently unsupported, also no request by users.	✗
FR-T5.1-11	xOpera should be able to parse get_artifact TOSCA function	SHOULD_HAVE	Supported since version 0.6.2.	✔✔✔
FR-T5.1-12	Orchestrator should consider each deployment separately	SHOULD_HAVE	Supported automatically with SaaS and manually with CLI.	✔✔✔
FR-T5.1-13	Orchestrator could support using ssh private keys	COULD_HAVE	Supported.	✔✔✔
R-T5.1-10	When modelling a FaaS, the user can select a specific function by referencing the location from Function Hub	COULD_HAVE	Function Hub artefacts are accessible by URL.	✔✔✔
R-T5.2-8	Support of FaaS_deployment to Google cloud platform	COULD_HAVE	Example available. TPS/Particles to be updated (D5.4), ok for orchestrator.	✔✔✔
R-T5.2-9	Support of FaaS deployment to Azure cloud platform	MUST_HAVE	Example available. TPS/Particles to be updated (D5.4), ok for orchestrator	✔✔✔
R-T5.2-10	Support deployment to regular VMs	MUST_HAVE	TPS/Particles to be updated (D5.4), ok for orchestrator	✔✔
R-T5.2-11	Support deployment to microservices architecture	MUST_HAVE	TPS/Particles to be updated (D5.4), ok for orchestrator	✔✔
R-T5.3-1	The TOSCA blueprint needs to be able to support the definition of security and privacy policy of specific serverless/FaaS provider.	MUST_HAVE	Included in Function/bucket config	✔✔
R-T5.3-2	The tool must be able to configure automatic scaling of the deployed components based on the auto scaling policies defined in the RADON models.	SHOULD_HAVE	Some ways of scaling are already applicable.	✔✔
R-T5.3-3	The tool must be able to support configuring AWS EC2 auto scaling service based on the TOSCA auto scaling policy.	SHOULD_HAVE	Supports Lambda configuration, TPS/Particles to be updated (D5.4), ok for	✔✔

			orchestrator	
R-T5.3-4	The tool should be able to support configuring automatic scaling of Docker services based on TOSCA auto scaling policies.	SHOULD_HAVE	Not finished yet.	✔
R-T5.3-5	The TOSCA blueprint must be able to enable users to define the usage of different FaaS/Serverless providers for different parts of their application.	MUST_HAVE	Examples available. TPS/Particles to be updated (D5.4), ok for orchestrator	✔✔✔

As it is presented in Table 4 a substantial amount of the requirements have been achieved during the Y2 also some of them still need further attention in the last, Y3 period. One of the *should have* with current low-support is definitely the R-T5.1-5 - "diff" command that should provide differences between current and desired state. The reason why this requirement did not receive much attention yet is two-fold. First, a basic diff is already provided in the GMT, where a user can check the differences between two versions of the same application; second, this is a *day two* operation and for the beta release we were mostly aiming to equip users with the day one operations to start using RADON. The next one is verification of the deployed software (R-T5.1-4) that can be covered by the multiple tasks as *opera validate* command to check CSAR, successful opera deployment to execute all commands and using policies and monitoring to initiate monitoring on the application resources. There is also a set of requirements (R-T5.2-10,R-T5.2-11, R-T5.3-1, R-T5.3-2) will be addressed more in detail together with the Template Library deliverable (D5.4) and abstraction layers.

5.1 Future work

The deliverable presents the improvements of runtime environment and delivery tools over Y2 of the project. As planned, the tools are integrated and examples are provided. The additional work will mainly include bug fixes, support and adaptation of tools to RADON users to become more user friendly.

However, even though the tools became better and more advanced during the Y2, the wish lists and ideas for new requirements are emerging everywhere, which is good, because that means that we are on a good path and the interest in our tools is rising. Therefore, beside working on the tasks that will allow us to make more ticks on the requirements above, we will focus on making delivery toolchain and runtime environment more mature and raising the TRL on tools that we develop and on their integration. One crucial part of the future work will be improving the orchestrator and the Template Library as they generated interest with the TOSCA Committee members. The orchestrator was already presented once in an ad-hoc TOSCA meeting and we have another invitation for presenting the orchestrator and Template Library. This time there will be a kick-off meeting among all TOSCA orchestrator creators with a goal to create a standard set of TOSCA template examples to be shared among developers.

Based on the collaboration with TOSCA community, we look forward to evolving and co-creating the (auto)scaling functionality further. Currently, the TOSCA standard already provides some information about scaling features. However, the standard does not explicitly describe how the orchestrators and other tools should invoke or use them. The TOSCA standard itself states that scaling workflows is something that will be further worked in the future. This is also tightly connected to RADON's scaling implementation and research since the scaling progress within TOSCA Simple Profile in YAML version 2.0 will be followed.

6 References

[RadD2.1] RADON Consortium, “Deliverable D2.1 - Initial requirements and baselines”, 2019

[RadD2.3] RADON Consortium, “Deliverable D2.3 - Architecture and integration plan I”, 2019

[RadD5.1] RADON Consortium, “Deliverable D5.1 - Runtime Environment 1”, 2019

[RadD5.3] RADON Consortium, “Deliverable D5.3 - Technology Library 1”, 2020

[Can20] Cankar M., Luzar A., Tamburri D.A. (2020) Auto-scaling Using TOSCA Infrastructure as Code. In: Muccini H. et al. (eds) Software Architecture. ECSA 2020. Communications in Computer and Information Science, vol 1269. Springer, Cham. https://doi.org/10.1007/978-3-030-59155-7_20