



Rational decomposition and orchestration for serverless computing

Deliverable D4.3 RADON Models I

Version: 1.0

Publication Date: 31-August-2019

Disclaimer:

The RADON project is co-funded by the European Commission under the Horizon 2020 Framework Programme. This document reflects only authors' views. EC is not liable for any use that may be done of the information contained therein.

Deliverable Card

Deliverable	D4.3
Title:	RADON Models I
Editor(s):	Michael Wurster (UST) and Vladimir Yusupov (UST)
Contributor(s):	Damian Tamburri (TJD), Michael Wurster (UST), Vladimir Yusupov (UST), Lulai Zhu (IMP)
Reviewers:	Mike Long (PRQ), Lulai Zhu (IMP)
Type:	Report
Version:	1.0
Date:	31-August-2019
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables
Copyright:	RADON consortium

The RADON project partners

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

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

Executive summary

This document presents an initial version of RADON modeling approach consisting of abstract and deployable entity modeling layers that comprise a set of abstract and technology-specific modeling constructs addressing the RADON modeling challenges.

The document outlines the modeling specifics with respect to the challenges in the context of RADON together with the detailed description of models in the companion document.

This deliverable will serve as a basis for the final RADON Models deliverable due in M18. The work presented in this document has been performed in the context of task T4.2. All models described in the document are publicly-available in the so-called “RADON Particles” repository. In addition, a respective **companion** document covers the detailed presentation of these “RADON Particles” in the form of TOSCA type specifications.

Glossary

AEML	Abstract Entities Modeling Layer
CML	Cloud Modeling Language
CSAR	Cloud Service Archive
DEML	Deployable Entities Modeling Layer
FaaS	Function as a Service
GMT	Graphical Modeling Tool
IDE	Integrated Development Environment
MSA	Microservice Architecture
NFR	Non-functional requirement
TOSCA	Topology and Orchestration Specification for Cloud Applications
VM	Virtual Machine

Table of contents

1. Introduction	6
1.1 Deliverable Objectives	6
1.2 Overview of Main Achievements	6
1.3 Structure of the Document	7
2. Requirements	7
3. TOSCA Fundamentals	9
4. Research and Development Approach	10
4.1 Research Assumptions and Design Issues	10
4.2 Modeling Approach Overview	10
4.3 RADON Modeling Profile	13
4.3.1 RADON Namespace	13
4.3.2 RADON Types Hierarchy	13
Modeling FaaS Functions	14
Data Pipelines Modeling	21
Microservices Modeling	23
Non-Functional Requirements Modeling	24
4.4 RADON Template Library	26
5. Conclusions	28
Future work	29
References	30

1. Introduction

The RADON project [Casale2019] aims to provide a unified DevOps experience allowing to employ serverless Function as a Service (FaaS) technology in modern software system engineering. In order to achieve this, RADON introduces a novel modeling approach and a set of standardized, reusable model components for orchestrating microservices, FaaS-based serverless applications, and data processing pipelines. This document elaborates on the initial modeling approach and describes in detail the set of required models. In addition, a **companion** document is provided presenting the specification of reusable TOSCA types used to model RADON applications.

1.1 Deliverable Objectives

This document presents the initial version of the RADON Models towards delivering the reported outcome from above. The baseline for this initial version is the list of reference application-level technologies agreed upon the consortium that has been published in deliverable D2.1 (Initial requirements and baseline). RADON models utilize and extend the TOSCA language for emerging compute contexts. With the formula “emerging compute contexts”, RADON intends to target (1) serverless and FaaS-enabled orchestration, (2) microservice orchestration, and (3) data pipeline orchestration.

This deliverable reflects that TOSCA is usable and adaptable for the aforementioned emerging compute contexts. The present deliverable has four main objectives:

1. Define best practices for the usage of TOSCA in such contexts
2. Publish a first working profile providing reusable TOSCA types for such contexts
3. Provide a series of abstractions that may consolidate or compound the usage of TOSCA

1.2 Overview of Main Achievements

With the work done as part of this deliverable, the consortium makes several contributions:

- Introduction of two modeling layers that can be used to model on abstract level as well as on concrete level to produce deployable blueprints
- Presentation of RADON’s type hierarchy and the intended use
- Detailed discussion of different modeling styles in the FaaS context by employing different function triggering semantics using the notion of “invocable” and “standalone” functions
- Presentation of an initial approach to model application components of different granularity, e.g., modeling of FaaS-based microservices with traditional components
- Introduction of a first step towards the modeling and orchestration of data pipelines using TOSCA to filter, transform, or analyze streams of data
- Implementation of RADON’s Template Library in the form of a publicly maintained GitHub repository
- Definition of initial reusable types as defined in detail in the companion document to cover the reference application level technologies agreed upon the consortium

1.3 Structure of the Document

The rest of the deliverable is structured as follows: Section 2 presents the requirements specifically defined for the RADON models. In Section 3, we provide fundamental information about TOSCA and the most important entities used in RADON. Section 4 elaborates on the research and development approach of creating the RADON Modeling Profile and introduces the RADON Template Library. Section 5 concludes the deliverable and points out future work.

2. Requirements

The RADON requirements analysis was presented in the deliverable D2.1 . The resulting set of requirements that are going to be used as a basis for guiding the package-related work activities as its part comprises RADON model-related requirements. This section briefly summarizes these requirements.

ID	R-T4.2-1
Title	Deployment types heterogeneity
Priority	Must have
Description	RADON model must allow expressing combinations of different deployment types including paradigm-specific elements, e.g., events and triggers.

ID	R-T4.2-2
Title	Reusable types and blueprints
Priority	Should have
Description	In RADON we should provide a repository (e.g., GitHub) to provide reusable types and blueprints.

