OASIS Committee Note

Emerging Compute Models: Recommendations and Sample Profile Version 1.0

Committee Note 01

28 April 2022

This stage:
https://docs.oasis-open.org/tosca/ECAH-rec/v1.0/cn01/ECAH-rec-v1.0-cn01.docx (Authoritative)
https://docs.oasis-open.org/tosca/ECAH-rec/v1.0/cn01/ECAH-rec-v1.0-cn01.html
https://docs.oasis-open.org/tosca/ECAH-rec/v1.0/cn01/ECAH-rec-v1.0-cn01.pdf

Previous stage:
N/A

Latest stage:
https://docs.oasis-open.org/tosca/ECAH-rec/v1.0/ECAH-rec-v1.0.docx (Authoritative)
https://docs.oasis-open.org/tosca/ECAH-rec/v1.0/ECAH-rec-v1.0.html
https://docs.oasis-open.org/tosca/ECAH-rec/v1.0/ECAH-rec-v1.0.pdf

Technical Committee:
OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC

Chair:
Chris Lauwers (lauwers@ubicity.com), Individual Member

Editors:
Dario Di Nucci (d.di.nucci@jads.nl), Jheronimus Academy of Data Science (JADS)
Damian A. Tamburri (d.a.tamburri@tue.nl), Jheronimus Academy of Data Science (JADS)

Related work:
This document is related to:

- *TOSCA Version 2.0*. Edited by Chris Lauwers and Calin Curescu. Latest stage: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html.

Abstract:

Emergent compute models reflect the adoption of one or more hybrid forms of cloud computing, thereby embedding at least two combinations of compute models such as (a) function-as-a-service (FaaS) computing; (b) internet-of-things/edge computing; (c) high-performance computing (HPC). In the scope of the Emergent Compute ad-Hoc (ECAH), we explored the aforementioned compute models with the key objectives of exploring and/or prototyping TOSCA-based constructs and patterns for (1) serverless and FaaS-computing orchestration; (2) edge-computing and edge-based data pipelines orchestration; (3) high-performance computing orchestration. In the scope of the aforementioned activities, this deliverable offers details over:

- A series of use-cases for TOSCA usage in the scope of any of the aforementioned emergent compute models;
- Best practices and tools for the use of TOSCA;

- A working profile example to the TOSCA Language which supports Emergent Compute Paradigms (e.g., FaaS);
- A set of recommended extensions w.r.t. features enrichment of the current TOSCA notation, e.g., policies or trusted data-sharing and orchestration in such an ambient.

Status:

Citation format:

When referencing this document, the following citation format should be used:

[ECAH-rec-v1.0]

Emerging Compute Models: Recommendations and Sample Profile Version 1.0. Edited by Dario Di Nucci and Damian A. Tamburri. 28 April 2022. OASIS Committee Note 01. https://docs.oasis-open.org/tosca/ECAH-rec/v1.0/cn01/ECAH-rec-v1.0-cn01.html. Latest stage: https://docs.oasis-open.org/tosca/ECAH-rec/v1.0/ECAH-rec-v1.0.html.

Notices:

Table of Contents

1 Introduction

The current technical space around cloud compute solutions is rich with architecture patterns [Bass2003] that deviate from the *de-facto* standard way in which cloud computing applications were being deployed until 24 months ago, specifically, said technical space is now rich with *deviant* compute forms including: (a) Serverless computing, a paradigm in which the cloud provider allocates machine resources on demand, taking care of the servers on behalf of their customers and where Serverless functions do not hold resources in volatile memory, a computing profile wherefore compute power is delivered rather in short bursts with the results persisted to storage [Baldini2017]; (b) High-Performance Computing (HPC), namely, the classical approach of computing by means of a supercomputer, a computer with a high-level of performance as compared to a general-purpose computers [Kumar2008]; (c) internet-of-things computing---often referred to or intermixed with *edge* or *internet* computing---which is known as a distributed computing paradigm that brings computation and data storage closer to the location where it is needed to improve response times and save bandwidth [Park2018]. In the scope of this deliverable---and within the research design and operations space addressed by the emergent compute ad-hoc group by which this document is prepared---We define the term *emergent compute models* as any combination of the aforementioned enumeration. The rest of the deliverable offers details over how the Emerging Compute Ad-Hoc proceedings have addressed emergent compute models, principles, practices, tools, and technical space.

1.1 Deliverable Objectives

This deliverable offers details of the aforementioned proceedings in the scope of the Emergent Compute ad-Hoc (ECAH), in which we explored---for the better part of the past 30 months---the aforementioned compute models with the key objectives of exploring and/or prototyping TOSCA-based constructs and patterns for (1) serverless and FaaS-computing orchestration; (2) edge-computing and edge-based data pipelines orchestration; (3) high-performance computing orchestration. In the scope of the aforementioned activities, this deliverable offers details over:

- A series of use-cases for TOSCA usage in the scope of any of the aforementioned emergent compute models;
- Best practices and tools for the use of TOSCA;
- A working profile example to the TOSCA Language which supports Emergent Compute Paradigms (e.g., FaaS);
- A set of recommended extensions w.r.t., feature enrichment of the current TOSCA notation, e.g., in terms of policies or trusted data-sharing and orchestration in such an ambient.

1.2 Structure of the Document

The rest of this document is structured as follows. Section 2 offers details over the challenges addressed by the ECAH, with a focus on the newest technical space within the domain of emerging compute, namely, serverless computing. Section 3 offers details over a tentative profile in support of emergent computing. Section 4 offers details about the principal use-cases around emerging compute models. Finally, Sections 5 and 6 offer a case study and conclude the deliverable.

2 Challenges in the State-of-practice of Emergent Compute Models: a focused look at Serverless Adoption

In the context of cloud computing, the serverless computing paradigm focuses on composing applications using provider-managed component types. The Function-as-a-Service (FaaS) cloud service model is frequently associated with the term serverless as it enables hosting arbitrary, user-provided code snippets that are fully managed by the target platform including scaling the instances to zero after the execution is completed. Furthermore, FaaS-hosted functions can easily be integrated with other provider-managed services in an event-driven manner: application owners can bind functions to events that originate from different services such as object storage or message queue offerings. Although provider-managed FaaS platforms such as AWS Lambda or Azure Functions serve as one of the main "business logic carriers" in the serverless world, multiple different types of provider-managed service types can constitute serverless applications, e.g., API gateways, Database-as-a-Service offerings, message queues, or various logging and monitoring solutions. Jonas et al. [Jon19] highlight the following core differences between serverless and serverful computing:

1. Computation and storage are decoupled: code and data are typically hosted using different provider offerings, e.g., FaaS for source code and object storage for storing results. This implies separate provisioning and pricing [Jon19];
2. Execution of the code does not require application owners to allocate resources [Jon19];
3. Pricing models are finer-grained, focusing on the amount of resources actually used instead of the allocated amount of resources [Jon19].

The paradigm shift caused by serverless computing introduces new challenges related to the development and operations (DevOps) domain. For example, deployment modeling aspects for such finer-grained applications are tightly coupled with the chosen provider's infrastructure, making it non-trivial to choose the "right tool for the job". In the following, we briefly discuss the core challenges for deploying and operating serverless applications as well as highlight how these challenges affect the current state in TOSCA-based application modeling.

2.1 Serverless Applications and Resulting DevOps challenges

Essentially, most challenges for developing and operating serverless applications emerge from the fact that component architectures become significantly finer-grained, with components being responsible for specific tasks, e.g., processing components hosted as FaaS functions, queueing topics hosted using provider-managed messaging offerings, etc. Furthermore, each component in a serverless application is inherently coupled with the chosen provider, which imposes additional requirements on how the component has to be implemented, e.g., using provider-specific libraries, implementing required interfaces, and operated, e.g., deployment automation, scaling configuration and component lifecycle management. The resulting combination of tasks is then, typically, provider-specific and cannot easily be adapted for other deployment targets. In the following, we formulate some notable challenges which application owners need to consider in the context of serverless applications' development and operations:

Increased complexity of component architectures. With decoupling of computation and storage components, use cases shift from simpler, virtualization management scenarios for smaller amounts of components to more loosely coupled and complex component compositions communicating in both, traditional, and event-driven fashion. This also results in additional requirements on the lifecycle management for application components. Furthermore, increased lock-in with provider offerings results in more specific DevOps requirements.

Increased complexity of operations. As a consequence of the increased complexity of component architectures, the complexity of underlying operations increases significantly. The amount of required work increases significantly due to the increases in numbers of components, the rapidity and scale of services. Moreover, the amount of manual work increases for achieving good automation: application owners need to complete a bigger amount of provider- and application-specific tasks, which becomes even more complex when considering tasks related to maintenance and fault handling during operations.

Increased complexity of artifacts' reuse. The paradigm shift caused by serverless computing also complicates reusing existing DevOps artifacts due to their finer-grained and custom-tailored nature. For example, small code snippets with their deployment scripts and required event bindings are not easily reusable across providers. The large number of resulting artifacts obtained due to the increased complexity of component architectures makes it also non-trivial to identify the reusable artifacts in differently structured application repositories. As a result, some of the efforts invested in DevOps artifacts can be potentially discarded if the application baseline is modified significantly.

Increased complexity of deployment modeling. Another consequence of having a higher number of components and relations among them is the increase of complexity in deployment models. Firstly, choosing the right deployment modeling approach, e.g., declarative or imperative models, becomes less trivial since application owners need to align it with the chosen provider, types of to-be deployed components, etc. Moreover, the declarative deployment modeling differs from technology to technology (also, deployment vs configuration management tools), which influences people's understanding of it based on existing experiences and biases.

Heterogeneity of deployment modeling approaches. One of the points that complicates the deployment modeling is the increased amount of deployment automation approaches for working with serverless applications. Firstly, application owners might refer to existing deployment technologies that typically offer plugins for working with serverless-specific component types and features such as establishing event bindings between functions and provider-managed services. However, as previously, application owners need to decide how to combine provider-specific and provider-agnostic deployment technologies for automating the deployment of a given application. For example, AWS-specific deployment can be achieved using AWS SAM (which uses AWS CloudFormation under the hood), while specific parts of the application can be configured using configuration management tools such as Chef (also provided by AWS as a service). On the other hand, for certain applications, it could be enough to use one automation technology such as Serverless Framework which enables deploying serverless applications to multiple cloud providers. In all cases, such heterogeneity results in an increased amount of custom solutions for each chosen technology, and, thus, more repetitive work with results that might be difficult to reuse.

Lack of standardization in deployment modeling approaches. A corollary to the previous challenge is the obvious lack of standardization in existing approaches: cloud providers typically consider only their requirements, whereas provider-agnostic solutions typically do not intend to provide a canonical modeling experience, focusing on point-to-point transformations instead. For example, the Serverless Framework enables deployment automation for different providers but following their specific requirements. This results in, essentially, in a need to produce provider-specific models using the Serverless Framework. On the other hand, deployment models in serverless applications provide a good opportunity for also representing data not directly related to deployment. More specifically, as deployment models for serverless applications essentially represent a given component architecture, application owners might use such models for representing various non-functional requirements, e.g., verify certain constraints on component placement w.r.t. geographic regions. Using technology-specific formats for such use cases further increases the amount of manual work and learning curve since for each specific technology, application owners need to understand whether this is possible and how to achieve this.

2.2 TOSCA for Modeling Serverless Applications

While TOSCA provides a standard-based deployment modeling experience, the aforementioned challenges also affect the way it can be used for modeling serverless applications. In particular, the adoption of TOSCA in serverless contexts is hindered by several connected issues.

Lack of usage guidelines. Application developers and operations engineers are more accustomed to relying on tooling-specific APIs and configuration management code. The same "reflexes" are then tried to be used when modeling with TOSCA, which has a different learning curve and requires additional learning efforts. The specification itself does not provide practical examples further hindering the usability in such cases.

"Lack of clear semantics". While TOSCA provides a powerful basis for representing any kind of relations for modeling the deployment, the lack of standardized approaches for representing additional semantics required in serverless applications makes it more difficult to adapt TOSCA for such cases. For example, the event-driven communication between functions or containers and other provider-managed services can be expressed in several ways, e.g., event bindings can be expressed using dedicated Node Types / Node Templates, or Relationship Types / Relationship Templates. The actual deployment of event bindings requires different implementation styles.

"Lack of deployable building blocks". Existing deployment automation tools offer out-of-the-box support for many cloud providers with multiple serverless-related plugins available, e.g., FaaS platforms, serverless containers, and databases, whereas in TOSCA, the component types library depends on the specific implementation of the chosen orchestrator and the available type repositories. In most cases, the amount of deployable components is limited which makes it difficult to start using TOSCA instead of existing deployment automation technologies.

3 A Sample Profile for Emergent Compute Models: Extending TOSCA for Function-as-a-Service

Following the normative types by the TOSCA 1.3 standard, the RADON Modeling Profile (RADON TOSCA) targets to extend those to fulfill the requirements and challenges of Serverless or Function-as-a-Service (FaaS) based applications. As a result, RADON TOSCA introduces a type hierarchy of new types, both abstract and concrete. Abstract types inherit directly from the normative types defined in TOSCA 1.3 and serve as a unification layer for the concrete types that represent specific technologies or cloud platforms and are deployable. Most of the leaf nodes in the introduced hierarchy represent concrete, deployable types.

The main segment of RADON TOSCA deals with the representation of serverless, FaaS-hosted functions in TOSCA service templates. To support the modeling of FaaS-hosted functions, several crucial aspects have to be addressed. In the following, we elaborate on these modeling aspects.

3.1 Cloud Providers and FaaS Platforms

Cloud providers, offering FaaS or services with a serverless notion, such as AWS or Microsoft Azure, often require a special configuration for provisioning. For example, authentication credentials or region settings must be provided. Further, each cloud provider provides SDKs or CLIs that must be utilized to automate the actual deployment of components. This motivated the decision to introduce a generic and abstract *CloudPlatform* node type to encapsulate such cloud platform-specific properties. Provider-specific types inherit from this type. Consequently, a concrete type is introduced by RADON TOSCA for each cloud provider. All further technology-specific node types representing services or offerings of these providers are connected using a HostedOn relationship. This way, configuration properties can be shared among all contained (aka. “hosted on”) components. Further, these types may employ implementations that install and configure respective CLIs or SDK utilized by the other provider-specific node types. For example, the “create” operation of the “AwsPlatform” type installs the “awscli” as well as the “boto” library in the orchestrator such that respective components on top can utilize them. Therefore, the CloudPlatform type exposes a “host” capability of type “Container”.

Similar to cloud providers, RADON TOSCA uses the *CloudPlatform* node type to also define the semantics for self-hosted FaaS platforms, such as OpenFaaS or OpenWhisk.

Name	URI	Version	Derived From
`CloudPlatform`	`radon.nodes.abstract.CloudPlatform`	1.0.0	`tosca.nodes.Root`

The *CloudPlatform*[1] node type is defined as follows:

```
radon.nodes.abstract.CloudPlatform:
  derived_from: tosca.nodes.Root
  metadata:
    targetNamespace: "radon.nodes.abstract"
  properties:
    name:
      type: string
      required: false
  capabilities:
    host:
      type: tosca.capabilities.Container
      occurrences: [ 1, UNBOUNDED ]
      valid_source_types:
        - radon.nodes.abstract.Function
        - radon.nodes.abstract.ObjectStorage
        - radon.nodes.abstract.ApiGateway
```

[1] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.abstract/CloudPlatform

```
        - radon.nodes.abstract.Database
```

Properties

Name	Required	Type	Constraint	Default Value	Description
`name`	`false`	`string`			Name of the cloud platform

Capabilities

Name	Type	Valid Source Types	Occurrences
`host`	`tosca.capabilities.Container`	`[radon.nodes.abstract.Function, radon.nodes.abstract.ObjectStorage, radon.nodes.abstract.ApiGateway, radon.nodes.abstract.Database]`	[1, UNBOUNDED]

Concrete Type Examples

- AzurePlatform[2]
- AwsPlatform[3]
- GoogleCloudPlatform[4]
- OpenFaasPlatform[5]

3.2 Functions and their triggering semantics

One of the main challenges is how to address different function triggering semantics. More specifically, functions need to be modeled differently based on what triggers them. For example, most of the functions deployed to commercial offerings such as AWS Lambda are event-driven and can be triggered by a plethora of events emitted by provider-specific services, e.g., AWS S3 object storage or AWS SNS message queue. Contrarily, there are functions that can be referred to as standalone as they do not require explicit modeling of event sources. For example, a scheduled function is typically triggered by an internally defined cron job, which does not need to be explicitly represented in the deployment model. As a result, the modeling semantics change depending on the kind of the function.

