
# Rational decomposition and orchestration for serverless computing

# Deliverable D6.6

# Outreach Demonstrator

# (Companion Document)

**Version: 1.0**

**Publication Date: 30-June-2021**

## Deliverable Card

| | |
|---|---|
| **Deliverable** | D6.6 |
| **Title:** | Outreach Demonstrator – Companion Document |
| **Editor(s):** | Alexandros Ilias Spartalis (EFI/PRQ) |
| **Contributor(s):** | Michael Wurster (UST), Vladimir Yussupov (UST), Mainak Adhikari (UTR), Eleftherios Anastasiadis (IMP), Ahmad Alnafessah (IMP) Zifeng Niu (IMP), Lulai Zhu (IMP), Shreshth Tuli (IMP), Mark Law (IMP), Jose Perusquia Cortes (IMP), André van Hoorn (UST), Thomas F. Düllmann (UST), Runan Wang (IMP), Stefano Dalla Palma (TJD), Dario Di Nucci (TJD), Alexandros Ilias Spartalis (EFI/PRQ), Matija Cankar (XLB), Anže Luzar (XLB), Špela Dragan (XLB), Anestis Sidiropoulos (ATC), Giorgos Giotis (ATC) |
| **Reviewers:** | Mainak Adhikari (UTR), Giuliano Casale (IMP) |
| **Type:** | Report |
| **Version:** | 1.0 |
| **Date:** | 30-June-2021 |
| **Status:** | Final |
| **Dissemination level:** | Public |
| **Download page:** | http://radon-h2020.eu/public-deliverables |
| **Copyright:** | RADON consortium |

## 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**

# Glossary

| | |
|---|---|
| CDL | Constraint Definition Language |
| CI/CD | Continuous Integration/Continuous Delivery |
| CLI | Command Line Interface |
| CSAR | Cloud Service Archive |
| CTT | Continuos Testing Tool |
| DPT | Defect Prediction Tool |
| DT | Decomposition Tool |
| FaaS | Function-as-a-Service |
| GMT | Graphical Modeling Tool |
| IaC | Infrastructure-as-Code |
| IDE | Integrated Development Environment |
| SUT | System under test |
| TI | Test infrastructure |
| TL | Template Library |
| TLPS | Template Library Publishing Service |
| TPS | Template Publishing Service |
| VT | Verification Tool |

# Table of contents

# 1. Overview

This companion document offers a static snapshot of how to use the final version of the "radon-demonstrator" online training course. The content provided here can also be found in RADON's Github repository: https://github.com/radon-h2020/radon-demonstrator.

# 2. Demonstrator

The demonstrator is practically an example application combining the usage of the RADON tools in a step by step walkthrough for the purpose of showcasing the functionality of the RADON framework.

## 1.1. Purpose of demonstrator

The demonstrator project provides a technical baseline for hands-on training activities which will be provided to the end user on the RADON project website. The purpose of the demonstrator itself is to offer the possibility for occasional visitors of RADON website, attendees of meetups and conferences or any other end user interested in testing the RADON framework to try out the solution and its capabilities in a matter of minutes. It is intended to require the minimum amount of manual configuration by the end user, while guiding him to a walkthrough that showcases the benefits of using RADON framework as a whole, based on a project that has vast applicability to many different business domains.

End users tend to be less willing to read a documentation file or instruction manual, especially if they are not yet convinced that a particular tool is able to improve efficiency. On the other hand, training material such as tool tutorials are the most effective way of getting the targeted audience familiar with a new tool or product. It saves them valuable time of reading long documentation material and offers them the possibility to try out effortlessly some of the key features of a product by just following a set of instructions. The demonstrator in this case serves the role of a RADON framework tutorial and PRQ intends to reuse the content of the demonstrator as a concrete example of using RADON framework as an end-to-end solution for serverless application development. Through this example, RADON promotes that it can take care of the resource modeling, architecture compliance, unit testing, decomposition optimization, resources deployment and monitoring. Training examples such as the demonstrator will also be part of hands-on training delivered by PRQ in the context of product dissemination showing the final version and overall capabilities of the framework.

## 1.2. Technical details

The underlying application of the demonstrator is one of the four different demo lab applications used by the consortium to validate the RADON tools and concepts as described in the deliverable D6.1, and has been decided collectively by the consortium's members. The "ServerlessToDoListAPI" as can be found in the available service templates in radon-particles is based on an existing project (https://github.com/iaas-splab/todo-api-nodejs) showcasing the creation and deployment of a serverless application using the well-known in serverless community framework, "Serverless". From a functionality point of view, it realizes a serverless API application, exposing several functions through an API Gateway that modify a database on the cloud. It is an ideal candidate for the demonstrator as it has high applicability to many different applications, but most importantly, it

demonstrates in practice how effortlessly the same application can be developed using an alternative, rather than a well trusted framework.

The chosen workflow in Figure 1.2.1, describes the procedure of developing an application from scratch using the RADON framework and includes a collection of RADON tools that benefit the lifecycle of an application development. Different combinations of tools or sequential order can be chosen according to the needs of each project, but for the purpose of this example we will stick to the one that helps us evolve the application in a logical manner. The end users are free to evaluate the tools and pick the ones that fit the best to their workflow plan.

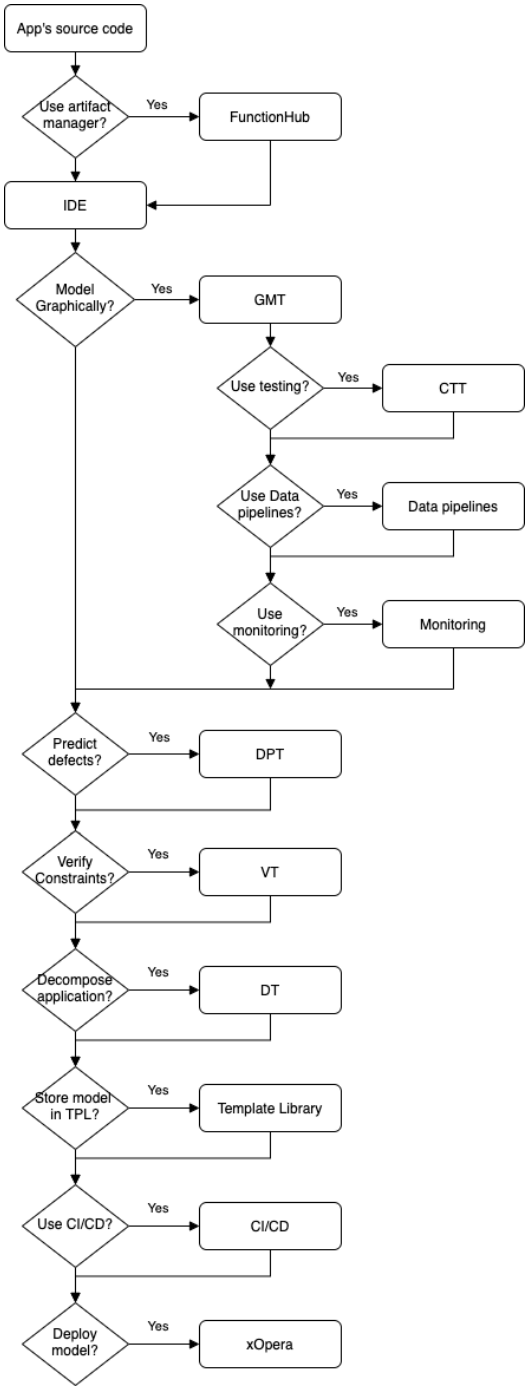


**Figure 1.2.1:** Demonstrator flow diagram with RADON

Table 1 represents the list of RADON tools used in the demonstrator example showing their runtime environment as well.

| # | **Tool** | **Usage** | **Runtime** |
|---|---|---|---|
| 1. | IDE | Get access and start a workspace | IDE |
| 2. | GMT | Model the application step by step | IDE plugin |
| 3. | VT | Set application constraints | IDE plugin |
| 4. | DPT | Invoke defect prediction tool on csar | IDE plugin |
| 5. | CTT | Perform endpoint testing | IDE plugin |
| 6. | Monitoring | Include monitoring libraries & manual set up | EC2 instance |
| 7. | FunctionHub | Upload artifacts in public repository | Local workspace |
| 8. | Orchestrator | Resources topology deployment | Local workspace/SaaS |
| 9. | Template Library | Upload csar to Template Library | IDE plugin |
| 10. | CI/CD | Execute a CI/CD pipeline | IDE plugin |
| 11. | Data Pipelines | Move data across multiple cloud vendors | IDE plugin |
| 12. | Decomposition | Decompose and optimize application | IDE plugin |

**Table 1.2.1:** Radon tools table in Demonstrator

### 10.2.1. FunctionHub

For the needs of the demonstrator, all the artifacts and functions that handle the back-end of the “SeverlessToDoListAPI" application will be hosted on FunctionHub. A short walkthrough of using

this tool will be presented in this section, while the detailed instructions on how to use the Cloudstash platform and FunctionHub CLI can be found in the documentation[1].

First, we need to retrieve the actual code that performs the functionality of the lambda functions. To do so we can just visit the GitHub repository[2] and download the files. For this example, we plan to create 5 lambda functions supporting the "create", "delete", "get", "list", "update" functionalities. In order to have the functions available anytime, aggregated in a common place, we will store them in FunctionHub. FunctionHub will provide access to them by a URL that we can later attach as a reference to the lambda's "artifacts" property in GMT. (If you wish to skip this part there is a public FunctionHub repository available containing the 5 necessary functions. The repository can be found in Cloudstash.io under the name "ServerlessToDoListApi-repo").

After creating a user and a repository in FunctionHub, all we need to do is to use the FunctionHub cli in order to upload our functions. Using the command "*fuhub create <project_name>*" a new directory is created in our local workspace containing a "config.ini" file. This file needs to be populated with necessary information such as the user credentials, the repository on which the function will be stored and various details about the function such as name, version, description, provider, runtime and handler. An example of the configuration file can be found in the following GitHub repository[3].

Next, the source code, in our case the "create.js" file has to be zipped and stored in the same directory as the configuration file. As a last step the user needs to invoke the deployment using the *"fuhub --token <deploy-token> upload <zipped_function>"* of the FunctionHub cli package.

```
→  create fuhub --token <deploy_token> upload create.js.zip
upload function create to repository ServerlessToDoListApi-repo
200
```

**Figure 10.2.1.1:** Deployment of function to FunctionHub

Figure 10.2.1.1 shows a successful deployment of the function "create" on FunctionHub, while figures 10.2.1.2 and 10.2.1.3 show the content of the repository "ServerlessToDoListAPI-repo" and the "create" function's details respectively.

[1] https://functionhub-cli.readthedocs.io/en/latest/

[2] https://github.com/iaas-splab/todo-api-nodejs

[3] https://github.com/radon-h2020/radon-functionhub-client/tree/master/test


**Figure 10.2.1.2:** Functions in "ServerlessToDoListAPI-repo"


**Figure 10.2.1.3:** Detailed information of the "create" function

The stored functions or artifacts can be downloaded from the web UI or by accessing the URL after constructing it providing the correct artifactID. An example of such a URL is shown in the documentation[4] under the section "Retrieving Functions from FunctionHub".

### 10.2.2. IDE

The RADON IDE (Integrated Development Environment) provides a web-based development environment that supports multi-user usage and acts as the host environment for the RADON framework by enabling the interaction with the RADON tools.

The RADON development environment is based on the Eclipse Che[5] technology: an open-source developer workspace server which comes with a cloud integrated development environment (IDE) used by developers to create applications without the need to install any software on their local system.

[4] https://functionhub-cli.readthedocs.io/en/latest/

[5] https://www.eclipse.org/che/

Essentially, the end user of RADON will have to get access to the IDE instance where all the RADON tools can be found within the RADON workspaces. In order to get access to the RADON IDE, the user must make a RADON Access Request using the form provided under "TEST RADON" available at the RADON website[6]. A more comprehensive documentation on the RADON IDE can be found in the IDE ReadTheDocs[7] webpage.

After acquiring the user credentials by the administrators of the IDE, a workspace that is representative of the RADON framework needs to be created. The process to do that is explained in the documentation of IDE on the link above under the section "Create a RADON workspace", which basically describes the creation of a custom workspace using the provided RADON Devfile. The RADON Devfile encloses a description of the Kubernetes containers and (Che) plugins that have been implemented to integrate the RADON tools on the RADON IDE. Once done with that, all the RADON tools and plugins are automatically installed in the user's workspace and are available to be used. Figure 10.2.2.1 depicts the RADON IDE workspace.

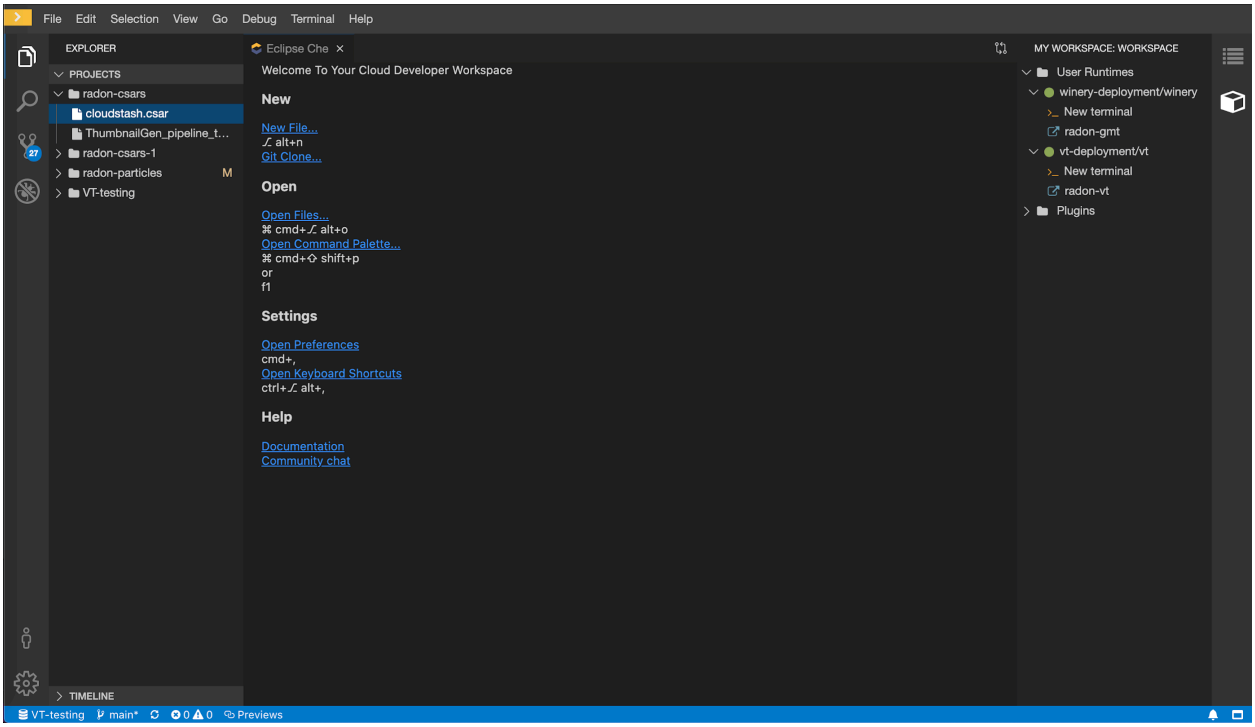



**Figure 10.2.2.1:** RADON IDE workspace

### 10.2.3. GMT

GMT (Graphical Modeling Tool) is the tool used for modeling the application topology. It relies on a set of pre-configured templates describing the various resource nodes and the relationships between them. This library of templates called "radon-particles" has been developed especially for RADON

[6] https://radon-h2020.eu/demo-homepage/

[7] https://radon-ide.readthedocs.io/en/latest/

with the contributions of all the RADON units and is already available in the generated workspace. It can be found on the left panel of the workspace as shown in Figure 10.2.2.1.

A very detailed user guide of the tool can be found in GMT documentation[8], but for the sake of the demonstrator, we will stick only to the necessary components for our application.

GMT can be started by using the "radon-gmt" button on the right panel of the IDE workspace. From there, a new web browser tab will appear with the available templates as depicted in Figure 10.2.3.1. A user can simply open one of the existing templates or create a new one by clicking on "Add new". Specifying the name, the versioning option and the namespace under which we want to create the model we create a new service template, as shown in Figure 10.2.3.2.