ID	R-T4.2-3
Title	Data processing modeling
Priority	Must have
Description	The models must be able to define different kinds of data processing tasks and control flow elements in order to express the behavior of my application.

ID	R-T4.2-4
-----------	----------

Title	Preconditions for data processing
Priority	Should have
Description	The models should be able to define certain preconditions for filtering which data objects to move/stream through the pipeline.

ID	R-T4.2-5
Title	Scaling of computing resources
Priority	Should have
Description	The models should be able to define how and when to scale certain computing resources.

ID	R-T4.2-6
Title	Data processing compression
Priority	Could have
Description	The models could define configurations regarding data compression and uncompression for certain processing components.

ID	R-T4.2-7
Title	Test case specification
Priority	Must have
Description	The models must be able to include the description of test cases for certain components (annotate test-related information).

3. TOSCA Fundamentals

This section of the deliverable briefly elaborates on the relevant fundamental TOSCA concepts required for introducing the RADON modeling approach.

TOSCA is a provider-agnostic cloud modeling language (CML) [Bergmayr2018] standardized by OASIS [OASIS2019]. With TOSCA, cloud applications can be modeled in the form of a graph that describes the connectivity of application's components in a declarative fashion [Binz2012, Lipton2018]. The resulting *topology*, contained in a so-called *service template* can then be exported as a Cloud Service Archive (CSAR) and automatically deployed by a TOSCA compliant orchestrator. Figure 1 shows graphically TOSCA's metamodel and depicts the connections between the entities.

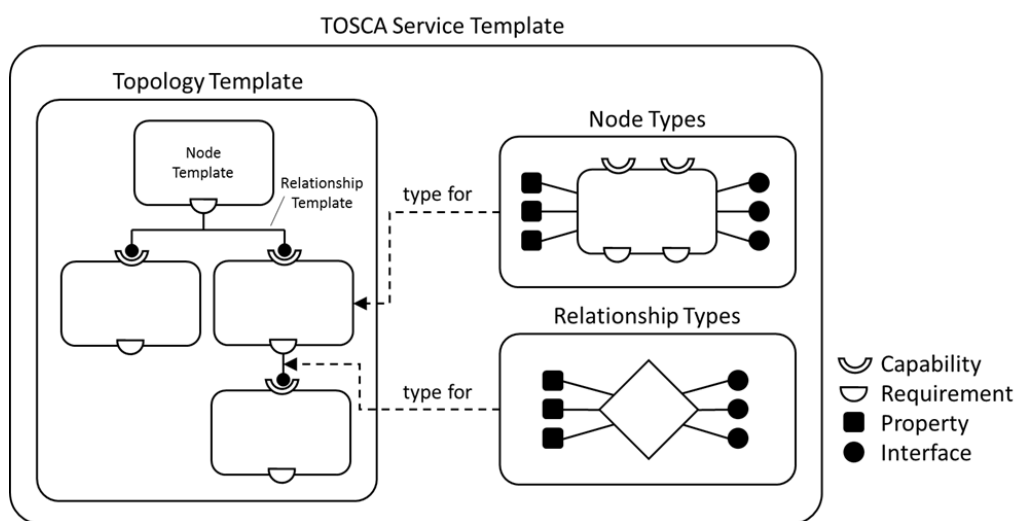


Figure 1 - TOSCA metamodel

The nodes of this topology in TOSCA terminology are called *node templates* whereas the edges are called *relationship templates*. Node templates and relationship templates are typed using *node types* and *relationship types* to define a particular semantics. For example, a node type could define a certain Ubuntu-based virtual machine exposing particular properties, i.e., to configure the required memory size. In contrast, a relationship type defines a certain type of dependency between two nodes, e.g., that a node is “hosted on” or “connects to” another node. Essentially, types define the semantics including sets of properties and attributes, which are then instantiated in concrete templates, e.g., node templates or relationship templates. Moreover, TOSCA introduces the concepts of *capabilities* and *requirements*, which, essentially, allow establishing the connections between nodes. A basic example is matching capabilities and requirements for HostedOn relationship, e.g., if a node representing MySQL DBMS needs to be hosted on the node that represents a database server, its requirement called *Container* has to be fulfilled by the self-titled capability in the database server node. TOSCA also provides the possibility to explicitly define a dependency between nodes, which helps prioritizing the deployment orchestration order in case the a specific deployment order is required. In addition, TOSCA uses policies, which are specified as

self-titled TOSCA constructs, as means to define non-functional requirements for the chosen part of the model.

Another important and relevant modeling concepts in TOSCA are *substitution* and *grouping*. The substitution allows combining different granularity levels in TOSCA templates, e.g., having more abstract nodes in the topology that have a composite structure and are described in separate service templates. The grouping feature in TOSCA allows defining groups in templates, e.g., for attaching policies to a group of nodes.

To represent an actual deployment logic, TOSCA allows defining *implementation artifacts* for node types and node templates. Depending on the orchestrator, the logic can be supplied using, e.g., Shell scripts or by using configuration management technologies such as Ansible. For the business logic, on the other hand, TOSCA allows specifying *deployment artifacts* that are instantiated by the orchestrator during deployment, e.g., a certain virtual machine image that shall be used to create a VM.

4. Research and Development Approach

In this section, we elaborate on the RADON modeling approach, which is based on the RADON type hierarchy defined using TOSCA CML. Prior to describing the introduced modeling approach, we briefly describe the assumptions and design issues that need to be taken into consideration.