3.2.1 Invocable and Standalone Functions

RADON TOSCA distinguishes between *invocable* and *standalone* functions with respect to the corresponding FaaS platforms. First and foremost, RADON TOSCA introduces an abstract *Function* node type. It defines a “name” property and serverless-specific requirements and capabilities. Standalone as well as invocable function inherit from this type. RADON TOSCA does not introduce separate abstract types since the triggering semantics of invocable functions can be modeled very genetically using “[0, UNBOUNDED]” occurrence constraints and, more significant, the scheduling semantics of standalone functions are too provider-specific to come up with a generic solution on the abstract level. For example, Azure Functions require cron settings while AWS Lambda supports a custom format (rate) and a cron setting. Therefore, RADON TOSCA introduces standalone types only on the concrete level for cloud providers supporting it.

Invocable functions, on the contrary, require an event source to trigger them. RADON TOSCA, herefore, introduces an *Invocable*[6] capability and a relationship type name *Triggers*[7]. The Invocable capability is exposed by each Function and indicates that it is invocable by an event from another cloud resource, such as object storage or API gateways. In terms of requirements, the Function type defines a “host” requirement, to host this function on a suitable platform, and a “endpoint” requirement, such that other components are able to connect to a function (using ConnectsTo relationships). Further, since functions

[2] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.azure/AzurePlatform
[3] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.aws/AwsPlatform
[4] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.google/GoogleCloudPlatform
[5] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.openfaas/OpenFaasPlatform
[6] https://github.com/radon-h2020/radon-particles/tree/master/capabilitytypes/radon.capabilities/Invocable
[7] https://github.com/radon-h2020/radon-particles/tree/master/relationshiptypes/radon.relationships/Triggers

are also able to emit events, it defines an “invoker” requirement, which can be satisfied by another Invocable capability of another function node.

Name	URI	Version	Derived From
`Function`	`radon.nodes.abstract.Function`	1.0.0	`tosca.nodes.Root`

The *Function*[8] node type is defined as follows:

```
radon.nodes.abstract.Function:
  derived_from: tosca.nodes.Root
  metadata:
    targetNamespace: "radon.nodes.abstract"
  properties:
    name:
      type: string
  requirements:
    - host:
        capability: tosca.capabilities.Container
        node: radon.nodes.abstract.CloudPlatform
        relationship: tosca.relationships.HostedOn
        occurrences: [ 1, 1 ]
    - invoker:
        capability: radon.capabilities.Invocable
        relationship: radon.relationships.Triggers
        occurrences: [ 0, UNBOUNDED ]
    - endpoint:
        capability: tosca.capabilities.Endpoint
        relationship: radon.relationships.ConnectsTo
        occurrences: [ 0, UNBOUNDED ]
  capabilities:
    invocable:
      occurrences: [ 0, UNBOUNDED ]
      type: radon.capabilities.Invocable
```

Properties

Name	Required	Type	Constraint	Default Value	Description
`name`	`true`	`string`			Name of the function

Requirements

Name	Capability Type	Node Type Constraint	Relationship Type	Occurrences
`host`	`tosca.capabilities.Container`	`radon.nodes.abstract.CloudPlatform`	`tosca.relationships.HostedOn`	[1, 1]
`invoker`	`radon.capabilities.Invocable`		`radon.relationships.Triggers`	[0, UNBOUNDED
`endpoint`	`tosca.capabilities.Endpoint`		`radon.relationships.ConnectsTo`	[0, UNBOUNDED

Capabilities

Name	Type	Valid Source Types	Occurrences
`invocable`	`radon.capabilities.Invocable`		[0, UNBOUNDED]

Concrete Type Examples

[8] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.abstract/Function

- AwsLambdaFunction[9]
- AwsTriggers[10]
- AwsApiGatewayTriggers[11]
- AzureTimerTriggeredFunction[12]
- AzureResourceTriggeredFunction[13]
- AzureContainerTriggers[14]

3.2.2 Trigger Semantics

To establish the actual trigger semantic of an invocable function, RADON TOSCA introduces the Triggers relationship type. A similar approach was proposed by Wurster et al. [Wurster2018] as they recommend using TOSCA's relationships to model the notion of events if a cloud resource triggers a cloud FaaS function. In RADON TOSCA this relationship type describes which event type triggers the function and may provide additional binding logic in the form of TOSCA implementation artifacts.

Name	URI	Version	Derived From
`Triggers`	`radon.relationships.Triggers`	1.0.0	`tosca.relationships.ConnectsTo`

The *Triggers*[15] relationship type is defined as follows:

```
radon.relationships.Triggers:
  derived_from: tosca.relationships.ConnectsTo
  metadata:
    targetNamespace: "radon.relationships"
  properties:
    events:
      type: list
      required: false
      entry_schema:
        type: radon.datatypes.Event
  valid_target_types: [ radon.capabilities.Invocable ]
```

Properties

Name	Required	Type	Constraint	Default Value	Description
`events`	`false`	`list: radon.datatypes.Event`			List of events

Concrete Type Examples

- AwsTriggers[16]
- AwsApiGatewayTriggers[17]
- AzureContainerTriggers[18]

[9] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.aws/AwsLambdaFunction
[10] https://github.com/radon-h2020/radon-particles/tree/master/relationshiptypes/radon.relationships.aws/AwsTriggers
[11] https://github.com/radon-h2020/radon-particles/tree/master/relationshiptypes/radon.relationships.aws/AwsApiGatewayTriggers
[12] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.azure/AzureTimerTriggeredFunction
[13] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.azure/AzureResourceTriggeredFunction
[14] https://github.com/radon-h2020/radon-particles/tree/master/relationshiptypes/radon.relationships.azure/AzureContainerTriggers

[15] https://github.com/radon-h2020/radon-particles/tree/master/relationshiptypes/radon.relationships/Triggers
[16] https://github.com/radon-h2020/radon-particles/tree/master/relationshiptypes/radon.relationships.aws/AwsTriggers
[17] https://github.com/radon-h2020/radon-particles/tree/master/relationshiptypes/radon.relationships.aws/AwsApiGatewayTriggers
[18] https://github.com/radon-h2020/radon-particles/tree/master/relationshiptypes/radon.relationships.azure/AzureContainerTriggers

3.2.3 Event Specification

The abstract Triggers relationship additionally defines an optional “events” property. This property is employed to define and specify the actual event that is used at runtime to trigger a function. RADON TOSCA introduces a special “Event” data type for this specification. The Event data type represents an event based on the CNCF CloudEvents schema.

Name	URI	Version	Derived From
Event	`radon.datatypes.Event`	1.0.0	`tosca.datatypes.Root`

The *Event*[19] data type is defined as follows:

```
radon.datatypes.Event:
  derived_from: tosca.datatypes.Root
  metadata:
    targetNamespace: "radon.datatypes"
  properties:
    spec_version:
      type: string
      required: false
      default: 0.3
    schema_url:
      type: string
      required: false
    data_content_encoding:
      type: string
      required: false
    type:
      type: string
    data_content_type:
      type: string
      required: false
```

Properties

Name	Required	Type	Constraint	Default Value	Description
`type`	`true`	`string`			Event type, e.g., `s3:ObjectCreated:Put`
`spec_version`	`false`	`string`		0.3	CloudEvents spec version
`data_content_encoding`	`false`	`string`			Event's content encoding
`data_content_type`	`false`	`string`			Type of event's data content, e.g., `text/xml`
`schema_url`	`false`	`string`			URL to the event's schema definition

3.3 Cloud Services as Event Sources

Ephemeral processing power in the form of FaaS offerings is mainly event driven. Therefore, cloud providers let users choose from a variety of cloud services that are enabled to emit certain events to trigger FaaS-based computing resources, aka. functions. For example, AWS enables users to trigger Lambda Functions whenever new files have been added to their object storage offering AWS S3 or when some arbitrary data has been pushed to a processing queue (AWS SQS) or topics (AWS SNS). Further, users may also invoke functions if new data entries are added to a DynamoDB table. This applies similarly to other cloud providers such as Google Cloud or Azure.

Due to a large number of possible event source categories and the fast development pace of public cloud offerings and open source FaaS platforms, RADON TOSCA tries to establish a first baseline providing suitable abstract types. RADON TOSCA introduces three abstract types for common cloud resources that

[19] https://github.com/radon-h2020/radon-particles/tree/master/datatypes/radon.datatypes/Event

emit events for FaaS: API Gateways, Object Storage, and Databases. In the following we present the abstract types as well as point to deployable example in RADON's type repository RADON Particles[20].

3.3.1 API Gateways

One common use case is to trigger functions based on the occurrence of HTTP events. To enable this, FaaS platforms typically rely on API Gateways. This applies to both commercial platforms such as AWS Lambda and Azure Functions as well as open-source solutions such as OpenFaaS. Hence, as a first step to enable modeling API gateways in the application topologies, RADON TOSCA provides an abstract type that represents a technology-agnostic API gateway which can be used as a parent for provider-specific API gateways, e.g., offered by AWS or Microsoft Azure.

Name	URI	Version	Derived From
`ApiGateway`	`radon.nodes.abstract.ApiGateway`	1.0.0	`tosca.nodes.Root`

The node type must be hosted on a suitable CloudPlatform type and can trigger Function node types based on the introduced Invocable capability. To model a relationship and to specify further details on the type of event the Triggers relationship type is used.

The *ApiGateway*[21] node type is defined as follows:

```
radon.nodes.abstract.ApiGateway:
  derived_from: tosca.nodes.Root
  metadata:
    targetNamespace: "radon.nodes.abstract"
  properties:
    name:
      type: string
  requirements:
    - host:
        capability: tosca.capabilities.Container
        node: radon.nodes.abstract.CloudPlatform
        relationship: tosca.relationships.HostedOn
        occurrences: [ 1, 1 ]
    - invoker:
        capability: radon.capabilities.Invocable
        node: radon.nodes.abstract.Function
        relationship: radon.relationships.Triggers
        occurrences: [ 0, UNBOUNDED ]
  capabilities:
    api_endpoint:
      occurrences: [ 0, UNBOUNDED ]
      type: tosca.capabilities.Endpoint
```

Properties

Name	Required	Type	Constraint	Default Value	Description
`name`	`true`	`string`			Name of the API

Requirements

Name	Capability Type	Node Type Constraint	Relationship Type	Occurrences
`host`	`tosca.capabilities.Container`	`radon.nodes.abstract.CloudPlatform`	`tosca.relationships.HostedOn`	[1, 1]
`invoker`	`radon.capabilities.Invocable`	`radon.nodes.abstract.Function`	`radon.relationships.Triggers`	[0, UNBOUNDED]

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

[21] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.abstract/ApiGateway

Capabilities

Name	Type	Valid Source Types	Occurrences
`api_endpoint`	`radon.capabilities.Endpoint`		[0, UNBOUNDED]

Concrete Type Examples

- AwsApiGateway[22]

To support deploying applications that rely on an API Gateway, a concrete node type must reflect the properties of the underlying provider-specific technology. For example, the AWS region and Amazon Resource Name (ARN) of a role to trigger functions must be defined on the type level and instantiated on the level of node templates. Further, concrete types may restrict the requirements of the abstract level. For example, a concrete API gateway node type for AWS will restrict the “host” requirement to nodes of type AwsPlatform.

Another important aspect of concrete types is the implementation of the node type itself and the concrete Triggers relationship type. For example, some cloud offerings might require a Swagger file to instrument the API Gateway. However, the specific event type, i.e., which HTTP operation shall invoke which function, is defined by a respective relationship template between an API Gateway node and a function node. We propose to implement the deployment logic using a template of a Swagger specification that is temporarily created by the “create” operation of the concrete API Gateway type. The Swagger specification can be substituted by the respective operation of the concrete Triggers relationship type using the properties specifying the actual endpoints and HTTP method. The TOSCA standard lifecycle interfaces for nodes and relationships help substantially in the implementation of such behavior.

3.3.2 Object Storage

There are a plethora of object storage offerings. Most well-established cloud providers offer a service that lets users store massive amounts of unstructured data. The main purpose is to store any kind of data in an unstructured but highly accessible way. Azure Blob Storage and AWS S3 are very well-known examples.

The TOSCA standard already defines an ObjectStorge node type to represent such a semantic. However, when users are dealing with functions, they have to specify what kind of events shall be used to trigger the envisioned function code. Therefore, RADON TOSCA introduces a new abstract ObjectStorage node type representing a managed object storage offering. It defines the respective “host” and “invoker” requirements to specify on what cloud platform it must be instrumented as well as to use the Triggers relationship type to specify the type of events.

Name	URI	Version	Derived From
`ObjectStorage`	`radon.nodes.abstract.ObjectStorage`	1.0.0	`tosca.nodes.Storage.ObjectStorage`

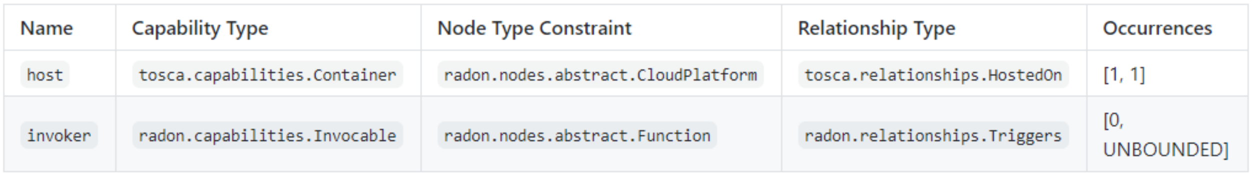
The *ObjectStorage*[23] node type is defined as follows:

```
radon.nodes.abstract.ObjectStorage:
  derived_from: tosca.nodes.Storage.ObjectStorage
  metadata:
    targetNamespace: "radon.nodes.abstract"
  requirements:
    - host:
        capability: tosca.capabilities.Container
        node: radon.nodes.abstract.CloudPlatform
        relationship: tosca.relationships.HostedOn
        occurrences: [ 1, 1 ]
    - invoker:
```

[22] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.aws/AwsApiGateway
[23] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.abstract/ObjectStorage

```
        capability: radon.capabilities.Invocable
        node: radon.nodes.abstract.Function
        relationship: radon.relationships.Triggers
        occurrences: [ 0, UNBOUNDED ]
```

Requirements



Name	Capability Type	Node Type Constraint	Relationship Type	Occurrences
`host`	`tosca.capabilities.Container`	`radon.nodes.abstract.CloudPlatform`	`tosca.relationships.HostedOn`	[1, 1]
`invoker`	`radon.capabilities.Invocable`	`radon.nodes.abstract.Function`	`radon.relationships.Triggers`	[0, UNBOUNDED]

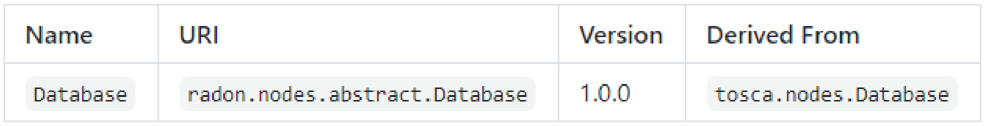
Concrete Type Examples

- AwsS3Bucket[24]
- AzureBlobStorageContainer[25]

Interestingly, the implementations for concrete node types may also be spread among the actual type and the concrete Triggers relationship type. For example, a concrete node type may provision and instrument the respective cloud service, i.e., create a new AWS S3 bucket using the AWS CLI. However, the actual event binding is established by the implementation of the respective Triggers relationship type. For this, the “post_configure_source” operation of the standard lifecycle interface may be used to coordinate the provisioning and actual event binding.