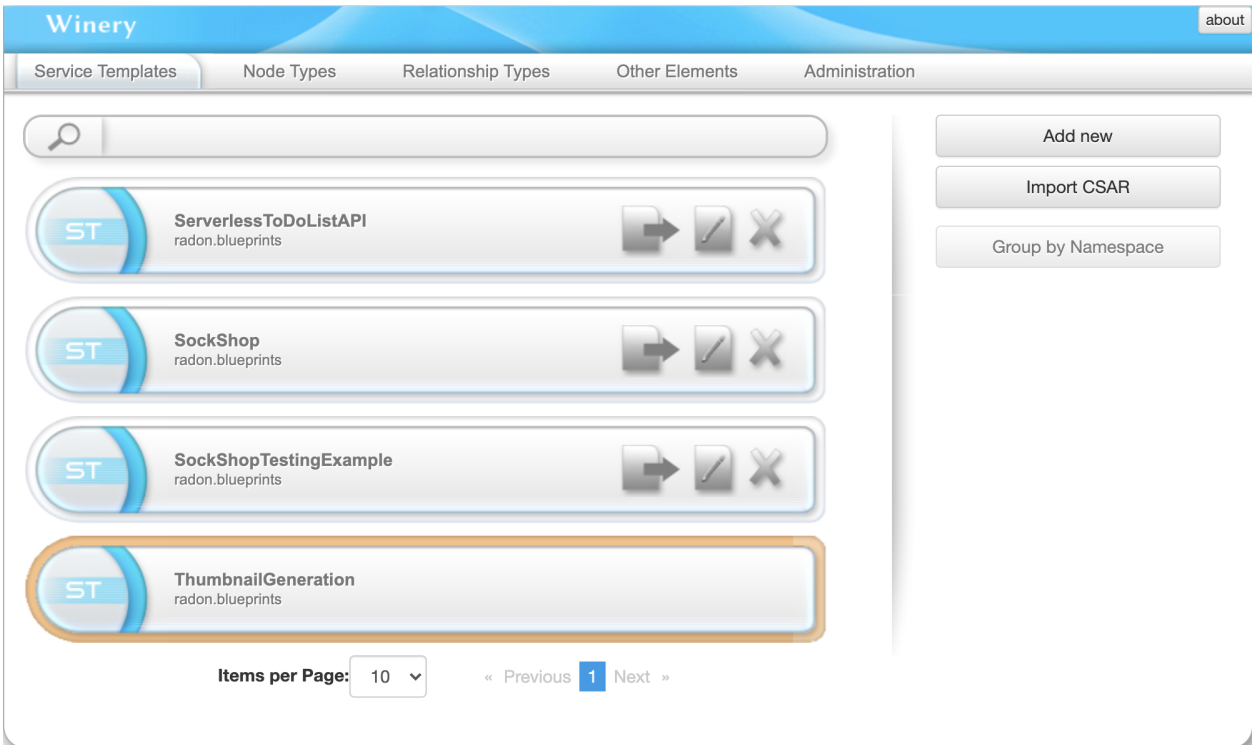



**Figure 10.2.3.1:** RADON GMT panel

[8] https://winery.readthedocs.io/en/latest/

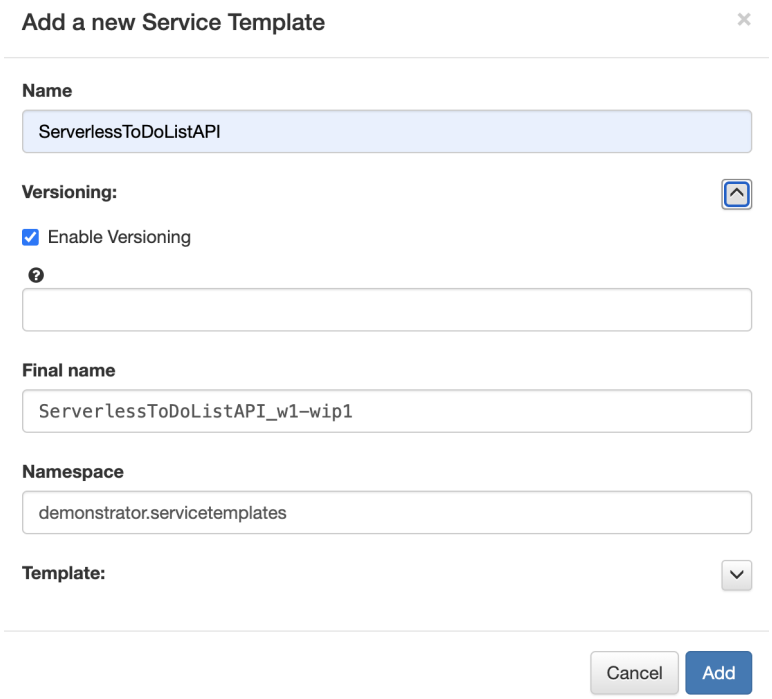



**Figure 10.2.3.2:** Add new Service Template

After creating a service template, the user can start modeling the application by accessing the "Open editor" button in the "Topology Template" tab. A new view is opened in a separate browser tab, the Topology Modeler workspace. In order to start modeling a topology, users can choose different types of nodes to model the application from the palette on the left side.

We have chosen to use AWS (Amazon Web Services) as the main cloud deployment platform, so all the necessary nodes can be found under the namespace "radon.nodes.aws" at the palette. The ServerlessToDoListAPI application consists of 1 "AwsApiGateway", 5 "AwsLambdaFunction" and 1 "AwsDynamoDBTable" TOSCA node types. An additional "AwsPlatform" node is required which represents the region on which the deployment will take place. For the purpose of this example, we will go through the configuration of each of the included nodes, taking out the duplicate ones such as the lambda functions to avoid redundancy.

Using the "drag and drop" method, we place all the required nodes on the topology editor and start configuring them. We can group the required configuration needed in "properties", "artifacts" and "relationships" configuration.

**AwsLambdaFunction - Properties**

On the tab properties we populate the different properties. GMT provides a real time error detector to prevent the user from filing false entries. An example of the properties set for the lambda function list is depicted in Figure 10.2.3.3.

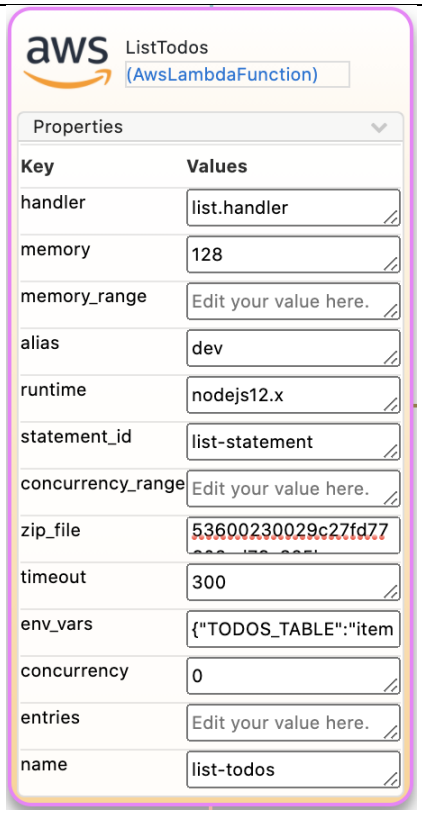



**Figure 10.2.3.3:** AwsLambdaFunction Properties

The necessary properties in this example are the **handler**, the allocated **memory** of the lambda function, the function **alias**, the **runtime** environment, the **statement_id**, the **zip_file**, the **timeout**, the **env_vars**, the **concurrency** and the **name**. Most of them are self-explanatory especially if you have some previous experience with AWS properties in Lambda functions. It is worth noting that the zip_file is basically a reference to the attached artifact of the lambda function. An example of how the zip_file property can be defined looks as follows:

**zip_file: {"get_artifact":["SELF","artifact_name"]},**

where "get_artifact" is the TOSCA's intrinsic function facilitating retrieval of attached artifacts during the deployment process. Another property that might not make sense at this stage is the env_vars. This is basically the definition of the environment variables that this lambda function has in order to access other AWS components on its backend, and in this example points to the DynamoDB table that we will create in the next steps.

**AwsLambdaFunction - Artifacts**

On the tab "Artifacts", the artifacts that accompany the lambda function can be attached either by uploading a file or by referencing a URL from FunctionHub. In Figure 10.2.3.4 the correct configuration for attaching an artifact stored in FunctionHub is presented. The URL can be constructed by adding the corresponding "ArtifactID" to the base download URL "http://cloudstash.io/artifact_download/". It is worth noting that, during CSAR generation or CSAR

export, all URL-referenced artifacts are downloaded by GMT to produce a self-contained CSAR package.

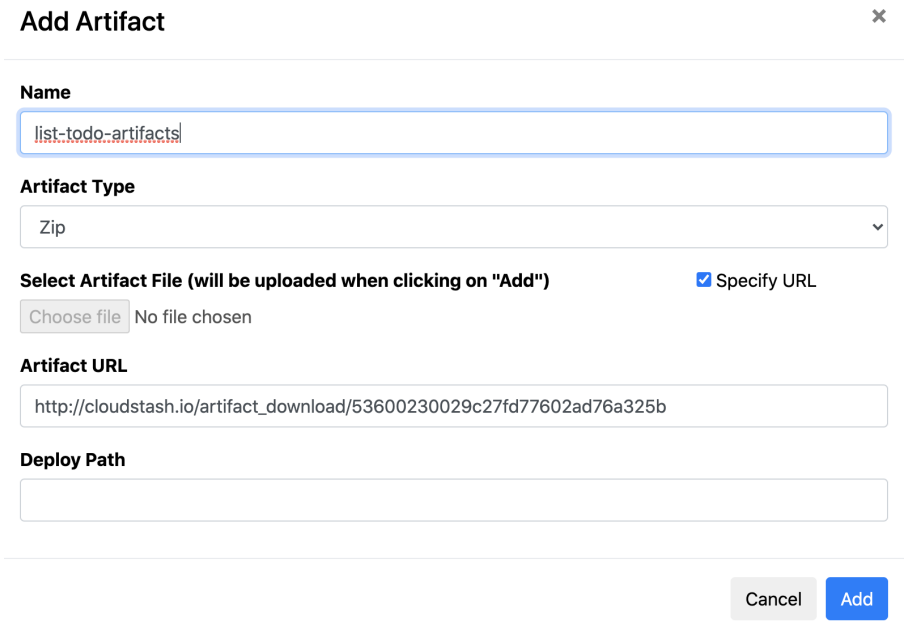



**Figure 10.2.3.4:** AwsLambdaFunction Artifacts

## AwsLambdaFunction - Relationships

As all nodes in the model, "AwsLambdaFunction" nodes require the proper relationship connections. Those can be found in the tab "Requirements & Capabilities in Figure 10.2.3.5.

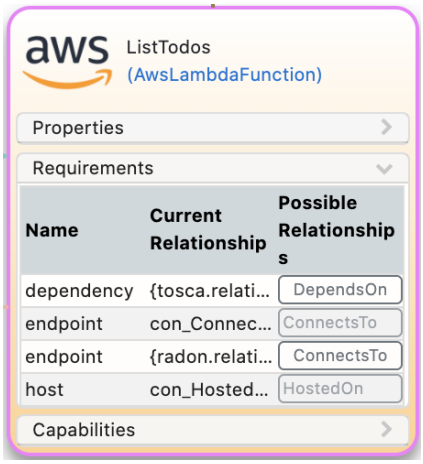



**Figure 10.2.3.5:** AwsLambdaFunction Relationships

- The requirement of the AwsLambdaFunction node "HostedOn" has to be matched with the capability of AwsPlatform node "Host".
- The requirement of the AwsLambdaFunction node "ConnectsTo" has to be matched with the capability of the DynamoDBTable node "Database".

## AwsApiGateway - Properties

Similarly, to the previous configuration, the properties must be set in the AwsApiGateway node as well. Figure 10.2.3.6 shows how this is done in this example.

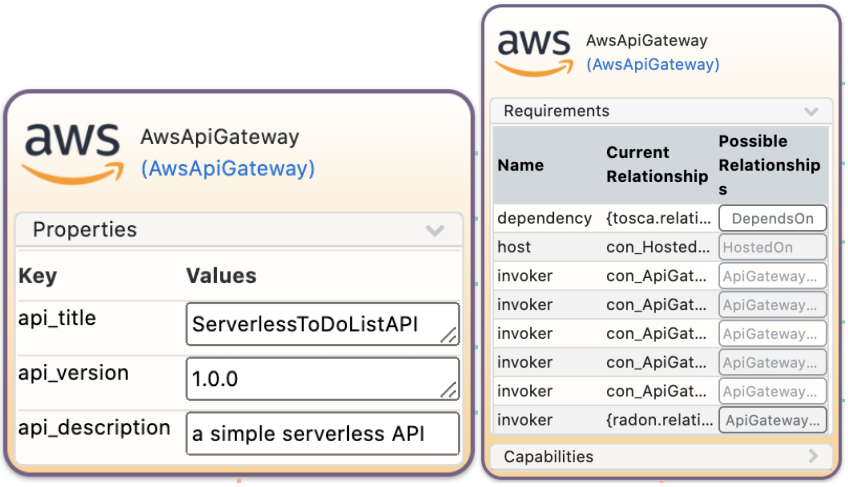



**Figure 10.2.3.6:** AwsApiGateway Properties

## AwsApiGateway - Relationships

Relationships are key parameters that define the AwsApiGateway node. Based on them, the whole API functionality is configured as they connect the API with the various cloud components that it serves. As depicted in the Figure 10.2.3.6,

- The requirement of the AwsApiGateway node “HostedOn” has to be matched with the capability of AwsPlatform node “Host”.
- The requirement “Invoker” of the AwsApiGateway node has to be matched with the capability “Invocable” of AwsLambdaFunction node.
- On the properties of the previously defined ApiGatewayTriggers relationship the keys “endpoint” and “http_methods” must be populated as shown in the Figure 10.2.3.7 below.

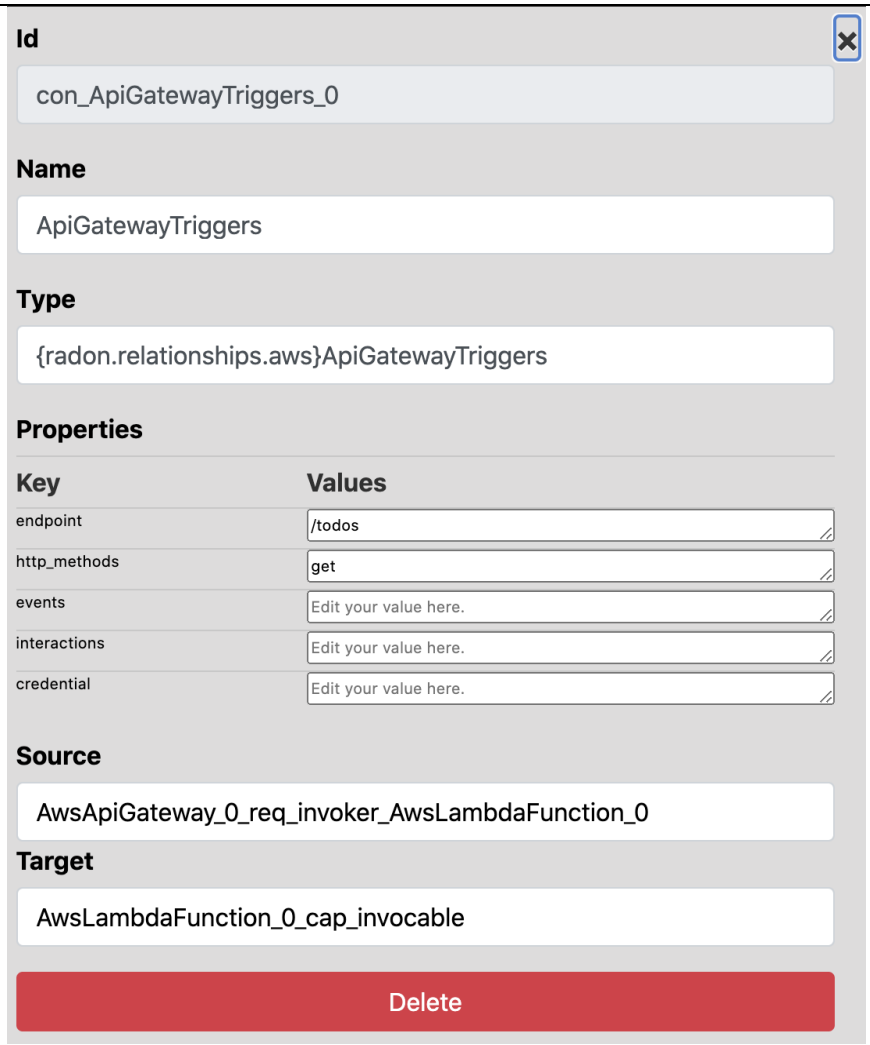



**Figure 10.2.3.7:** ApiGatewayTriggers Relationship Properties

## AwsDynamoDB - Properties

The properties of the AwsDynamoDB node are configured as depicted in Figure 10.2.3.8.

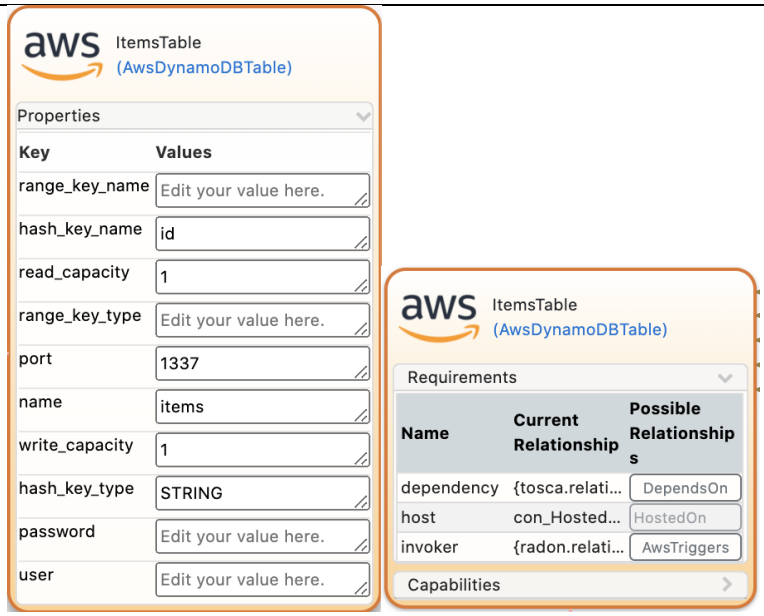



**Figure 10.2.3.8:** AwsDynamoDB Properties

## AwsDynamoDB - Relationships

Same as for other resource types, the requirement of the AwsDynamoDBTable node "HostedOn" has to be matched with the capability of AwsPlatform node "Host". The final topology of ServerlessToDoListAPI modeled on GMT is presented in Figure 10.2.3.9 below.

