H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable 5.1

Runtime Environment 1

Version: 1.0

Publication Date: 19-December-2019

Deliverable Card

Deliverable	5.1
Title:	Runtime Environment 1
Editor(s):	Matija Cankar (XLAB)
Contributor(s):	Anestis Sidiropoulos (ATC), Hans Georg Næsheim (PRQ), Mainak Adhikari (UTR), Satish Srirama (UTR), Matija Cankar (XLAB), Kristian Žarn (XLAB), Vladimir Yussupov (UST)
Reviewers:	Michael Wurster (UST), Domenica Presenza (ENG)
Type:	Report
Version:	1.0
Date:	19-December-2019
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

The aim of this deliverable is to provide a technical overview of the RADON tools that comprise the Runtime Environment. Through the review, users will learn the current year one (Y1) status of each tool and understand the technical details that are required for RADON.

The status of each tool is presented with the description of tool, listing the interface options, and listing of basic and required features. The tool capabilities are also demonstrated with a simple thumbnail generator example, which is our FaaS toy example for such demonstration. Further, the tools are evaluated by their current maturity and level of compliance with RADON requirements. Finally, each tool also has a short description about the future plans and immediate issues to be tackled.

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

The Runtime environment deliverable presents the RADON toolset that bridges the Dev part, where applications are developed and configured, to the Ops part, where the testing, staging and the whole application lifecycle is instantiated. From the RADON actors perspective, the Runtime environment uses the outputs from the tools used mainly by Software designer and Software developer and equips Operation engineer, Release manager and QoS engineers with necessary tools for their daily work.

This deliverable provides a more detailed and technical overview of the delivery toolchain tools and supporting tools introduced in D2.1 and D2.3 with a goal to present the current state - in this document referred to as year one (Y1) - of each component and provide the basis for the detailed specification of the development and integration for year two (Y2).

1.1 Deliverable objectives

The main objective of the deliverable is to present the current status of the work and research done in Runtime environment technologies. For each technology or idea we choose a particular instantiation as a tool already available or a tool that still needs development to fulfil the RADON Runtime toolchain environment.

Objectives of this deliverable are:

- Present the general picture of RADON Runtime environment and delivery toolchain.
- For each tool provide a particular example and describe:
    - Currently available interfaces.
    - Review the requirements/complexity of handling with the tool (installation, setup, configuration)
    - Maturity level:
        - Which RADON requirements can be already fulfilled with the tool.
        - Which tool interfaces are already sufficient for the RADON development
    - Demonstrate (if possible) how the tool can be used with RADON toy example, e.g., thumbnail generator.
    - Future plans and expectations for Y2.
- Populate the WP5 requirements table with achieved levels of compliance.

1.2 Overview of all achievements

The main achievement evident from this deliverable is the progress from the architectural planning to the concrete development and integration state. The tools and technologies presented in previous deliverables as (D2.3) are evaluated through detailed inspection of the available functionality and the examples of usage like for example the xOpera orchestrator in section 3.4 Orchestrator. The descriptions of the tools are described in detail to show all missing functionality that still needs to be developed, like for example the Template library in section 3.2 Template library was not deployed as it needs to be developed first. Therefore a detailed description was included.

This presents our status on the *RADON field* where all the implementation will be done. This status report was achieved by all tool owners that deployed, installed and configured their tools and prepared the instructions on how this can be done and which are the current limits to integrate them in RADON.

1.3 Structure of the document

Section 2 presents an overview of the Runtime environment and the high level architecture. At the end of the section, two requirements of the Runtime environment are stated and the methodology of determining a tool maturity level is presented.

Section 3 presents tool by tool integrated in the Runtime environment and explains how the tools are used, which technical skills are required to manage this tool and how this tool can be currently tested in action on particular example. Where needed, the section invites the reader to read more technical readmes from the Internet or walkthroughs from the Appendix of this document.

Section 4 concludes the document with the requirement compliance table and future work.

As already mentioned, Appendix includes detailed installation and configuration descriptions that were too detailed for section 3.

2 Runtime environment

In this document we describe the overall runtime environment of the RADON framework. A preliminary presentation of the runtime tools, planned to be developed in RADON, has been provided in D2.1 [RadD2.1]. The main focus of this section is to describe the overview of the runtime tools and how the tools interact with each other in order to meet the objectives of the project. In particular, Section 2.1 provides an overview of the runtime tools and how the tools are connected with RADON IDE.

2.1 Overview of Runtime Tools

The RADON framework provides a set of components that realize a set of runtime tools for application development and deployment. Table 1 lists the set of runtime tools with brief descriptions.

Table 1. Overview of Runtime RADON components

Name	**Description**
Monitoring System	A back-end system to collect the resource utilization from the runtime environment for supporting the quality assurance of the system including auto-scaling and security.
Template Library	A central point for retrieving and storing the application artifacts as modules, blueprints, and FaaS abstraction modules.
Delivery Toolchain	A set of tools for software delivery and inherits the functionalities from the particular enclosed tool.
Orchestrator	A tool for putting an application on the run-time environment by enforcing the state described by the application blueprint (CSAR) onto the target provider.
Data Pipeline Orchestration Module	A module that extends the orchestrator with the capability to control the life cycle of data pipelines, introducing the ability to automate the movement and transformation of data.
Continuous Integration	A tool for managing the code, functions and configuring together and finally test it.
Continuous Deployment	A tool that deploys the new application version in staging or production environment.
Function Hub	A public repository for open-sourced contribution to reusable functions.

2.2 High level Architecture

In the runtime environment, initially, the RADON user employs the Graphical Modelling Tool (GMT) to create a new application blueprint in the form of a TOSCA topology template. The CSAR contains everything that is required for deployment including all business component artifacts (e.g., FaaS function code) for deployment execution. For modeled application components, the respective source code project can be opened in the RADON IDE.

The template library is a repository of application runtime management definitions including the blueprints, high-level system abstractions, application abstractions (including data pipeline components) and TOSCA language extensions. The template library is responsible for application planning, CSAR generation, and blueprint design. During runtime, the blueprint or its parts could be designed separately. Here, it is assumed that a RADON user would like to support new functionalities or new providers that require new blueprint artifacts or blueprints that are not created only for deployment but rather as an example or closed workflow to use.

The orchestrator puts the application into the runtime environment by enforcing the state described by application blueprint (CSAR) onto the targeted cloud/FaaS provider. The common operations are application deployment, resource scaling, and un-deploy. At first, the orchestrator interprets the CSAR and enforces its blueprint on the infrastructure. Beside pure deployment of the application, orchestrator must be able to configure triggering events for FaaS and setting up the monitoring metrics. In addition, the data pipeline orchestration module of the tool controls the life cycle of data pipelines, introducing the ability to automate the movement and transformation of data. Next, to support the application flow and scalability of the computing resources in the form of CPU and memory, the events need to be configured and appropriate monitoring service needs to be subscribed in the orchestrator.

The resource monitor component is mainly responsible for monitoring the computing resources in each time instance (e.g. every minute). Here, we use Prometheus and Grafana as resource monitors, which are described in detail in Section 3.1. The orchestration scale-up or scale-down the computing resources or applications after receiving an alert signal from the monitoring system.

Finally, when the application lifecycle reaches the endpoint, the application instances are removed from the environment. The overall runtime workflow of the applications is shown in Figure 1.