3.3.3 Database

Similar to object storage offerings, most cloud providers offer some kind of managed database service (aka. database as a service). In terms of abstract modeling, it does not matter if it is a relational or schema-free database technology. Hence, RADON TOSCA introduces a new abstract type that represents exactly such managed databases, which, in turn, are enabled to emit events for invoking function code. However, it is derived from TOSCA’s Database type to reuse the existing configuration. Further, it extends it with appropriate requirements to be usable within the RADON TOSCA modeling profile (cf.”host” and “invoker” requirements).



Name	URI	Version	Derived From
`Database`	`radon.nodes.abstract.Database`	1.0.0	`tosca.nodes.Database`

The Database[26] node type is defined as follows:

```
radon.nodes.abstract.Database:
  derived_from: tosca.nodes.Database
  metadata:
    targetNamespace: "radon.nodes.abstract"
  requirements:
    - host:
        capability: tosca.capabilities.Container
        relationship: tosca.relationships.HostedOn
        occurrences: [ 1, 1 ]
    - invoker:
        capability: radon.capabilities.Invocable
        node: radon.nodes.abstract.Function
        relationship: radon.relationships.Triggers
        occurrences: [ 0, UNBOUNDED ]
  capabilities:
    database_endpoint:
```

[24] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.aws/AwsS3Bucket
[25] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.azure/AzureBlobStorageContainer
[26] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.abstract/Database

```
      occurrences: [ 0, UNBOUNDED ]
      type: tosca.capabilities.Endpoint.Database
```

Requirements

Name	Capability Type	Node Type Constraint	Relationship Type	Occurrences
`host`	`tosca.capabilities.Container`		`tosca.relationships.HostedOn`	[1, 1]
`invoker`	`radon.capabilities.Invocable`	`radon.nodes.abstract.Function`	`radon.relationships.Triggers`	[0, UNBOUNDED]

Capabilities

Name	Type	Valid Source Types	Occurrences
`database_endpoint`	`tosca.capabilities.Endpoint.Database`		[0, UNBOUNDED]

Concrete Type Examples

- AwsDynamoDBTable[27]
- AzureCosmosDB[28]

Similar to object storage deployments, the provisioning of such a managed database service and the event binding can be split into the node type and the respective Triggers relationship type. By using TOSCA lifecycle interface (e.g., “create” for node types and “post_configure_source” the deployment can be well coordinated.

3.4 Namespace Best Practices

To identify and maintain TOSCA types a best practice to use namespaces is introduced. The general schema of RADON’s namespace is defined as follows:

radon.[entity-type].[purpose-identifier*].[entity]

It consists of four parts, namely:

- The first part of the namespace is the fixed keyword *radon* that separates all TOSCA types developed under the umbrella of RADON from TOSCA’s normative types.
- Next, i.e., *[entity-type]*, specifies a corresponding TOSCA entity type. The following table maps all TOSCA types to the respective keyword:

TOSCA Types	RADON Keywords
Relationship Types	relationships
Requirement Types	requirements
Capability Types	capabilities
Policy Types	policies
Artifact Types	artifacts
Data Types	datatypes

[27] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.aws/AwsDynamoDBTable
[28] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.azure/AzureCosmosDB

Group Types	groups
Interface Types	interfaces

- The *[purpose-identifier*]* part of the namespace serves as an identifier of the entity's purpose. More specifically, this part of the namespace: *(i) separates* technology-agnostic types from technology-specific types using the keyword *abstract, (ii)* describes particular technologies or providers, e.g., *aws* for Amazon Web Services or *kafka* for a popular streaming platform, and *(iii)* identifies the purpose of the TOSCA entity, e.g., *scaling* for grouping scaling policies.
- Finally, the *[entity]* part refers to an actual entity, e.g., *S3Bucket* to describe the bucket created in AWS S3 object storage service.

The table below shows several examples of how namespaces are defined for some RADON types.

Examples of the RADON namespace usage

Example	Description
radon.nodes.abstract.Function	Identifies an abstract node type that represents a FaaS function, which is hosted on an abstract FaaS platform.
radon.nodes.nifi.NifiPipeline	Identifies a technology-specific node type that represents a data pipeline defined for and hosted on Apache NiFi.
radon.policies.scaling.AutoScale	Identifies one of RADON's scaling policies, namely an autoscaling policy.

3.5 Collaborative Repository Layout

RADON TOSCA introduces a best practice to share node types and service templates among the community. It establishes a collaborative environment where users can contribute individually. Further, many of these repositories can be utilized as a source to build and compose TOSCA service templates out of reusable node types.

As TOSCA types and templates are stored in files, we propose a repository layout that can be used locally or collaboratively with version control systems. Independently of the used mechanism, the following information needs to be stored:

- TOSCA entity types, such as TOSCA node types, relationship types, or policy types as reusable modeling entities.
- Respective TOSCA implementations attached to TOSCA node and relationship types (Ansible Playbooks) to make those types executable[29].
- Additional files such as README or LICENSE files.
- Application blueprints or TOSCA service templates, which are composed of reusable TOSCA entity types.
- In the following, we propose a file-based layout to store such information, which is schematically shown below (an example is shown afterward).

```
<root>
  |-<entity type>
  |  |-<namespace>
  |  |  |-<identifier>
  |  |  |  |-files
```

[29] Types without TOSCA implementations are called "abstract types".

```
  |  |  |  |  |-<implementation>
  |  |  |  |  |  |-<name>.yml
  |  |  |  |-<entity type>.tosca
  |  |  |  |-README.md
  |  |  |  |-LICENSE
```

Generally, a repository is structured by the `<TOSCA entity type>`, a `<namespace>`, and an `<identifier>` for the respective TOSCA entity, plus additional versioning and metadata information. Under a root directory we use dedicated directories to separate the TOSCA entity types, e.g., separate directories to store all available TOSCA node types, policy types, or service templates. One level below, we employ the previously introduced namespace scheme to ensure that all maintained entities have unique names so that they can be easily identified, e.g., `radon.nodes.aws` to group all AWS-related entity types. The third level identities the entity itself. It is a name identifier that, together with the namespace, uniquely identifies the entity. This level then also contains the actual information about the entity in form of respective TOSCA syntax. For example, a `NodeType.tosca` file contains the actual TOSCA syntax to define the respective TOSCA node type. Besides that, additional metadata information can be stored, e.g., a README and LICENSE file. Similar to metadata, we store respective TOSCA implementations, e.g., Ansible Playbooks attached to TOSCA node types, in a dedicated "files" directory and store them in a structured way to identify implementations for "create" or "delete" operations.

```
root
├── ...
├── nodetypes
│   ├── ...
│   ├── radon.nodes.aws
│   │   ├── ...
│   │   ├── AwsLambdaFunction
│   │   │   ├── LICENSE
│   │   │   ├── NodeType.tosca
│   │   │   ├── README.md
│   │   │   └── files
│   │   │       ├── create
│   │   │       │   └── create.yml
│   │   │       └── delete
│   │   │           └── delete.yml
│   │   └── ...
│   └── ...
└── ...
```

The key advantages of this convention are that each populated TOSCA type can be (1) developed and maintained separately and all metafiles files or implementations are stored along with the type definition itself, (2) import statements in the type definitions can be employed to make use of other TOSCA types, and (3) multiple such repositories can be used a source to compose TOSCA service templates.

An instantiated example of such a file-based repository is the RADON Particles[30] repository. It is a public and open-source repository to publish and provide all executable TOSCA entity types (e.g., TOSCA node types and Ansible Playbooks) and RADON Models (application blueprints in the form of TOSCA service templates) that have been developed by the RADON consortium.

3.6 Modeling Profile in Action

In this section, we show how users may use RADON TOSCA to model application deployments graphically. In the following, we present a brief tutorial on how to use Eclipse Winery[31] to graphically model an application deployment based on the introduced types above.

However, experienced TOSCA developers may use any text editor or IDE to develop a TOSCA service template based on the provided RADON TOSCA entity types. Eclipse Winery, which an open-source[32]

30 https://github.com/radon-h2020/radon-particles

31 https://projects.eclipse.org/projects/soa.winery

32 https://github.com/eclipse/winery

modeling tool for TOSCA, provides a graphical abstraction and syntax-agnostic way to browse node types or compose blueprints.

3.6.1 Start Eclipse Winery using Docker

Start your Docker environment and execute the following command:

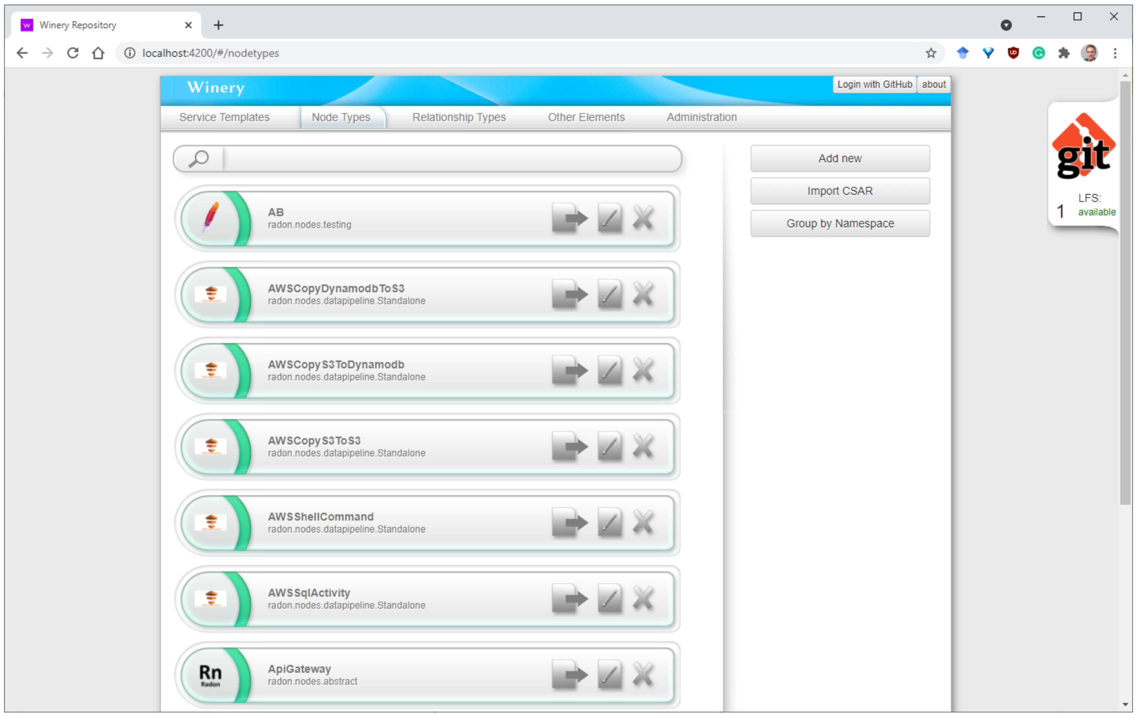
```
docker run -it -p 8080:8080 \
  -e PUBLIC_HOSTNAME=localhost \
  -e WINERY_FEATURE_RADON=true \
  -e WINERY_REPOSITORY_PROVIDER=yaml \
  -e WINERY_REPOSITORY_URL=https://github.com/radon-h2020/radon-particles \
  opentosca/winery
```

Afterwards, launch a browser window and navigate to: `http://localhost:8080`

3.6.2 Browse the reusable modeling types

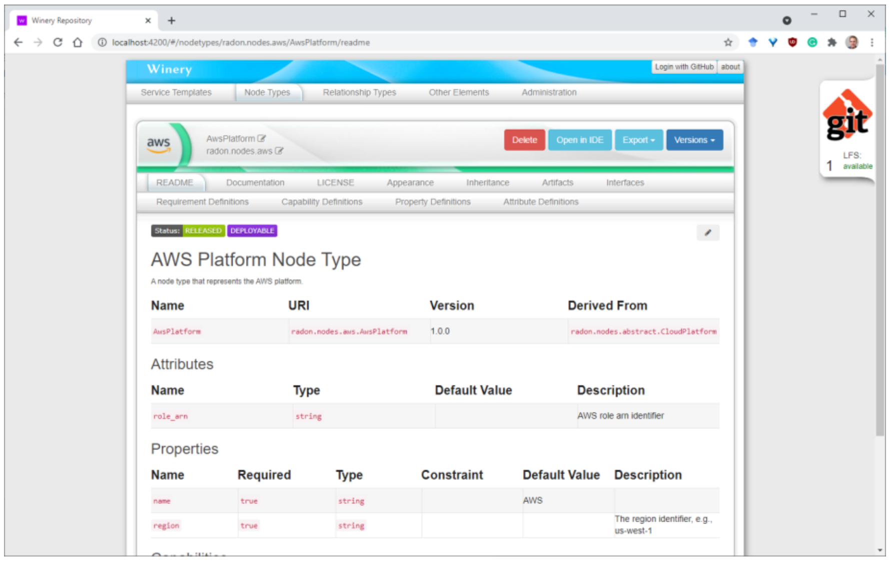
The Eclipse Winery web application is in general divided into two main user interfaces. Users start with the TOSCA Management UI to manage all the TOSCA entities inside the configured data store (a.k.a. repository). The second UI is the TOSCA Topology Modeler used to graphically compose a TOSCA service template based on the reusable types inside the current repository.

Eclipse Winery starts the TOSCA Management UI by default showing the “Service Templates” view. From here, users can use the top navigation menu to change between the main views within the TOSCA Management UI. By clicking on “Node Types”, the users find a list of reusable TOSCA node types contained in the configured repository.



Users may use the “Group by Namespace” option to navigate faster through the list. However, for more details, click on the node type you are interested in. For example, try to find the “AwsPlatform” node type and open the detail view. The overall TOSCA management UI behaves similarly for every TOSCA entity. The detailed view window provides comprehensive information relevant for the chosen element type, e.g., relationship types, capability types, or data types.

The detail view for TOSCA node types presents the current definition in a graphical manner. Users may use the respective submenus to modify the type respectively. For example, users are able to define or adapt certain property definitions. Further, additional capabilities or requirements can be defined for a type. The UI also supports users to upload and assign TOSCA artifacts (deployment and implementation artifacts) to a type. Lastly, users can get an idea of the type hierarchy in the “Inheritance” submenu and quickly navigate to the respective parent type. The detail view is shown in the next screenshot.



AWS Platform Node Type

A node type that represents the AWS platform.

Name	URI	Version	Derived From
AwsPlatform	radon.nodes.aws.AwsPlatform	1.0.0	radon.nodes.abstract.CloudPlatform

Attributes

Name	Type	Default Value	Description
role_arn	string		AWS role arn identifier

Properties

Name	Required	Type	Constraint	Default Value	Description
name	true	string		AWS	
region	true	string			The region identifier, e.g., us-west-1

Apart from just modifying the configuration of the node type, Eclipse Winery provides the possibility to export a chosen element as a TOSCA CSAR file or control its versions. Interestingly, Eclipse Winery always serializes the current configuration to its file-based repository. Whenever a user changes a type, e.g., adds properties or capabilities, Eclipse Winery will save these changes to its data store compliant to the TOSCA standard.