4.1 Research Assumptions and Design Issues

To automate the deployment and provide means for verification and testing of fine-grained and loosely-coupled microservices and serverless applications, traditional deployment modeling approaches have to take several new requirements into consideration. Essentially, these challenges can be clustered into two categories related to functional and non-functional aspects.

Functional aspects are concerned with representing particular novel features and properties of new component types in the application such as microservices, serverless functions, e.g., event-driven nature of FaaS-hosted functions [Baldini2017], or how several components can be composed in a data shipping architecture, to represent a deterministic data flow within the application's model, i.e., adding behavioral information into the deployment model.

Non-functional aspects are concerned with non-functional application requirements, for example, how certain constraints can be defined within the application model, e.g., security or performance constraints, which facilitates constraint verification process. Moreover, continuous testing of such finer-grained application topologies is significantly harder to achieve. As a result, the models must also support specification of tests-related requirements.

4.2 Modeling Approach Overview

While TOSCA CML provides multiple normative types, which can be used for modeling cloud-native applications, due to its abstract nature and high flexibility it is not sufficient for tackling the

described design issues. To fulfill the requirements described in Section 2, the modeling approach employed by RADON builds on top of TOSCA and introduces a hierarchy of new RADON-specific types, both abstract and concrete, that represent specific technologies and are deployable. The former types constitute RADON Abstract Entities Modeling Layer (AEML), whereas the latter constitute RADON Deployable Entities Modeling Layer (DEML). Figure 2 presents a global view on the RADON type hierarchy combining the abstract and deployable types. Basically, mostly the leaf nodes in the figure represent concrete, deployable types, whereas the remaining nodes are abstract and serve as a basis for deployable types.

The majority of introduced types describe nodes in the application topology including traditional PaaS-hosted components such as nodes related to Kafka Streaming Platform, e.g., *Kafka Broker* or *Kafka Topic*, or FaaS-hosted functions and data pipeline components. Since TOSCA provides multiple generic normative types such as *tosca.nodes.SoftwareComponent* and *tosca.nodes.Root*, to avoid introducing redundancy, multiple types in the hierarchy inherit from suitable normative types, which results in relatively-large number of leaf nodes without any explicitly present abstract parent nodes.

For example, technology-specific nodes such as *Apache Nifi* and *Kafka Broker* originate from *tosca.nodes.SoftwareComponent* node type. In cases where there was a need to introduce additional abstract types, e.g., *Function* and *CloudPlatform* node types, *tosca.nodes.Root* type was used as a parent. A large segment of introduced node types focuses on modeling FaaS-hosted functions as well as serverless and cloud platforms. The decision to introduce multiple abstract and deployable types in the context of different providers and open source technologies is motivated by several factors that are explained in the following sections.

The next large segment in the hierarchy is dedicated to relationship types. This segment also consists of abstract and concrete types. Most of the concrete relationship types in the current state originate from an abstract *Triggers* relationship type, which is described in the following sections. Having technology-specific, concrete relationship types such as *PublishToKafkaTopic* facilitates specification of provider- and technology-specific properties. To support modeling with the newly-introduced function types, the special capability *Invocable* is introduced, its semantics is also described in the subsequent subsections. Finally, to specify RADON-specific non-functional requirements, a set of policies is introduced. In its current state, the hierarchy supports specifying scaling and performance-related requirements using the corresponding policy types.