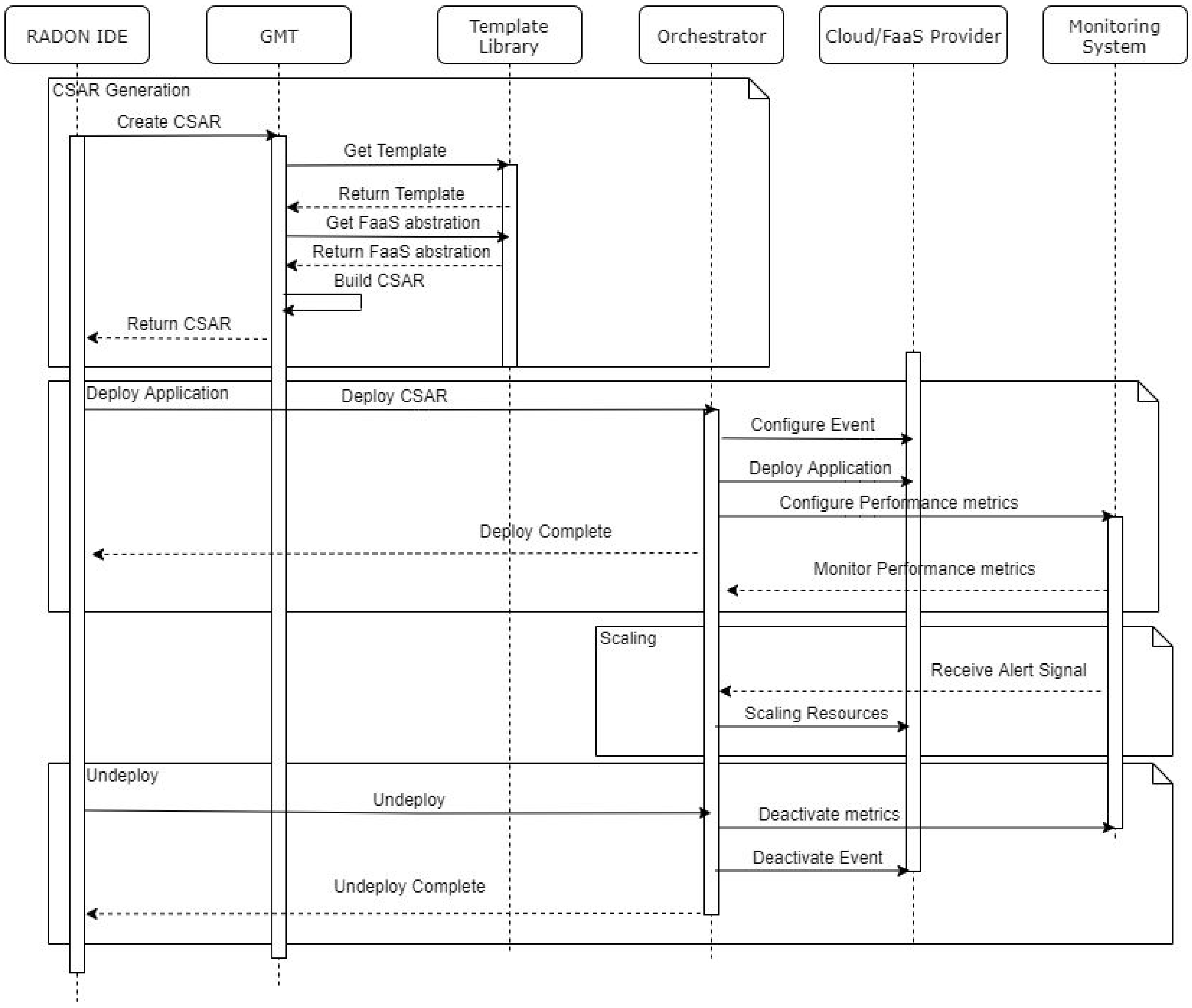


Figure 1. Runtime Tools integration

2.3 Methodology and Requirements

To review the current state of the tools in the Runtime environment, the following methodology was chosen. First, for each tool we present a short description and if possible demonstrate how the tool is deployed, configured and used in case of the thumbnail generator - the RADON toy example. These technical details provide us an important insight of the complexity, compatibility and current tool maturity. To understand the compliance with the final RADON results, for each tool we determine compliance with the requirements from Table 2. As the project is currently in year one (Y1), many tools will still have unfulfilled requirements.

The tool reviews, maturity levels and compliance levels will serve the WP5 tasks to prepare a detailed specification and work plan for Y2. As the goal of Y1 is to understand how each piece of the runtime environment and deployment toolchain works individually and interfaces with other pieces, in Y2 we would like to achieve the automated deployment and move closer to automated configuration of all tools. Achieving this automation will allow us easier deployment of the whole RADON environment and easier preparation of the toolchain for each application.

Table 2. List of Runtime environment requirements.

Id	Requirement	Priority
R-T5.1-1	The orchestrator tasks must be executable through CLI	Should have
R-T5.1-2	A user describes the application architecture and dependencies at least in TOSCA YAML 1.2	MUST_HAVE
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
R-T5.1-4	At the end of deployment, the deployment of services needs to be verified along with its dependencies	MUST_HAVE
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.	MUST_HAVE
R-T5.1-6	Support of FaaS deployment to OpenFaas	MUST_HAVE
R-T5.1-7	Support of FaaS deployment to AWS cloud platform	MUST_HAVE
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
R-T5.1-9	When modelling a FaaS, the user can select a specific function by referencing the location from Function Hub	COULD_HAVE
R-T5.2-1	Support of FaaS_deployment to Google cloud platform	COULD_HAVE
R-T5.2-2	Support of FaaS deployment to Azure cloud platform	MUST_HAVE
R-T5.2-3	Support deployment to regular VMs	MUST_HAVE
R-T5.2-4	Support deployment to microservices architecture	MUST_HAVE
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
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.	MUST_HAVE
R-T5.3-3	The tool must be able to support configuring AWS EC2 auto scaling service based on the TOSCA auto scaling policy.	MUST_HAVE
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
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

The maturity of each tool will be discussed in the section describing the tool itself. Overall compliance level of the tools with the presented requirements is included at the end of the document.

Presented runtime environment and included tools have different levels of maturity as some of them are already developed and RADON is only exploiting their functionality and others are yet to be fully developed. To demonstrate the current ability of the tools, each tool has been used for managing the toy example - the thumbnail generator application. This hands-on exercise done by the tool owners, provides a clear understanding what the purpose of each tool is and, in the context of RADON's application deployment workflow, precisely presents what are the current shortcomings.

3 Tools

This section presents Runtime environment tools focusing on interfaces, installation, setup, current maturity level, appliance to the thumbnail generator example and future work.

3.1 Monitoring

3.1.1 Description and interfaces

Monitoring comprises an integral part of the RADON runtime environment, since it aims to provide monitoring, logging and potentially event notification capabilities for supporting quality assurance and assessment. In this section, we describe this system and the context of use in the RADON project.

The Monitoring system is a collection of existing open source tools that are integrated together in RADON to facilitate functions with respect to the acquisition, processing and visualisation of metrics from several layers of the infrastructure, aiming at the continuous observation of the progress and quality of an application. The Monitoring system is designed to be agnostic of the platform on which the monitored application is deployed and thus the adopted approaches are:

- Monitoring of General Metrics. Based on this approach the metrics are collected on:
    - Virtual Machine level (CPU, Memory)
    - Docker Container level (CPU, Memory)
    - FaaS level (Duration, Invocations, Errors, Throttles, Global Concurrent Executions)
    - Application level (CPU, Memory, I/O Traffic)
- Monitoring of Specific Metrics. Based on this approach, the metrics are collected only on application level. In order for this level of monitoring to be feasible, code snippet injection in the source code of the application is a mandatory prerequisite. The purpose of the injected code is to define, collect and expose these application specific metrics to be monitored.

Since the monitoring system is performed on several layers, the following software units are embedded and configured in order to interface and report metrics on the visualization dashboard of this Monitoring System:

- Docker, a set of “platform as a service” products that use OS-level virtualization to deliver software in packages called containers. Containers are lightweight, isolated from one another and bundle together their own software, libraries and configuration files. They can communicate with each other through well-defined channels.
- cAdvisor, provides an understanding of the resource usage and performance characteristics of the running containers to the users. The main component is a running daemon that collects, aggregates, processes, and provides the information about running containers. For

each container, it keeps historical resource usage, resource isolation parameters, network statistics, and histograms of complete historical resource usage. This data is exported by a container and is machine-wide."

- Prometheus, a systems' monitoring and alerting toolkit. Prometheus scrapes metrics from instrumented jobs. This can be achieved directly or via an intermediary push gateway for short-lived jobs. Internally it stores all scraped samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts.
- Prometheus Pushgateway, an intermediary service that allows you to push metrics from jobs, which cannot be scrapped. The purpose of invoking the Pushgateway is to cover cases where Prometheus on its own is not sufficient to acquire the metrics e.g. to capture the outcome of service-level batch job.
- Grafana, an open source metric analytics & visualization suite. It is most commonly used for visualizing time series data for infrastructure and application analytics but many use it in other domains including industrial sensors, home automation, weather, and process control.
- Application Configuration, which is a piece of code injected in the source code of the application to expose strictly application coupled metrics.
- Centralized Monitoring Dashboard, a centralized monitoring dashboard UI that combines and visualizes in a dynamic way the metrics acquired from all the layers described above.

In Figure 2, we present the architectural components of the Monitoring system and their orchestration towards providing the expected capabilities for collecting raw monitoring information and distributing them as monitoring metrics and/or events.

The monitoring metrics are collected by cAdvisor on container level, by a node exporter on VM level and by injecting code on the source application code. In case of a FaaS, the metrics are collected from the proprietary Cloud provider monitoring service (for example in case of AWS the monitoring tool is Cloudwatch). Subsequently, the collected metrics are forwarded to Prometheus (either directly or through Prometheus Gateway) and Prometheus is configured via queries to collect and group them. Finally, Grafana is configured (through the action of defining sources) to ingest these metrics from Prometheus or the Cloud Proprietary Monitoring tools and visualize them in the relevant dashboards.