Users have access to the actual serialized TOSCA files by mounting a directory from their local workstation to the Docker container. The user guide shows in detail how to do this[33].

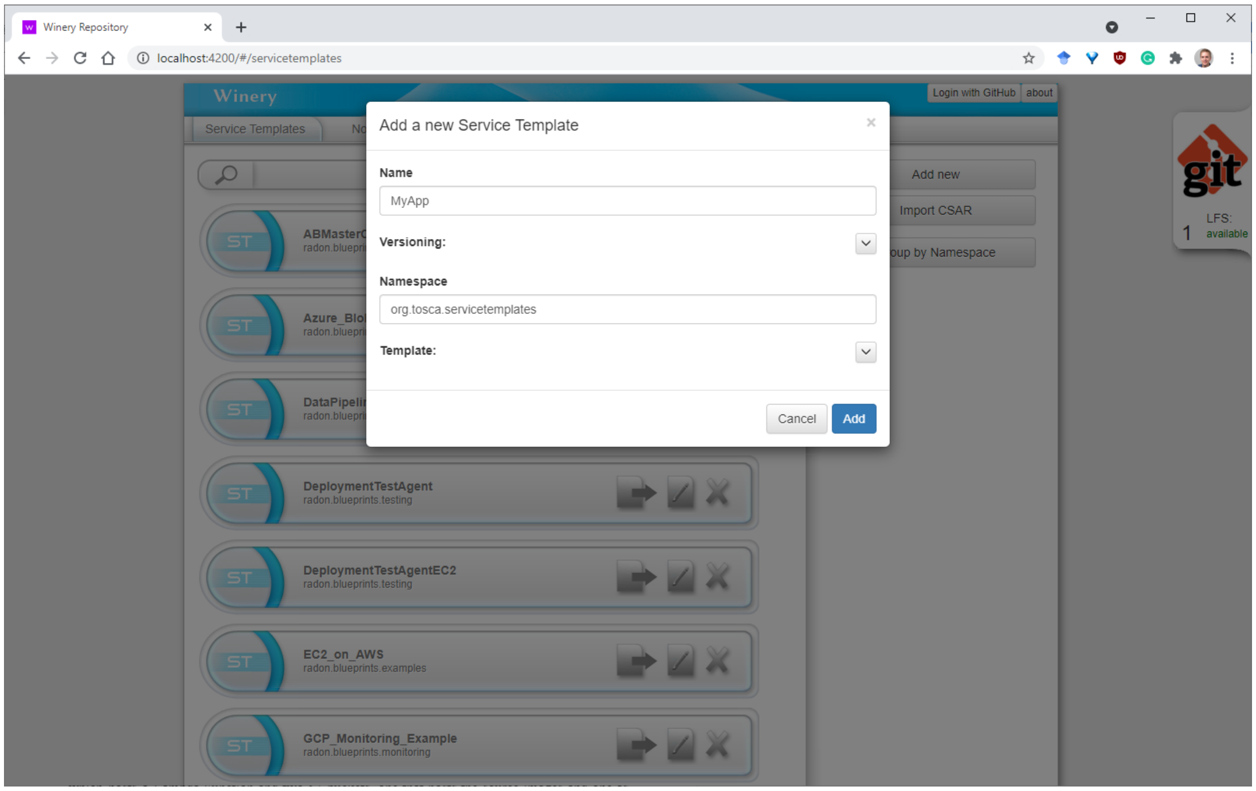
From here, you can use the node type view to get familiar with the node types provided by the RADON modeling profile. For each abstract type, as mentioned in the previous section, there are several concrete and deployable node types.

3.6.3 Compose TOSCA blueprints

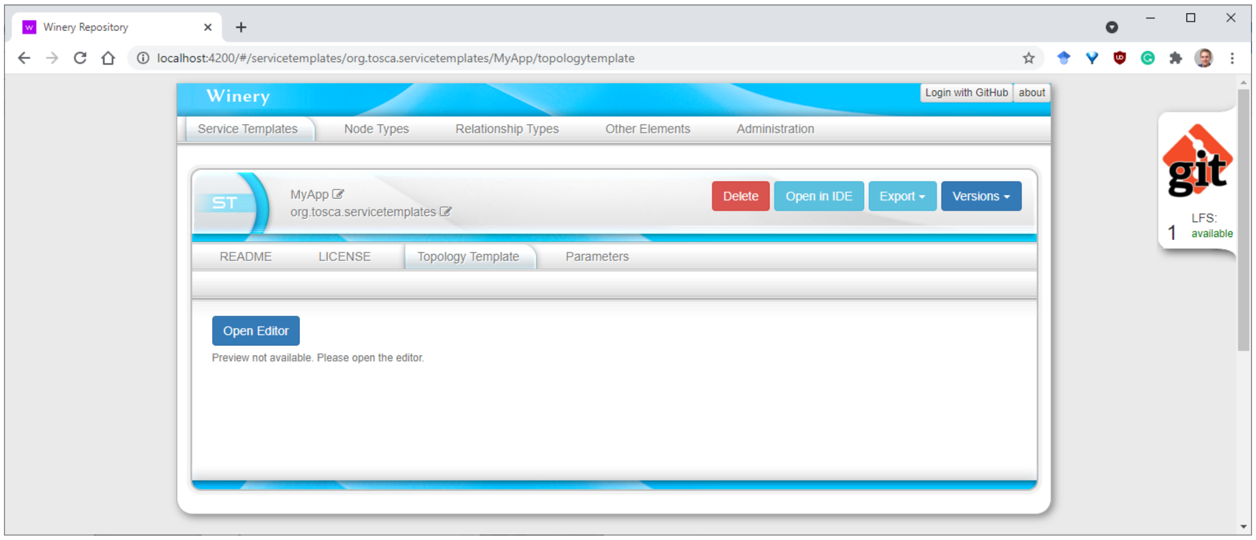
Next, we briefly present the capabilities of Eclipse Winery’s TOSCA Topology Modeler to graphically compose TOSCA service templates and modify its configuration.

In the TOSCA Management UI, go back to “Service Templates”. Users may use Eclipse Winery to inspect the configuration of existing TOSCA blueprints (similar to TOSCA node types) or create new ones. Click “Add new” and set a name as well as a suitable namespace.

[33] https://winery.readthedocs.io/en/latest/user/getting-started.html#use-a-custom-tosca-model-repository



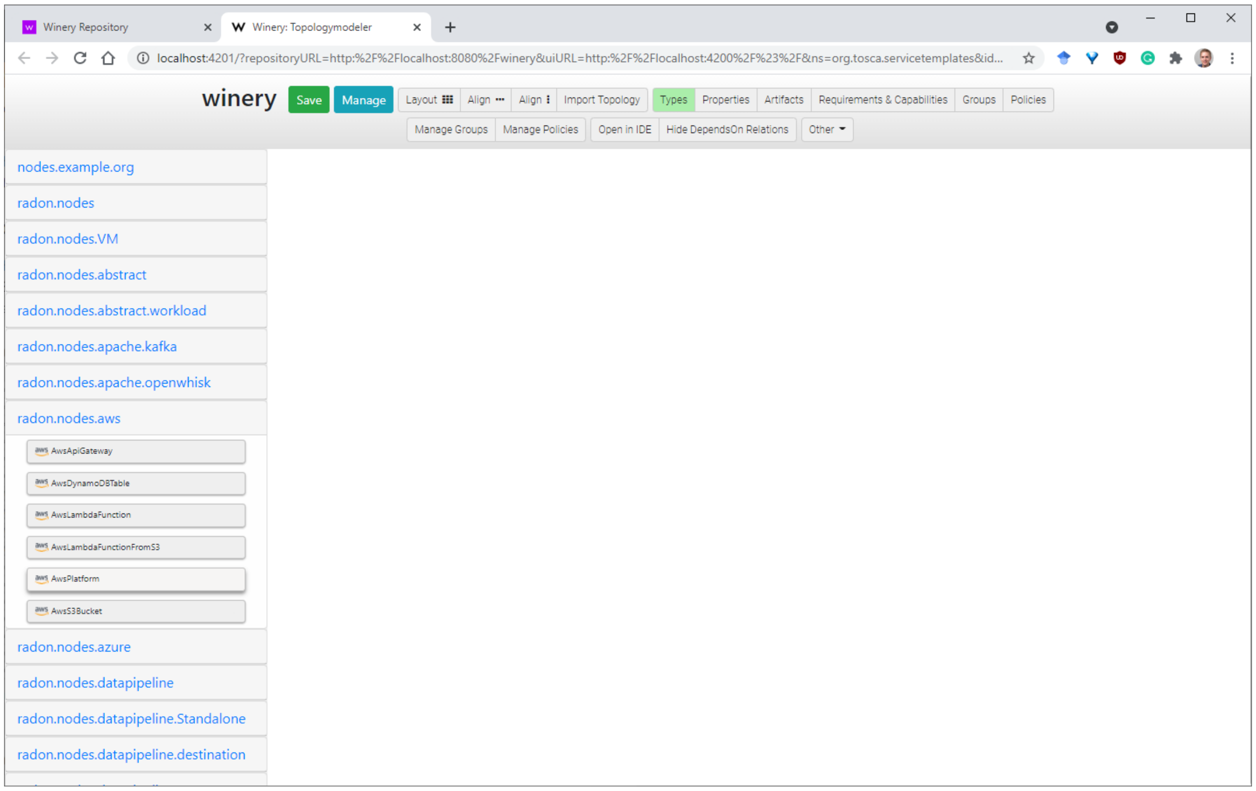
The main actions in the service template detail view are to generate a CSAR file for the complete blueprint or to open the TOSCA Topology Modeler. Users open the graphical editor by clicking on “Open Editor” in the “Topology Template” submenu.



Eclipse Winery’s user guide shows interactively how to user the TOSCA Topology Modeler[34].

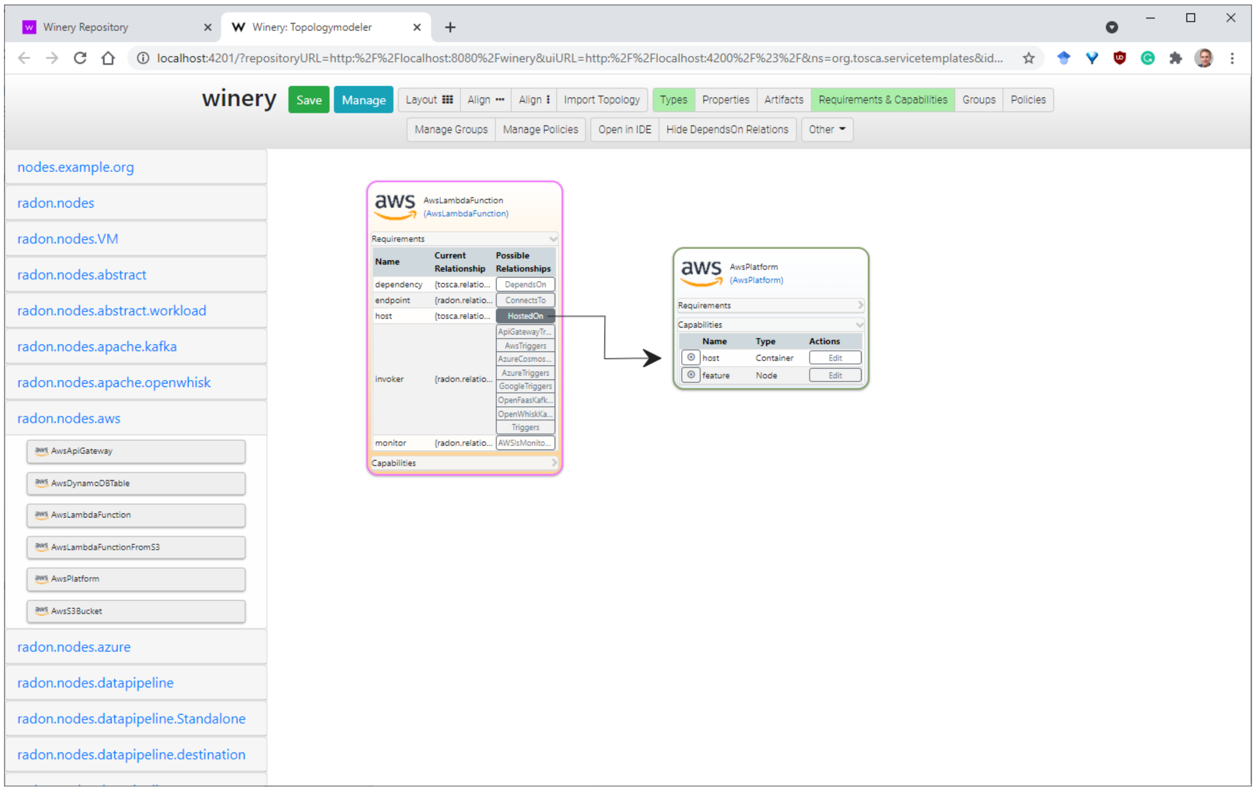
When users create new TOSCA service templates, the TOSCA Topology Modeler is started with an empty modeling canvas.

[34] https://winery.readthedocs.io/en/latest/user/yml/index.html



Users can drag-and-drop components from the palette on the left-hand side of the editor to the canvas to model their intended application structure. For example, you can drag the “AwsPlatform” entry to the canvas to create a new TOSCA Node Template of this type. In addition, you may want to add an additional node template of type “AwsLambdaFunction”.

To create a relationship between these new nodes, e.g., to express that the Lambda function is hosted on the AWS platform node, you may enable the “Requirements & Capabilities” view from the top menu. This lets you inspect the current requirements and capabilities each node exposes, by expanding either the “Requirements” or “Capabilities” box of a respective node. For the example above, users may drag from the “HostedOn” relationship of the requirement “host” to the “host” capability of the “AwsPlatform” node. By this, users are able to establish any kind of relationship between different kinds of nodes.



Name	Current Relationship	Possible Relationships
dependency	{tosca.relatio...	DependsOn
endpoint	{radon.relatio...	ConnectsTo
host	{tosca.relatio...	HostedOn
invoker	{radon.relatio...	ApiGatewayTr... AwsTriggers AzureCosmos... AzureTriggers GoogleTriggers OpenFaasKafk... OpenWhiskKa... Triggers
monitor	{radon.relatio...	AWSIsMonito...