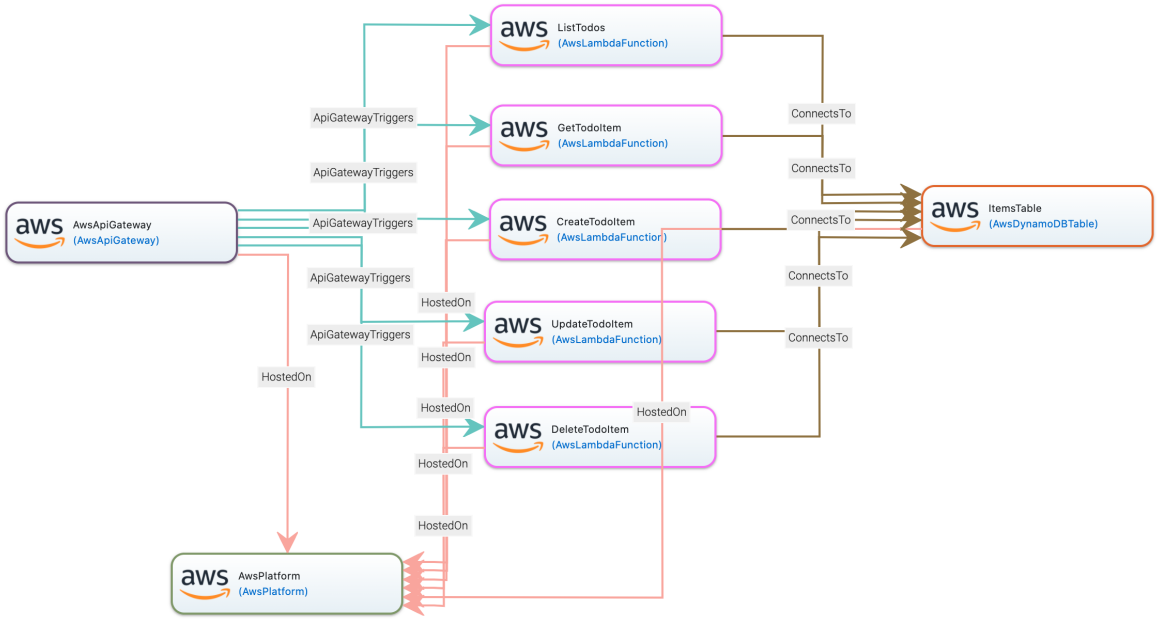



**Figure 10.2.3.9:** ServerlessToDoListAPI complete topology model

### 10.2.4. VT

The primary purpose of the verification tool is to allow a user to verify that a given FaaS architecture complies with a set of functional and non-functional requirements. These requirements are encoded using a new language, based on logic, called the *constraint definition language* (CDL).

The tool takes as input a FaaS architecture and a CDL specification, and outputs a list of issues with the FaaS architecture (i.e. a list of violations of the CDL specification). The user can then use this information to correct their FaaS architecture. As correction of a FaaS architecture, and specification of requirements can be a hard task for non-expert users, the tool also provides two additional modes in order to help a user correct a non-compliant architecture, and to aid the user to specify requirements in the CDL.

A more detailed technical overview can be found in the VT documentation.[9]

For the demonstrator's purpose, a simple set of constraints have been created in the corresponding CDL file named as main.cdl. The constraints state that if a lambda function accesses sensitive data (where the nodes which contain sensitive data are given in the CDL specification), then it should be hosted on the same AwsPlatform as the data. In this simple case, the constraints mean that the resource components in the model must be hosted on the same AwsPlatform. ServerlessToDoListAPI has been implemented as such, so when we invoke the verification test importing the service template of the created application, no consistencies are found as expected. The VT can be invoked choosing the option “Verify” by right clicking on the cdl file stored in our IDE workspace. Figure 10.2.4.1 shows the structure of the CDL file along with the output of the verification test on the bottom. The topology upon which the VT is invoked can be seen in Figure 10.2.3.9 above.

[9] https://radon-vt-documentation.readthedocs.io/en/latest/

```
import "_definitions/radonblueprints__ServerlessToDoListAPI.tosca";

types = {
  radon.nodes.aws.AwsLambdaFunction,
  radon.nodes.aws.AwsPlatform,
  radon.nodes.aws.AwsDynamoDBTable,
  radon.nodes.aws.AwsApiGateway,
  radon.nodes.aws.AwsS3Bucket
};

$X.host_node := $X.requirements[$Y].host.node;
lambdas      ::= $N : $N.type = radon.nodes.aws.AwsLambdaFunction;

sensitive_data = { AwsDynamoDBTable_0 };

AwsLambdaFunction_0.endpoints = { AwsDynamoDBTable_0 };
AwsLambdaFunction_1.endpoints = { AwsDynamoDBTable_0 };
AwsLambdaFunction_2.endpoints = { AwsDynamoDBTable_0 };
AwsLambdaFunction_3.endpoints = { AwsDynamoDBTable_0 };
AwsLambdaFunction_4.endpoints = { AwsDynamoDBTable_0 };


INCONSISTENCY sensitive_data_issue {
  lambda <- lambdas;
  sensitive_node <- lambda.endpoints;
  sensitive_node <- sensitive_data;
  ASSERT((lambda.host_node != sensitive_node.host_node));
};

@show lambda;
@show sensitive_node;

Problems   >_ RADON Verification Tool ×   >_ RADON Verification Tool
No inconsistencies detected.
```

**Figure 10.2.4.1:** CDL file & constraints verification test

In order to put the VT to the test, we will slightly modify the topology model of the ServerlessToDoListAPI hosting 1 of the 5 lambda functions (DeleteToDoItem) to a different region than the rest of the components and invoke the VT again. As we can see, the modified topology in Figure 10.2.4.2 does not comply with the CDL specification (AwsLambdaFunction_4 is stored on a different platform to AwsDynamoDBTable_0, and because the table contains sensitive data and is accessed by the lambda, this is a violation of the constraints). Figure 10.2.4.3 shows the output of the VT, which prompts the modeler to make modifications accordingly to resolve the inconsistency between the CDL specification and the model.

When presented with an inconsistency, the user can manually correct the model. If the user is unsure how to correct the inconsistency, they can instead define a space of possible changes to the model and select the "correct" option of the VT. The VT will then suggest (if one exists) a modification to the model which has no inconsistencies.

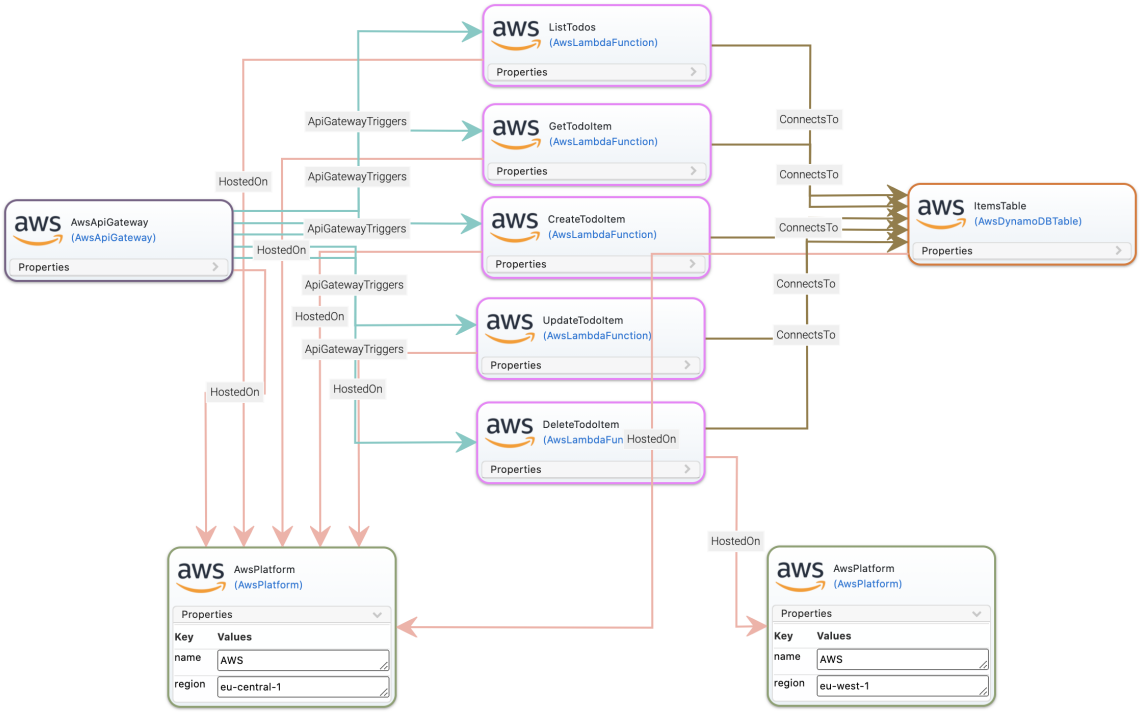



**Figure 10.2.4.2:** Modified ServerlessToDoListAPI Service Template

```
main.cdl
1  import "_definitions/radonblueprints__ServerlessToDoListAPI.tosca";
2
3  types = {
4    radon.nodes.aws.AwsLambdaFunction,
5    radon.nodes.aws.AwsPlatform,
6    radon.nodes.aws.AwsDynamoDBTable,
7    radon.nodes.aws.AwsApiGateway,
8    radon.nodes.aws.AwsS3Bucket
9  };
10
11 $X.host_node := $X.requirements[$Y].host.node;
12 lambdas      ::= $N : $N.type = radon.nodes.aws.AwsLambdaFunction;
13
14 sensitive_data = { AwsDynamoDBTable_0 };
15
16 AwsLambdaFunction_0.endpoints = { AwsDynamoDBTable_0 };
17 AwsLambdaFunction_1.endpoints = { AwsDynamoDBTable_0 };
18 AwsLambdaFunction_2.endpoints = { AwsDynamoDBTable_0 };
19 AwsLambdaFunction_3.endpoints = { AwsDynamoDBTable_0 };
20 AwsLambdaFunction_4.endpoints = { AwsDynamoDBTable_0 };
21
22
23 INCONSISTENCY sensitive_data_issue {
24   lambda <- lambdas;

Problems   >_ RADON Verification Tool

============================== Inconsistency 1 ==============================

Detected sensitive_data_issue inconsistency. The following assertions are sufficient to demonstrate the inconsistency:
  (base[0])  "lambda" = "AwsLambdaFunction_4"
  (base[0])  "lambda" = fn((fn("AwsApiGateway_0","requirements"),(at,6)),"invoker").node
  (base[0])  "sensitive_node" = "AwsDynamoDBTable_0"
  (base[0])  "sensitive_node" = fn((fn("AwsLambdaFunction_0","requirements"),(at,1)),"endpoint").node
  (base[0])  "sensitive_node" = fn((fn("AwsLambdaFunction_1","requirements"),(at,1)),"endpoint").node
  (base[0])  "sensitive_node" = fn((fn("AwsLambdaFunction_2","requirements"),(at,1)),"endpoint").node
  (base[0])  "sensitive_node" = fn((fn("AwsLambdaFunction_3","requirements"),(at,1)),"endpoint").node
  (base[0])  "sensitive_node" = fn((fn("AwsLambdaFunction_4","requirements"),(at,1)),"endpoint").node
```

**Figure 10.2.4.3:** CDL file & constraints verification test

### 10.2.5. Defect Prediction Tool

The Defect prediction tool is used for quantifying the characteristics of an IaC blueprint and predicting its defect-proneness.

The DPT can be used in the IDE as a plugin, providing metrics of TOSCA and Ansible files as output. Besides, it can also be used in the context of a CI/CD pipeline. A detailed explanation of the tool can be found in DPT documentation.

In this example, we invoke the DPT by performing "Run detection" on the deployment csar of our final application. A new tab called "Receptor" opens in our IDE workspace, showing a list with all the suitable for defect detection TOSCA files. A more analytical set of metrics is given to the user by clicking on them, highlighting whether each file is defective or not and providing an interpretation of the prediction.

Figure 10.2.5.1 depicts a usage example of DPT, characterizing one of the tosca files found inside the csar structure.

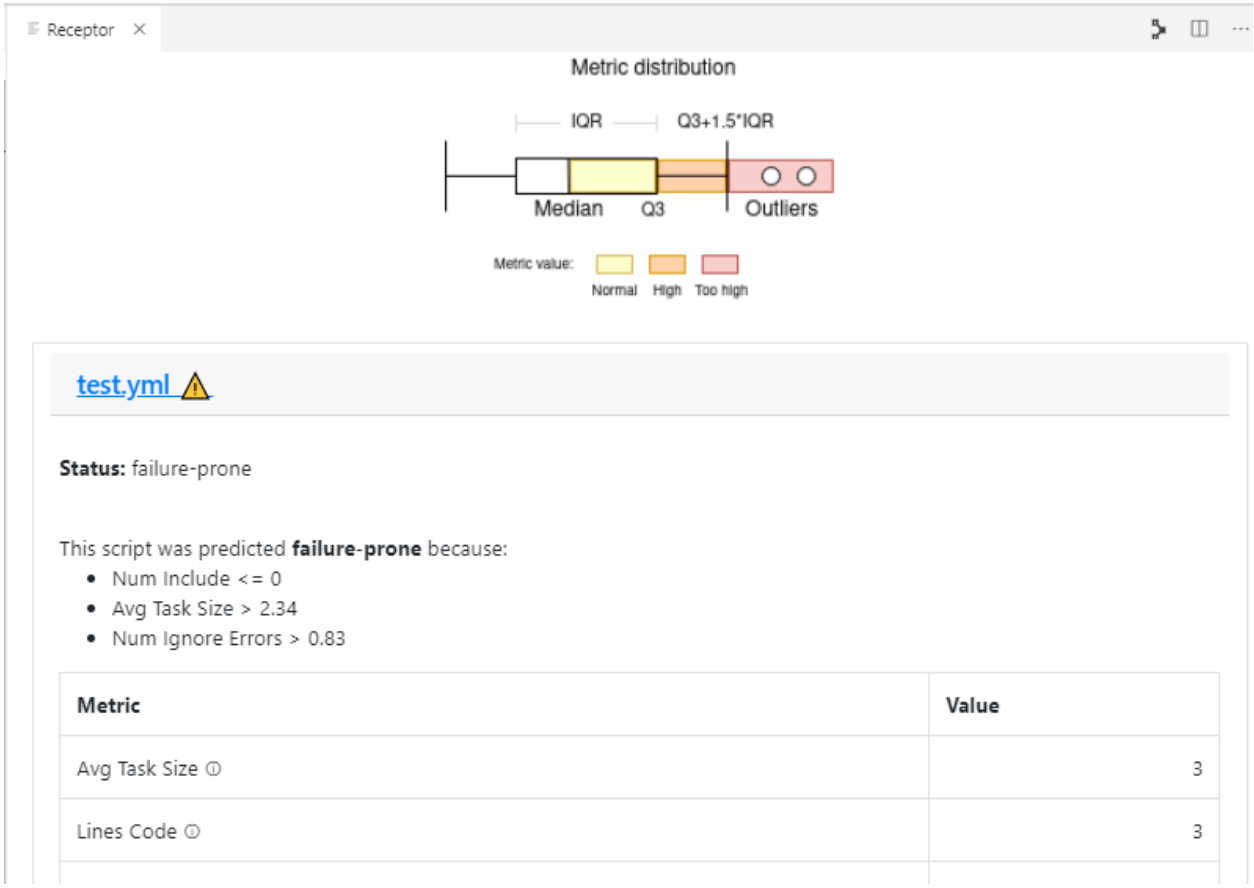



**Figure 10.2.5.1:** The “Receptor” panel of DPT

### 10.2.6. CTT

The Continuous Testing Tool (CTT) provides the means to deploy the application that is supposed to be tested, the so-called system under test (SUT), and a testing agent, the so-called test infrastructure (TI), that executes the defined tests against the SUT. After the deployment has succeeded, the defined

test is executed, and the results are obtained. Tests and test-related information are defined in the TOSCA models of the SUT and TI. The complete functionality of the tool is described in the CTT documentation[10].

In this usage description, we go through the test of the "ServerlessToDoListAPI" and an endpoint test that makes sure that the deployment was successful. The SUT is a FaaS-based implementation of a ToDo-list using AWS services, especially AWS-Lambda functions. The TI consists of a Docker container of a test agent for CTT that is deployed on top of an AWS EC2 instance.

To make this example work, some information is needed beforehand: AWS Access Key ID, AWS Secret Access Key, AWS EC2 SSH Key Type (e.g., OPENSSH, RSA), AWS EC2 SSH Key (without the header and footer sections), AWS EC2 SSH Key Name, AWS VPC Subnet ID.

The concrete steps are as follows:

1. **Preparing the Workspace with Credentials**

   In order to use CTT in the context of the RADON IDE, some credentials need to be provided when the workspace is created. The said credentials are required in order to deploy the SUT and the TI on the respective service providers' infrastructures (e.g., AWS).

   These credentials need to be filled in into the workspace configuration devfile.yaml before the workspace is created. Listing 10.2.6.1 shows an exemplary excerpt of the devfile.yaml's CTT env-section on how the fields need to be populated with the credentials.

```
env:
  - name: OPERA_SSH_USER
    value: "ubuntu"
  - name: OPERA_SSH_IDENTITY_FILE
    value: "/tmp/aws-ec2"
  - name: AWS_ACCESS_KEY_ID
    value: "AKSDF4353SFD3NMGXHERWQ"
  - name: AWS_SECRET_ACCESS_KEY
    value: "6QYMAS4sdfhAHDJ1L+pfgqZt/9OcxUN8a1/vg/ly"
  - name: KEY_TYPE
    value: "OPENSSH"
  - name: SSH_PRIV_KEY
    value: >
      c3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAA
      NhAAAAAwEAAQAAAxUA9DcKpAwyCTystithD
```

[10] https://continuous-testing-tool.readthedocs.io/en/latest/

```
[..]
Akawm0cQ55NZ76el6jzUWBDePeT7mmWUCfm
kVpfAebH2+m6/F/KpFE2Q8aFBhWSVD3SmX5
YPAAAAAAECCwQ=
```

**Listing 10.2.6.1:** Exemplary devfile.yaml environment-section for CTT credentials

Once these variables are set, the workspace can be created.