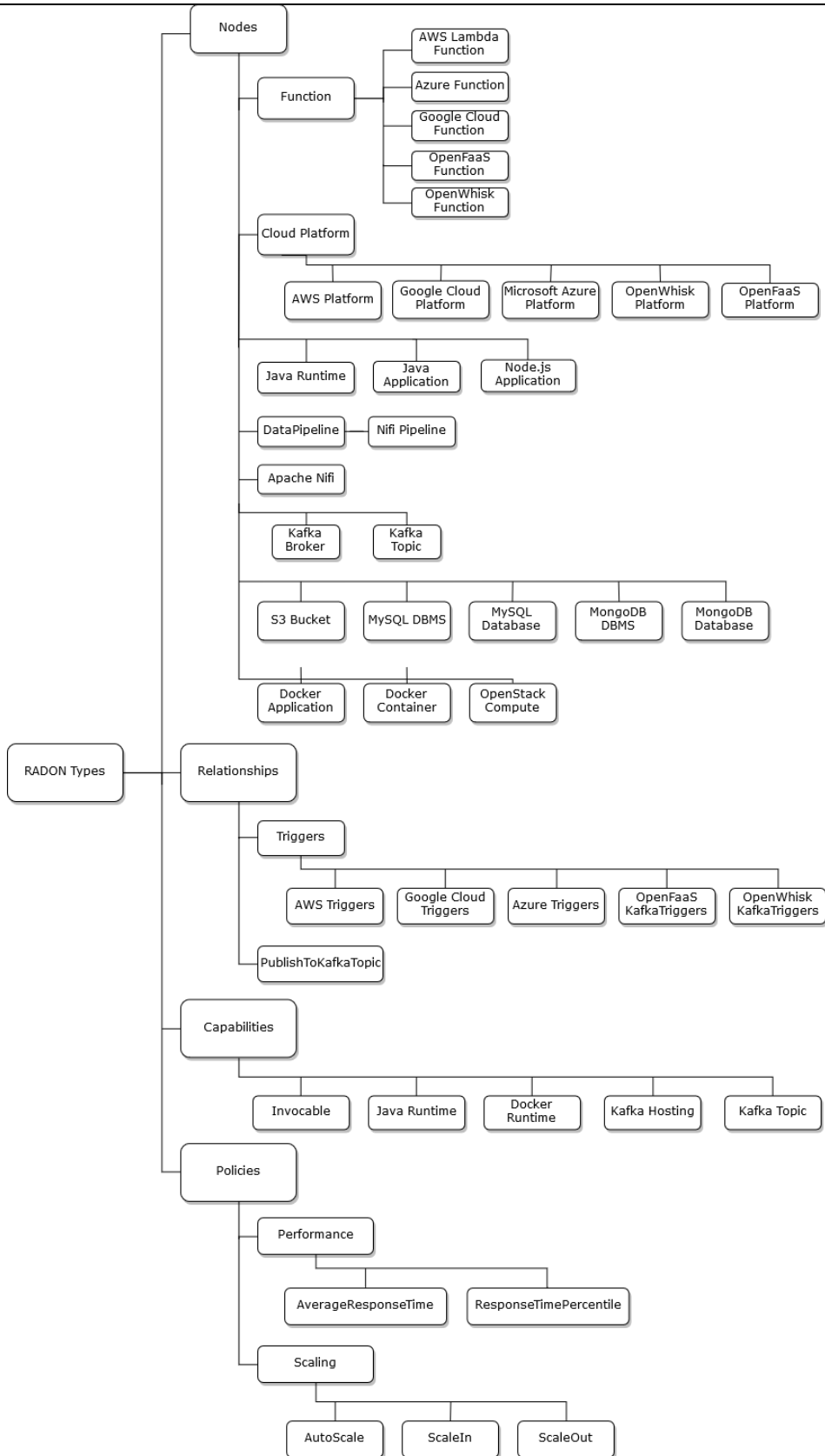


Figure 2 - RADON Types hierarchy

4.3 RADON Modeling Profile

This section provides the details on the RADON Modeling Profile together with examples of how the introduced modeling entities can be used in RADON models.

4.3.1 RADON Namespace

Firstly, to uniquely identify RADON types a separate namespace 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 a fixed keyword *radon* that separates all TOSCA types developed under the umbrella of RADON.
- Next part, i.e., *[entity-type]*, specifies a corresponding TOSCA entity type, e.g., nodes, relationships, or policies.
- 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. Table 1 demonstrates several examples of how the namespace is defined for some of the RADON types.

Table 1 - Examples of the RADON namespace usage

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

4.3.2 RADON Types Hierarchy

The main parent nodes in the hierarchy of introduced types are of TOSCA origin, i.e., nodes, relationships, policies. As discussed previously, Figure 2 demonstrates this hierarchy, with the largest amount of new types being node types. In the following we explain the modeling approach for representing the new component types using TOSCA.

 Modeling FaaS Functions

The important segment of RADON types deals with representation of serverless, FaaS-hosted functions [Jonas2019] in deployment models. To support modeling of FaaS-hosted functions, several crucial aspects have to be addressed. In the following, we elaborate on these modeling aspects.

A. Serverless and Cloud Platforms

To simplify modeling of provider-specific services, e.g., AWS Lambda FaaS platform or Microsoft Azure Object Storage service, RADON types hierarchy has a generic *CloudPlatform* node type in the AEML, which is used as a parent for such technology-specific nodes. Consequently, technology-specific node serve as a target for HostedOn relationship with respect to multiple heterogeneous node types. For example, AWS S3 bucket is modeled as hosted on AWS Cloud Platform without specifying the exact service to avoid model cluttering. This decision is motivated by the fact that provider-managed services do not require a detailed configuration process requiring multiple separate node types, thus it is possible to encapsulate cloud platform specific properties, such as authentication credentials or region settings using generalized node types. Figure 3 shows an example how to use a “AwsPlatform” node as a hosting component for a Lambda function as well as a S3 bucket. In this case, the “AwsPlatform” acts as a serverless and cloud platform and provides technology-specific semantics for AWS. The RADON profile also provides a serverless platform node to define the semantics for a reusable OpenFaaS node. Listing 1 shows an excerpt of such respective node template as an example.

```

tosca_definitions_version: tosca_simple_yaml_1_0
topology_template:
  node_templates:
    Platform:
      type: radon.nodes.openfaas.OpenFaaSPlatform
      properties:
        basic_auth_user: JohnDoe
        basic_auth_password: abc123#!a
        api_gateway_host: https://openfaas.host.me
  
```

Listing 1 - Example of an OpenFaaS serverless platform node

B. Function triggering semantics

The first important modeling challenge 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 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 changes depending on the function kind.

To separate these function types, RADON type hierarchy specifies *invocable* and *standalone* function types with respect to the corresponding FaaS platforms. The invocable function types requires modeling a relationship of a specific type called *Triggers* in RADON types hierarchy that connects the specific resource, i.e., event source, and the function itself. A similar approach was proposed by Wurster et al. [Wurster2018] as they recommend to use TOSCA's relationships to model the notion of events if a cloud resource triggers a cloud FaaS function. Moreover, this relationship must describe which event types trigger the function and provide a binding logic that links the event source with the function. A graphical representation of the invocable function that is triggered by the event emitted from AWS S3 bucket is depicted in Figure 3, whereas the actual listing describing this use case is shown in Listing 2.