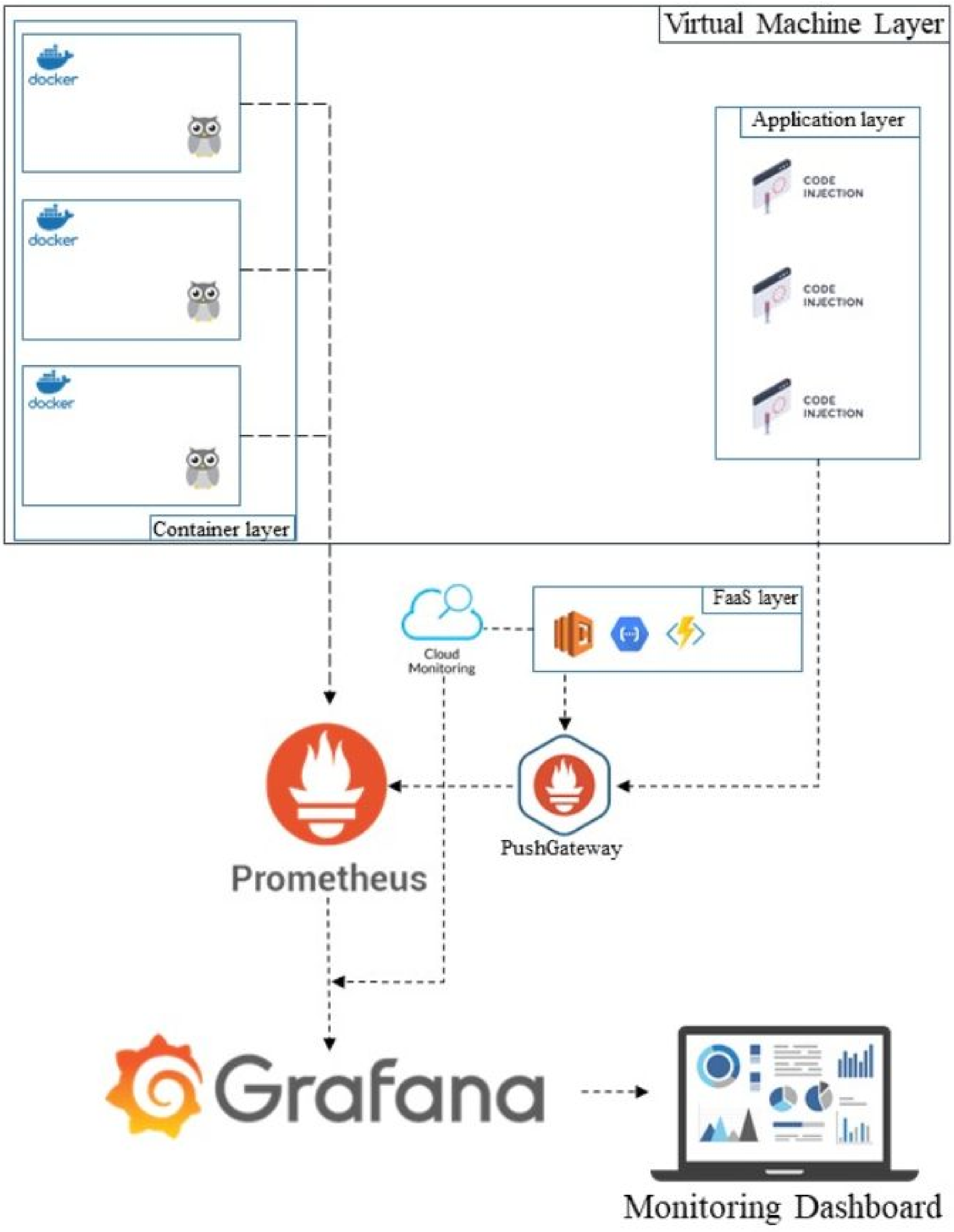


Figure 2. Orchestration/Architecture of the RADON Monitoring System.

3.1.2 Installation and setup

In order for the user to setup and install the monitoring tool, the following expertise is needed:

- AWS console management and EC2 instance configuration (or any other equivalent Cloud provider to setup a Virtual Machine)
- Linux bash
- PromQL query language (or equivalent)
- Docker deployment
- Prometheus setup and configuration (this includes cAdvisor and push Gateway)
- Grafana setup and configuration
- Application under monitoring code competence (in order to insert code snippets)

For the current prototype version of the Monitoring system, the following prerequisites are needed and have to be configured:

- **Docker**, a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.
- **cAdvisor**, which provides an overview and understanding of the resource usage and performance characteristics of the running containers to the container users
- **Prometheus Monitoring**, system monitoring and alerting toolkit
- **Prometheus PushGateway**, allow ephemeral and batch jobs to expose their metrics to Prometheus
- **App Configuration**
    - **Prom Client**, in order to export the application specific ,etrics
- **Grafana Dashboards**, which is a software for visualization, alerts management and dashboards creation

Detailed instructions on how to setup and configure the Monitoring System are available in the Appendix: Walkthrough: General purpose Monitoring Tool Setup

3.1.3 Current maturity level

The current state of the Monitoring System provides a variety of metrics to be monitored, such as: RAM, CPU, Traffic, concurrent executions, execution time, cost, etc.

Manual configuration is needed as well to make all the used components to work with each other. Current level of maturity fulfills a subset of requirements expressed by the RADON requirements.

Requirements that remain to be fulfilled in future releases of the Monitoring System are:

1. Alert generation when specified metrics thresholds are exceeded;

2. Deployment of the Monitoring System in all the Cloud Providers;
3. Enhancement of the Monitoring System in order to fully interface with the other RADON Tools. In future releases, other RADON tools will be able not only to acquire and expose monitoring metrics but also filter them by time unit and also aggregate them.

Future releases of this system will include a wider range of possible metrics to monitor, unified User Interface for monitoring dashboards and higher level of automation in deploying and configuring the system.

3.1.4 Toy example demo

In this section, the application/configuration of the Monitoring System prototype on the Toy example is described. A basic prerequisite is that the toy application can be deployed on any of the supported serverless computing platforms. In this specific case, AWS Lambda function (and the Amazon cloud infrastructure in general) has been selected.

The monitoring metric pipelines are described in the Figure 3:

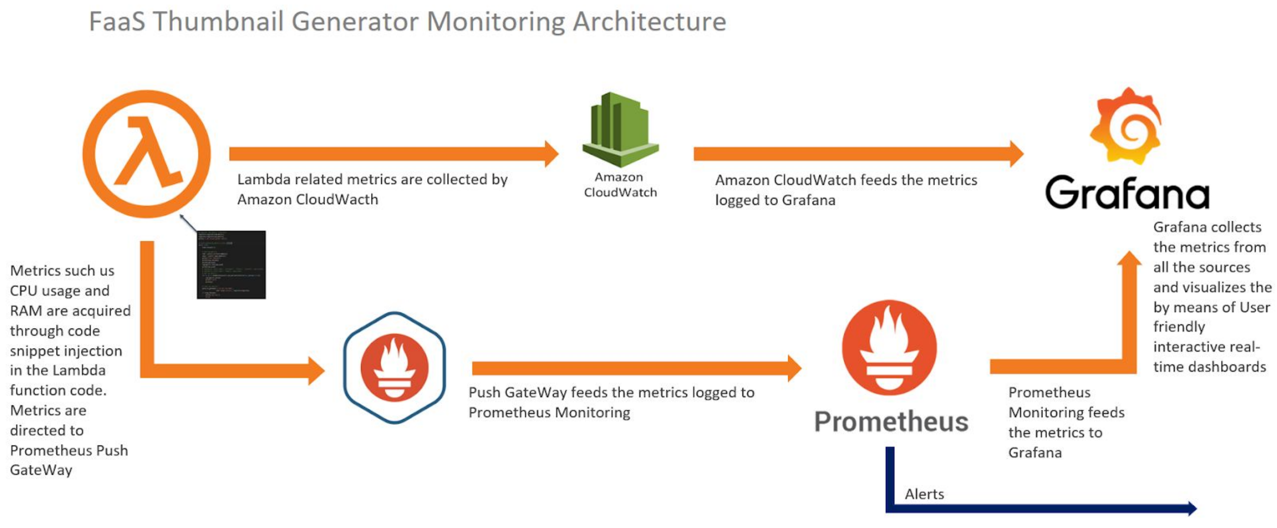

FaaS Thumbnail Generator Monitoring Architecture
Lambda related metrics are collected by Amazon CloudWacth
Amazon CloudWatch
Amazon CloudWatch feeds the metrics logged to Grafana
Grafana
Grafana collects the metrics from all the sources and visualizes the by means of User friendly interactive real-time dashboards
Metrics such us CPU usage and RAM are acquired through code snippet injection in the Lambda function code. Metrics are directed to Prometheus Push GateWay
Push GateWay feeds the metrics logged to Prometheus Monitoring
Prometheus
Prometheus Monitoring feeds the metrics to Grafana
Alerts

Figure 3. Toy example monitoring approaches.

Detailed instructions on how to setup and configure the Monitoring System are available in the Appendix: Walkthrough: Toy Example Monitoring Tool Setup

3.1.5 Future plans

The next steps include the finalization of the tool to include all possible metrics that can be monitored and provided in a wide collection of dashboards. Furthermore, the Monitoring System will also be able to trigger the generation of user specified alerts. This is feasible either through Prometheus monitoring or Grafana Dashboards. The principle for setting up this alert based system is to define rules that trigger the sending of alerts to users when specified metrics thresholds are exceeded.

In addition, investigation is ongoing on possible interactions of the tools comprising the Monitoring System with other tools of the RADON arsenal. The goal is to adapt the tool in order to be able to interface with other tools. In the current RADON architecture (as reported in Deliverable D2.3 [RadD2.3]), the following tools are considered:

- Continuous Testing Tool
- Data Pipelines Tool
- Decomposition Tool
- Eclipse Che (IDE)

The aforementioned interactions are bidirectional, meaning that

- The Monitoring System will provide dataset towards other tools (e.g. Decomposition Tool in order to identify vulnerabilities)
- The Monitoring System may receive metric related data from other Tools (e.g. CTT) in order to create new dashboards.

Finally, Grafana is quite flexible and the user can easily generate graphs and most importantly, dynamically embed them in any Web based application (e.g. IDE).

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 template library with current tools, while a more detailed description of the content will be in the deliverable *D5.3: Technology Library*.

3.2.1 Description and interfaces

Essentially, the template library is a self-contained set of artifacts that enables modeling and orchestrating the deployments of various microservice set-ups including containerized and FaaS-based applications together with data and network flows. To support modeling and orchestration, the library must provide (i) *a set of modeling constructs* called RADON particles that cover all required component types such as provider-specific FaaS-hosted functions, e.g., AWS Lambda or OpenFaaS, (ii) *a set of artifacts supporting orchestration* of these modeling constructs, i.e., the so-called FaaS abstraction layer, and (iii) *a set of blueprints* that, essentially, are modeled, orchestrator-ready example applications representing a spectrum of typical RADON use cases. In the following, we briefly describe the purpose of template library's constituents.

RADON particles.

RADON particles is a set of TOSCA modeling constructs enabling specification of the application components required by RADON use cases such as microservice and FaaS-based applications and data pipelines. The first version of RADON particles was defined in the scope of the deliverable D4.3 - RADON Models I; it is open source and is hosted on GitHub[1].

FaaS abstraction layer.

FaaS abstraction layer is a set of configuration management scripts allowing to orchestrate the deployment for components represented by RADON particles. For example, to successfully deploy a FaaS function to AWS Lambda, a set of actions have to be performed by the orchestrator. FaaS abstraction layer provides a set of executable Ansible playbooks that support deploying RADON particles-based components.

RADON blueprints.

The purpose of RADON blueprints is twofold: (i) showcase the deployable demo applications that represent use case applications typical for RADON's scope, and (ii) provide a set of boilerplate (but orchestrator-ready) application architectures that can be reused by RADON modelers to implement more advanced use cases.

From the modeler's perspective, the interaction with the template library typically follows a specific workflow. In the following, we elaborate on the aspects of interaction with the template library including such details as the modeling entry points, i.e., how IDE and GMT can facilitate interacting with the template library, which interfaces are needed, and which features, e.g., versioning using Git, can be a part of the workflow.