## 2. Attaching a Test Policy to the "ServerlessToDoList"

In order to assign the required information for testing an application (named "system under test" or abbreviated as "SUT") with CTT, first a model in GMT needs to be created. Within the model of the SUT, so-called policies add the information about the tests that CTT will later execute.

In GMT, open your SUT in the topology modeler. On the top menu, click on the 'Manage Policies' button. In the now opened dialogue, click on the blue button labeled 'Add'. Then, enter a name for the policy you are about to create. For example, "ToDoListEndpointTestPolicy" and choose the matching policy type for the test, which is "{radon.policies.testing}HttpEndpointTest" in this case. Once you click on "Add", the new policy is created and is shown in the lower part of the dialog. To provide test-specific details, select the newly created policy, which extends the dialog with the available properties, as seen in Figure 10.2.6.1.

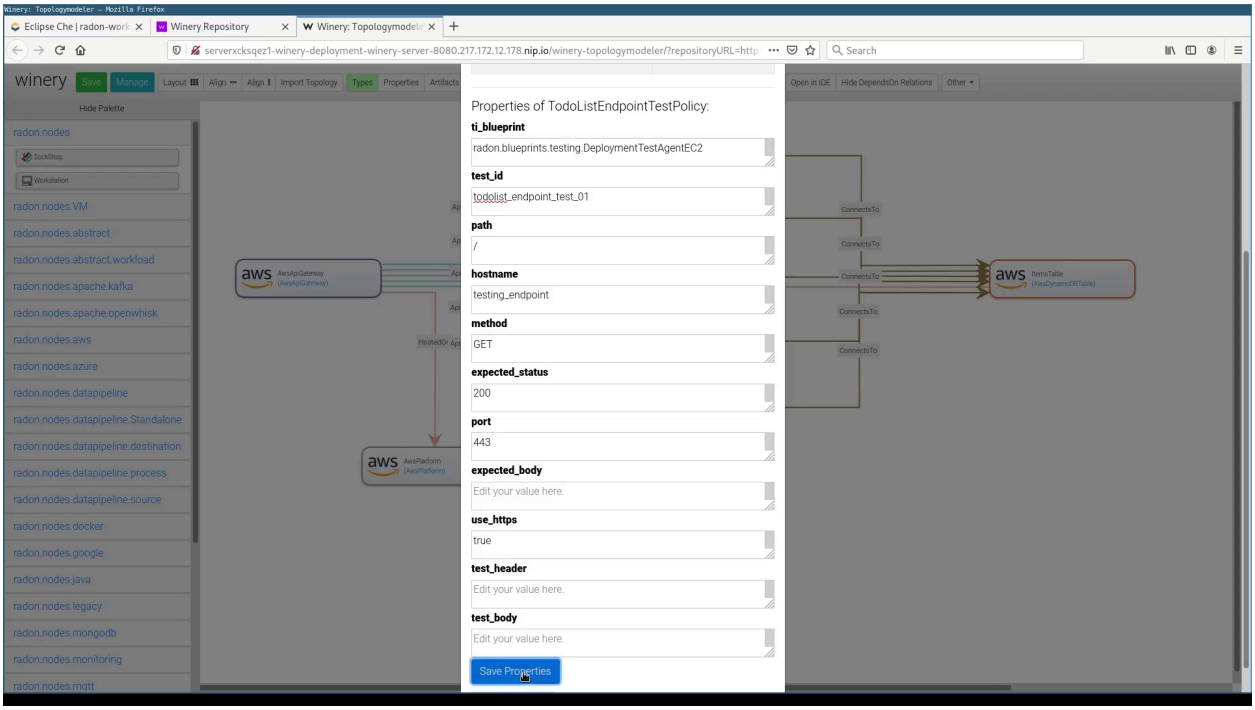



**Figure 10.2.6.1:** Screenshot of Test Policy Property Dialog

In the following, only the fields required for the current scenario will be covered.

The property ti_blueprint defines the blueprint of the test infrastructure the test is supposed to be executed with. The test_id is a unique identifier for the test that can be defined individually.

The following properties, comprising path, hostname, method, expected_status, port, and use_https represent the HTTP parameters for a request to the target system. The property hostname can either be filled with a fixed hostname (e.g., google.com) or can take the name of a TOSCA output of the SUT (e.g., the dynamically created hostname of a system deployed on an AWS EC2 instance).

The remaining fields are not mandatory in the current example and can be left empty.

Once the properties of the testing policy have been entered and saved, the testing policy needs to be assigned to a component in the model.

In order to do so, click on the "Policies" button on the top menu and extend the policies for the component of your choice. The previously created policy is listed there and can be activated by checking the checkbox under the "Is Activated?" label as depicted in Figure 10.2.6.2.

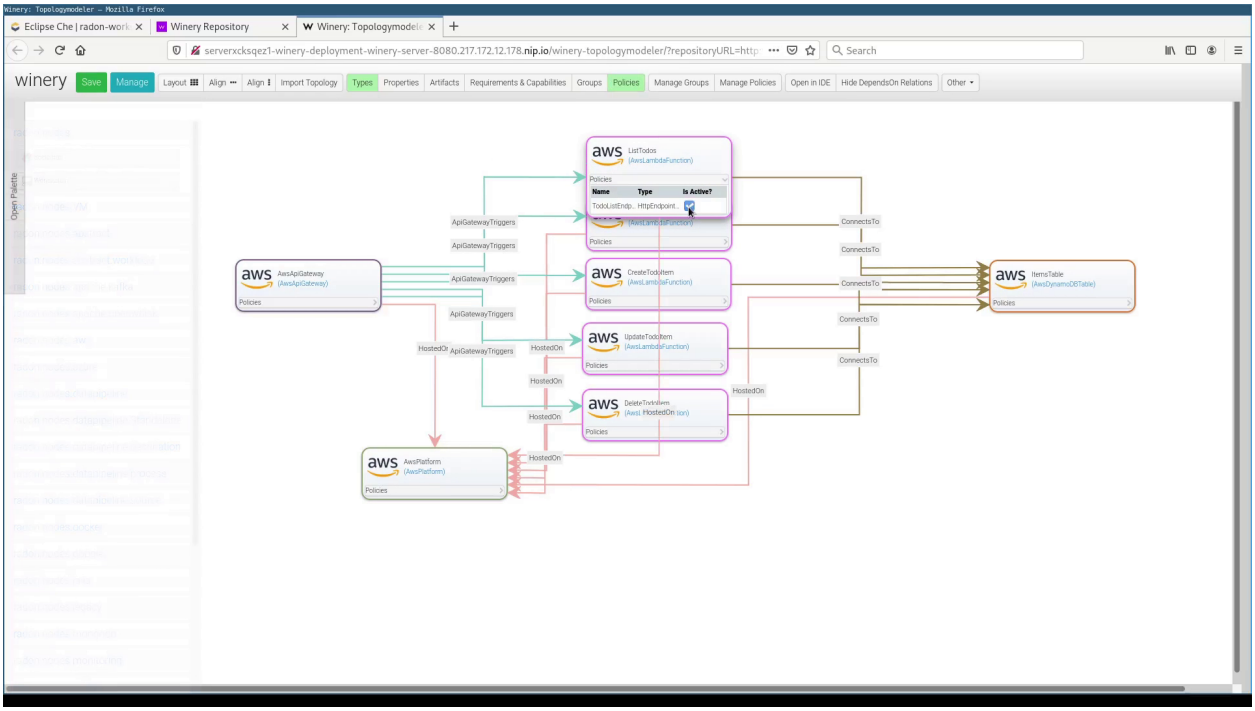



**Figure 10.2.6.2:** Screenshot of Test Policy Activation

### 3. Configuring the Test Scenario

Once the workspace is started and completely loaded, we create a new directory that holds all files that are needed to execute CTT. In this example, we name it "ServerlessToDoList". The CSAR files of the Serverless ToDo-List API service template and the CTT DeploymentTest

agent are put into this directory, as well as an inputs.yaml file that provides some inputs needed for the deployment of the TI. The fields to fill are highlighted in bold in Listing 10.2.6.2.

```
---
vpc_subnet_id: "subnet-04706a8b41abdefa5"
ssh_key_name: "awsec2"
ssh_key_file: "/tmp/aws-ec2"
...
```

**Listing 10.2.6.2:** Exemplary inputs.yaml file

The configuration of the CTT execution itself is specified by a YAML configuration file. In this file, the following properties need to be defined:

- Name for the test configuration
- Folder, the artifacts are placed in
- SUT CSAR path (relative to the folder)
- SUT inputs file (optional, relative to the folder)
- TI CSAR path (relative to the folder)
- TI inputs file (optional, relative to the folder)
- Test Id of the test to be executed (not yet taken into account)
- Results output file path (relative to configuration file)

In 10.2.6.3, you can find an exemplary CTT configuration file named ctt_config.yaml

```
{
  "name": "ServerlessToDoList-DeploymentTest",
  "repository_url": "ServerlessToDoList",
  "sut_tosca_path": "todolist.csar",
  "ti_tosca_path": "deploymentTestAgent.csar",
  "ti_inputs_path": "inputs.yaml",
  "test_id": "test_1",
  "result_destination_path": "serverless-test-results.zip"
}
```

**Listing 10.2.6.3:** Exemplary ctt_config.yaml file

Please note that the folder property is currently named repository_url for historical reasons. In the future, this property will be renamed.

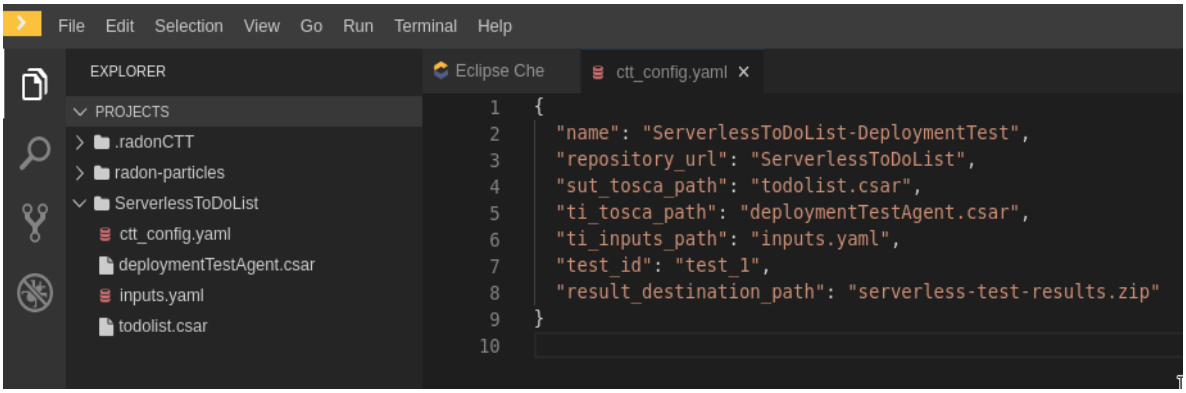



**Figure 10.2.6.3:** ServerlessToDoListAPI scenario in the RADON IDE

### 4. Executing CTT (using the RADON IDE)

After all preparations are finished, you can right-click on the ctt_config.yaml file and choose the option RadonCTT: Execute test configuration.

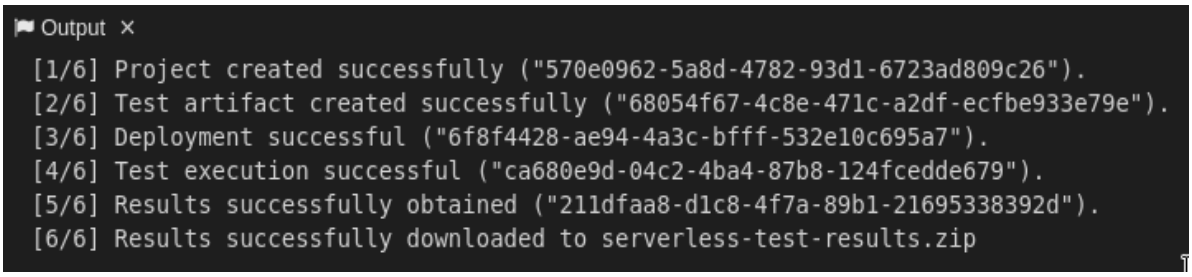



**Figure 10.2.6.4:** Progress log in the output panel

The progress can be seen in the output panel (see Figure 10.2.6.4) and a progress bar appears on the lower right. Depending on the underlying infrastructure, this process can take some time until the process is finished. Once the process is finished, you find the results in a ZIP-file located where you specified the result_destination_path.

### 5. Executing CTT (using the CTT CLI Tool)

In addition to the possibility to execute CTT from within the RADON IDE, we also provide a command-line tool called the “CTT CLI Tool” which allows the execution of CTT from the command line. The CLI Tool can be used as part of continuous integration or in any other kind of automated process.

Similarly, to the execution using the RADON IDE, the CTT CLI Tool, which is written in Python, uses the same configuration file format (see Listing 10.2.6.3) to define the parameters for a test execution.

Listing 10.2.6.4 shows the usage of the CTT CLI Tool and the respective invocation for the example configuration file. The two mandatory parameters are the URL of the CTT server as well as the configuration file that should be used for the execution.

```
% ./ctt_cli.py --help
ctt-cli.py [PARAMS]
   Mandatory parameters:
      -u, --url=CTT_SERVER_URL    URL of the CTT server
      -c, --config=CTT_CONFIG     Path to the CTT configuration file

   Other parameters:
      -v, --verbose               Be verbose
      -h, --help                  Print this help

% ./ctt_cli.py -u "http://localhost:18080/RadonCTT" -c ctt_config.yaml
```

**Listing 10.2.6.4:** Usage and invocation of the CTT CLI Tool

### 10.2.7. Orchestrator

There are multiple ways how users can use the orchestrator inside RADON. First, and most convenient option is to a) exploit the possibilities of the xOpera SaaS orchestrator, which means that user can directly deploy things from RADON IDE, and also can call the xOpera SaaS from other tools, e.g., inside CI/CD jobs. In this approach all secret management is done inside xOpera SaaS. The second option is to ii) exploit the CLI version of xOpera, where the user can download TOSCA CSAR from IDE or TPS and deploy directly from his machine, where he has full control of the orchestrator environment and in this case all responsibilities of secret management are in users' domain. Likewise, CI/CD can use the CLI version of xOpera, by installing xOpera first and then deploying everything with it.

**Using xOpera SaaS**

The documentation how the xOpera SaaS is used is available on the xOpera documentation site[11] and for the seamless usage from IDE, xOpera SaaS needs to be configured first. The part of xOpera SaaS configuration, which cannot be done from IDE directly - due to the security reasons - is managing secrets, like SSH keys and cloud provider credentials and assigning those to the workspaces.

Deploying ServerlessToDoList API using IDE and xOpera SaaS is quite straightforward. All you need to do is use a right click on a CSAR file in IDE (in this case ServerlessToDoListAPI.csar) and choose "Create a project from CSAR" option (see Figure 10.2.7.1). Afterwards, a user can follow the

[11] https://xlab-si.github.io/xopera-opera/saas.html

instructions that result in creating a new xOpera SaaS project from the CSAR and, if user requests, deploying it directly from IDE. In detail instructions collect the following data from the user:

- Choosing workspace where the project will be created (or creating a new one)
- A new name for a project
- Choose if the user wants to deploy CSAR directly after the project is created. If yes, then
    - a name of the service template (starting one) needs to be provided.
    - The inputs file (it can be empty) needs to be provided.

After the successful project creation or start of the deployment, the user can check the progress of the deployment directly in xOpera SaaS page (see Figure 10.2.7.2), which automatically opens after the successful creation of xOpera SaaS project from IDE. Users can further monitor or create new orchestrator invocations directly with xOpera SaaS.