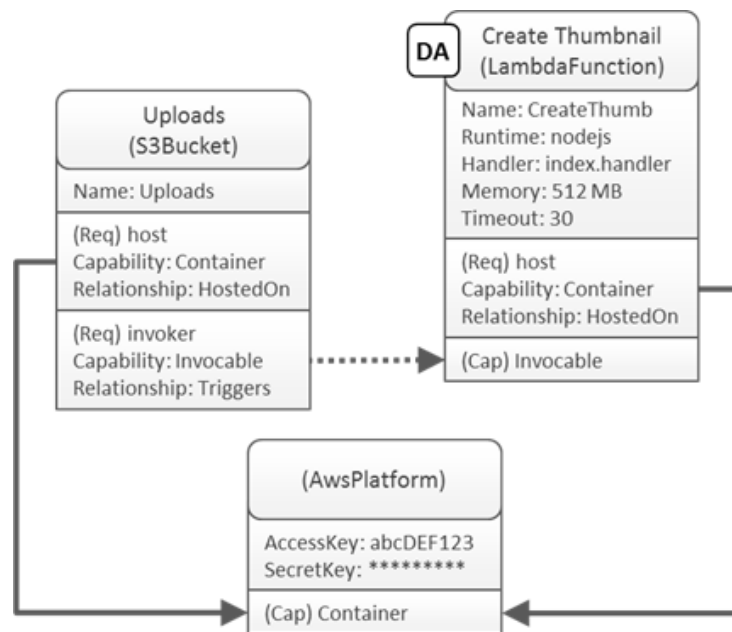


Figure 3 - Modeling an Invocable function for thumbnail generation use case; function code is attached as deployment artifact (DA)

```

tosca_definitions_version: tosca_simple_yaml_1_0
topology_template:
  node_templates:
    Platform:
      type: radon.nodes.aws.AwsPlatform
      properties:
        access_key_id: asdv5846qasd2134153311
  
```

```

secret_access_key: asddv54653724932asd165
region: eu-central-1
CreateThumbnail:
  type: radon.nodes.aws.LambdaFunction
  properties:
    name: CreateThumb
    role_name: CreateThumbRole
    runtime: nodejs
    handler: index.handler
    memory: 512
    timeout: 30
  artifacts:
    deployment_package:
      file: thumbnail.zip
      type: radon.artifacts.archive.Zip
  requirements:
    - host: Platform
Uploads:
  type: radon.nodes.aws.S3Bucket
  properties:
    name: Uploads
  requirements:
    - host: Platform
    - invoker:
      node: CreateThumbnail
      relationship: ResourceTrigger

```

Listing 2 - Modeling an Invocable function for thumbnail generation use case

Compared to invocable functions, standalone functions do not require a relationship, since all binding logic can be defined directly within the function. As an example, for scheduled functions, the only information that is required is the actual timeout and the configuration logic, i.e., implementation artifact in TOSCA terms, that sets up the appropriate trigger, i.e., a cron job, on the provider's side. A graphical model specifying a scheduled function deployed to Azure Functions platform is shown in Figure 4. Another example of a standalone function might be the endpoint exposed via API Gateway. While this type of functions can also be modeled as invocable functions, e.g., with a *Client* node serving as the triggering node, it can also be represented as a standalone node with all the required semantics and logic, e.g., an OpenAPI specification to configure the API Gateway, attached directly to the function. Moreover, providing the API Gateway Configuration for some FaaS platforms is not obligatory, e.g., OpenFaaS automatically exposes deployed functions

using the endpoints of a predefined format, which can also be reconfigured using the corresponding implementation artifacts attached directly to the function.

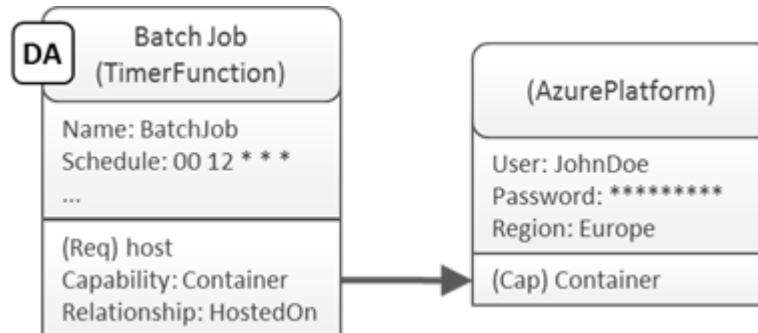


Figure 4 - Modeling a Standalone function for scheduled task use case

```

tosca_definitions_version: tosca_simple_yaml_1_0
topology_template:
  node_templates:
    Platform:
      type: radon.nodes.azure.AzurePlatform
      properties:
        user_name: JohnDoe
        password: asddv54653724932asd165
        region: europe
    BatchJob:
      type: radon.nodes.azure.TimerTriggeredAzureFunction
      properties:
        function_name: BatchJob
        timeout: 30
        schedule: 00 12 * * *
        app_name: BatchJob
        app_runtime: node
      artifacts:
        deployment_package:
          file: batchjob.zip
          type: radon.artifacts.archive.Zip
      requirements:
        - host: Platform
  
```

Listing 3 - Modeling a Standalone function for scheduled task use case

C. Event triggers

As described above, for invocable functions there is a dedicated relationship type called *Triggers*. The name of the relationship follows the TOSCA naming approach where the name represents an action, e.g., *HostedOn* or *ConnectsTo*. In this case, the relationship's name signifies that the event source *triggers* a function, based on one or more events. To achieve this, *Triggers* relationship type as one of its properties has a list of events. For example, if AWS Lambda function must be triggered by several AWS S3 events, these events can be specified in the corresponding *Triggers* relationship connecting the respective S3 bucket and Lambda function. Moreover, to establish an actual binding between the bucket and a function, the implementation artifact needs to be attached to this relationship type. This is required since both the function and the bucket are initially deployed independently of each other, thus, requiring to establish a binding between them.

In RADON, we introduce an abstract *Triggers* relationship type that is prepared to be used together with any kind of *Invocable* capability. However, as each cloud provider or serverless platform provides different interfaces to configure the event binding, we further introduce specific relationship types for them.