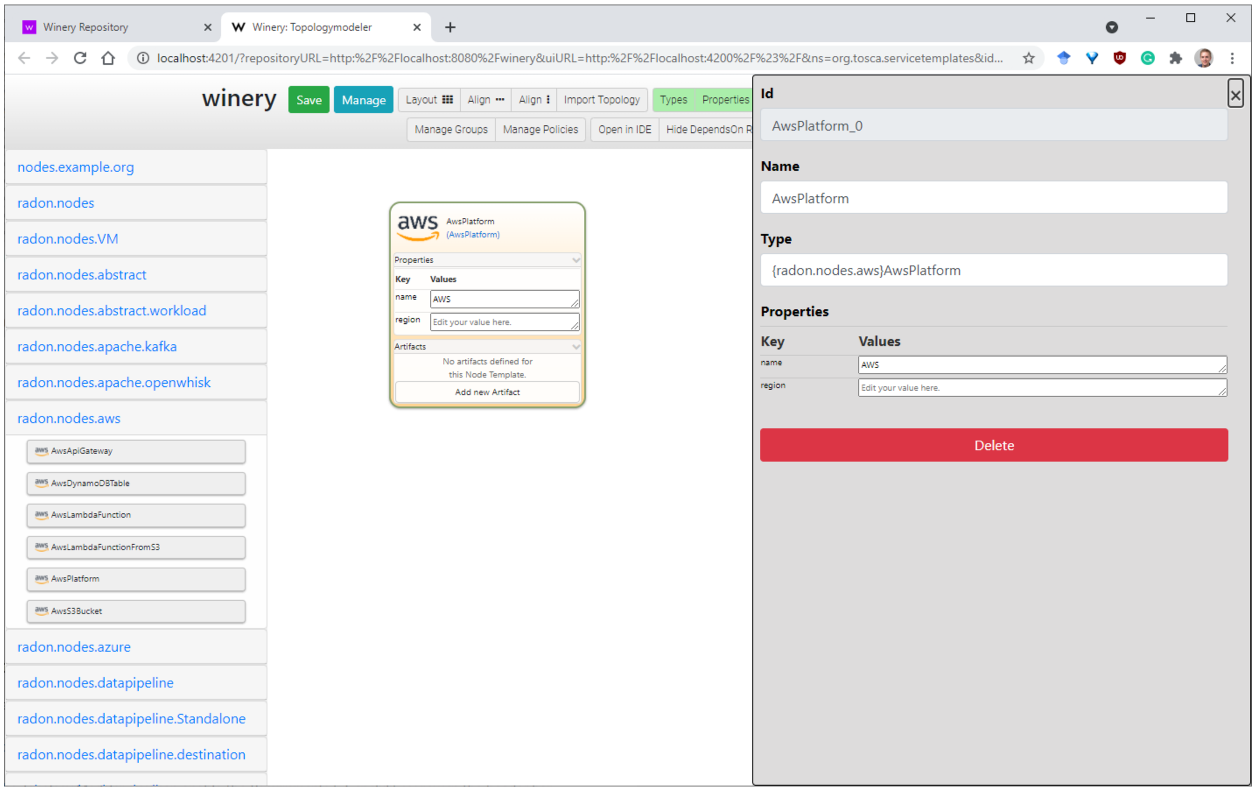
Name	Type	Actions
host	Container	Edit
feature	Node	Edit

3.6.4 Managing Properties and Artifacts

Usually, deployable TOSCA node types define properties that have to be set before being able to deploy the application. Further, TOSCA deployment artifacts representing the business logic of a node have to be specified individually on node template level. Eclipse Winery lets users to modify properties within the TOSCA Topology Modeler as well as upload and define deployment artifacts to the new TOSCA blueprint.

Users may enable the “Properties” and “Artifacts” view from the top menu. From here, users can specify and manage properties and artifacts for each individual node. Further, properties can be set and edited through the TOSCA Topology Modelers edit pane. The edit pane will be activated whenever a user selects a node. On top of properties, in the edit pane users may also change the name of a modeled node.

Editing properties for relations works similar to nodes. Click on the name of the relationship you want to edit and use the edit pane on the right-hand side to modify the available properties.



Id

AwsPlatform_0

Name

AwsPlatform

Type

{radon.nodes.aws}AwsPlatform

Properties

Key	Values
name	AWS
region	Edit your value here.

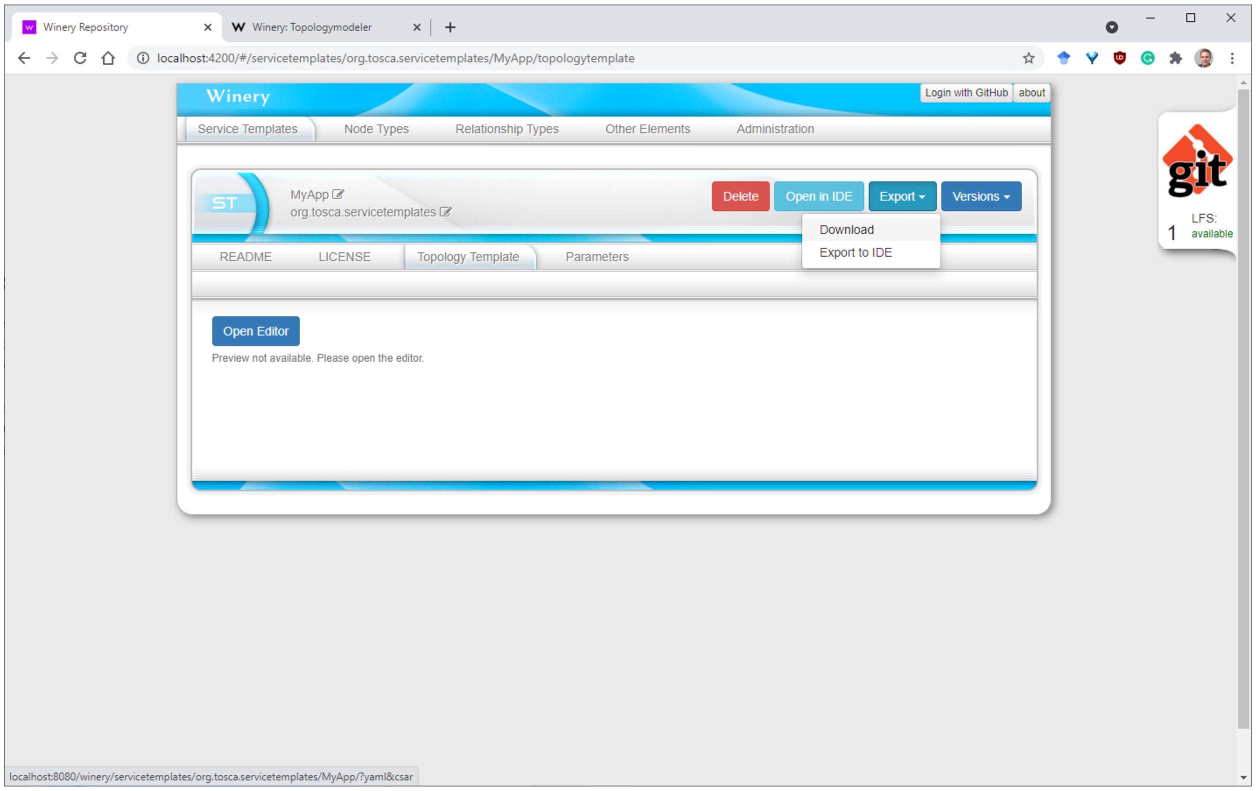
3.6.5 Create a TOSCA CSAR of a TOSCA blueprint

TOSCA's exchange format is a Cloud Service Archive (CSAR). A CSAR is essentially a ZIP archive following a certain directory layout and contains all necessary files and templates to execute the deployment of the modeled application.

Go back to the TOSCA Management UI and open the "Service Templates" view. Search for your service template and open it. In the service template detail view users may click on "Export" either to *download* the CSAR or to save it to the filesystem (relative to the configured repository path).

The generated CSAR is self-contained as it contains all type definitions, implementation and deployment artifacts as well as the TOSCA service template itself to deploy the application by a TOSCA-compliant orchestrator. For example, users may use xOpera[35], a lightweight TOSCA orchestrator, to execute the deployment using the provided "opera" CLI.

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

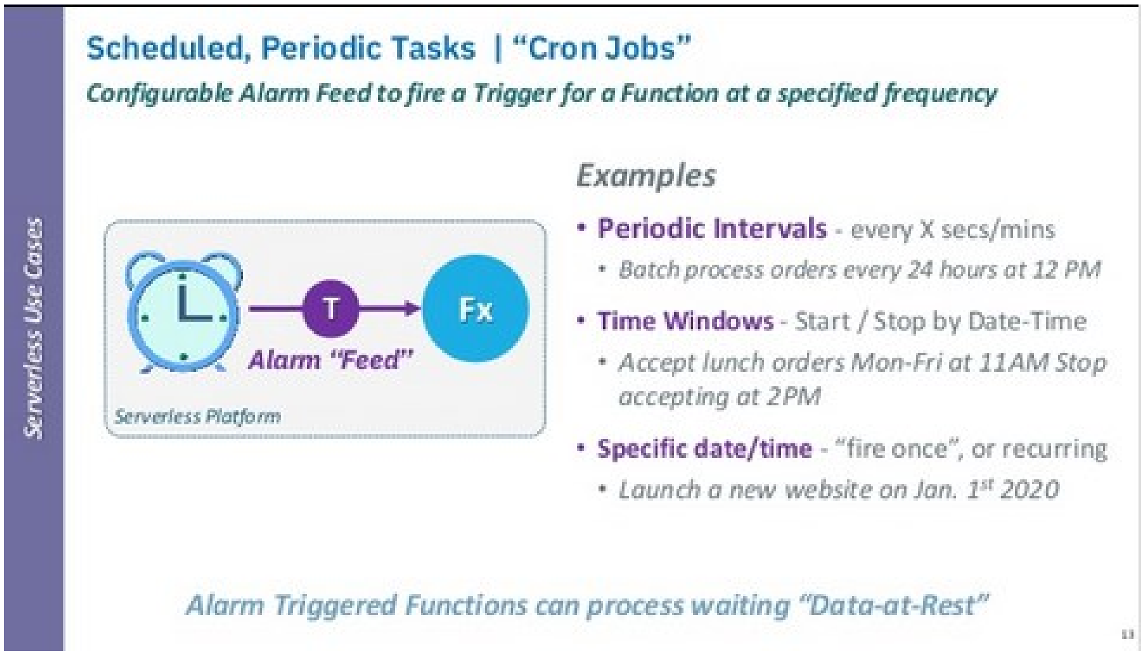

Winery Repository
Winery: Topologymodeler
localhost:4200/#/servicetemplates/org.tosca.servicetemplates/MyApp/topologytemplate
Winery
Login with GitHub
about
Service Templates
Node Types
Relationship Types
Other Elements
Administration
ST
MyApp
org.tosca.servicetemplates
Delete
Open in IDE
Export
Versions
Download
Export to IDE
README
LICENSE
Topology Template
Parameters
Open Editor
Preview not available. Please open the editor.
git
LFS:
1
available
localhost:8080/winery/servicetemplates/org.tosca.servicetemplates/MyApp/?yaml&csar

4 Function-as-a-Service: Principal Use Cases

In the following, we show three main use cases of using RADON TOSCA to model serverless applications. We provide the modeling abstraction provided by the RADON instance of Eclipse Winery[36] customized for serverless-based development for each of them.

4.1 Periodic Tasks

Use Case Description and Typical Policies. The use-case reflects tasks carried out periodically through a specified or driven interval of time and require recurrent but pausable and event-driven processing. This behavior is typical in the scope of Internet-of-Things systems that regularly check sensorial input from the outside and trigger reactive tasks to such inputs. Typical policies to employ this operational model reflect (a) recurrent and periodic intervals with simple execution consequences or (b) windowed operational procedures triggered by pre-specified events (including unknown ones) or even (c) planned events (e.g., nightly builds).


Serverless Use Cases
T
Fx
Alarm "Feed"
Serverless Platform
13

Scheduled, Periodic Tasks | "Cron Jobs"

Configurable Alarm Feed to fire a Trigger for a Function at a specified frequency

Examples

- **Periodic Intervals** - every X secs/mins
  - *Batch process orders every 24 hours at 12 PM*
- **Time Windows** - Start / Stop by Date-Time
  - *Accept lunch orders Mon-Fri at 11AM Stop accepting at 2PM*
- **Specific date/time** - "fire once", or recurring
  - *Launch a new website on Jan. 1st 2020*

Alarm Triggered Functions can process waiting "Data-at-Rest"

Technical Characteristics

- Event-driven semantics combined with various operational models (e.g., windowed, planned);
- Event derivation and factoring.

Challenges

- Multi-policy orchestration;
- Hybrid system architecture (e.g., computing with IoT and HPC blends).

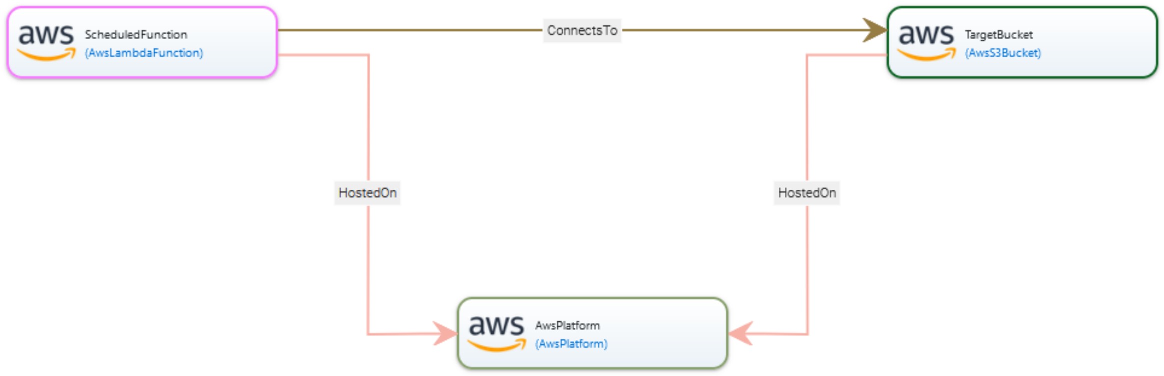
Implementation using RADON TOSCA. An example of this use case modeled using the RADON Profile for AWS is shown below. In this example Service Template[37], an AWS Lambda Function is triggered on a scheduled basis to query the file names for the previous day from a public Open Air Quality dataset available in the AWS registry for open data[38]. After getting the file names, this AWS Lambda Function stores them in .json format in a user-specified AWS S3 bucket. This is also a good example of a standalone function, since the scheduler-based invocation is modeled implicitly. In case schedulers need to be present in the mode explicitly, another option for representing such behavior would be to introduce a

[36] https://github.com/radon-h2020/radon-gmt

[37] https://github.com/radon-h2020/radon-particles/tree/master/servicetemplates/radon.blueprints/ScheduledLambdaInvocation

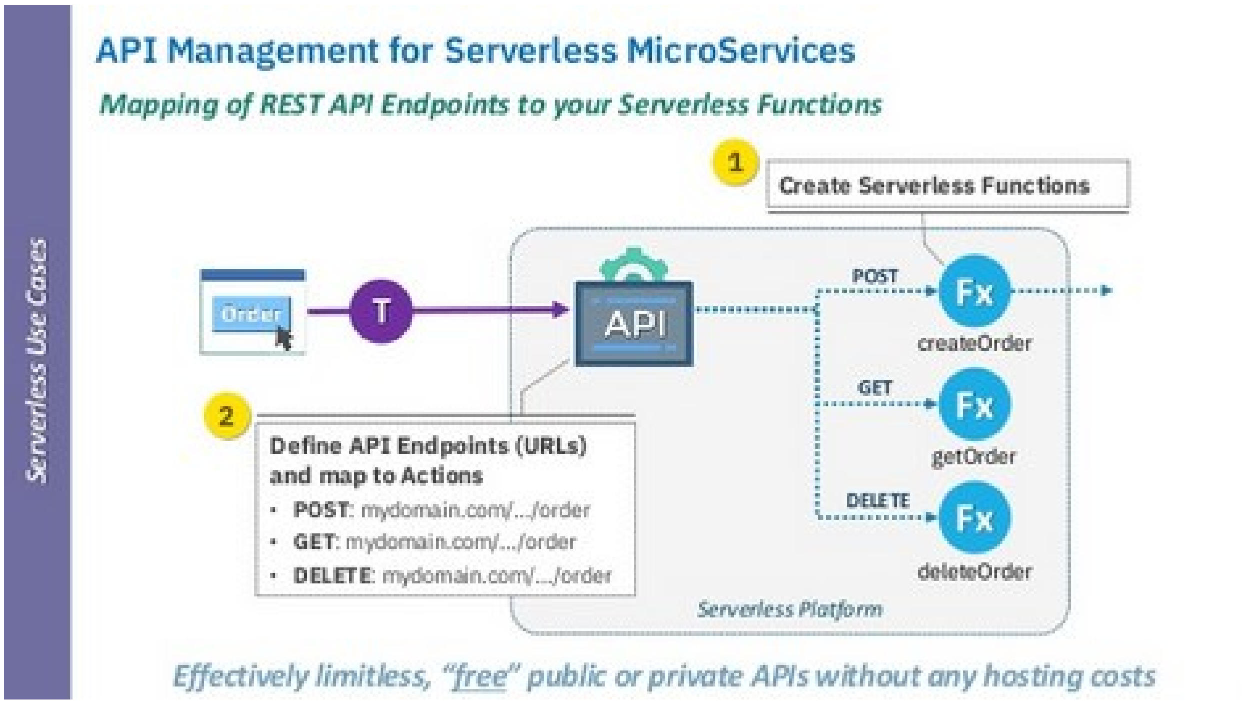
[38] https://registry.opendata.aws/openaq/

dedicated Node Type representing the scheduling expression which can be deployed as Amazon EventBridge rule in case of AWS-based applications.


aws ScheduledFunction (AwsLambdaFunction)
ConnectsTo
aws TargetBucket (AwsS3Bucket)
HostedOn
HostedOn
aws AwsPlatform (AwsPlatform)

4.2 Serverless API

Use Case Description and Typical Policies. The use-case reflects the dynamic operations of Application Programmer Interfaces (APIs) as mapped with specific functions based on contextual information or other meta-data available upon runtime (e.g., operational phases, geolocation). In such a scenario, an API management stratum can be used in addition to containerization technologies (amounting to a microservices architecture style) to properly manage the execution of said APIs in the desired fashion. Within this execution model, the API manager mapped the pre-specified endpoints (in the form of Universal Resource Locators, URLs) to concrete actions reflected by stateless functions and managed entirely at runtime (i.e., instantiated, executed, and possibly destroyed). Policies enacting this execution scenario reflect dynamic orchestration of cloud applications requiring specific fine-grained management (e.g., geolocation services) or even services that need specific execution properties to be guaranteed and specific service-level agreements to be dynamically maintained sometimes even regardless of operational costs.