Interaction entry points.

There are several possible ways to interact with the template library, namely a standalone usage of the library, interacting with it via the RADON IDE, or interacting with the library via the GMT.

A. Standalone Template Library

Since the template library is hosted on GitHub, it can be accessed and used as a regular Git repository. This, however, mostly applies to extending the existing sets of models, configuration management scripts, and blueprints following the recommended GitHub workflow.

B. IDE and Template Library

RADON IDE is based on Eclipse Che and can serve as another entry point for interacting with the template library. The envisioned workflow is to support initialization and synchronization of the template library with the project environment, created in the IDE. As a result, modelers can use the available types directly from the IDE or by accessing them by means of the GMT.

C. GMT and Template Library

Graphical modeling of application topologies is another entry point for interacting with the template library, since all the types and artifacts available in it will be used by modelers to

[1] https://github.com/radon-h2020/radon-particles

construct application topologies in a graphical fashion. This entry point is envisioned to be one of the main ways to reuse the elements from the template library in modeled applications.

Interfaces and features.

1. GMT interfaces

As the main storage mechanism, Eclipse Winery relies on the file-based repository which stores all the available modeling constructs. To interact with the repository, Winery provides graphical user interfaces for managing the repository and for application modeling. Internally, both graphical interfaces rely on Winery's REST API, which can also be used separately. When imported into Winery's repository, the template library becomes available for reuse to modelers. This includes modifying available library elements and creation of graphical models using these elements. As an extension point, Winery's REST API can be also used by other tools to query the information about the available modeling constructs, application blueprints, etc.

2. Envisioned IDE interfaces

RADON IDE is intended to support creating the RADON modeling projects and facilitate communication with the RADON toolchain. As a part of the envisioned workflow, the IDE will support initializing new projects together with the template library made accessible, e.g., for Winery's repository. Additionally, the template library initialized for a project needs to be synchronized with the original template library hosted on GitHub.

3. GitHub interfaces

The interaction with the template library via GitHub relies on the APIs and tooling provided by GitHub and follows the standard GitHub workflow.

4. Versioning of elements in the template library

The file-based repository in Eclipse Winery supports integration with Git, which makes it easy to manage versions of repository elements, revert changes, etc. This also provides a possibility to directly synchronize with the template library in the case if it follows the structure of Winery's repository: it can be cloned from GitHub and used as-is for management and modeling purposes.

3.2.2 Installation and setup

A Git-based repository of a predefined structure that groups TOSCA entities in the corresponding clusters such as "node types" or "service templates" and considers the namespaces is initialized automatically in Eclipse Winery. The installation of Winery is described in-detail on Winery's web-site. The template repository has to be manually imported into Winery, and Git-related operations are currently decoupled from available GUIs. This means that all Git and synchronization operations have to be performed manually.

3.2.3 Current maturity level

The set of RADON particles is described in detail in the deliverable D4.3 “RADON Models I”. The definitions of the described application components, both abstract and provider-specific, are open sourced and available vie GitHub[2]. At this stage, the particles repository contains more than 30 distinct elements. In addition, the particles repository contains two preliminary examples of a modeled application. However, the element definitions are not complete and expected to be changed based on the requirements arising from the use case and other tool owners, e.g., the orchestrator. Moreover, apart from enriching existing particles, new elements will need to be added to support modeling of relevant provider-specific services.

3.2.4 Toy example demo

While there is a preliminary version of the toy example modeled in the particles repository, it is not yet compatible with the orchestrator tool. Since some of the defined particles will be updated and might also change based on the refined requirements, the deployable definitions of the toy example applications will be produced in the next iteration of RADON Models deliverable.

3.2.5 Future plans

As discussed previously, in the future, the template library will combine the refined version of RADON models, corresponding artifacts for orchestrating the deployment of these models, and a set of deployable toy application examples. Additionally, the technical aspects of interacting with the Template Library will be enhanced, e.g., improved versioning, support for integration of several template libraries with different content.

3.3 Delivery toolchain

3.3.1 Description and interfaces

Delivery toolchain represents a unified entry point to configure all components that are part of the application lifetime (CI/CD, orchestrator and monitoring). It also describes how the other standalone tools are used together to provide complete delivery of the application. Continuous integration (CI) tool validates the provided CSAR (Cloud Service Archive) that defines the application from an integration perspective using unit tests or even canary testing. It also provides the functionalities to compile, test and package the application. Continuous delivery (CD) expands on CI and provides the tools for deployment of the application. The main steps in this process are a check of the packaged application, deployment, testing the response in production and update. The orchestrator is an integral part of CI and CD. It puts the application into the runtime environment and enforces the state described in CSAR for testing or production environment. It also configures the monitoring component that provides the crucial information required to trigger events and take action. More general description of the delivery toolchain is given in previous deliverables D2.1

[2] https://github.com/radon-h2020/radon-particles

Initial requirements and baselines [RadD2.1] and D2.3 *Architecture and integration plan I* [RadD2.3].

3.3.2 Installation and setup

At this point every tool of the delivery toolchain is provided as a standalone component and has its own installation and setup instructions that are described in the other sections of this document. The delivery toolchain will need a detailed setup in a sense of connecting the whole application delivery and management pipeline, that will be further investigated and prepared for Y2.

3.3.3 Current maturity level

In terms of functionality the maturity level of delivery toolchain depends on each of the tools it comprises. Maturity level of other tools is described in other sections of this document. A unified setup point that represents a single place for configuration and execution of the involved tools has not yet been realized and it is planned for Y2.

3.3.4 Toy example demo

At this point the toy example makes use of the monitoring, orchestrator and the CI workflow which is an expected result for Y1. Mentioned tools currently work isolated with minimal or no interaction between them and this deficiency will be the focus for Y2. Examples, how each tool works with the toy example is presented in tool related section.

3.3.5 Future plans

Future plan is to provide a realization of a unified configuration management of the delivery toolchain that will allow IDE and other tools to work with only a single entity instead with each intrinsic component individually. To achieve this step by step, the expected interfaces from the architecture deliverable D2.3 will be aligned with current available interfaces presented in this document. The mapping will allow us to understand what is missing and the D5.1 provides us a first overview how the missing parts can be achieved or started to be tackled.

3.4 Orchestrator

3.4.1 Description and interfaces

The orchestrator puts the application into the runtime environment with enforcing the state described by application blueprint (CSAR) onto the targeted provider(s). The common operations are deployment, scaling and cleanup or un-deploy and are executed on different target environment as staging, development and production. More detailed description of the functionalities is given in deliverable D2.3 Architecture and Integration plan. Functionalities are a part of the application lifecycle. The first step is to deploy the application and the last step is to un-deploy it. Scale and update functions can happen anytime during the application lifecycle.

RADON will use xOpera orchestrator, which aims to be a TOSCA-compliant orchestrator with a core design presented on Figure 4. At its core, xOpera orchestrator is composed of:

1. TOSCA parser that transforms input YAML documents into a topology template, and
2. execution engine that executes user-defined operations.

The parser part is pluggable and is designed to allow adding support for different TOSCA versions. At the moment, xOpera orchestrator only supports a subset of TOSCA Simple Profile 1.2. We have a TOSCA Simple Profile 1.3 compliant parser in the works but is not ready for prime time yet.

A topology template is a directed acyclic graph (DAG) of nodes, modeling the dependencies between the nodes. These dependencies are used by the execution engine to determine the processing order.

The execution engine will run operations in parallel if possible. What this means is that the operations on the nodes with satisfied dependencies will be executed concurrently. In the current implementation, this parallel execution is not implemented yet. Instead, we determine the order of the operations by topologically sorting the DAG and executing operations one after another. Note that at the moment, operations defined in the Configure interface of the relationships are not executed. This is also something that will change with the next release of the xOpera orchestrator orchestrator.

And while the parser parts of the xOpera orchestrator only deal with YAML documents, the executor is also responsible for executing other artifacts from the CSAR. Currently, xOpera orchestrator expects to be running at the top level of the extracted CSAR, but this limitation will be lifted in the near future. Another limitation of current implementation is related to secondary artifacts. At the moment, xOpera orchestrator only knows how to handle primary implementation artifacts. All other dependencies are ignored.

When the scheduler part of the executor is improved, xOpera orchestrator will be able to execute multiple operations at the same time. And since those operations are rarely processor and memory intensive, we should be able to run quite a few of them on a single reasonably sized control node. Since xOpera orchestrator uses Ansible as a final effector, and we do have quite some experience with it, our estimate would be that running 10-20 parallel operations should be feasible even on a low-level hardware.

xOpera orchestrator does not offer any service facilities. It is designed to be used from the command-line in interactive mode. But its core is a Python library that can be embedded in other services if so desired. For example, we could create a web interface that would offer another way of managing our deployments.

If the TOSCA standard gets a well-defined support for listening for events, we might need to make at least part of the xOpera orchestrator available as a service, but event mechanism are not in a state where we could implement them yet.