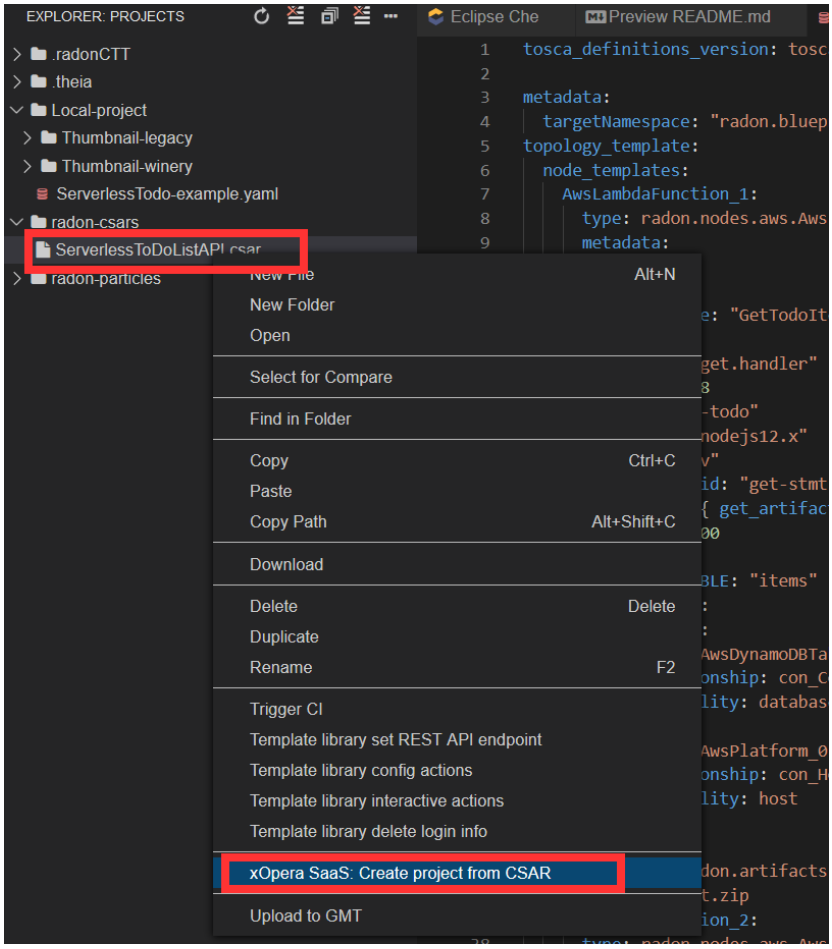



**Figure 10.2.7.1:** Creating a new project from CSAR using xOpera SaaS plugin

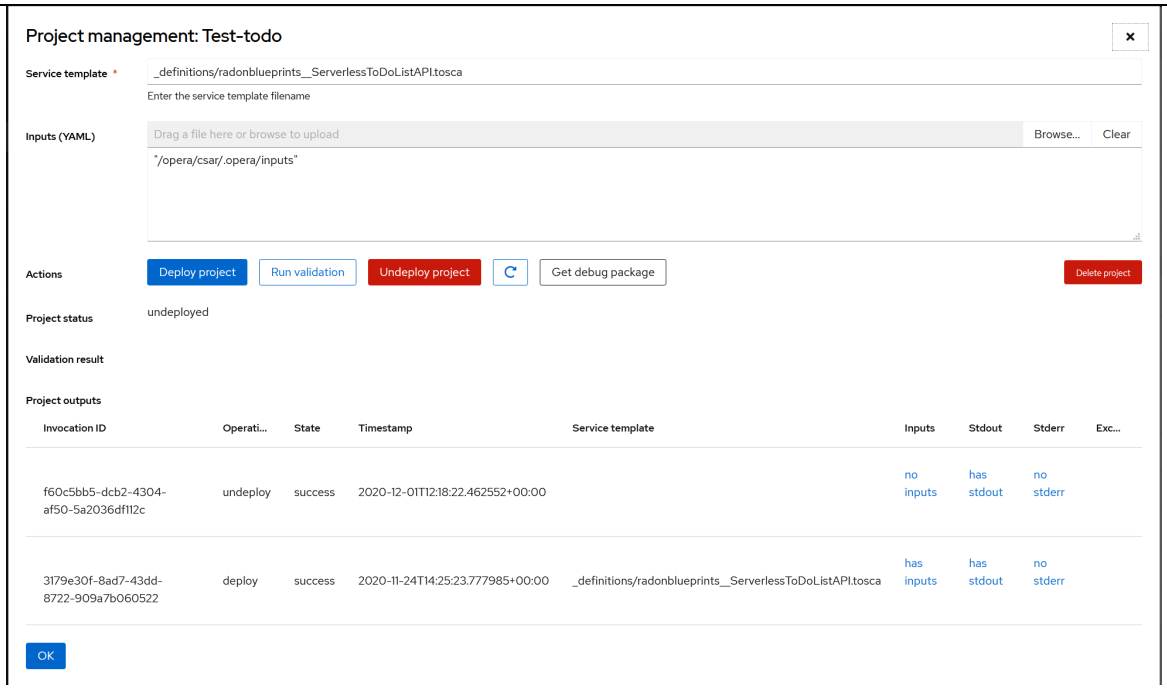



**Figure 10.2.7.2:** xOpera SaaS project management

## Using xOpera CLI

Even though the xOpera SaaS presents a very convenient way to orchestrate the project, the user can still use the power of xOpera CLI if this suits him better for whatever reason. In this case user needs to install xOpera CLI first, set up the required secrets and download the CSAR file from the IDE. Afterwards, user can issue a command “opera deploy ServerlessToDoListAPI.csar” and application will be deployed on his behalf on the AWS.

```
→  ServerlessToDoListAPI . ~/opera/.venv/bin/activate

(.venv) →  ServerlessToDoListAPI opera init ServerlessToDoListAPI.csar
CSAR was initialized
(.venv) →  ServerlessToDoListAPI opera deploy
[Worker_0]   Deploying AwsPlatform_0_0
[Worker_0]     Executing configure on AwsPlatform_0_0
[Worker_0]   Deployment of AwsPlatform_0_0 complete
[Worker_0]   Deploying AwsDynamoDBTable_0_0
[Worker_0]     Executing create on AwsDynamoDBTable_0_0
[Worker_0]   Deployment of AwsDynamoDBTable_0_0 complete
[Worker_0]   Deploying AwsLambdaFunction_1_0
[Worker_0]     Executing create on AwsLambdaFunction_1_0
[Worker_0]   Deployment of AwsLambdaFunction_1_0 complete
[Worker_0]   Deploying AwsLambdaFunction_2_0
[Worker_0]     Executing create on AwsLambdaFunction_2_0
[Worker_0]   Deployment of AwsLambdaFunction_2_0 complete
[Worker_0]   Deploying AwsLambdaFunction_0_0
[Worker_0]     Executing create on AwsLambdaFunction_0_0
[Worker_0]   Deployment of AwsLambdaFunction_0_0 complete
[Worker_0]   Deploying AwsLambdaFunction_3_0
[Worker_0]     Executing create on AwsLambdaFunction_3_0
[Worker_0]   Deployment of AwsLambdaFunction_3_0 complete
[Worker_0]   Deploying AwsLambdaFunction_4_0
[Worker_0]     Executing create on AwsLambdaFunction_4_0
[Worker_0]   Deployment of AwsLambdaFunction_4_0 complete
[Worker_0]   Deploying AwsApiGateway_0_0
[Worker_0]     Executing create on AwsApiGateway_0_0
[Worker_0]     Executing pre_configure_source on AwsApiGateway_0_0--AwsLambdaFunction_0_0
[Worker_0]     Executing pre_configure_source on AwsApiGateway_0_0--AwsLambdaFunction_1_0
[Worker_0]     Executing pre_configure_source on AwsApiGateway_0_0--AwsLambdaFunction_2_0
[Worker_0]     Executing pre_configure_source on AwsApiGateway_0_0--AwsLambdaFunction_3_0
[Worker_0]     Executing pre_configure_source on AwsApiGateway_0_0--AwsLambdaFunction_4_0
[Worker_0]     Executing configure on AwsApiGateway_0_0
[Worker_0]   Deployment of AwsApiGateway_0_0 complete
(.venv) →  ServerlessToDoListAPI
```

**Figure 10.2.7.3:** Deployment of ServerlessToDoListAPI using Opera CLI

### 10.2.8. Template Library

The Template Library is the place to store the TOSCA modules, corresponding Ansible playbooks and TOSCA CSARs describing a particular application. It was divided to support two different online repositories. RADON Particles is a community-based repository on GitHub, the second one is Template Library Publishing Service (TPS), which holds published TOSCA content that can be publicly available or closed.

TPS can be used directly from the RADON IDE workspace. When a new CSAR has been successfully exported to filesystem with GMT, it appears in the radon-csar folder on the left (see RADON IDE screenshot on Figure 10.2.8.1). Right clicking on the CSAR opens a dropdown menu, containing Template Library Publishing Service plugin's actions. Publishing the content requires some additional data about the template which a user can add interactively - by choosing Template Library interactive actions in the dropdown - or using a config file - by choosing the Template Library config

actions in the dropdown. After providing the data and RADON user credentials for TPS the CSAR is published to the configured endpoint. The details of this process are described in the documentation[12].

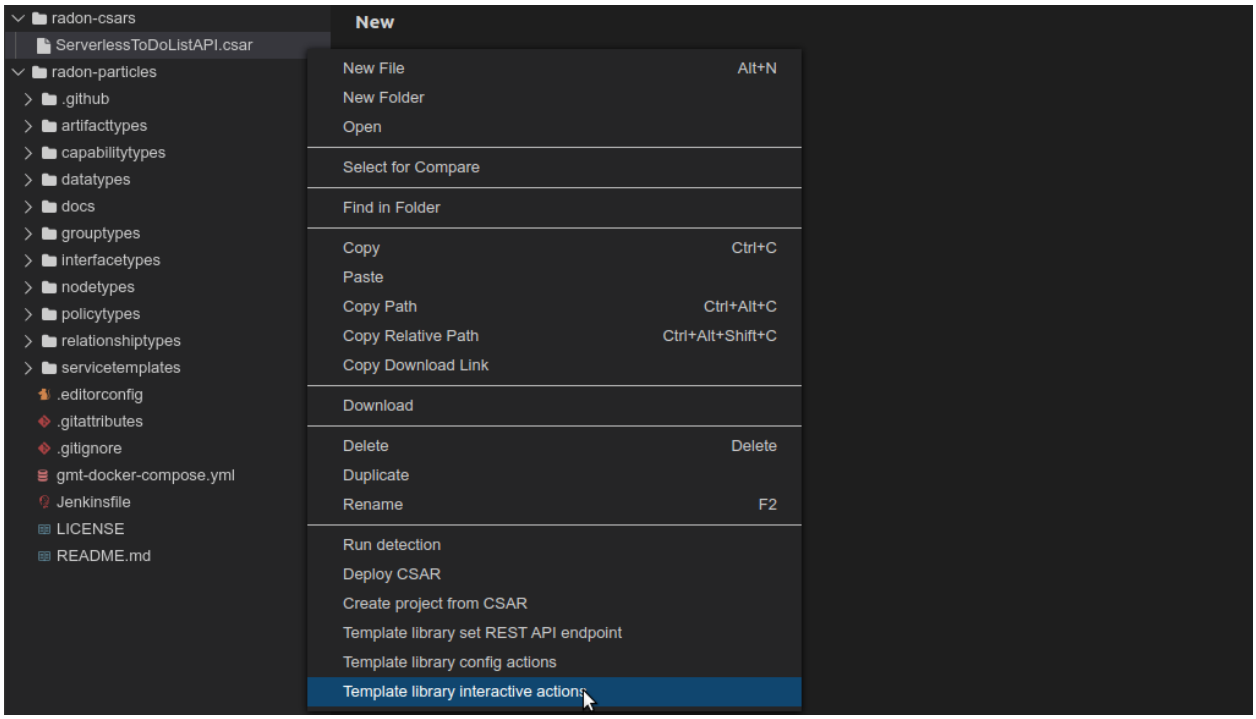



**Figure 10.2.8.1:** Right click on a file opens a dropdown menu with TPS actions

### 10.2.9. CI/CD

Most of the RADON tools support command-line execution and so can be integrated in the context of a CI/CD pipeline in the platform of the user's choice.

A set of CI/CD templates are provided by RADON to guide the user through the implementation of the tools in such a pipeline. The templates are stored in GitHub[13]. In the Github repository a directory including CI/CD templates is maintained and frequently updated for every tool that has been successfully validated and used in either CI or CD.

The officially supported CI/CD platform is Jenkins, thus the CI/CD plugin integrated in the IDE is configured to trigger a "job" on a Jenkins platform, but nevertheless templates have been created and validated for Jenkins[14] and CircleCI[15] so that the user can have more flexibility choosing CI/CD technology.

Necessary prerequisites to enable CI/CD using Jenkins:

[12] https://template-library-radon.xlab.si/docs/

[13] https://github.com/radon-h2020/radon-cicd-templates

[14] https://www.jenkins.io/

[15] https://circleci.com/

- Access to a Jenkins instance and project creation
- Set up of the Jenkins project to execute a pipeline or a GitHub repository configured with a webhook trigger
- Set up of AWS credentials as secret environment variables in the Jenkins server

The user can simply use one of the existing templates[16] for his/her own project after modifying the necessary properties or combining multiple templates in order to compose one of higher complexity that includes multiple tools in one single pipeline.

In Listing 10.2.9.1 an exemplary Jenkinsfile for VT is shown. This example is configured to execute VT assuming that the deployment csar file is stored in the same GitHub. directory as the Jenkinsfile.

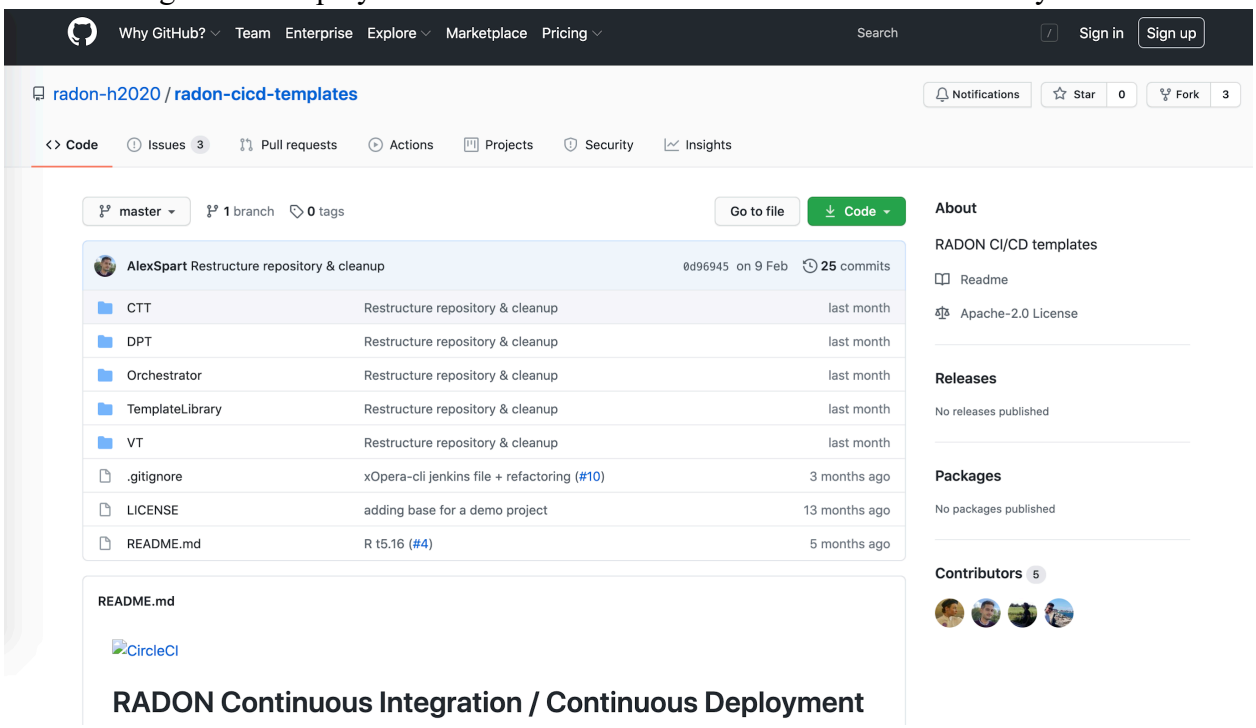



**Figure 10.2.9.1:** CI/CD templates Github repository

```
pipeline {
    agent any
    stages {
        stage('Run VT') {
            environment {
                // The deployable csar file as exported from Radon-GMT
                DEPLOY_FILE = '<name_of_csar.csar>'
                // Specify the name of the container
                VT_DOCKER_NAME = 'RadonVT'
```

[16] https://github.com/radon-h2020/radon-cicd-templates

```
            // The VT docker image published in Dockerhub
            VT_DOCKER_IMAGE = 'marklawimperial/verification-tool'
            // The path to the main.cdl file inside the container
            VT_FILES_PATH = '{"path":"/tmp/main.cdl"}'
        }
        steps {
            // Pull the latest image of VT
            sh 'docker pull $VT_DOCKER_IMAGE'
            // Unzip the csar
            sh 'unzip $DEPLOY_FILE'
            // Move relevant files to temp folder.
            sh 'mkdir -p tmp/radon-vt && cp -r _definitions tmp/radon-vt/_definitions && cp radon-vt/main.cdl tmp/radon-vt'
            // Run Verification Tool as Docker image and open port 5000
            sh 'docker run --name $VT_DOCKER_NAME --rm -d -p 5000:5000 -v $PWD/tmp/radon-vt:/tmp $VT_DOCKER_IMAGE'
            // Wait some sec for the container to spin up
            sh 'sleep 5'
            // Verify the model with the main.cdl restrictions. - Detect inconsistencies
            sh 'curl -X POST -H "Content-type: application/json" http://localhost:5000/solve/ -d $VT_FILES_PATH'
            // Correct the model to comply with the main.cdl restrictions. - Propose correction of inconsistencies
            sh 'curl -X POST -H "Content-type: application/json" http://localhost:5000/correct/ -d $VT_FILES_PATH'
            // Stop the container
            sh 'docker stop $VT_DOCKER_NAME'
        }
    }
  }
  post {
    always {
      cleanWs()
    }
  }
}
```

**Listing 10.2.9.1:** Exemplary Jenkinsfile for VT

A (che) plugin has been implemented and integrated in the RADON IDE so that a user can configure and trigger CI/CD pipelines directly from the RADON IDE. In particular, the user can edit within the

RADON IDE a configuration file (i.e. a yaml file) in order to specify the CI/CD pipeline to trigger in a Jenkins server. The parameters to set are listed below and in Figure 10.2.9.3:

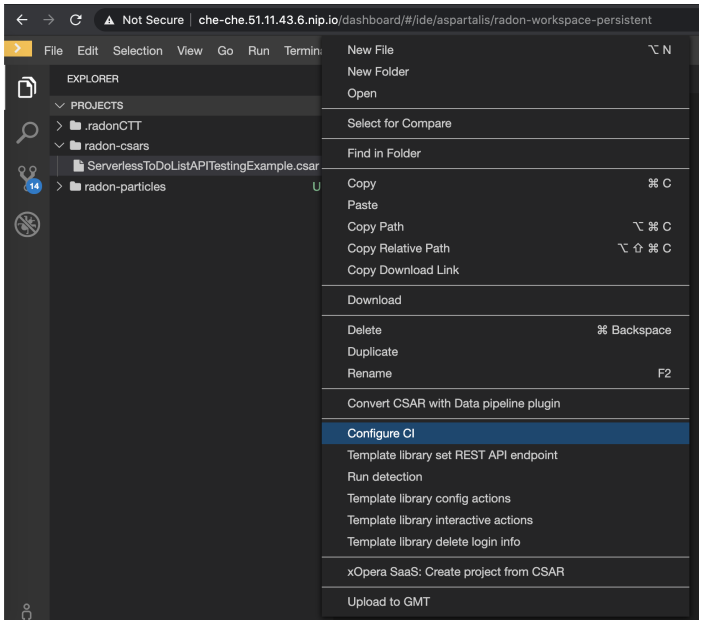



**Figure 10.2.9.2:** CI/CD Instantiate configuration file

- *CSAR_name*: The name of the CSAR as uploaded in the Template Library;
- *CSAR_version*: The version of the CSAR as as uploaded in the Template Library;
- *Jenkins_URL*: The URL of the Jenkins server;
- *Jenkins_username*: The username of Jenkins credentials;
- *Jenkins_password*: The password of Jenkins credentials;
- *Jenkins_job*: The job (i.e. CI/CD pipeline) that must be triggered;
- *Jenkins_job_token*: The Authentication Token associated to the job;
- *cookie_jar*: Parameter used to get a Jenkins crumb. Use the value /tmp/cookies

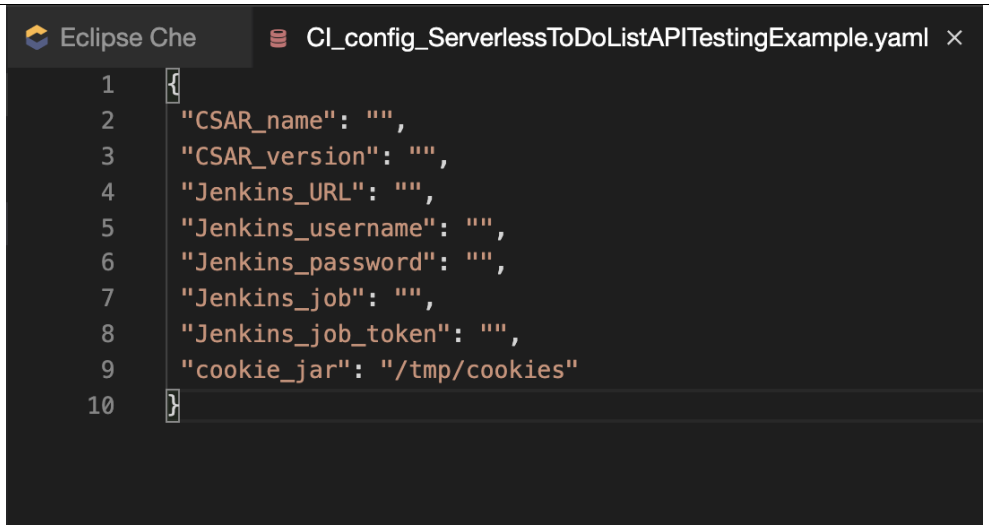



**Figure 10.2.9.3:** CI/CD configuration file

A detailed description of the CI/CD functionalities integrated in the RADON IDE is provided in the deliverable D2.7.

### 10.2.10. Data Pipelines

Data pipeline focuses on flow of data across multiple private and public clouds. Serverless platform is integrated allowing the developers to process the data on the fly. In addition, a number of features such as encrypting the data, support for logging are provided by the data pipeline TOSCA models. The developed TOSCA models can be found in the official radon-particle GitHub repo[17]. However, the given "ServerlessToDoListAPI" example can be further improved with additional features such as taking backup of the AWS DynamoDB data into S3 bucket and Google Cloud Storage. In precise, we will add the following capabilities to the example for backup purpose.

- Copy the data from AWS DynamoDB to AWS S3 bucket.
- Copy the data in AWS S3 Bucket to Google Cloud Storage (GCS) bucket for backup purpose.

This would demonstrate the advantage of the data pipeline in smooth data flow from one cloud platform to another addressing the vendor lock-in issue. For this, the "ServerlessToDoListAPI" service blueprint is modified by adding a number of pipeline nodes highlighted in the Figure 10.2.10.1.

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

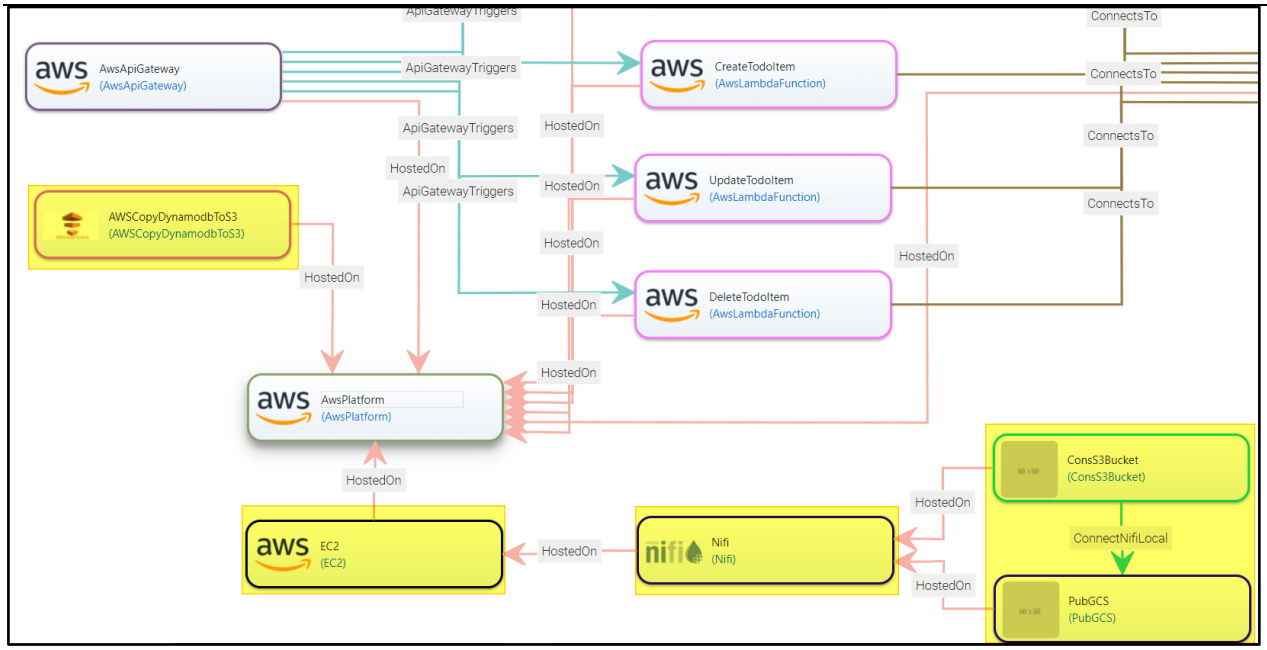



**Figure 10.2.10.1:** Service blueprint of ServerlessToDoListAPI with highlighted data pipeline nodes.

### Pipeline Platform

For the implementation of pipeline related nodes, we need mainly two pipeline platforms: AWS data pipeline (as commercial data management solution) and Apache Nifi (open-source data management solution).

1. ***AWS data pipeline platform***: To setup AWS data pipeline platform, no specific node type is needed and all the AWS data pipeline related TOSCA nodes (in this example it is *AWSCopyDynamodbToS3*) will be hosted on AWSPlatform TOSCA node, as shown in Figure 10.2.10.1.

2. ***Nifi-based platform:*** To setup Apache Nifi platform, an EC2 instance is needed along with the Nifi TOSCA node. The EC2 instance can be created using *EC2* TOSCA node type. The required properties of the *EC2* node are given in the Figure 10.2.10.2. Below figure can be used as reference. We recommend using centOS image over Ubuntu image.

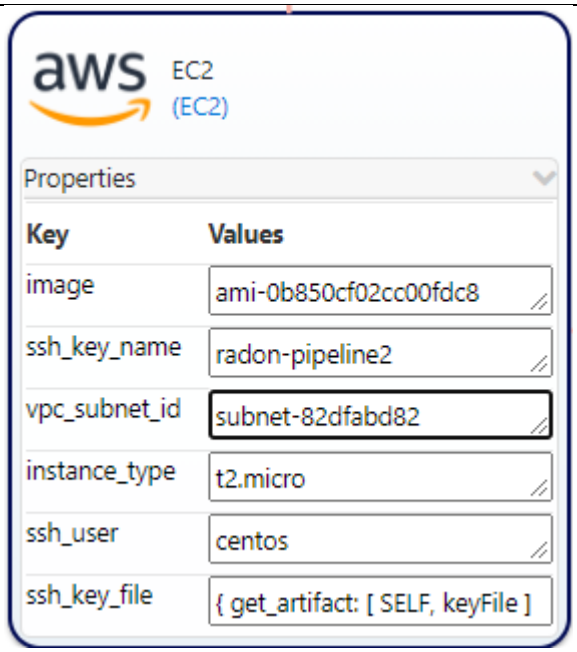



**Figure 10.2.10.2:** Properties of TOSCA *EC2* node

For Apache Nifi, the user needs to provide the required version, which is a mandatory property.

### Data Flow from AWS DynamoDB to S3 bucket

For this, the "ServerlessToDoListAPI" service blueprint is modified by adding an "*AWSCopyDynamodbToS3*" node with the following properties.

**Properties**

| Key | Values |
|---|---|
| log_directory | log |
| schedule | 1 Days |
| throughput_ratio | 0.25 |
| dynamodb_table | items |
| output_directory | items |
| destination_s3 | radon-utr-pipeline |
| log_bucket | radon-utr-pipeline |
| dynamodb_region | eu-west-1 |
| dp_name | backupToS3 |
| configure_file_path | {"get_artifact":["SELF","config"]} |
| credential_file_path | {"get_artifact":["SELF","credentials"]} |
| schedulingStrategy | EVENT_DRIVEN |
| name | backup |
| schedulingPeriodCRON | * * * * * ? |

**Figure 10.2.10.3:** Properties of *AWSCopyDynamodbToS3* TOSCA node

*AWSCopyDynamodbToS3* TOSCA data pipeline node requires *AWSPLatform* node as the host. List of properties and possible values are shown in Figure 10.2.10.3. The config and the credential files can be passed through the artifacts. Below are the content of config and credential files.

**Content of the config.txt**

---

```
[default]
region = eu-west-1
```

**Content of credentials.txt**

---

```
[default]
aws_access_key_id = <secret>
aws_secret_access_key = <secret>
```

### Data flow from AWS S3 to Google cloud storage bucket

Now to move the data from AWS S3 to Google cloud storage bucket, we need to use *PubsGCS* and *ConsS3Bucket* nifi-based node types. All NiFi-base data pipeline node types require a *NiFi* node as the host. Further, NiFi TOSCA node requires a container type TOSCA node as a hosting environment. In this example we will use *EC2* as the hosting environment for NiFi, as shown in Figure 10.2.10.2.

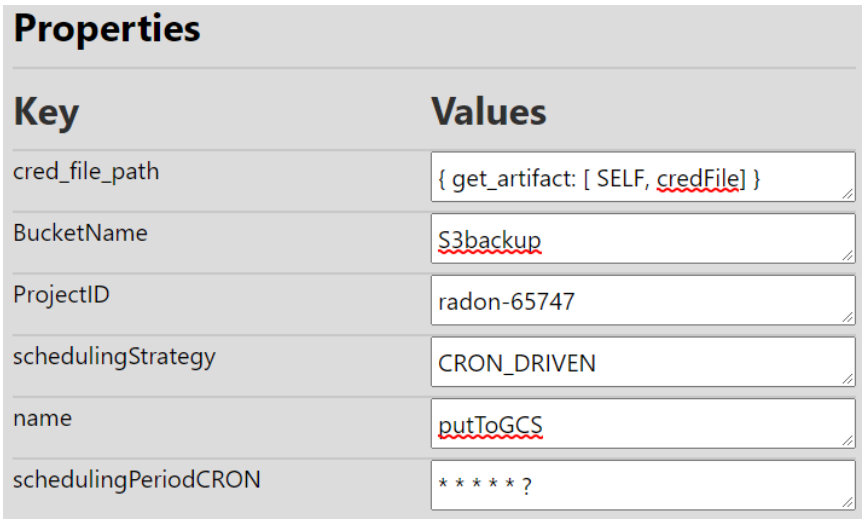



**Figure 10.2.10.4:** Properties of PubsGCS TOSCA node

Figure 10.2.10.4 shows the list of properties and possible values for PubGCS TOSCA node. One possible way to provide the credential files is through the artifacts. MAke sure that the credential file is in the JSON format which can be downloaded from the Google console. The Scheduling strategy can be left as default. *BucketName* is the name of the storage bucket, which you may get from https://console.cloud.google.com/storage/. You may also get your project id from here https://console.cloud.google.com/home/dashboard . Figure 10.2.10.5 shows all the possible properties' values for the ConsS3Bucket TOSCA node. Make sure that the S3 bucket is already created.

**Properties**

| Key | Values |
|---|---|
| cred_file_path | { get_artifact: [ SELF, credFile] } |
| BucketName | getDataFromS3 |
| Region | eu-west-1 |
| schedulingStrategy | EVENT_DRIVEN |
| name | getS3 |
| schedulingPeriodCRON | * * * * * ? |

**Figure 10.2.10.5:** Properties of *ConsS3Bucket* TOSCA node

### 10.2.11. Decomposition tool

The decomposition tool aims to help in finding the optimal decomposition solution for an application based on the microservices architectural style and the serverless FaaS paradigm. It supports three typical usage scenarios: (i) architecture decomposition, (ii) deployment optimization, (iii) accuracy enhancement.

In what follows, we show how to use the decomposition tool to obtain the optimal deployment scheme of the ServerlessToDoListAPI application, minimizing its total operating cost on the AWS platform under the specified performance requirement. The ServerlessToDoListAPI application can be considered as exposing five RESTful endpoints, namely *list*, *get*, *create*, *update* and *delete*. For brevity, we only take the *list* endpoint for example to clarify the steps toward deployment optimization. The TOSCA model including a full version of the service template, function packages and Python utilities present below can be found here in RADON Particles[18].

**Performance Modeling**

The development of the decomposition tool is based on layered queueing networks (LQNs), a performance modeling formalism applicable to most modern distributed systems. A dedicated set of data types and an abstract layer of node and relationship types are defined in RADON Particles, allowing the use of LQN annotations to describe the behavior of an application. We also define

[18] https://github.com/radon-h2020/radon-particles/tree/master/servicetemplates/radon.blueprints.optimization/ServerlessToDoListAPIOptimizationExample

practical policy types for specifying performance requirements. D3.2 Decomposition Tool I and D3.3 Decomposition Tool II present an introduction to LQNs and details about the definitions of TOSCA types specific to the decomposition tool.

Suppose that the ServerlessToDoListAPI application receives an infinite stream of requests arriving on average every 0.2 seconds and each request may be destined for the *list* endpoint with a probability of 0.2. Such a reference workload can be modeled using the *OpenWorkload_0* node defined in Listing 10.2.11.1. This node has one entry named *start*, which contains a single activity that sends requests in a stochastic pattern. The *ClosedWorkload_0* node connects to the *AwsApiGateway_0* node through the *con_ConnectsTo_5* relationship. Listing 10.2.11.2 shows the definition of the *con_ConnectsTo_5* relationship, where a synchronous interaction from the *start.a1* activity of the *ClosedWorkload_0* node to the *list* entry of the *AwsApiGateway_0* node is specified. The latter represents the *list* endpoint.

```
ClosedWorkload_0:
  type: radon.nodes.abstract.workload.ClosedWorkload
  properties:
   interarrival_time:
     mean: 0.2
   entries:
     start:
       a1:
         service_time:
           mean: 0.001
         bound_to_entry: true
         request_pattern: stochastic
  requirements:
   - endpoint:
       node: AwsApiGateway_0
       relationship: con_ConnectsTo_5
       capability: api_endpoint
```

**Listing 10.2.11.1:** Definition of the *ClosedWorkload_0* node

```
con_ConnectsTo_5:
  type: radon.relationships.ConnectsTo
  properties:
    interactions:
     - type: synchronous
```

```
      source_activity: start.a1
      target_entry: list
      number_of_requests: 0.2
```

**Listing 10.2.11.2:** Definition of the *con_ConnectsTo_5* relationship

Upon receipt of a *list* request, the *AwsApiGateway_0* node triggers the *AwsLambdaFunction_0* node through the *con_AwsApiGatewayTriggers_0* relationship. Listings 10.2.11.2 and 10.2.11.3 show the definitions of the *AwsApiGateway_0* node and the *con_AwsApiGatewayTriggers_0* relationship. The *list* entry of the *AwsApiGateway_0* node has a single activity that sends requests in a deterministic pattern and replies with responses upon completion as *list* requests are synchronous. The interaction between the *list.a1* activity of the *AwsApiGateway_0* node and the entry of the *AwsLambdaFunction_0* node is defined by the *con_AwsApiGatewayTriggers_0* relationship.

```
AwsApiGateway_0:
  type: radon.nodes.aws.AwsApiGateway
  properties:
    entries:
      list:
        activities:
          a1:
            service_time:
              mean: 0.125
            bound_to_entry: true
            replies_to_entry: true
            request_pattern: deterministic
  requirements:
    - invoker:
        node: AwsLambdaFunction_0
        relationship: con_AwsApiGatewayTriggers_0
        capability: invocable
```

**Listing 10.2.11.3:** Definition of the *AwsApiGateway_0* node

```
con_AwsApiGatewayTriggers_0:
  type: radon.relationships.aws.AwsApiGatewayTriggers
  properties:
    interactions:
      - type: synchronous
        source_activity: list.a1
```

```
      target_entry: execute
      number_of_requests: 1.0
```