Serverless Use Cases
API Management for Serverless MicroServices
Mapping of REST API Endpoints to your Serverless Functions
1
Create Serverless Functions
Order
T
API
POST
Fx
createOrder
GET
Fx
getOrder
DELETE
Fx
deleteOrder
2
Define API Endpoints (URLs) and map to Actions
• POST: mydomain.com/.../order
• GET: mydomain.com/.../order
• DELETE: mydomain.com/.../order
Serverless Platform
Effectively limitless, "free" public or private APIs without any hosting costs

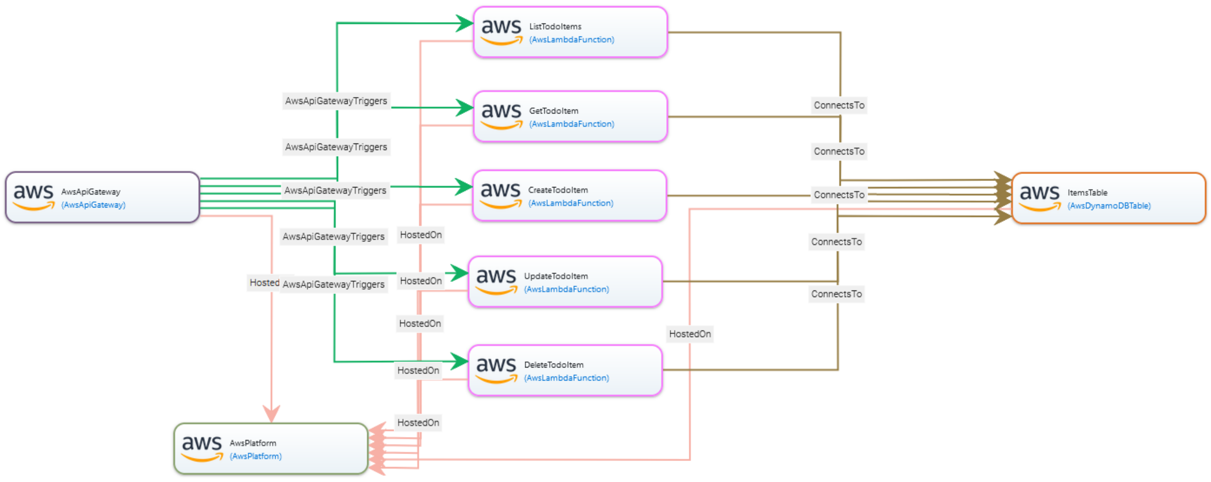
Technical Characteristics

- Microservice architecture;
- Dynamic orchestration;
- Policy-based API management;

Challenges

- Functional mapping is non-trivial;
- Dynamic orchestration has no guarantees for cost minimization;

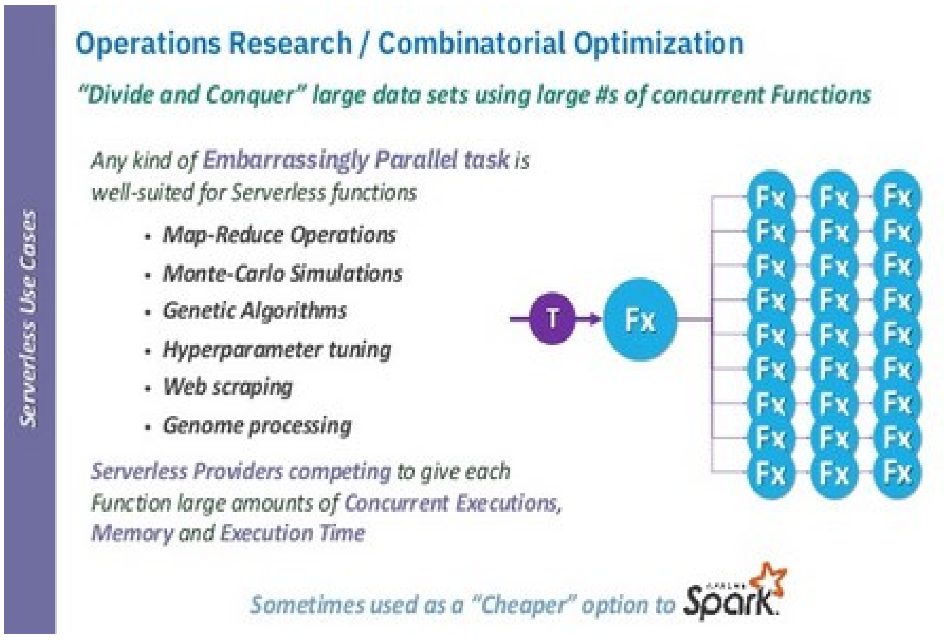
Implementation using RADON TOSCA. An example of this use case modeled with the RADON Modeling Profile is a serverless API of a simplified ToDo List application hosted on AWS[39]. Here, AWS Lambda functions responsible for creation, retrieval, modification, and deletion of ToDo items are exposed as an API hosted using AWS API Gateway. The data is stored using the AWS DynamoDB table. Here, the triggering semantics is represented using the corresponding relationships from the API Gateway to functions, with required events specified as properties of Relationship Templates. After all functions are deployed, the corresponding API is created on the API Gateway.


aws ListTodoItems (AwsLambdaFunction)
aws GetTodoItem (AwsLambdaFunction)
aws CreateTodoItem (AwsLambdaFunction)
aws UpdateTodoItem (AwsLambdaFunction)
aws DeleteTodoItem (AwsLambdaFunction)
aws AwsApiGateway (AwsApiGateway)
aws ItemsTable (AwsDynamoDBTable)
aws AwsPlatform (AwsPlatform)
AwsApiGatewayTriggers
AwsApiGatewayTriggers
AwsApiGatewayTriggers
AwsApiGatewayTriggers
AwsApiGatewayTriggers
ConnectsTo
ConnectsTo
ConnectsTo
ConnectsTo
ConnectsTo
HostedOn
HostedOn
HostedOn
HostedOn
HostedOn
HostedOn
Hosted

4.3 Embarrassingly Parallel Computing

Use Case Description and Typical Policies. Specific types of data-intensive computing require non-necessarily recurrent stream analytics to be computed with cheaper compute options than deployed and operational data-intensive middleware (e.g., nightly map-reduce operations computed by specific function-as-a-service assets). An example is hyperparameter tuning specifically designed for deep-learning jobs triggered via active learning or similar advanced extract-transform-load (ETL) pipelines. In the scope of such jobs, streaming data becomes the event source that the hybrid computing facilities manage as if it were a running event-driven system. A typical application scenario for this configuration reflects jobs enacted under specific operational requirements and execution conditions that reflect immediate and bursty data to be processed at high-speed and manageable costs. Policies in this scenario, therefore, reflect constrained evaluation over (a) the expenditure of data, (b) the expenditure of machine-time, (c) the expenditure of compute resources, (d) the job-completion time or any other policy which regulates the timeliness and performance/throughput parameters of the executed process.

[39] https://github.com/radon-h2020/radon-particles/tree/master/servicetemplates/radon.blueprints/ServerlessToDoListAPI



Serverless Use Cases

Operations Research / Combinatorial Optimization

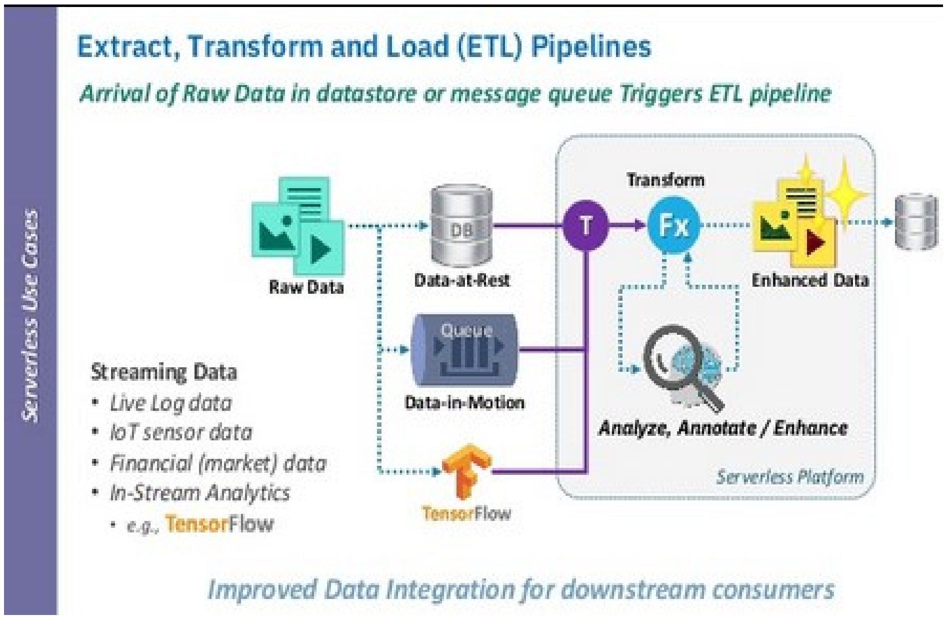
"Divide and Conquer" large data sets using large #s of concurrent Functions

Any kind of Embarrassingly Parallel task is well-suited for Serverless functions

- Map-Reduce Operations
- Monte-Carlo Simulations
- Genetic Algorithms
- Hyperparameter tuning
- Web scraping
- Genome processing

Serverless Providers competing to give each Function large amounts of Concurrent Executions, Memory and Execution Time

Sometimes used as a "Cheaper" option to Spark



Serverless Use Cases

Extract, Transform and Load (ETL) Pipelines

Arrival of Raw Data in datastore or message queue Triggers ETL pipeline

Streaming Data
- Live Log data
- IoT sensor data
- Financial (market) data
- In-Stream Analytics
  - e.g., TensorFlow

Improved Data Integration for downstream consumers

Technical Characteristics

- Policy-based execution parameters management;
- Alternative to more compute-intensive data-driven processes and middleware (e.g., Apache Spark);
- Data-intensive computing.

Challenges

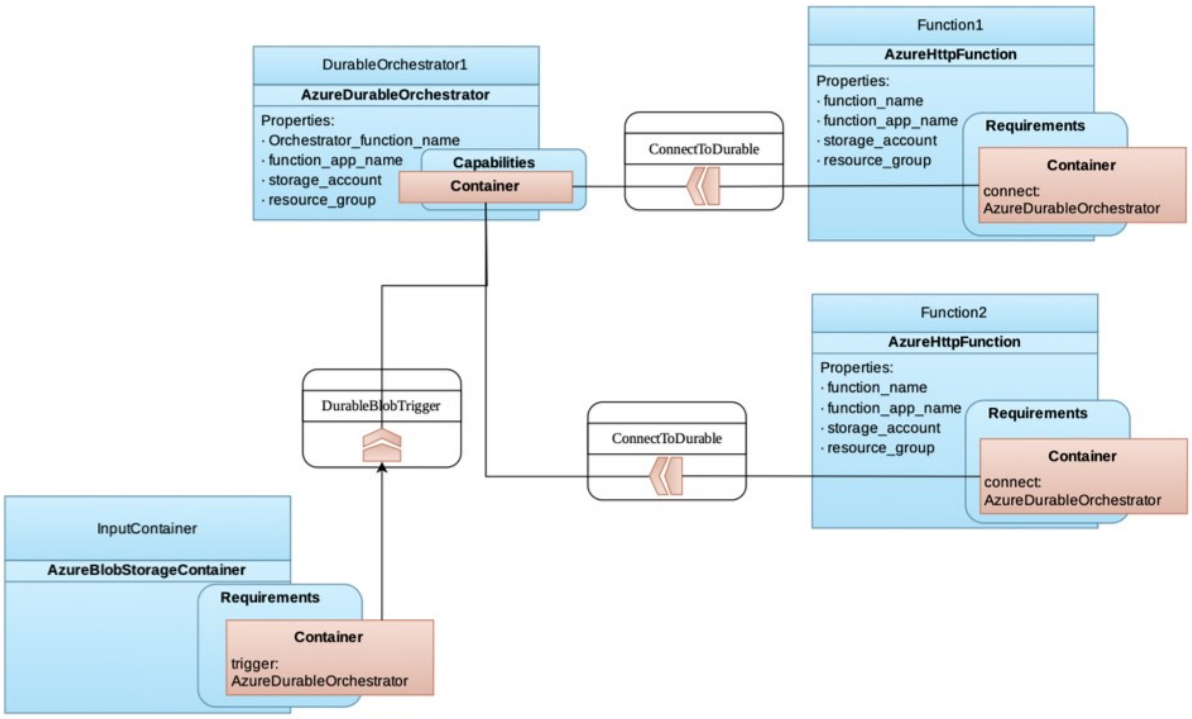
- Orchestration will lack runtime management opportunities (large-scale jobs);
- Comes with the difficulties of data-intensive computing.

Implementation using RADON TOSCA. RADON nodes and relationships allow for deploying projects for different serverless platforms. However, new trigger relationships are required to support more complex scenarios like implementing algorithms based on combinatorial optimization. Users can extend their node and relationship templates by customizing RADON TOSCA to support user-defined use cases or services and the relation between different node templates. The extension of the new relationship would involve changes like adding new valid source nodes in the existing nodes.

For example, to deploy Azure Durable Function for HTTP-triggered function chaining, we defined new nodes and relationships, including

1. nodes for Azure Durable Orchestrator function and Azure HTTP-triggered function;
2. a relationship between an Azure Blob Storage-triggered event and Orchestrator function;
3. a relationship connecting Azure HTTP-triggered function to the Azure Orchestrator function.

The node *AzureDurableOrchestrator* represents the orchestrator function that can chain other Azure functions in specific execution order. We also customize a node *AzureHttpFunction* for deploying HTTP-triggered functions. This node is used for individual functions in function chaining. The orchestrator function is triggered by uploading events in *Azure Blob Storage*. *DurableBlobTrigger*, a newly defined relationship template, configures the app settings for the orchestrator function and starts the orchestrator function when a new uploading event occurs. *ConnectToDurable* represents a relationship between them, connecting the local HTTP-triggered functions to the orchestrator. More in detail, this relationship helps to pass parameters like the URL of HTTP-triggered functions to *AzureDurableOrchestrator*. A logical diagram for an Azure Durable function example with two functions in the chain is shown in the figure below.