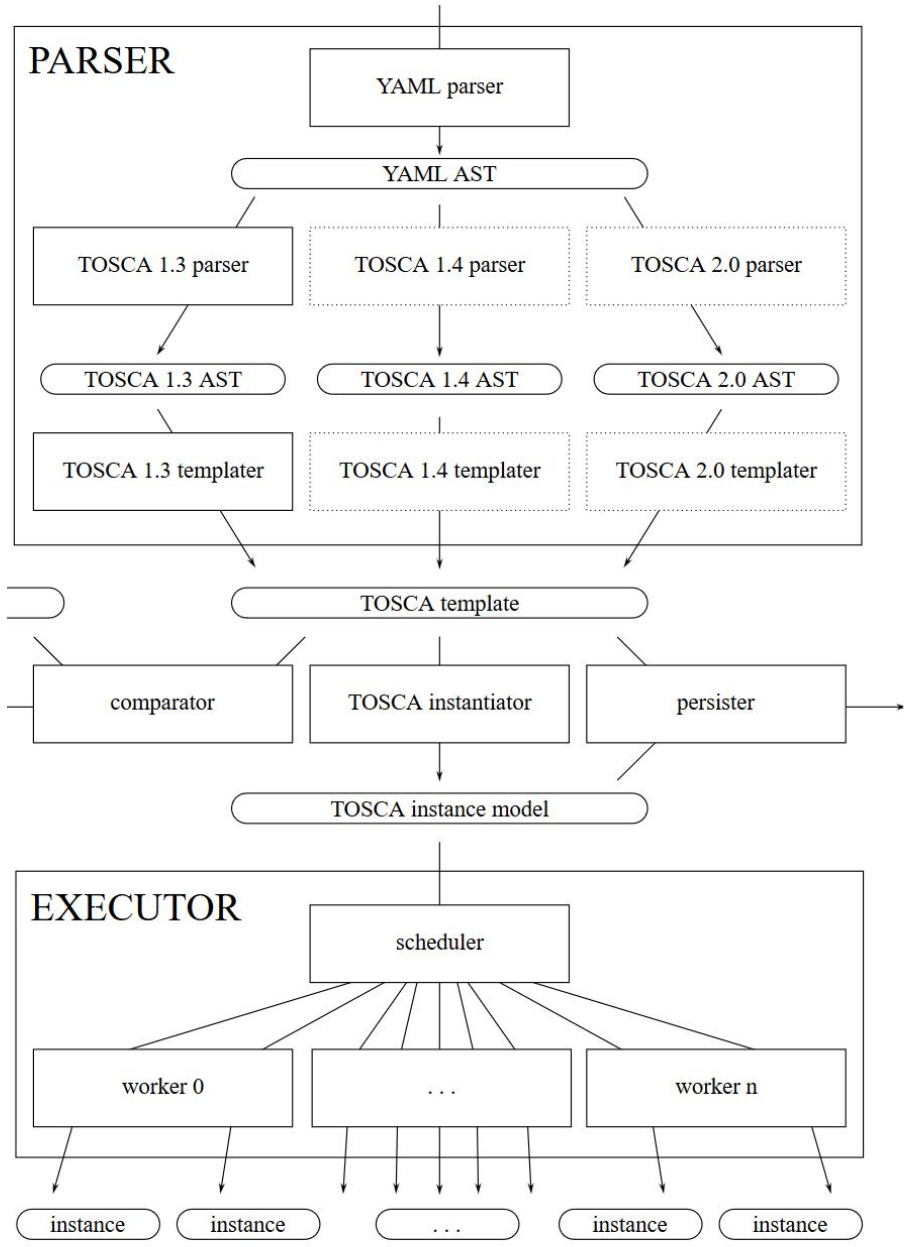


Figure 4. The xOpera orchestrator core design.

The xOpera orchestrator is packed in *opera* Python package and invoked by running the `opera` command. At this time two functionalities are supported, namely deploy and undeploy. They can be utilized with additional positional arguments. Console help message that describes the usage of the orchestrator is shown below.

```
usage: opera [-h] {deploy,undeploy} <name> <template.yaml>

positional arguments:
  {deploy,undeploy}
      deploy              Deploy service template from CSAR
      undeploy            Undeploy service template
optional arguments:
  -h, --help       show this help message and exit
```

At this stage the orchestrator commands can only be executed through command line interface (CLI), which is an appropriate way to execute for CI/CD tools. In the future it is intended to use the orchestrator as a deployed server supporting REST API service. This additional interface would allow an easier integration with other RADON components (e.g. monitoring) and also ease the management of the *day two* operations such as scaling, reshaping and un-deploying an application.

3.4.2 Installation and setup

The orchestrator can be downloaded from the xOpera GitHub[3] and installed following the instructions from a GitHub readme page. The requirements to run xOpera are Python virtual environment and Python interpreter.

Currently there is no particular setup solely for the orchestrator, the only requirement is to have a whole deployment TOSCA blueprint yaml downloaded and accessible from xOpera. However, the blueprint can have dependencies or special requirements related to the deployment. One example of those would be the libraries for particular providers (e.g. boto3 in case of Amazon) or credential management for particular provider.

3.4.3 Current maturity level

The xOpera orchestrator, which is used in RADON delivery toolchain is a lightweight TOSCA compliant orchestrator. The version 0.1.0 issued on May 2019 is compliant with TOSCA YAML v1.2 with limited functionality on specific executions of TOSCA types.

The recent release tagged with version 0.5.0 includes a new parser, which is compliant with TOSCA YAML v1.3 standard. The new parser is more human-friendly with more informative outputs of the YAML validation. Additionally the latest version supports uncompressed CSAR blueprints.

Current level of maturity fulfills a subset of requirements expressed in the RADON requirements table. Orchestrator already supports CLI commands and is able to deploy toy example Amazon Lambda. Still there are necessary modifications to make on autoscaling and more advanced functions that are not yet clear by TOSCA standard. On this matters RADON team will try to improve the orchestrator in a best possible way to achieve this functionality and propose the TOSCA standardisation group our result as a candidate for adoption into the standard.

[3] https://github.com/xlab-si/xopera-opera

3.4.4 Toy example demo

Orchestrator can successfully deploy the RADON toy example to prove the functionality and to demonstrate the complexity and completeness of this task. The necessary skills to run the example is being familiar with Linux shell, Git and Python virtual environment, additionally it is beneficial to be familiar with Amazon Lambda and setting AWS credentials. The whole list of tasks that consist of the walkthrough steps of the orchestrator installation and toy example deployment is available in the Appendix: *Walkthrough: Use the Orchestrator to deploy RADON toy example*.

3.4.5 Future plans

The immediate improvements to be made on the orchestrator depend on the development of template library and the integration of monitoring and scaling requirements. Firstly, with template library improvements, orchestrator will be able to deploy more complex applications to the wider range of cloud providers. Secondly, the monitoring capabilities and scaling requirements of the use cases will provide the guidance for a development of (auto)scaling functionality. The (auto)scaling is also interesting for the TOSCA standardisation group, as currently it is not yet fully covered by the standard.

Next important step that goes in hand with all future plans is the integration of the orchestrator into the whole delivery toolchain. The orchestrator has an important role in the delivery toolchain and is tightly connected with deployment, CI/CD and monitoring process, therefore a careful planning of the integration needs to be outlined as soon as possible.

3.5 Continuous integration

3.5.1 Description and interfaces

Continuous integration is the principle of frequently merging your work with the main branch. Doing this, enables faster feedback for your changes and lets your colleagues build upon your contribution. CI is not a RADON tool in itself, but rather a workflow we want to adopt in the runtime environment. We achieve this by including the RADON tools into the integration workflow, where we follow infrastructure from the design stage through passed tollgate and into the master branch. An important assumption for CI is the adoption of infrastructure as code, and as a result, version control. The process of CI is commonly implemented with a CI server. The CI server works as an objective validator of code quality and a tollgate for integration. Furthermore, it is common practice not to allow direct push to the master branch, but let the commits go through pull requests where they can be reviewed and discussed.

There is a vast set of alternatives in this space, although the functionality is more or less the same. It is common to divide them into two main categories; self hosted and SaaS. Within self hosted alternatives, you find tools like Jenkins, a stand alone web-based server. With Jenkins, you have to configure and maintain the master node and worker nodes that handle the workload. Alternatively, in the SaaS area, you find tools like Circle-CI, where you pay for the amount of concurrent jobs and size of the machines needed.

Praqma's main contributions around CI are twofold. One is to set requirements towards the other tool providers so that the relevant tools can be integrated into a CI pipeline. The other part is to provide quality and security toll gates as functions, for continuous, automated validation.

3.5.2 Installation and setup

Select CI provider based on the needs of your project. CI as a cloud service is definitely becoming the new standard and is also the easiest way of kickstarting CI with your project. In this example we focus on Circle-CI. Their free tier subscription is often sufficient for small projects.

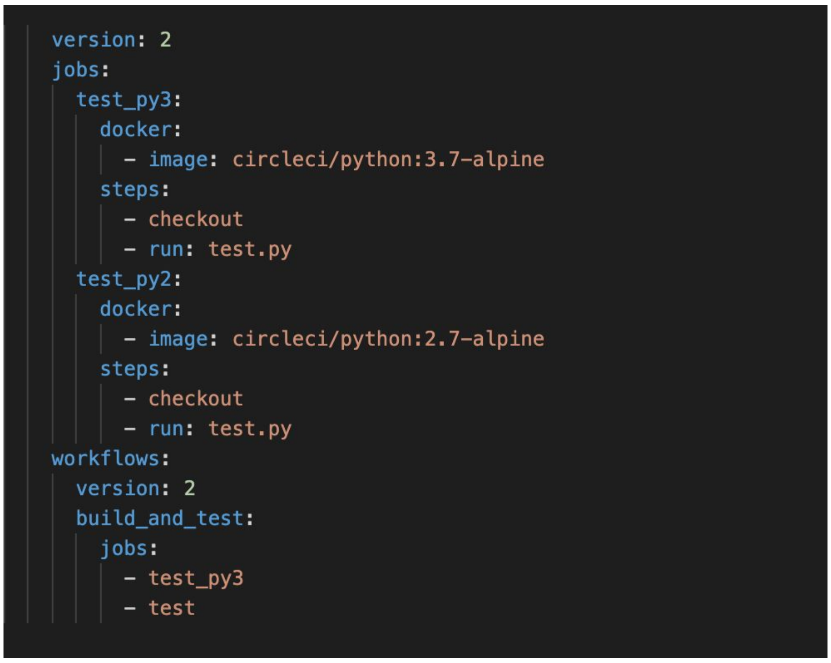
Create account

Log into Circle-CI with your GitHub or Bitbucket account. Once logged in, the browser will give you an overview of existing projects related to your user and organization where you are an owner. By pressing the 'Add project' in the left menu, you can select which project you want to follow based on the list of available projects.