**Listing 10.2.11.4:** Definition of the *con_AwsApiGatewayTriggers_0* relationship

To capture the behavior of the *AwsLambdaFunction_0* node, we introduce an entry named *execute*, as shown in Listing 10.2.11.5. This entry is defined similarly to the *list* entry of the *AwsApiGateway_0* node, because the *AwsLambdaFunction_0* node simply performs a *scan* operation on the *AwsDynamoDBTable_0* node and returns the result to the *AwsApiGateway_0* node. Listing 10.2.11.6 shows the definition of the *con_ConnectsTo_0* relationship, which connects the *AwsLambdaFunction_0* node to the *AwsDynamoDBTable_0* node and defines the interaction between the *execute.a1* activity of the former and the *scan* entry of the latter.

```
AwsLambdaFunction_0:
  type: radon.nodes.aws.AwsLambdaFunction
  properties:
    entries:
      execute:
        activities:
          a1:
            service_time:
              mean: 0.046
            bound_to_entry: true
            replies_to_entry: true
            request_pattern: deterministic
  requirements:
    - endpoint:
        node: AwsDynamoDBTable_0
        relationship: con_ConnectsTo_0
        capability: database_endpoint
```

**Listing 10.2.11.5:** Definition of the *AwsLambdaFunction_0* node

```
con_ConnectsTo_0:
  type: radon.relationships.ConnectsTo
  properties:
    interactions:
      - type: synchronous
        source_activity: execute.a1
        target_entry: scan
```

```
      number_of_requests: 1.0
```

**Listing 10.2.11.6:** Definition of the *con_ConnectsTo_0* relationship

The definition of the *AwsDynamoDBTable_0* node is shown in Listing 10.2.11.7. In the *scan* entry of the *AwsDynamoDBTable_0* node, an activity with no request pattern is defined due to the fact that it is the end of the request chain. Note that the name of each entry in an *AwsDynamoDBTable* node must be prefixed with the name of the operation that it is associated with, e.g. "get", "get_item" and "getItem". This enables the decomposition tool to compute the operating cost of the node. The supported operations for an *AwsDynamoDBTable* node include *get*, *put*, *update*, *delete*, *query* and *scan*.

```
AwsDynamoDBTable_0:
  type: radon.nodes.aws.AwsDynamoDBTable
  properties:
    entries:
      scan:
        activities:
          a1:
            service_time:
              mean: 0.003
            bound_to_entry: true
            replies_to_entry: true
```

**Listing 10.2.11.8:** Definition of the *AwsDynamoDBTable_0* node

### Benchmarking

To parametrize the mean service times of activities, we need to create a benchmark of the ServerlessToDoListAPI application. However, it is difficult to apply commonly used benchmarking methods to Lambda functions as quantities required by these techniques are not measurable on AWS. Lambda allocates dedicated memory and CPU resources to each instance of a function. The service times of activities performed by Lambda functions are invariant to the intensity of the workload, which means that we can in fact estimate the mean service times under a reference workload through direct measurement. This simple method is also applicable to API gateways and DynamoDB tables since they essentially only introduce pure delays.

Again, take the *list* endpoint for example. The execution time of the single activity carried out by the *AwsDynamoDBTable_0* node includes the service time spent in completing the activity itself and the waiting time spent in accessing the *scan* entry of the *AwsDynamoDBTable_0* node, both of which

cannot be measured separately. Let $E$, $S$ and $W$ be the mean execution, service and waiting times of this activity respectively. Then, we have

$$E = S + W \tag{1}$$

The mean service time $S$ of the activity is in inverse proportion to the memory $m$ of the function:

$$S = \frac{K}{m} \tag{2}$$

where $K$ is the corresponding proportionality constant. Note that (2) holds when running an instance of the function does not require more memory resources. Applying (2) to (1) yields

$$E = \frac{K}{m} + W \tag{3}$$

(3) and (2) provides a two-step method for service time estimation. One can estimate the proportionality constant $K$ and the mean waiting time $W$ by fitting (3) through linear regression across different memories and then calculate the mean service time $S$ for a particular memory $m$ by applying (2).

The ServerlessToDoListAPI application can be benchmarked by simulating a single client that continually sends requests to different API endpoints and recording timestamps immediately before or after critical operations at appropriate nodes. This can be done through a load testing tool such as JMeter[19] and Locust[20]. Locust is a simple Python-based tool that supports simulation of a workload with thousands of concurrent clients on a single machine. The *utilities* folder of the TOSCA model includes a locustfile, *client_benchmark.py*, for benchmarking the ServerlessToDoListAPI application. After installing Locust v0.11.0, you can run the locustfile using the command shown in Listing 10.2.11.8. The web UI of Locust will then be available locally at http://localhost:8089/, where the benchmarking procedure can be started, as shown in Figure 10.2.11.1. Note that the number of users to simulate must be set to 1. The benchmarking procedure will automatically stop when the relative errors of the estimations are less than 5% at the 99th confidence level.

```
$ locust --locustfile=client_benchmark.py --host=http://example.com
```

**Listing 10.2.11.7:** Command to run the locustfile

[19] https://jmeter.apache.org/

[20] https://locust.io/

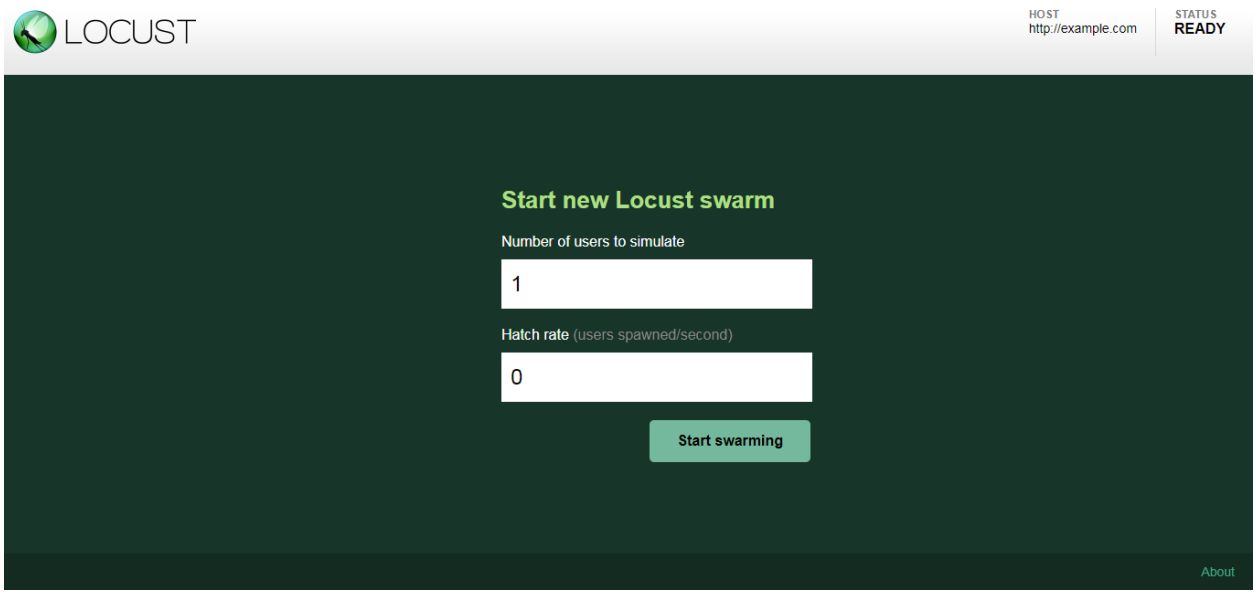



**Figure 10.2.11.1:** Web UI of Locust

Locust only provides the mean response times of API endpoints under test. To obtain the mean service times of all the activities, you also need to record timestamps at critical points within Lambda functions. The *files* folder of the TOSCA model includes a modified version of the function packages that collect such timestamps and log them to CloudWatch. Listing 10.2.11.9 shows the code of the modified *list_items* function. We record timestamps at four points: (i) the start of the function, (ii) the start of the *scan* operation, (iii) the end of the *scan* operation, (iv) the end of the function, which can be used to calculate the execution times of three activities. We combine these activities into one in the service template as the benchmarking result shows that the first and second activities are trivial.

```
'use strict';

const AWS = require('aws-sdk');
const dynamoDb = new AWS.DynamoDB.DocumentClient();

module.exports.handler = (event, context, callback) => {
    let timestamps = new Array(4);
    timestamps[0] = Date.now();
    const headers = {
        "Access-Control-Allow-Origin": "*",
        "Access-Control-Allow-Credentials": "true",
        "Content-Type": "application/json",
        "X-Requested-With": "*",
        "Access-Control-Allow-Headers":          'Content-Type,X-Amz-Date,Authorization,x-api-key,x-requested-
```

```
with,Cache-Control',
    };

    const params = {
        TableName: process.env.TODOS_TABLE
    };

    timestamps[1] = Date.now();
    dynamoDb.scan(params, (error, result) => {
        timestamps[2] = Date.now();
        if (error) {
            console.error(error);
            callback(null, {
                headers: headers,
                statusCode: error.statusCode || 501,
                body: JSON.stringify('Couldn\'t fetch the todos.'),
            });
            return;
        }

        const response = {
            statusCode: 200,
            headers: headers,
            body: JSON.stringify(result.Items)
        };
        timestamps[3] = Date.now();
        console.log(timestamps);
        callback(null, response);
    });
};
```

Listing 10.2.11.9: Code of the modified *list_items* function

To compute the mean execution time of the activity, you need to export the log data of the *list_items* function from CloudWatch to a S3 bucket for download. Please refer to the user guide of CloudWatch[21] for the detailed steps on how to do so. In the *utilities* folder of the TOSCA model, we provide a Python script, *service_time_estimator.py*, for extracting timestamps from the log data and calculating the average execution time. Suppose that the log data is saved locally as a text file named

[21] https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/S3Export.html

*list_items.log*. You can obtain the mean execution time *E* by running the command shown in Listing 10.2.11.10.

```
$ python service_time_estimator.py list_items.log
```

Listing 10.2.11.10: Command to run the locustfile

As mentioned earlier, the mean service time *S* and the mean waiting time *W* can be computed through linear regression on (3), which requires benchmarking the *list_items* function under at least two different memories. Table 10.2.11.1 reports the benchmarking results for the *list* endpoint and the *list_items* function. When the memory *m* of the function is set to 128 MB and 258 MB, the mean execution time *E* is estimated at 0.049 s and 0.026 s respectively. Applying (3), we obtain the proportionality constant *K* and the mean waiting time *w*as 29.44 s·MB and 0.003 s. The mean service time *S* can then be calculated from (2) as 0.046 s when the memory *m* of the function is 128 MB. Since the mean response time *R* of the *list* endpoint is 0.174 s in this case, the mean network delay *D* of the *list* entry at the API gateway is 0.125 s.

Table 10.2.11.1: Benchmarking results for the *list* endpoint and the *list_items* function

| Function Memory | Mean Response Time (*R*) | Mean ServiceTime (*S*) |
|---|---|---|
| 128 MB | 0.174 s | 0.049 s |
| 256 MB | 0.130 s | 0.026 s |

### Deployment Optimization

When used for deployment optimization, the decomposition tool aims to find the optimal deployment scheme that minimizes the total operating cost while satisfying the specified performance requirement. We also need to describe the expected performance in the service template before starting the optimization procedure. Two performance policy types are defined in RADON Particles for this purpose: *MeanResponseTime*[22] and *MeanTotalResponseTime*[23], which can be used to specify performance requirements on the mean response time and the mean total response time respectively. The main difference between these two policy types is that the former applies to each target node or entry individually while the latter applies to all the target nodes or entries together. Suppose that we want the mean response time of the application to be no more than 0.2 s. This performance requirement can be expressed using the *MeanResponseTime_0* policy shown in Listing 10.2.11.11.

[22] https://github.com/radon-h2020/radon-particles/tree/master/policytypes/radon.policies.performance/MeanResponseTime

[23] https://github.com/radon-h2020/radon-particles/tree/master/policytypes/radon.policies.performance/MeanTotalResponseTime

```
- MeanResponseTime_0:
    type: radon.policies.performance.MeanResponseTime
    properties:
      upper_bound: 0.2
    targets:
      - AwsApiGateway_0
```

Listing 10.2.11.11: Definition of the *MeanResponseTime_0* policy

Each time a new RADON workspace is created, RADON Particles is cloned in the *projects* folder of the workspace. The aforementioned TOSCA model for the ServerlessToDoListAPI application is thus available in the workspace, as shown in Figure 10.2.11.2. To optimize the TOSCA model, right-click on either the service template, *ServiceTemplate.tosca*, and select the *Optimize* option. The execution of the optimization procedure will be displayed in the *Output* window (Ctrl+Shift+U to open), as shown in Figure 10.2.11.3. After the optimization procedure completes, the service template will be updated according to the optimal deployment scheme, and extra information about the optimal solution will also be given in the *Output* window, including the total operating cost and the performance measures.

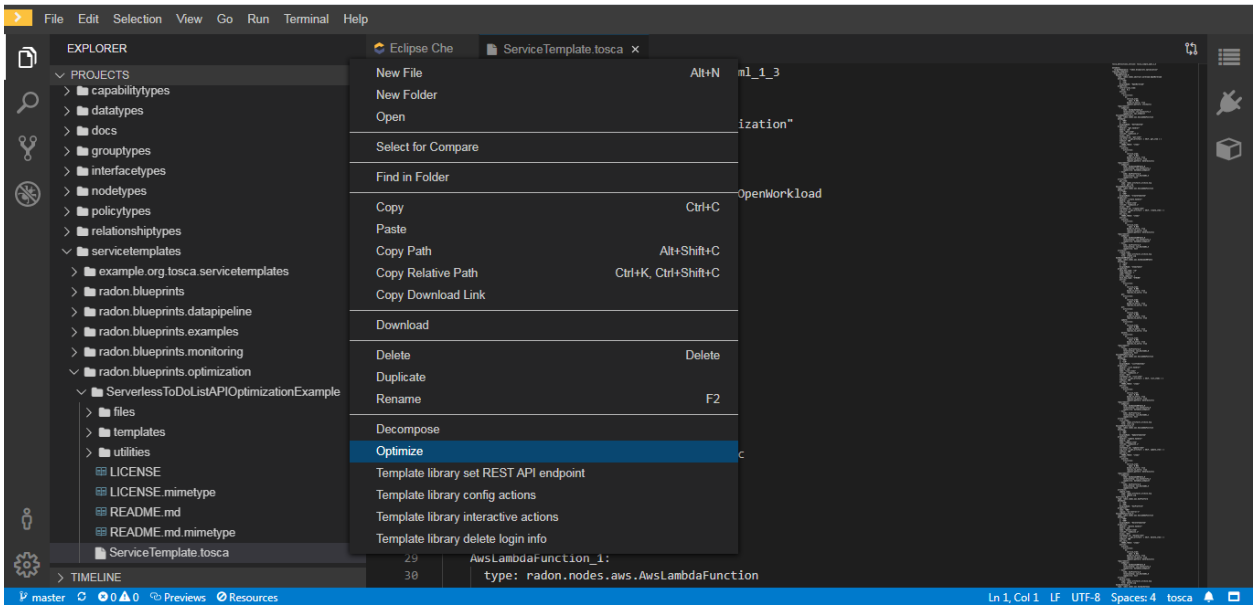



Figure 10.2.11.2: Start of deployment optimization

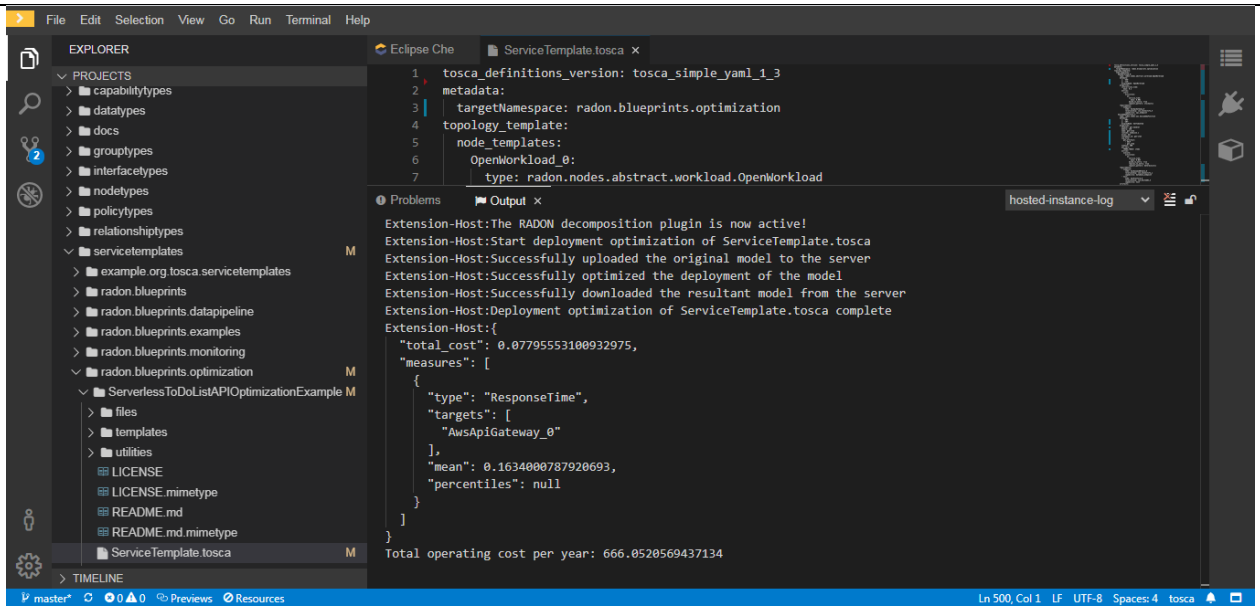



Figure 10.2.11.3: End of deployment optimization

For the ServerlessToDoListAPI application, the decomposition tool will try to find the optimal configuration for the memories of the Lambda functions and the write and read capacities of the DynamoDB table. These properties specify resources available for the application, thus being optimized to make a trade-off between the total operating cost and the performance measures. Table 10.2.11.2 reports the optimization results for the ServerlessToDoListAPI application. The memories of the Lambda functions do not change, because these functions are too lightweight in the sense that the performance requirement is satisfied even when the memories are 128 MB. To validate this deployment scheme, we also provide another locustfile, *open_workload_test.py*, for load testing the ServerlessToDoListAPI application. You can use a command similar to the one shown in Listing 10.2.11.8 to run this locustfile, which will simulate the reference workload specified by the *ClosedWorkload_0* node. In particular, the number of users to simulate should again be set to 1, as we show before in Figure 10.2.11.1. Table 10.2.11.3 reports the load testing result for the ServerlessToDoListAPI application. It can be seen that the mean response times of all the API endpoints are less than 0.2 s. Thus, the optimal deployment scheme satisfies the specified performance requirement.