Furthermore, data pipeline node types can be used to design data streams and ingest data from MQTT, file systems, or GridFTP. Functions can be chained calling AWS Lambda functions directly or other functions through REST with the final goal of streaming data through the functions. They can contain the application logic, representing how data is analyzed. An example of these nodes is the AWSEMRactivity data pipeline node type, which initiates an AWS Elastic MapReduce job (tested with Spark Python), takes input from an S3 bucket, and writes output to another S3 bucket. Full details concerning the implementation can be found online[40].

[40] https://github.com/radon-h2020/radon-particles/tree/master/nodetypes/radon.nodes.datapipeline.Standalone/AWSEMRactivity

5 A More Complex Scenario: Towards Edge Deployments

The goal of this scenario is to provide a more complex scenario using the same type hierarchy and modeling approach to enable min.io and openfaas deployment on Kubernetes cluster on RaspberryPIs. The use case has been developed by the RADON[41] and SODALITE[42] Horizon 2020 projects. It provides a concrete example that demonstrates the challenges and solutions found in adopting RADON TOSCA in situations that fell outside the scope of the project requirements.

The following sections report on these technical activities and the joint experience, commenting on achievement of joint work and mutual feedback on the tools.

The assets developed in this collaboration have been released and documented in the organization repositories[43]. In particular, the integrated RADON-SODALITE node templates, that we have termed the Hybrid Compute Profile, are available at:

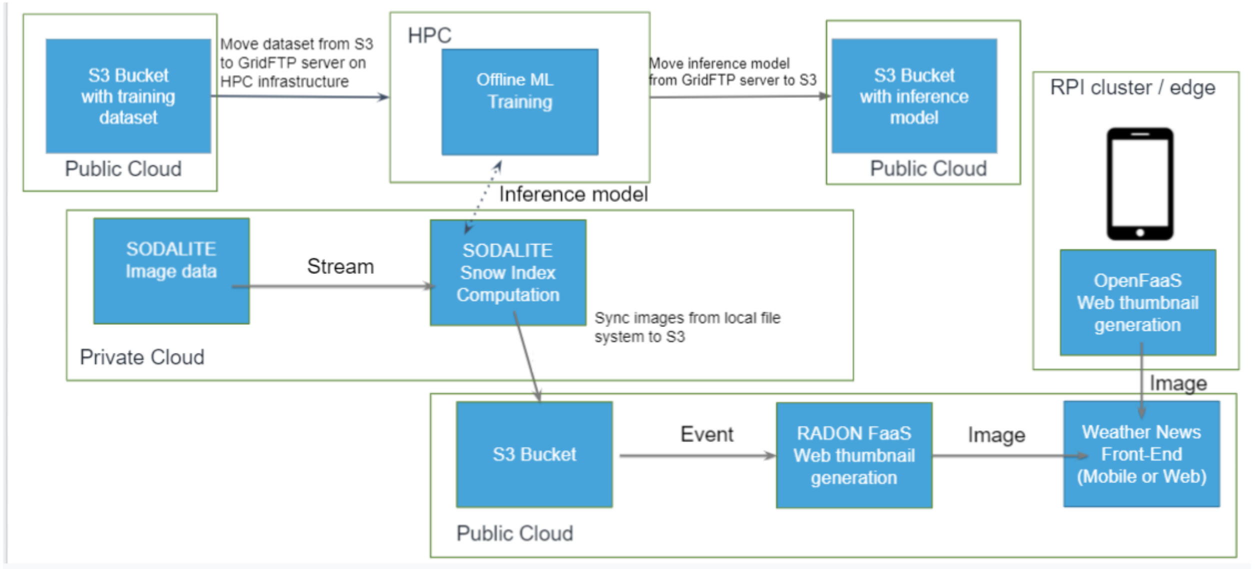
https://github.com/RADON-SODALITE/hybrid-compute-profile

The release of the asset is open source under Apache 2.0 licensing.

5.1 Weather News Front-End Case Study

The main aim of the integrated scenario was to demonstrate the ability to extend and customize the R-S frameworks in a complex end-to-end scenario falling outside the requirements of either project. It was observed that, on the one hand, RADON work emphasized FaaS technology and SODALITE similarly emphasized HPC, while on the other hand RADON did not consider HPC and similarly SODALITE did not focus on FaaS. This offered an opportunity for both projects to consider a joint scenario that posed the question on how to support deployment over heterogeneous cloud targets.

The overall technical architecture of the use case is shown in the figure below. The goal of the use case was to be able to design and deploy this end-to-end application, called the Weather News Use Case. Weather News has the objective of demonstrating an image processing and visualization pipeline stretching across a “continuum cloud”, including HPC, raspberry PI devices, and a backend public cloud with FaaS support.


S3 Bucket with training dataset
Public Cloud
Move dataset from S3 to GridFTP server on HPC infrastructure
HPC
Offline ML Training
Move inference model from GridFTP server to S3
S3 Bucket with inference model
Public Cloud
RPI cluster / edge
Inference model
SODALITE Image data
Stream
SODALITE Snow Index Computation
Sync images from local file system to S3
Private Cloud
OpenFaaS Web thumbnail generation
Image
S3 Bucket
Event
RADON FaaS Web thumbnail generation
Image
Weather News Front-End (Mobile or Web)
Public Cloud

A set of technical requirements was drafted to describe the planned work:

[41] https://radon-h2020.eu/

[42] https://www.sodalite.eu/

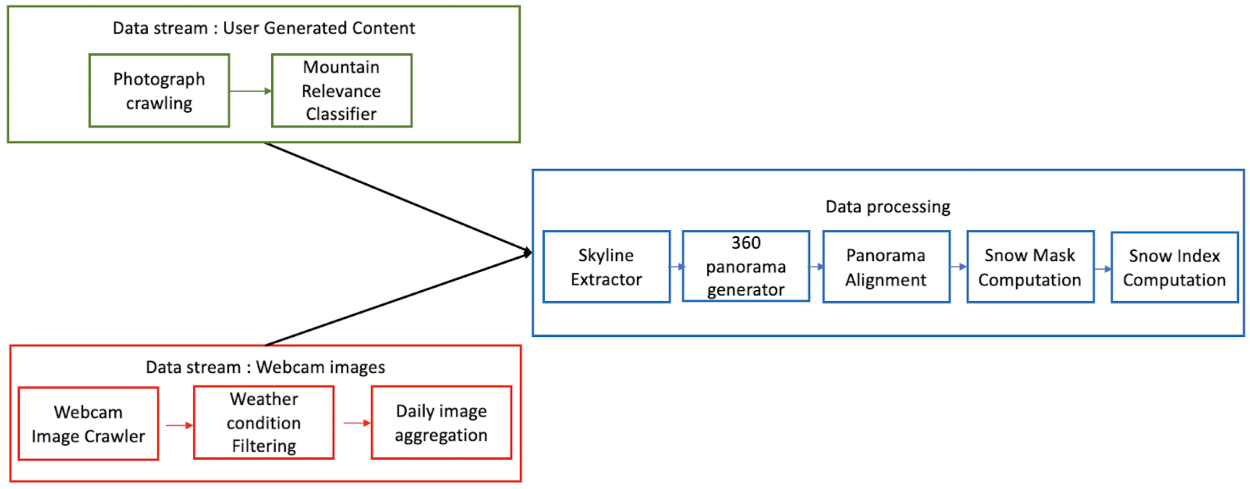
[43] https://github.com/RADON-SODALITE/

- R-SR-01: Move snow index mask and values from executing VM to remote S3 bucket
- R-SR-02: Move snow index mask and values from executing VM to remote Google Cloud storage
- R-SR-03: Define S3 thumbnail generation FaaS pipeline
- R-SR-04: Define Google Cloud thumbnail generation FaaS pipeline
- R-SR-05: Create news front-end website
- R-SR-06: Training data pipeline between cloud storage and HPC
- R-SR-07: Develop node templates for MinIO storage buckets and OpenFaaS functions.
- R-SR-08: Develop OpenFaaS function triggers.

We now discuss individually the components of the Weather News application across the different deployment environments.

5.1.1 HPC Layer

This section of the Weather News application is an adaptation of the Snow Use Case (Snow UC) developed within SODALITE as one of its use cases. The original Snow UC goal is to exploit the operational value of information derived from public web media content, specifically from mountain images contributed by users and existing webcams, to support environmental decision making in a snow-dominated context. At a technical level, Snow UC consists of an automatic system that crawls geo-located images from heterogeneous sources at scale, checks the presence of mountains in each photo and extracts a snow mask from the portion of the image denoting a mountain. Two main image sources are used: touristic webcams in the Alpine area and geo-tagged user-generated mountain photos in Flickr in a 300 x 160 km Alpine region. The figure below shows the different components of the original Snow UC pipeline. Ultimately, the pipeline consists of three sub-pipelines, where certain image processing steps are applied. As such, there are two sub-pipelines that filter and classify source images (user generated photos and webcam images) and a computational sub-pipeline that at the end computes a snow index based on the processed source images. All the components are deployed on VMs of the private OpenStack cloud.



The task force leveraged Snow UC to implement the Weather News case study. The original Snow UC pipeline was extended with the offline ML-training of one of the components of Snow UC (Skyline Extractor) and with a thumbnail generation for the resulting images of Snow UC, as presented in Figure 6.1.2. These thumbnails are then eventually consumed by Weather News Front-End, which displays images to the end user. The ML-training demonstrates deployment on an HPC cluster with the GPU accelerators. The training dataset is stored in an Amazon S3 bucket and then moved from the bucket to the HPC infrastructure. Once the ML-training is executed, the resulting inference model is moved from the HPC cluster into another S3 bucket, from where it can be further consumed by the Snow UC Skyline Extraction component.

5.1.2 Cloud and Edge Layers

The cloud layer consists of a hybrid setup, with some services hosted on OpenStack in a private cloud and some other services, in particular S3 buckets and functions, hosted on Amazon AWS.

The images made available on the S3 bucket can then be picked up by FaaS functions for further processing, which in the Weather News Case study focuses on thumbnail generation.

The thumbnail generation is implemented in two cases, both relying on Function-as-a-Service (FaaS) technology. In the first case, the thumbnails are generated in the Cloud with Amazon Lambda service and stored in Amazon S3. In this classic FaaS application scenario, a function hosted on AWS Lambda generates thumbnails for each image uploaded to a source AWS S3 bucket. More precisely, an image upload event triggers the function, which generates a thumbnail for the uploaded image and stores it in the target AWS S3 bucket.

In the second case, the same function is deployed on Edge (a Raspberry Pi cluster) with OpenFaaS and the results are stored in MinIO (an S3-compatible storage). With similar APIs of both MinIO and S3, the pipeline is integrated with Android smartphone gateway. The weather photos can be uploaded by an android application using access and secret keys. Similar to geo-tagged mountain photos, images with geo-tagging information can be uploaded within the app to apply the UC pipeline. Basic image filtration and processing like image-subsampling can be performed at the edge to minimize the response time of the workloads.

5.2 Tool Baselines

The case study leverages the following tools and assets by RADON[44] and SODALITE[45] Horizon 2020 projects:

- RADON Graphical Modelling Tool (GMT)[46]
- RADON Particles[47]
- RADON Data Pipelines[48]
- xOpera orchestration (both projects)[49]
- SODALITE IaC Modules[50]
- SODALITE Snow Use Case (Image data, Snow Index Computation)[51]

The baseline tools had several limitations, making it challenging to meet the requirements from the start:

- Although both projects were based on TOSCA 1.3 they were each tailored to their target environment and used respectively RADON Particles and SODALITE IaC Modules. The feasibility of their integration had to be established through initial tests and be followed by integration of the TOSCA service templates of the two projects in a unified hierarchy.
- The SODALITE IDE uses a TOSCA-like model definition language supported by ontology-based semantic modeling which is translated to TOSCA blueprints as an intermediate execution language to provision resources, deploy and configure applications in heterogeneous environments. Supporting the RADON node type definition in SODALITE IDE was a cross-project validation of approaches used in both projects. At the same time, while RADON GMT can address the need to have a graphical TOSCA node and template representation by design, it could not show at start SODALITE node templates as part of the visualization.
- The RADON Modeling Profile had neither abstract nor concrete node types to support HPC, as this was out of the scope of the project. Integrating the SODALITE types into the RADON

[44] https://radon-h2020.eu/
[45] https://www.sodalite.eu/
[46] https://github.com/radon-h2020/radon-gmt
[47] https://github.com/radon-h2020/radon-particles
[48] https://github.com/radon-h2020/radon-datapipeline-plugin
[49] https://github.com/xlab-si/xopera-opera
[50] https://github.com/SODALITE-EU/iac-modules
[51] https://www.sodalite.eu/sites/sodalite/files/public/content-files/deliverables/D6_2-Initial_implementation_and_evaluation_of_the_SODALITE_platform_and_use_cases.pdf section 4.1: POLIMI Snow UC.

Particles repository required introducing a completely new hierarchy of node types, resulting in a corresponding palette of components available for modeling in the RADON GMT.

- The RADON Data pipelines offered a method to move data using TOSCA and Apache NiFi. However, the solution was not designed to operate in the HPC context, where tools such as GridFTP are required to transfer data out of HPC nodes.
- RADON has developed a specialized set of raspberry PI node templates used within the Assisted Living UC; however, these were used internally by the industrial use case owner and are not available for repeatable use as part of the RADON Particles hierarchy.

Overall, the above limitations of the baselines prompted sustained actions across the two projects to extend and integrate the baseline across multiple fronts, whose main outcomes are described below.

5.3 Additional Components needed to Complete the Scenario

The technical components to complete the scenario are a hybrid compute profile, support for GridFTP, and support for edge deployment on Raspberry Pi devices, which are summarized below.

5.3.1 Hybrid Compute Profile

Since both RADON and SODALITE rely on TOSCA for modeling application topologies, most integration efforts were focused on porting the modeling constructs introduced on the SODALITE side into the format compatible with RADON tools, and RADON GMT in particular. This mainly involved modularizing introduced types and templates and organizing them according to the RADON GMT's conventions: all distinct modeling constructs such as Node Types and Relationship Types have to be stored in dedicated folders and grouped by corresponding namespaces. Furthermore, GMT employs the full TOSCA notation, meaning that modifications were needed whenever shorthands were used in the SODALITE modeling constructs. To promote reusability, GMT also requires importing all used types inside Service Templates instead of embedding these definitions directly, which also required splitting such combined models into reusable building blocks and storing them as described previously.

Another important part of the integration is organization of the deployment logic, i.e., Ansible playbooks enabling the deployment of corresponding modeling constructs. RADON GMT requires to store these implementations also following specific file organization conventions: inside the files/{artifact_name} folder related to the corresponding modeling construct such as Node Type.

A summary of the hybrid compute profile can be found in Appendix A.