Create config file

In your git project root folder, create a folder called ".circleci". In the folder, create a file called config.yml: this config file will be a declarative description of the integration pipeline. When a CI job is triggered, the CI server interprets the config file and carries out the work. Examples of pipeline description are 'when should it run', 'in what environment should it run', 'the order of execution for each job', 'the caching of data between jobs'.

The following config shows a workflow where a test runs in parallel. The two jobs are defined as 'test_py2' and 'test_py3', and use a Docker image to setup the testing environment. The idea is to automate the testing across the two versions of python.



```
version: 2
jobs:
  test_py3:
    docker:
      - image: circleci/python:3.7-alpine
    steps:
      - checkout
      - run: test.py
  test_py2:
    docker:
      - image: circleci/python:2.7-alpine
    steps:
      - checkout
      - run: test.py
workflows:
  version: 2
  build_and_test:
    jobs:
      - test_py3
      - test
```

Figure 5. config.yml example

Configure necessary tools

The next step is to include the available RADON tools into the config file, and let the CI server automate the validation of code.

3.5.3 Current maturity level

At this stage, the RADON toy example is used as a simple integration example[4]. This will work as a baseline for the continuation of CI and CD. The current status is a single job triggered by new commits to a feature branch.

3.5.4 Toy example demo

GitHub, Gitlab and Bitbucket all support integration with common CI servers like Jenkins, Travis, Circle-CI. In the Toy Example scope, let's assume the following scenario:

Let's say you need to change the memory allocation of the lambda function. You then make a branch and continue to do your code changes. Now you have a newly generated RADON model blueprint and want to integrate your changes with the "master" branch. Push your changes to the git server and open a pull request. The last commit will trigger the CI server to run a predefined pipeline definition. This integration pipeline will take your changes and validate the quality according to your defined policies. In the RADON framework environment, a set of toll gates would be the same as shown on Figure 6:

[4] https://github.com/naesheim/tosca-blueprint-aws-lambda

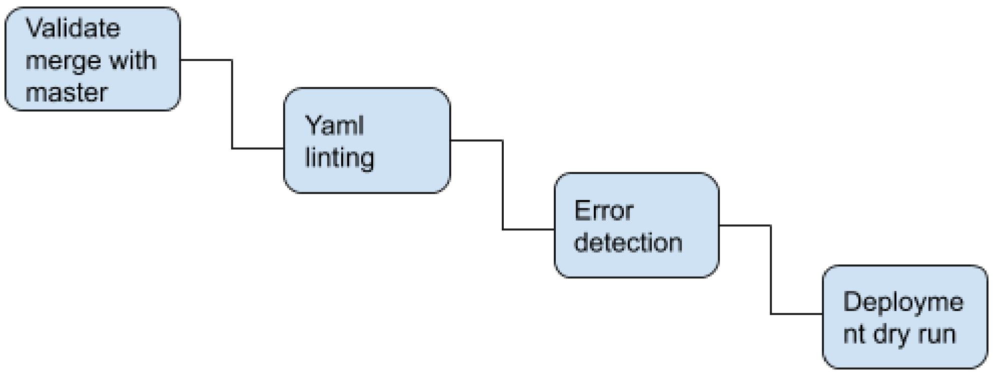


Figure 6. Tollgate example

Now the result from the CI server will be visible in the Pull Request. If the tollgate fails, you will need to reiterate the development process and make a new commit. If it succeeds, you can ask your colleagues for a code review before the final integration. The more tollgates you integrate and automate in the CI pipeline, the less time your colleague will spend reviewing your changes and the faster the integration cycle will be.

3.5.5 Future plans

The future plan is divided into three action points:

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.
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.
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.

3.6 Continuous deployment

3.6.1 Description and interfaces

There are several benefits of using continuous deployment; for one, it speeds up shipping new features to end users and shortens feedback loops. Thus, teams become more agile. Additionally, decreasing the size of each deployment statistically reduces the risk of introducing bugs and further improves quality. Continuous deployment shares many similarities with continuous integration. In fact, for many, it is a continuation of CI. Once your changes have gone through the CI pipeline and are integrated on the master branch. A similar CD pipeline gets triggered; this time to deploy the changes to production.

For implementing CD, you also use a CI server. By deploying from a CI server where source code is co-located with the deployment configuration, you achieve traceability and reproducibility. Traceability is achieved in the form of having audit trails of the deployment events and people who were involved in them. Reproducibility is achieved in the form of being able to reproduce a specific deployment from the version control history. Alternatively, deploying from a local machine opens up the risk of a 'dirty workspace' where that machine can contain local state which is not available on other machines. This prevents others from deploying since they lack that local state.

3.6.2 Installation and setup

The configuration of the descriptive deployment recipe is also merged with the CI config. Which pipeline that is executed is then determined by the branch that was changed. Commonly, feature branches trigger the *integration* pipeline; changes to master branch, trigger the *deployment* pipeline.

3.6.3 Current maturity level

Similar to the status of CI, we have a simple example of deploying AWS resources to production upon new commits to master.

3.6.4 Toy example demo

If we continue the example from continuous integration, we assume the CI pipeline succeeded, code review was carried out and the changes were validated for integration with the master branch. The set of changes related to the latest collection of commits merged into master, will constitute the delta of the new deployment.

3.6.5 Future plans

The main area of focus as the project matures, will be the integration of the different tools into the pipeline. Within the delivery toolchain, CD plays an important role. Securing a seamless connection with the monitoring setup will be the initial focus. Secondly we’re looking into alternative deployment methods. Dark launches, canary deployment and A/B testing are methods we want to experiment with in the delivery toolchain. Similar to the approach with CI, we don’t want to enforce a tool on the end user, but rather facilitate an easy integration of the RADON CD practice into the CI tool. That being said, we will provide CI/CD pipeline suggestion as pseudo code, CircleCI and Jenkins.

3.7 Function Hub

3.7.1 Description and interfaces

Serverless functions are packaged as compressed files containing code and configurations for running the functions on a targeted cloud platform. When deploying a function to the cloud, one must provide the compressed file of function’s code and configurations. Just like other software, function would evolve overtime with bug fixes and improvements and will have multiple versions. Additionally, in some cases, one might need to replicate the same functionality of a function for another cloud provider. Software providers and teams will need to store, share and collaborate on functions.

Function Hub is a serverless service to host and share functions between teams and/or between providers and users. Similar to other package managers, Function Hub allows creators of serverless functions to publish their functions and allows functions' users to pull and use those functions.

In the RADON context, functions are modeled within the GMT and then orchestrated by the runtime environment. Function Hub will be integrated in the GMT to enable RADON users to model functions and specify the function package from Function Hub. The Runtime environment will pull the package and deploy it to the target cloud at runtime.

3.7.2 Installation and setup

As a part of Praqma's use case, the Function Hub will be validated through the RADON framework. This means we will share the setup instructions as the use ase matures. Until this stage, Function Hub is available as a SaaS service at cloudstash.io.

3.7.3 Current maturity level

An alpha version of the Function hub is available at cloudstash.io. The Hub will support both public and private repositories but currently only public repository for reusable functions are available. Interaction with Function Hub is limited to browse and retrieval of functions through the web UI or a serverless REST API.

3.7.4 Toy example demo

The toy example currently produces one function. This function will be published and served via the Function Hub. The TOSCA model created in GMT will refer to the function location in the Function Hub using a URL (e.g. repo.cloudstash.io/public-functions/aws/toy-app). Then at runtime, the xOpera orchestrator pulls the function from the Function Hub and deploy it to production.

3.7.5 Future plans

The future work is grouped in three work packages.

- *Common function format.* The different cloud providers operate with different formats for functions. In order to store and evolve the functions in a consistent way, we need a coherent format. At this time, we are evaluating the adoption of the Helm[5] repository format. Helm is the Kubernetes package manager and is a CNCF[6] project. The helm repository format is simply a tar file with the package contents and an index file that contains metadata about the

[5] https://www.helm.sh
[6] https://www.cncf.io

packages. The metadata for functions will include description of the target runtime, the handler name, version, etc.

- *Integration with GMT and xOpera.* When creating a FaaS in GMT and specifying the location of the function, you can reference the URL of the specific function from Function Hub.
- *General functionality.* This backlog of new features will evolve as the tool matures and user feedback creates new requirements. These requirements, feature requests and new functionality will be handled as the project continues. A common CLI client, support for private repositories are some examples of potential features.

4 Conclusions and Future Work

This deliverable presents an overview of the current status of the Runtime environment and all the tools that are/will be integrated. The overview includes all the required information to equip the reader with the ability to deploy tools and understanding the requirements related to dependencies and potential interfaces. However, to have a compliance overview on one place, Table 3 shows an overview of the level of fulfillment for each of the agreed requirements. 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.

Currently most of the requirements does not fulfill the requirements as we are still in the early stage of the project. During the project the table will be updated and this tracking of the work will be useful for the tasks in WP2-WP5.

Table 3. Achieved level of compliance to RADON requirements

Id	Requirement Title	Priority	Level of compliance
R-T5.1-1	The orchestrator tasks must be executable through CLI	SHOULD_HAVE	✔✔✔
R-T5.1-2	A user describes the application architecture and dependencies at least in TOSCA YAML 1.2	MUST_HAVE	✔
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	X
R-T5.1-4	At the end of deployment the deployment of services needs to be verified and its dependencies	MUST_HAVE	X
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.	MUST_HAVE	X
R-T5.1-6	Support of FaaS deployment to OpenFaas	MUST_HAVE	✔
R-T5.1-7	Support of FaaS deployment to AWS cloud platform	MUST_HAVE	✔✔
R-T5.1-8	The xOpera command line interface needs to have a dry	COULD_HAVE	✔

	run mode to verify changes without asking for input in execution		
R-T5.1-10	When modelling a FaaS, the user can select a specific function by referencing the location from Function Hub	COULD_HAVE	X
R-T5.2-8	Support of FaaS_deployment to Google cloud platform	COULD_HAVE	X
R-T5.2-9	Support of FaaS deployment to Azure cloud platform	MUST_HAVE	✔
R-T5.2-10	Support deployment to regular VMs	MUST_HAVE	✔✔
R-T5.2-11	Support deployment to microservices architecture	MUST_HAVE	✔
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	X
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.	MUST_HAVE	X
R-T5.3-3	The tool must be able to support configuring AWS EC2 auto scaling service based on the TOSCA auto scaling policy.	MUST_HAVE	X
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	X
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	X

4.1 Future work

The deliverable presents runtime environment tools and explains together the future development plans for each tool.