For example, with OpenFaaS there is a special *KafkaTriggers* relationship type that is used to establish a corresponding event binding between a Kafka topic and a function hosted on OpenFaaS. In addition, there is a AWS-specific *Triggers* relationship type having the semantics and the deployment logic attached to setup and establish several resource-triggered relationships between AWS services and Lambda functions.

An example of resource-triggered relationship modeling is shown in Figure 5, which depicts a detailed excerpt of Figure 3. In this example, the model describes how the corresponding Lambda function is triggered whenever a new file is uploaded or changed inside the modeled S3 bucket. For describing different events, there is a special event datatype in RADON type repository. Currently, this data type's structure is based on the CloudEvents specification, however, to simplify the modeling process most of the properties are made optional. This allows specifying only particular event names, e.g., AWS S3 Put event with the corresponding AWS event name string as it is depicted in Figure 5. Lastly, Listing 4 shows how such a model translates to TOSCA YAML showing how RADON's event data type is used.

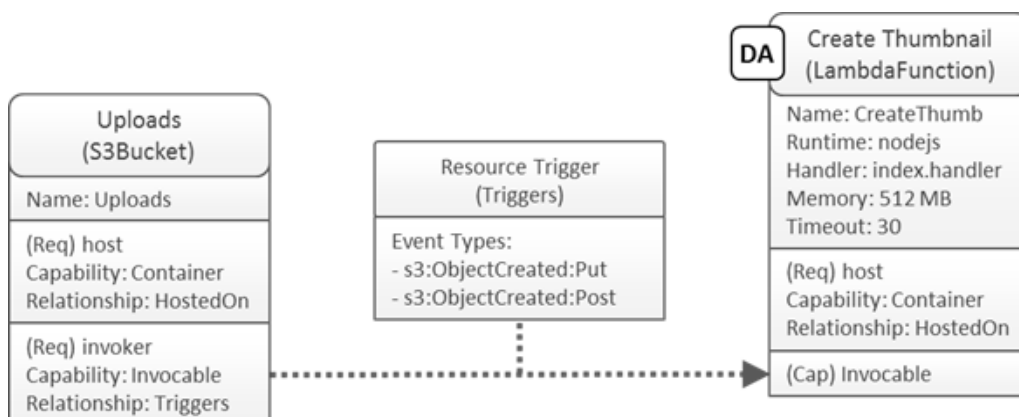


 Figure 5 - Modeling of a “Triggers” relationship using the toy example use case

```

...
relationship_templates:
  ResourceTrigger:
    type: radon.relationships.aws.Triggers
    properties:
      event_types:
        - putEvent:
            type: radon.datatypes.Event
            properties:
              type: s3:ObjectCreated:Put
        - postEvent:
            type: radon.datatypes.Event
            properties:
              type: s3:ObjectCreated:Post
  
```

Listing 4 - Modeling of a “Triggers” relationship using the toy example use case

D. Abstracting FaaS providers

The function-related node types are represented by both abstract and deployable modeling layers, i.e., AEML and DEML. All function node types originate from a generic *Function* type, which represents a FaaS-hosted function in a provider-agnostic fashion. For example, the already mentioned toy example use case could be modeled using abstract types as shown in Figure 6.

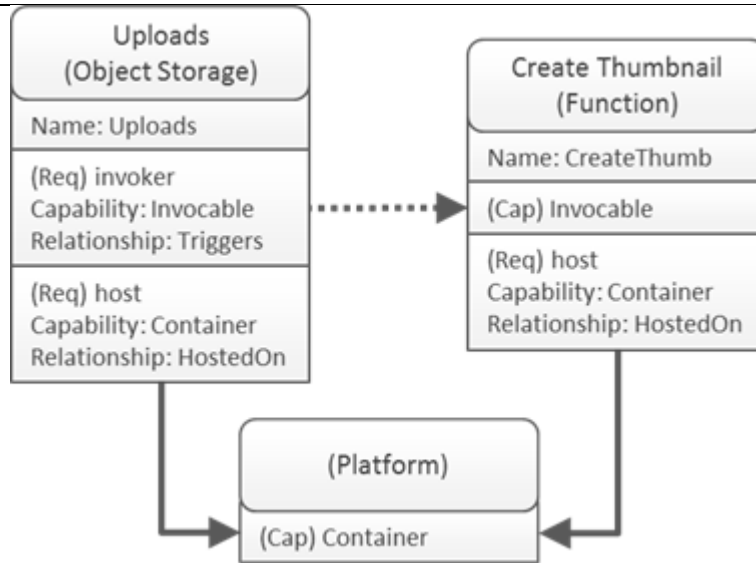


Figure 6 - Abstract modeling of the toy example use case

Further, abstract provider-specific function types comprise different sets of properties, but are not yet deployable due to the aforementioned difference between the invocable and standalone function types. As a result, every provider has their invocable and standalone function types that originate from the corresponding provider-specific abstract type shown in Table 2.

Table 2 - Abstract function types in RADON type hierarchy belonging to AEML

Name	Namespace	Description
Function	radon.nodes.abstract.Function	A generic, provider-agnostic function node type which serves as a basis for provider-specific function nodes.
AWS Lambda Function	radon.nodes.aws.LambdaFunction	An abstract AWS Lambda function node type that serves as a basis for AWS Invocable and AWS Standalone function types.
Azure Function	radon.nodes.azure.Function	An abstract Microsoft Azure function node type
Google Cloud Function	radon.nodes.google.CloudFunction	An abstract Google Cloud Function function node type
OpenWhisk Function	radon.nodes.openwhisk.Function	An abstract OpenWhisk function node type
OpenFaaS Function	radon.nodes.openfaas.Function	An abstract OpenFaaS function

		node type
--	--	-----------

Data Pipelines Modeling

For the first version of the RADON models, the consortium agreed to implement data processing pipelines using Apache NiFi as technology. To model an application containing Apache NiFi processing elements, two TOSCA node types are required. On the one hand, a node type is required that represents Apache NiFi as a middleware component. This node type is capable of installing and starting the respective software components and can only be hosted on traditional compute infrastructure, such as an Ubuntu operating system. Further, a node type is required that represents the actual Apache NiFi processing pipeline.