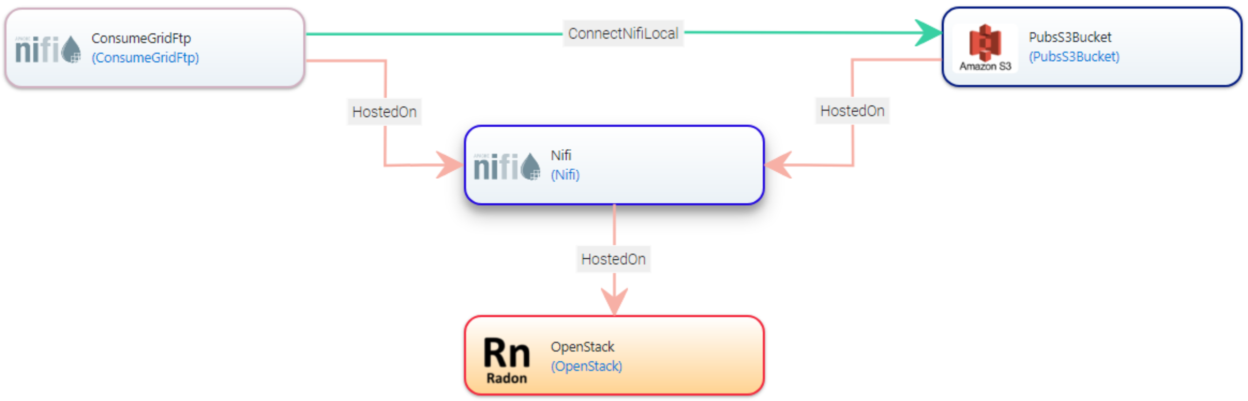
5.3.2 GridFTP Support

There was a need to continuously migrate data between the HPC and Cloud layers to implement the selected use case. In most HPC systems, the main protocol for data transfer between the Grid servers and the outside world is GridFTP due to its performance, reliability, and security. Data pipeline TOSCA node types have been previously developed inside the RADON project for deploying and orchestrating data migration service across Cloud and on-premises environments. However, GridFTP was not supported by both the existing TOSCA models and also by the underlying technologies (Apache NiFi and AWS).

To solve this issue a new GridFTP data pipeline node types were designed and implemented to set up real-time data migration services. While a custom approach was needed (as the underlying technology did not officially support GridFTP) using low level GridFTP Linux client libraries, it demonstrated that RADON data pipeline node types can be adapted for custom data sources with a medium amount of effort. Two GridFTP related node types were created:

- ConsumeGridFtp - listens for new files in a specific GridFTP server folder;
- PublishGridFtp - transfers files into a specific GridFTP server folder.

Both node types require GridFTP certificates to be available and must be hosted on a NiFi node type, which can be installed either on OpenStack VM, AWS EC2 VM or as a Docker container. The figure below illustrates a sample data pipeline, which receives data through ConsumeGridFTP and transfers files into an AWS S3 bucket using a PubsS3Bucket data pipeline node type.



The unified data management services, such as FTS3 and Globus, do provide a similar real-time data migration between HPC and Cloud storages, however the variety of supported storage types and platforms is limited. Using the RADON data pipeline node types means that data can be transferred not only between the protocols such as GridFTP and S3, but also to other supported data pipeline node types (e.g., MQTT, Azure storage, Google Cloud storage, FTP) and can be further extended for support of additional storage services (e.g. Kafka, SQL databases). Furthermore, the data pipeline node types can be used to migrate data to multiple external datastores at the same time and data can be transformed or processed on the way using FaaS functions. Therefore, the developed RADON data pipelines enable data migration between heterogeneous platforms, such as HPC, Cloud storages, data streams and serverless platforms.

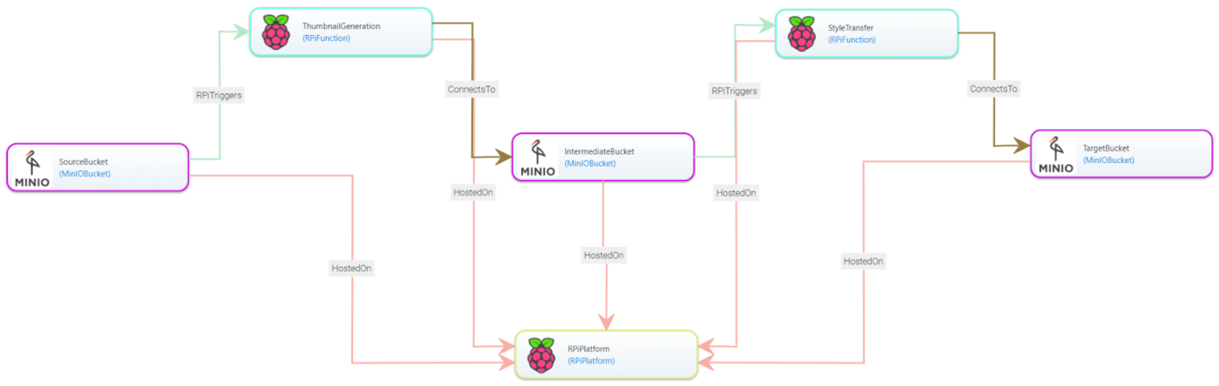
5.3.3 Support for Edge Deployment on Raspberry Pi Devices

New node templates were defined to deploy serverless functions on Raspberry Pi based private edge clusters. These include node definitions for MinIO data buckets and OpenFaaS deployed on a lightweight Kubernetes (k3s) cluster. MinIO is used for creating storage buckets in a private LAN node and follows APIs similar to the Amazon S3 bucket. We also integrated this with an Android based smartphone gateway to directly upload or download images to these MinIO buckets.

We then implemented custom OpenFaaS triggers that call the serverless functions on a `s3:ObjectCreated:*` event. We assume that the docker images are already available as part of a local docker registry in the master node of the k3s cluster.

To test the node templates, we created a sample application that generates thumbnails for input images, uploaded to a MinIO `SourceBucket`. This bucket triggers the `image-resize` function, hosted on an `RPi-Platform`, that saves the output image to a `TargetBucket`. We extend an open-source Android application to create a smartphone front-end UI.

An overview of this demo application is shown in the figure below. As visible from the figure, the RADON GMT has been extended to display customized icons for the new Raspberry Pi and MinIO data buckets.



6 Conclusion and Future Work

Emerging compute models blend any technology which deviates from the de-facto state-of-the-art architecture pattern for cloud-native applications, namely, the microservices architecture pattern. In this scope, this deliverable has explored best practices, tools, and a potential TOSCA profile.

In the short term, future work will reflect the proceedings around this document. The present document will undergo the following likely steps as per the bylaws and operational procedures of the TOSCA technical committee: (a) reported conclusions and recommendations will be discussed and possibly adopted/approved by the TOSCA TC; (b) a presentation to the general TOSCA public may ensue; (c) the TOSCA ECAH will revise and consolidate its future objectives stemming from what was reported in this document.

Concerning future work, the proceedings of ECAH have highlighted three key areas for further refinements:

1. There needs to be a holistic platform that unites and consolidates hybrid compute design and operations.
2. Verification and validation of hybrid compute applications deserve further study.
3. Specifications of hybrid compute applications demand more advanced and container-based facilities in TOSCA.

It is the recommendation of this office to discuss these limitations as next steps for the coming time of ECAH, specifically, wherefore its operations were to be confirmed as part of the TOSCA TC. Future deliverables along the lines of the aforementioned recommendation will most likely overlap or continue the present one and specifically focus on the points reported above.

7 References

[Baldini2017] Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P. & Tardieu, O. (2017). The serverless trilemma: function composition for serverless computing. In E. Torlak, T. van der Storm & R. Biddle (eds.), *Onward!* (p./pp. 89-103): ACM. ISBN: 978-1-4503-5530-8.

[Bass2003] Bass, L., Clements, P., Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley. ISBN: 9780321154958.

[Jon19] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N. and Gonzalez, J.E., 2019. Cloud programming simplified: A berkeley view on serverless computing. arXiv preprint arXiv:1902.03383.

[Kumar2008] Kumar, B., Delauré, Y. & Crane, M. (2008). High-Performance Computing of Multiphase Flow. *ERCIM News*, 2008.

[Park2018] Park, D. S. (2018). Future computing with IoT and cloud computing. *J. Supercomput.*, 74, 6401-6407.

[Wurster2018] Wurster, M., Breitenbücher, U., Képes, K., Leymann, F., & Yussupov, V. (2018, November). Modeling and automated deployment of serverless applications using TOSCA. In 2018 IEEE 11th conference on service-oriented computing and applications (SOCA) (pp. 73-80). IEEE.

Appendix A. Hybrid Compute Profile

A.1 Artifact Types

Namespace	Types
radon.artifacts	• Ansible • Repository
radon.artifacts.archive	• JAR • Zip
radon.artifacts.datapipeline	• InstallPipeline
radon.artifacts.docker	• DockerImage

A.2 Capability Types

Namespace	Types
radon.capabilities	• Invocable
radon.capabilities.container	• DockerRuntime • JavaRuntime
radon.capabilities.datapipeline	• ConnectToPipeline
radon.capabilities.kafka	• KafkaHosting • KafkaTopic
radon.capabilities.monitoring	• Monitor

A.3 Data Types

Namespace	Types
radon.datatypes	• Activity • Entry • Event • Interaction • Precedence • RandomVariable
radon.datatypes.function	• Entries
radon.datatypes.workload	• Entries
sodalite.datatypes.OpenStack	• env • SecurityRule
sodalite.datatypes.OpenStack.env	• OS
sodalite.datatypes.OpenStack.env.Token	• EGI

A.4 Node Types

Namespace	Types
radon.nodes	• SockShop • Workstation
radon.nodes.abstract	• ApiGateway • CloudPlatform • ContainerApplication • ContainerRuntime • DataPipeline • Function • ObjectStorage • Service • WebApplication

	• WebServer • Workload
radon.nodes.abstract.workload	• ClosedWorkload • OpenWorkload
radon.nodes.apache.kafka	• KafkaBroker • KafkaTopic
radon.nodes.apache.openwhisk	• OpenWhiskFunction • OpenWhiskPlatform
radon.nodes.aws	• AwsApiGateway • AwsDynamoDBTable • AwsLambdaFunction • AwsLambdaFunctionFromS3 • AwsPlatform • AwsRoute53 • AwsS3Bucket
radon.nodes.azure	• AzureCosmosDB • AzureFunction • AzureHttpTriggeredFunction • AzurePlatform • AzureResource • AzureResourceTriggeredFunction • AzureTimerTriggeredFunction
radon.nodes.datapipeline	• DestinationPB • MidwayPB • PipelineBlock • SourcePB • Standalone

radon.nodes.datapipeline.destination	● PubGCS ● PublishDataEndPoint ● PublishGridFtp ● PublishLocal ● PublishRemote ● PubsAzureBlob ● PubsMQTT ● PubsS3Bucket ● PubsSFTP
radon.nodes.datapipeline.process	● Decrypt ● Encrypt ● FaaSFunction ● InvokeLambda ● InvokeOpenFaaS ● LocalAction ● RemoteAction ● RouteToRemote
radon.nodes.datapipeline.source	● ConsAzureBlob ● ConsGCSBucket ● ConsMQTT ● ConsS3Bucket ● ConsSFTP ● ConsumeDataEndPoint ● ConsumeGridFtp ● ConsumeLocal ● ConsumeRemote
radon.nodes.datapipeline.Standalone	● AWSCopyDynamodbToS3 ● AWSCopyS3ToDynamodb ● AWSCopyS3ToS3 ● AWSEMRactivity ● AWSShellCommand ● AWSSqlActivity
radon.nodes.docker	● DockerApplication ● DockerEngine

radon.nodes.google	• GoogleCloudBucket • GoogleCloudBucketTriggeredFunction • GoogleCloudFunction • GoogleCloudPlatform • GoogleCloudResource
radon.nodes.java	• JavaApplication • JavaRuntime
radon.nodes.mongodb	• MongoDBDatabase • MongoDBMS
radon.nodes.monitoring	• NodeExporter • PushGateway
radon.nodes.mqtt	• MosquittoBroker
radon.nodes.mysql	• MySQLDatabase • MySQLDBMS
radon.nodes.nifi	• Nifi • NiFiDocker • Pipeline
radon.nodes.nodejs	• NodeJSApplication
radon.nodes.openfaas	• OpenFaasFunction • OpenFaasPlatform
radon.nodes.testing	• AB • CTTAgent • DataPipeline • DeploymentTestAgent • JMeter • LoadTestAgent • Locust • QT

radon.nodes.VM	• EC2 • OpenStack
sodalite.nodes	• ConfigurationDemo • DockerHost • DockerizedComponent • DockerNetwork • DockerRegistry • DockerVolume • RegistryCertificate • RegistryServerCertificate • TestComponent
sodalite.nodes.OpenStack	• Keypair • SecurityRules • VM

A.5 Policy Types

Namespace	Types
radon.policies	• Performance
radon.policies.performance	• MeanResponseTime • MeanTotalResponseTime
radon.policies.scaling	• AutoScale • ScaleIn • ScaleOut
radon.policies.testing	• ABLoadTest • DataPipelineLoadTest • HttpEndpointTest • JMeterLoadTest • LoadTest • LocustLoadTest • PingTest • QTLoadTest • TcpPingTest

	• Test

A.6 Relationship Types

Namespace	Types
radon.relationships	• ConnectsTo • Triggers
radon.relationships.apache.kafka	• PublishToKafkaTopic
radon.relationships.apache.openwhisk	• OpenWhiskKafkaTriggers
radon.relationships.aws	• ApiGatewayTriggers • AwsTriggers
radon.relationships.azure	• AzureCosmosDBTriggers • AzureTriggers
radon.relationships.datapipeline	• ConnectNifiLocal • ConnectNifiRemote
radon.relationships.google	• GoogleTriggers
radon.relationships.monitoring	• AWSIsMonitoredBy • GCPIsMonitoredBy
radon.relationships.openfaas	• OpenFaasKafkaTriggers

A.7 Service Templates

Namespace	Types
example.org.tosca.servicetemplates	• NiFiDocker
radon.blueprints	• ServerlessToDoListAPI • ServerlessToDoListAPI_withDNS_w1-wip1 • SockShop

	● SockShopTestingExample ● ThumbnailGeneration ● ThumbnailGeneration_CDL-w1-wip1 ● ThumbnailGeneration_FromS3-w1-wip1 ● ThumbnailGeneration_GCP-w1-wip2
radon.blueprints.datapipeline	● ToyExampleNifi
radon.blueprints.examples	● AWS_EMR_Example ● Azure_Blob_Datapipeline_Example ● DataPipelineExample ● EC2_on_AWS ● gridFTPtoS3pipeline ● MQTT_Data_Pipeline_Encryption_Decryption_Example ● S3toGridFTPpipeline ● SFTP_Datapipeline_Example ● TestPython
radon.blueprints.monitoring	● GCP_Monitoring_Example
radon.blueprints.testing	● ABMasterOnly ● DataPipelineAWSdemoSUT ● DeploymentTestAgent ● DeploymentTestAgentEC2 ● JMeterMasterOnly ● JMeterMasterOnlyEC2 ● LocustMasterOnly ● NiFiTIDocker ● QTMasterOnly ● ServerlessToDoListAPITestingExample
sodalite.blueprints	● demo-snow-cloud

Appendix B. Acknowledgments

Dario Di Nucci and Damian Tamburri are supported by the European Commission grant no. 825040 (RADON H2020).

Damian Tamburri is supported by the European Commission grant no. 825480 (SODALITE H2020).

B.1 Special Thanks

Pelle Jakovits, University of Tartu

Shreshth Tuli, Imperial College of London

Michael Wurster, University of Stuttgart

Vladimir Yussupov, University of Stuttgart

B.2 Participants

Calin Curescu, Ericsson

Dario Di Nucci, Jheronimus Academy of Data Science (JADS)

Paul Jordan, Individual Member

Chris Lauwers, Individual Member

Matthew Rutkowski, IBM

Damian Tamburri, Jheronimus Academy of Data Science (JADS)

Appendix C. Revision History

Revision	Date	Editor	Changes Made
1.0	04 Oct 2021	Dario Di Nucci Damian A. Tamburri	First version of the note
1.0 Rev 2	09 Apr 2022	Dario Di Nucci	Incorporate comments from Paul Jordan

Appendix D. Notices