In a nutshell, the document forecasts future release of the monitoring that will support a wider range of metrics to monitor and improve on the interaction with the other tools. The level of automation in deployment of monitoring configuration will also be improved. The template library will be refined and improved in terms of improved versioning and support for integration of several template libraries. Delivery toolchain will provide a unified configuration management for the other tools. The orchestrator functionality depends on the template library. Its future improvements will enable deployment of more complex applications to the wider range of cloud providers powered by template library. The (auto)scaling functionality and integration into the whole delivery toolchain is also planned for future work. The future plan for the CI/CD is to provide guidance for integration of other tools into a CI environment. Templates and pseudo code config files will also be prepared for easier adoption. Some alternative deployment methods will also be experimented with in the delivery toolchain. Future work for the function hub consists of using a common function format for storage, integration with GMT and xOpera and other general functionality improvements.

However, to coordinate the development of the Runtime environment through Y2 also the integration plan is required. One important way is to plan the scaling and auto scaling which means that the configuration of monitoring and development of the orchestrator has higher priority. On the other hand, data pipelines integration and support for different providers is maintained through template library. This means that Template library and integration of monitoring are currently most important.

5 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

6 Appendix A

6.1 Walkthrough: Use the Orchestrator to deploy RADON toy example

In this section we present a detailed instructions on how to deploy the application of thumbnail generation with xOpera. The application can be deployed on one of the supported serverless computing platforms, in this case AWS Lambda[7]. TOSCA Blueprint is used to describe the storage, functions, triggers and relationships. The commands given in the instructions assume the usage of Linux operating system.
First step is to clone the repository that contains the TOSCA blueprint for xOpera orchestrator with command:

```
git clone https://github.com/radon-h2020/tosca-blueprint-aws-lambda.git
```

Change your working directory to the downloaded project. Before we deploy the application, we will prepare the python environment and install necessary requirements. It is recommended that a Python virtual environment is used. This is an isolated and self-contained directory tree that contains a particular version of Python and installation of additional packages. Virtual environment can be created and activated with commands:

```
python3 -m venv .venv
source .venv/bin/activate
```

Before we can deploy the application we need to configure our AWS credentials. Some additional Python packages are needed in order to perform this step, namely botocore and boto3. The user then needs to input the following information: AWS Access Key ID, AWS Secret Access Key, Default region name and Default output format. Package installation and AWS credentials configuration is done with commands:

[7] https://aws.amazon.com/lambda/

```
pip install botocore boto3
aws configure
```

The next step is to install the xOpera orchestrator. The orchestrator depends on the ansible package. In this case we will install the development version, but version 2.9 should also be fine. The installation is done with the following commands:

```
pip install git+https://github.com/ansible/ansible.git@devel
pip install opera
```

At this point our environment is set up and ready to deploy applications. In this case we will deploy a toy example, i.e., the application of thumbnail generation. A ready to deploy zip package can be downloaded from GitHub repository given below. The instructions on how to prepare a zip package for a custom function are also given in this repository.

```
https://github.com/radon-h2020/FaaS-thumbnail-generator-python/tree/master/binar
y-zip
```

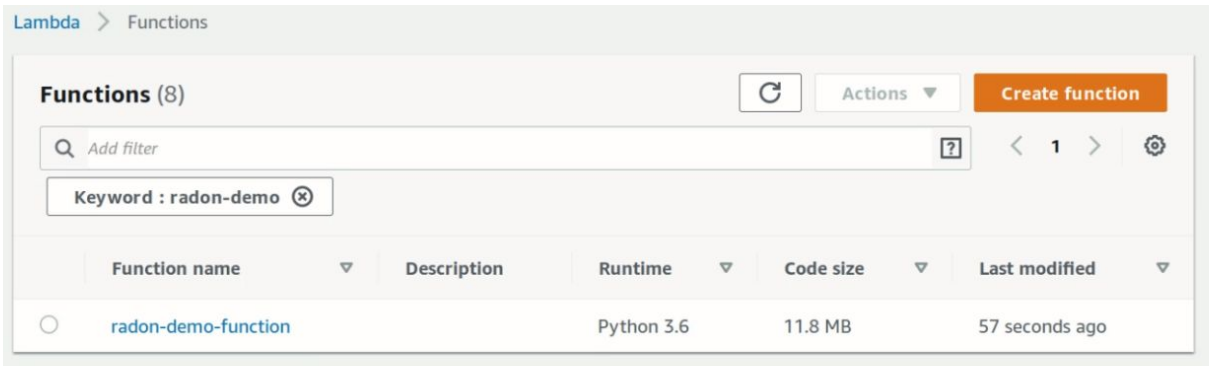
The current version of xOpera orchestrator does not support the lookup function. Because of this the location of the application package and AWS policy file is set to a fixed path, namely the /tmp folder. The application zip package and policy file (contained in the first cloned repository) needs to be copied to this folder. Note that this is a temporary limitation. If your working directory has not changed and the downloaded toy example is located in the default Downloads folder, then the following commands can be used to place the files in the appropriate folder:

```
cp playbooks/aws_role/policy.json /tmp/
mv ~/Downloads/X-test-ImageRes.zip /tmp/
```

The last step of the setup before we are ready to deploy the application is to set the desired names for the function and storage buckets on AWS. This is done by editing the properties in the configuration YAML file located in the TOSCA blueprint repository. Open the resize_service_opera.yml and edit the values of properties function_name and bucket_name in the topology_template section. Toy example can now be deployed using the following command:

```
opera deploy demo-deploy resize_service_opera.yml
```

If the deployment was successful then the created function and storage buckets should appear in the AWS console as shown in Figure 7 and Figure 8.

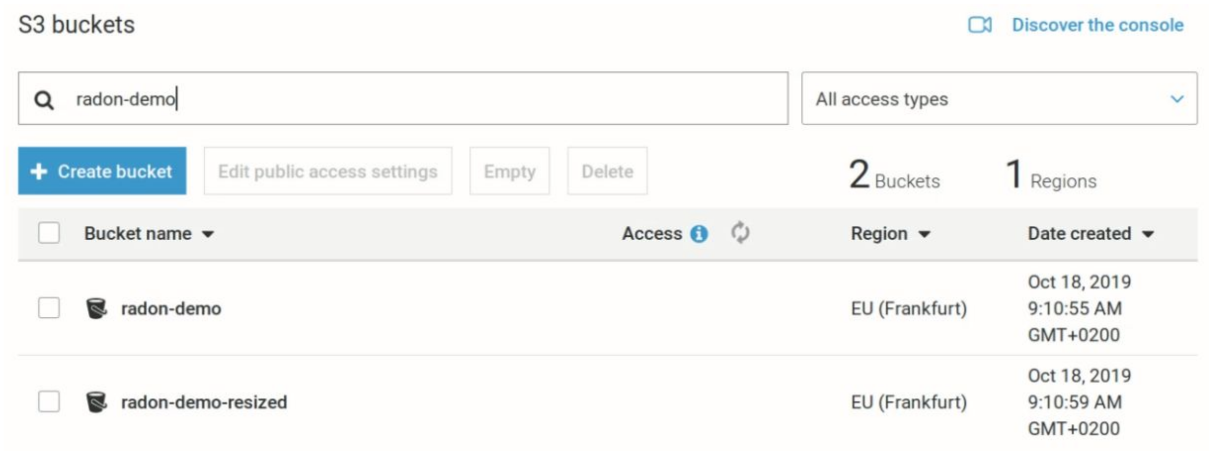

Lambda > Functions

Functions (8)

Keyword : radon-demo

Function name	Description	Runtime	Code size	Last modified
radon-demo-function		Python 3.6	11.8 MB	57 seconds ago

Figure 7. Toy example function deployed on AWS Lambda.


S3 buckets

radon-demo

All access types

2 Buckets 1 Regions

Bucket name	Access	Region	Date created
radon-demo		EU (Frankfurt)	Oct 18, 2019 9:10:55 AM GMT+0200
radon-demo-resized		EU (Frankfurt)	Oct 18, 2019 9:10:59 AM GMT+0200

Figure 8. Storage Buckets configured on AWS S3.

To demonstrate the usage of the deployed toy example we can upload the image in the input bucket named radon-demo in this case. This will trigger the image resize function and the output image will appear in the output bucket named radon-demo-resized in this case.

6.2 Walkthrough: General purpose Monitoring Tool Setup

Installation and Setup

- **Docker**
  Read the instructions from the official website https://docs.docker.com/install/

- **cAdvisor**
  cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about running containers. Specifically, for each container it keeps resource isolation parameters, historical resource

usage, histograms of complete historical resource usage and network statistics. This data is exported by container and machine-wide.

To set up a cAdvisor container instance run the following:

```
docker run \
--volume=/:/rootfs:ro \
--volume=/var/run:/var/run:rw
--volume=/sys:/sys:ro \
--volume=/var/lib/docker/:/var/lib/docker:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor:latest
```

- **Prometheus Pushgateway**
  The Prometheus Pushgateway exists to allow ephemeral and batch jobs to expose their metrics to Prometheus. Since these kinds of jobs may not exist long enough to be scrapped, they can instead push their metrics to a Pushgateway. The Pushgateway then exposes these metrics to Prometheus.