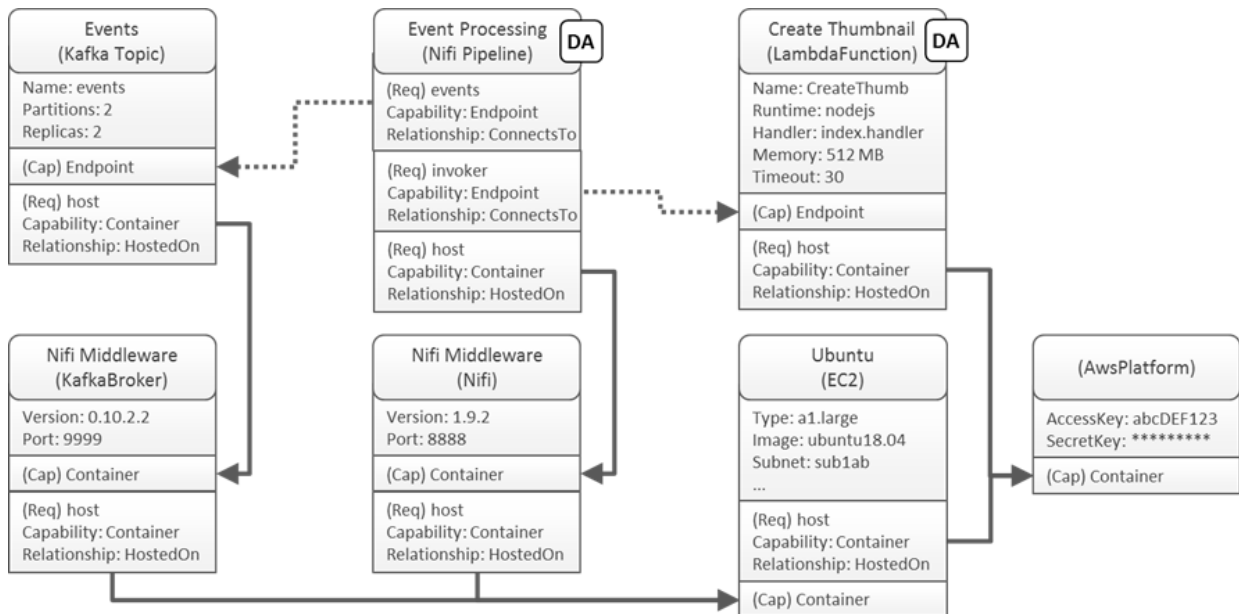


Figure 7 - Modeling example of a data pipeline processing Kafka events and calling a Lambda backend function

Figure 7 shows a modeling example of an Apache NiFi data pipeline that processes arbitrary events coming from a Kafka topic. In this example, the NiFi pipelines *connects to* the Kafka topic and listens for events. After processing events, i.e., filtering or transforming data, the pipeline invokes a Lambda function hosted on AWS for further processing (cf. relation between “Event Processing” and “Create Thumbnail” nodes in Figure 5). Further, the NiFi pipeline and the Kafka topic are hosted on respective software middleware components: on a “Apache NiFi” node template and “Kafka Broker” node template respectively. Listing 5 shows an excerpt of how the connection between the Kafka topic, the NiFi pipeline, and the Lambda function can be expressed in TOSCA. The advantage of this modeling approach is that the data pipeline can be specified using Apache NiFi’s native XML specification syntax, which, in turn, can be attached as a TOSCA deployment artifact to the pipeline

node type. On top of that, Apache NiFi pipelines can be connected using TOSCA's *connectsTo* relationship. Application developers have to supply the respective implementations in its blueprint to establish the physical connection between two pipelines.

```

tosca_definitions_version: tosca_simple_yaml_1_0
topology_template:
  node_templates:
    EventProcessing:
      type: radon.nodes.apache.nifi.NifiPipeline
      artifacts:
        pipeline_template:
          file: pipeline.zip
          type: radon.artifacts.archive.Zip
      requirements:
        - host: Nifi
        - events:
            node: Events
            capability: endpoint
            relationship: tosca.relationships.ConnectsTo
        - invoker:
            node: CreateThumb # omitted for brevity
            capability: endpoint
            relationship: tosca.relationships.ConnectsTo
    Nifi:
      type: radon.nodes.apache.nifi.NifiPipeline
      properties:
        component_version: 1.9.2
        port: 88888
      requirements:
        - host: Compute # omitted for brevity
    Events:
      type: radon.nodes.apache.kafka.KafkaTopic
      properties:
        topic_name: events
        partitions: 2
        replicas: 2
      requirements:
        - host: Broker # omitted for brevity
  
```

Listing 5 - Modeling example of a data pipeline processing Kafka events and calling a Lambda backend function

One disadvantage of this modeling approach is that the actual pipeline is hidden from the modeler, which limits the possibilities to model non-functional requirements and constraints for testing, defect prediction, and decomposition. However, the granularity of modeled pipeline processing nodes can be adjusted by modelers, i.e., nodes will represent separate pipeline actions instead of groups of actions, providing more flexibility for specifying NFRs. Resulting pipeline models are flexible enough to fulfill the given modeling requirements, which makes this modeling approach suitable for employing in RADON.