Table 10.2.11.2: Optimization results for the ServerlessToDoListAPI application

| Node | Property | Before | After |
|---|---|---|---|
| AwsLambdaFunction_0 | memory | 128 MB | 128 MB |
| AwsLambdaFunction_1 | memory | 128 MB | 128 MB |
| AwsLambdaFunction_2 | memory | 128 MB | 128 MB |
| AwsLambdaFunction_3 | memory | 128 MB | 128 MB |
| AwsLambdaFunction_4 | memory | 128 MB | 128 MB |

| | | | |
|---|---|---|---|
| AwsDynamoDBTable_0 | read_capacity | 1 | 1 |
| | write_capacity | 1 | 3 |

Table 10.2.11.3: Load testing results for the ServerlessToDoListAPI application

| Endpoint | Mean Response Time | Min Response Time | Max Response Time |
|---|---|---|---|
| list | 0.144 s | 0.082 s | 0.447 s |
| get | 0.134 s | 0.076 s | 0.503 s |
| create | 0.129 s | 0.072 s | 0.390 s |
| update | 0.137 s | 0.080 s | 0.521 s |
| delete | 0.129 s | 0.081 s | 0.454 s |

### 10.2.12. Monitoring tool

The Monitoring Tool offers the capability to monitor serverless functions and obtain monitoring metrics through the automatic creation of personalized, user-proprietary Monitoring dashboards. This is feasible by the deployment of a Push Gateway node and the definition of an AWSISMonitoredBy relationship, provided by the Monitoring tool.

**Overview**

Initially, the function code has to be injected with a code snippet through which the metrics are pushed towards the Prometheus Push Gateway instance. Since the serverless function is hosted on a nodeJS runtime environment, the code snippet is triggering a parallel worker thread to push metrics parallel to the execution of the cloud function. Monitored metrics are collected by PushGateway.

Finally through the defintion of the AWSISMonitoredBy relationship, user proprietary Grafana dashboards are created and the monitoring metrics collected by the Prometheus PushgateWay node are visualized towards the user.

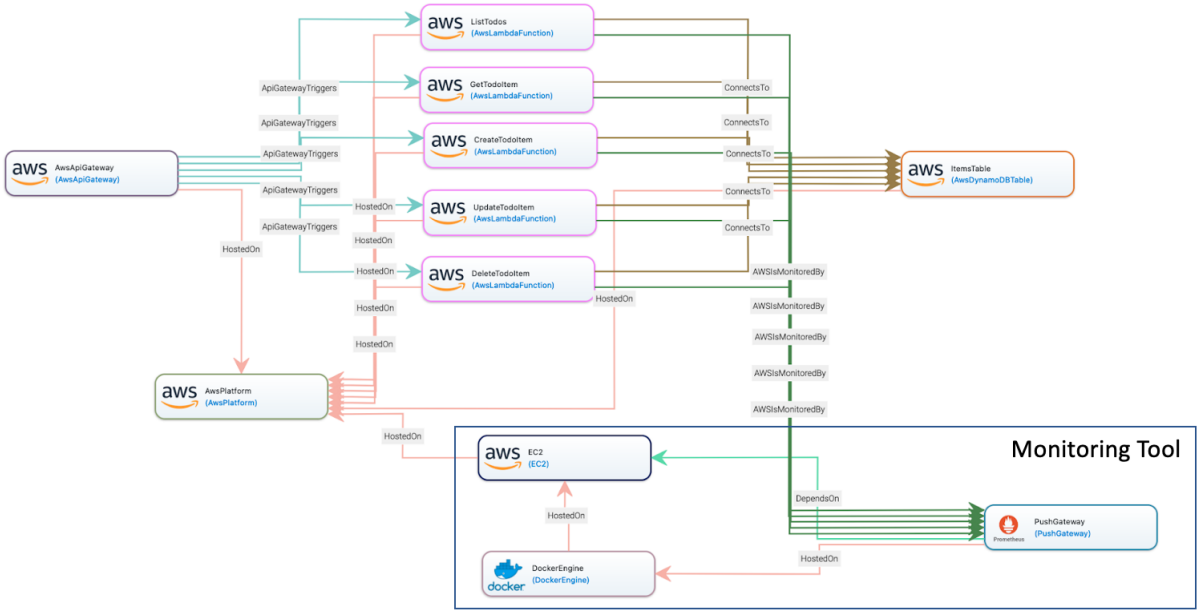



Figure 10.2.12.1: Generic Service blueprint of Monitoring Tool on ServerelessToDoAPI.

## Detailed Steps

### 1. Setup the Nodes

In order to integrate the monitoring tool to the ServerelessToDoAPI, the user must initially create an EC2 instance. This **EC2** instance will host a **Docker Engine** node which subsequently will host the **Pushgateway** Node. In addition, the Pushgateway node must depend on the EC2 to get an ip dynamically. The above mentioned 3 nodes are needed to deploy the monitoring tools. In detail:

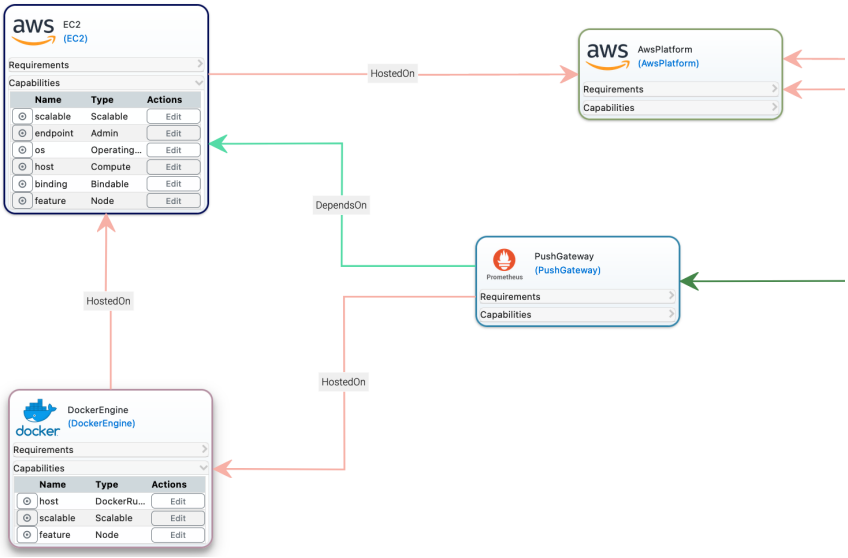
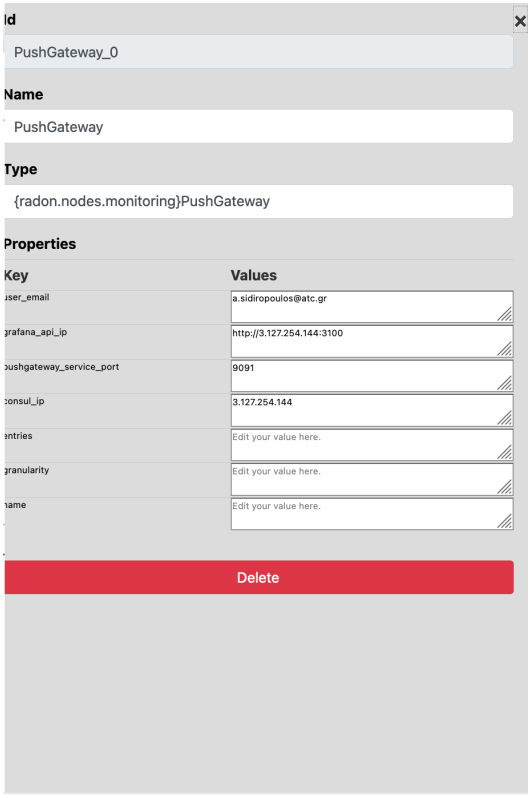



Figure 10.2.12.1: Monitoring nodes and their relationships

Additionally, the Push Gateway node must be configured with the following information:




**User_email** : email used for Keycloak

**Grafana_api_ip**: ip to the Grafana API. This ip can be used fixed as shown on the left.

**Pushgateway_service_port**: 9091

**Consul_ip:** console service ip. This ip can be used fixed.

Figure 10.2.12.3: Configuration of Push Gateway

## 2. Setup the monitoring relationships

Apart from the above mentioned relationships, for every function to be monitored the user has to configure the AWSIsMonitoredBy/GCPIsMonitoredBy relationships between the function and the Pushgateway Node (newly added capability **monitor** in pushgateway). Hence the Monitoring tool is configured to receive metrics from the function.

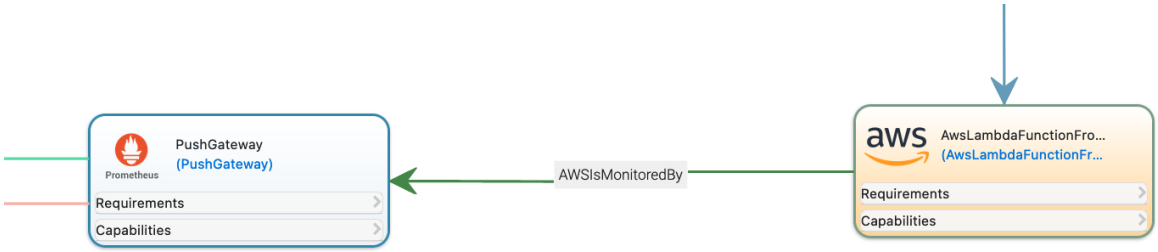



Figure 10.2.12.4: IsMonitored relationship

This relationship through its configuration scripts takes care of all the steps. No additional configuration is required.

### 3. Inject the function code

As stated in the beginning, for the monitoring to be feasible the function code has to be code injected or wrapped depending on the function runtime.

Since in ServerelessToDoAPI the functions are implemented in Node runtime (single thread operations), the function codes have to be wrapped with service workers to expose these metrics towards Pushgateway. The acquired metris are RAM and CPU.

```
"use strict";

const worker = require("./service/worker.js");
const timer = (ms) => new Promise((res) => setTimeout(res, ms));

module.exports.handler = async (event) => {
  //
  const _worker = worker.run(); // start worker

  // ************************************* //
  //
  // This is the original function code block //
  //
  for (let i = 0; i < 5; i++) {
    console.log("This is the iteration:", i);
    await timer(3000);
  }

  const response = {
    statusCode: 200,
    body: JSON.stringify("Hello from Lambda!"),
  };
  //
  // ************************************* //

  _worker.terminate(); // stop worker
  return response;
};
```

Figure 10.2.12.5: Code injection to expose function metrics

The code that wraps a Node JS application can be found here:

https://github.com/radon-h2020/radon-monitoring-tool/blob/master/lambda/nodejs-runtime/toDoGet-nodeJS-monitored-function.zip

In the link above, the toDoGet cloud function is wrapped to expose the RAM and CPU metrics

### 4. Deploy the tosca model

OPERA_SSH_USER=ubuntu OPERA_SSH_IDENTITY_FILE=/tmp/{{**your key**}}.pem opera deploy --clean-state {{**path to tosca file**}}.tosca

### 5. Access the Dashboards

Log in http://3.127.254.144:3000 using your keycloak credentials. Once the user is logged in the relevant dashboards are available. The dashboards are user proprietary and can be accessed only on the provided email account.

### 6. Invoke the functions

After function invocation the Grafana dashboards are updated in real time. The user can see the function performance metrics.

### 7. Trigger Alerting and Scaling Capabilities

By invoking the Node Policy definition mechanism along with an xOpera SaaS-adapted way of deployment, the Monitoring Tool can be extended to generate Alarms and trigger scaling events towards xOpera SaaS. Eventually, every serverless function can be assigned with one or more scaling policies. In these policies, the user can define thresholds and adjustments to resources (RAM, CPU) whenever these thresholds are violated.

In GMT, the following policy is attached to a Lambda function and is used to generate an alert every time the RAM monitored metrics exceed 80 % of the total capacity.

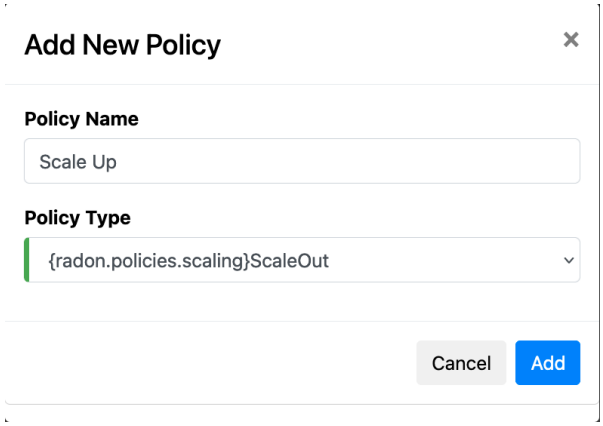



Figure 10.2.12.6: New policy definition

In the service template, the policy is attached to the Lambda function.

```
policies:
  - scale_up:
      type: radon.policies.scaling.ScaleUp
      properties:
        cpu_upper_bound: 60.0
        ram_upper_bound: 80
        adjustment: 2
        callbackUrlCPU: { get_input: callbackUrlCPU }
      targets: [ AwsLambdaFunctionFromS3_0 ]
```

Figure 10.2.12.7: Policy attachment to Lambda function

A callback url has to be provided per scaling event. This url is generated by the user in the xOpera SaaS prior to the deployment.

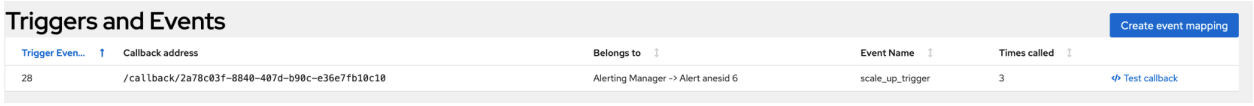



Figure 10.2.12.8: Callback URL provision

The callback url is provided through the inputs.yml file.

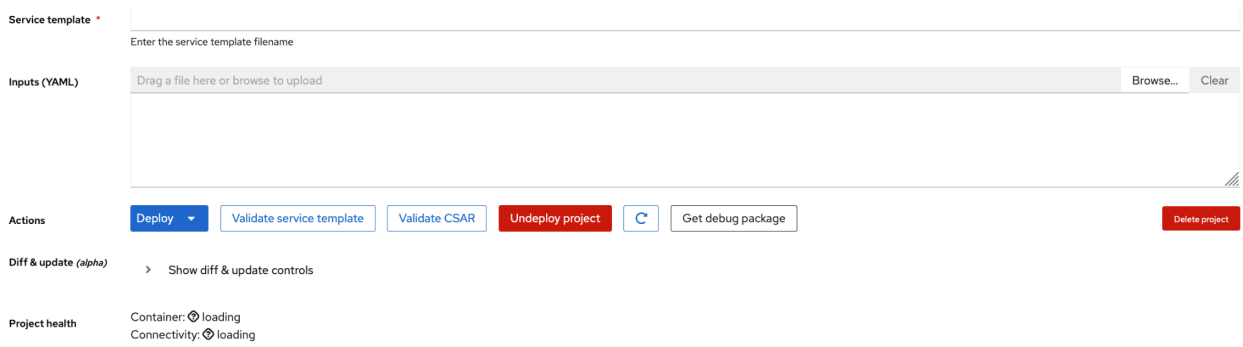



Figure 10.2.12.9: Callback url provision through inputs

The alert will be forwarded to xOpera SaaS which subsequently will trigger the execution of a scale action .yml file.

```yaml
tasks:
  - name: Start scaling up of Lambda function {{ monitored_function_name }}
    debug:
      msg: "Scaling up initiated."

  - name: See what's in the notification file from monitoring that was supplied to opera notify "{{ notification }}"
    debug:
      msg: "{{ notification }}"

  - name: Fecth the Labda function current memory size in order to increase it.
    shell: aws lambda get-function-configuration --region eu-central-1 --function-name {{ monitored_function_name }}
    register: current_function

  - name: Get Current AWS Memory Size
    set_fact:
      memory: "{{ (current_function.stdout_lines | join('\n') | from_json).MemorySize }}"

  - name: Adjust AWS Memory size
    set_fact:
      memory: "{{ 2*memory|int }}"

  - name: Scale Up Lambda function by incesing the Memory Size.
    shell: aws lambda update-function-configuration --function-name {{ monitored_function_name }} --region eu-central-1 --memory-size {{ memory }}
```

Figure 10.2.12.10: Lambda function RAM scaling yml file

Hence through the execution of this playbook the RAM of the monitored Lambda function will be increased according to the adjustment (duplicate the value).

After the successful deployment, during the execution phase, whenever there is a threshold violation an alert will be generated by the Grafana API, forwarded to xOpera SaaS, trigger the scaling event and scale up the resources.