  To set up Prometheus Gateway :

```
docker run -d -p 9091:9091 prom/pushgateway
```

- **Prometheus**
  Prometheus is an open-source system monitoring and alerting toolkit originally built at SoundCloud. Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community. It is now a standalone open source project and maintained independently of any company. To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

  To set up Prometheus:

```
docker run -p 9090:9090 -v /tmp/prometheus.yml:/etc/prometheus/prometheus.yml \
prom/prometheus
```

To set up prometheus we have to provide a configuration file like the following

```
scrape_configs:
-
 job_name: monitor
scrape_interval: 5s
static_configs:
-
targets:
- "cadvisor(ip):port"
- "pushgateway(ip):port"
```

For now we only care about targets (locations of metrics to scrape) and scrape interval (frequency of scrape).

You can find more details about configuration here: https://prometheus.io/docs/prometheus/latest/configuration/configuration/

- **App configuration**

In order to get applications metrics we have to export them first. To achieve that, follow the instructions below:

1. Download metricsfn.js and reqtimes.js and import them into your root js file (or the relevant root file of the user application)

```
var reqtimes = require('./path_to_reqtimes.js')
var metricsfn = require('./path_to_metricsfn')
metricsfn.init('ip_of_pushGateWay')
```

2. Add as middleware the following functions:

```
app.use(metricsfn.requestCounters)
app.use(metricsfn.pushgtw)
app.use(reqtimes(metricsfn.client))
```

3. Get function Execution time:

```
metricsfn.execution_time.setToCurrentTime();
let end = metricsfn.execution_time.startTimer();

function test(){
// code ...
metricsfn.exec_time_end(end, arguments.callee.name)
// exit function
}
```

4. (Optional) You can also inject /metrics route to access metrics from the App and not through prometheus gateway.

```
app.get('/metrics', function (req, res) {
res.end(metricsfn.client.register.metrics());
});
```

After everything is done, visit prometheus and you will able to access metrics.

- **Grafana**

  For better visualization,alerts management and dashboards you can use grafana.You can check the documentation from here: https://grafana.com/docs/

  Set up grafana with docker:

  ```
  docker run -d -p 3000:3000 grafana/grafana
  ```

  Once Grafana is installed, the sources of the incoming metrics have to be defined. The available sources can be:

  - Prometheus
  - AWS Cloudwatch
  - Azure Monitor
  - Elasticsearch
  - Google Stackdriver
  - Graphite
  - InfluxDB
  - Loki
  - Microsoft SQL Server (MSSQL)
  - Mixed
  - MySQL
  - OpenTSDB
  - PostgreSQL
  - Testdata

  Information on how to install and configure the sources can be found here[8].

  After the sources are configured, the user can either create dashboards or import them from the provided available dashboards found here[9].

[8] https://grafana.com/grafana/dashboards
[9] https://grafana.com/grafana/dashboards

For each created dashboard the user has to provide:

- The Source (listed above)
- The query (which is the configuration that collects the metrics-data)

For example:

- If the user wants to monitor the CPU usage of a container the source is Prometheus (which ingests data from CAdvisor) and the relevant query is:

  *rate(container_cpu_user_seconds_total{name="Container_name"}[30s])*100*

- If the user wants to measure the RAM usage of a container the source is Prometheus (which ingests data from cAdvisor) and the relevant query is:

  container_memory_usage_bytes{name="Container_name"}/1000000

- If the user wants to measure the incoming traffic towards an application deployed in a container the source is Prometheus (which ingests data from cAdvisor) and the relevant query is:

  rate(container_network_receive_bytes_total{name="Container_name"}}[30s])/1000000

- If the user wants to measure the outcoming traffic from an application deployed in a container the source is Prometheus (which ingests data from cAdvisor) and the relevant query is:

  irate(container_network_transmit_bytes_total{name="viarota"}[5m])

In general the query syntax depends on the source from which the data are ingested. In case of prometheus the user needs to get competent around PromQL. More info about PromQL can be found in the basic section and the examples of Prometheus.

In case the user wants to invoke an already predefined dashboard once again the relevant source has to be defined and the relevant dashboard can be downloaded and configured according to the list of available Grafana Dashboards.

6.3 Walkthrough: Toy Example Monitoring Tool Setup

The thumbnail generation application is deployed on an AWS Lambda function. Prior to deployment, the source code of the application is injected with code in order to expose application specific metrics (e.g. traffic, CPU and RAM usage).

Metrics directly exposed by Lambda functions are collected by Cloudwatch service (or any equivalent Cloud Monitoring Services) and they are forwarded directly to Grafana for the generation of the relevant real-time updated dashboards. These metrics include Lambda function invocation duration, cost, errors, etc. On the other hand, metrics that are exposed through code injection are pushed towards the Prometheus Pushgateway and subsequently towards Prometheus. These metrics are RAM and CPU usage. After Prometheus collects these metrics it performs the following parallel actions:

- Alerts can be generated based on defined rules on thresholds by the user. These alerts can trigger subsequent actions.
- Metrics are forwarded towards Grafana for the generation of the relevant dashboards.

Through this way, Grafana is the common ground for both General Metrics and Application Specific Metrics monitoring. Once the dashboards in Grafana are created, the user can dynamically define and embed them in the Centralized Monitoring Dashboard (see Figure 9).

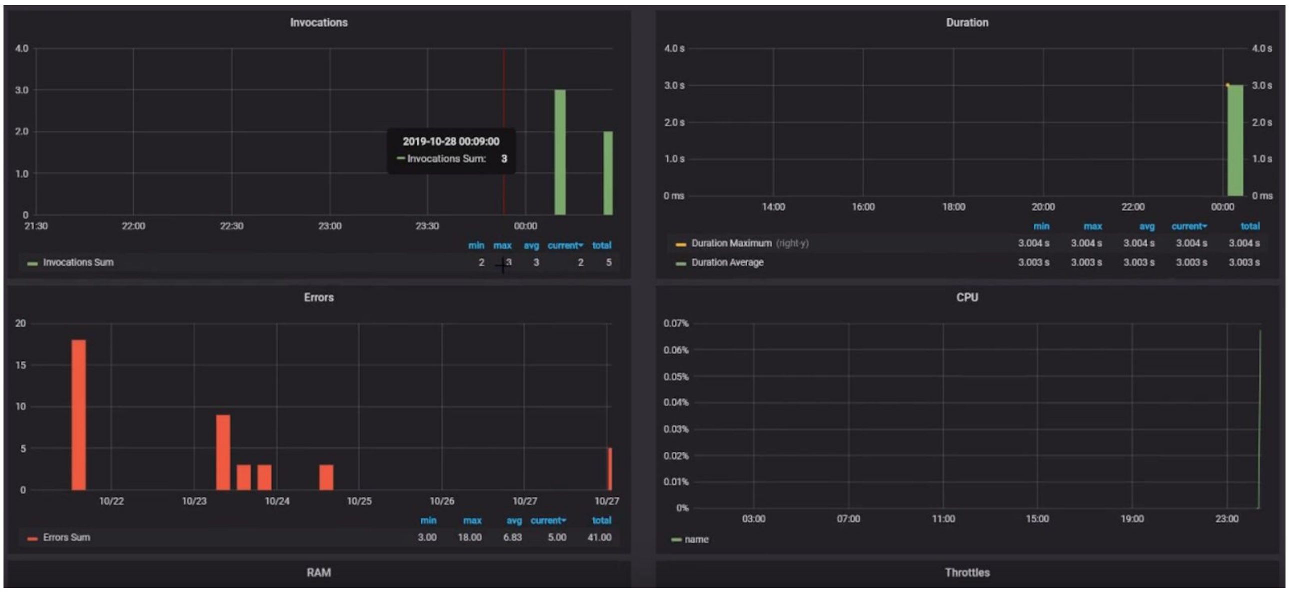


Figure 9. Centralized Monitoring tool snapshot.

Below you can find detailed steps to deploy, setup and configure the Monitoring Tool adjusted on the Thumbnail generation toy example:

Installation and Setup

- **Prometheus Pushgateway**

  The Prometheus Pushgateway exists to allow ephemeral and batch jobs to expose their metrics to Prometheus. Since these kinds of jobs may not exist long enough to be scrapped, they can instead push their metrics to a Pushgateway. The Pushgateway then exposes these metrics to Prometheus.

  To set up Prometheus Gateway :

  docker run -d -p 9091:9091 prom/pushgateway

- **App configuration**

  In order to get the toy set applications metrics such as RAM and CPU the user has to expose them first. To achieve that, the user has to inject some code in the application. This code pushes the metrics to prometheus gateway and subsequently to Prometheus and Grafana dashboards. The updated image_resize.py file can be found here.

- **Grafana**

  For better visualization,alerts management and dashboards you can use grafana.You can check the documentation from here: https://grafana.com/docs/

  Set up grafana with docker:

  docker run -d -p 3000:3000 grafana/grafana

  Once Grafana is installed, the sources of the incoming metrics have to be defined. The available sources can be:

  - Prometheus
  - AWS Cloudwatch

  Information on how to install and configure the sources can be found here.

  The metrics that are monitored in the case of thumbnail generation are:

  - CPU
  - RAM
  - Duration
  - Invocations
  - Errors
  - Throttles
  - Global Concurrent Executions

  In order to visualize this metrics in Grafana the following dqshboards have to be configured:

  1. CPU: The source of the dashboard is Prometheus and the relevant query is:

     rate(cpu_usage_percent{exported_job="image_resize",instance="prometheus_push_gateway_url:PORT",job="cadvisor"}[30s])

  2. RAM: The source of the dashboard is Prometheus and the relevant query is:

     memory_usage_bytes{exported_job="image_resize",instance="prometheus_push_gateway_url:PORT",job="cadvisor"}

3. The rest of the metrics are visualized in predefined imported dashboards. this predefined dashboards as well as how to configure them can be found here.

Note: Since one of the sources is prometheus, the user needs to get competent around PromQL to syntax the queries for dashboards like CPU and RAM. More info about PromQL can be found in the basic section and the examples of Prometheus.