Microservices Modeling

Microservice architectures [Newman2015] typically consist of multiple units. In TOSCA, a single microservice can be represented as a *node template* or as a *service template*. For example, a container-based microservice can be represented by using a node template of type *DockerContainer* inside the RADON template library. Further, a microservice itself can consist of multiple components such as a database or block storage to store its state. In such a case, a microservice can be modeled based on a service template that, in turn, contains the detailed structure of it, e.g., several serverless FaaS functions hosted on a serverless platform using a database to store its state. Hence, service templates are TOSCA's native construct to group the components of a microservice.

Using this, a service template containing multiple small serverless functions represents at the bigger picture a single microservice. By employing TOSCA's substitution feature, one can orchestrate such independent service templates by referencing them in a separate service template using substitutable nodes based on abstract node types. Figure 8 shows an example of how microservice applications can be developed using RADON's approach. On the left hand side of the figure, a simple service template is shown containing a serverless FaaS function hosted on AWS. This function is using a "User" microservice to query and modify the data provided by this service.

However, this node template is of a special substitutable type *Microservice*. This node template points to another service template containing the details of the respective service. In this case, and as shown on the right hand side in Figure 8, this service consists of multiple serverless FaaS functions hosted on Azure. By using RADON's envisioned template substitution mechanisms, a user is able to develop single microservice-based service templates which can be later on referenced and orchestrated by the mentioned constructs.

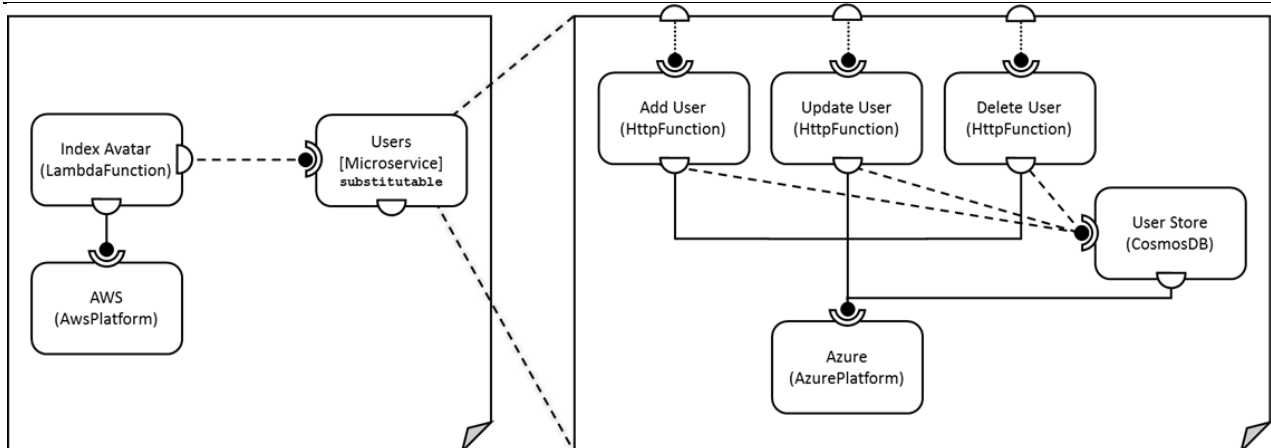


Figure 8 - Using RADON's template substitution to implement microservice applications

Non-Functional Requirements Modeling

A typical way to specify non-functional requirements in TOSCA is to use policies. This entity type can be attached to various constructs and describe arbitrary information related to the chosen target entity. To support attachment of various NFR relevant in the context of RADON, the type hierarchy comprises a list of dedicated policy types.

As an initial set, RADON offers policies for specification of scaling behavior. Further, RADON provides policies to express certain performance-related requirements. Finally, a first set of functional testing policies are provided used to specify model-based test cases that can be executed by the tools provided in RADON.

As an example, Figure 9 shows how an auto-scaling policy can be attached to a group of nodes. In this case, the model defines that the whole stack must auto-scale between 1 and 5 instances. On top of that, Listing 6 shows exemplary how such a model is translated into TOSCA syntax.

Further, in Figure 10 we show how a performance requirement policy can be attached to a serverless FaaS function. Essentially, RADON will provide tools to interpret, process, and enforce such policies. These types will be part of the envisioned template library and a user can extend them with additional properties if required.

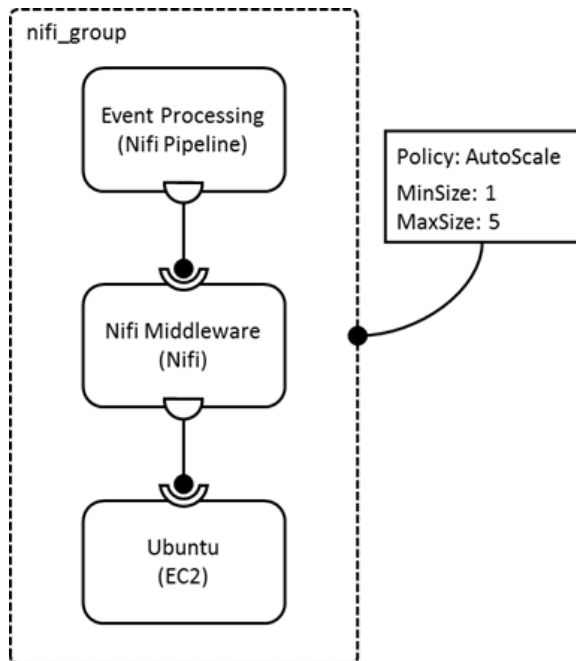


Figure 9 - Auto-scaling policy for a group of nodes

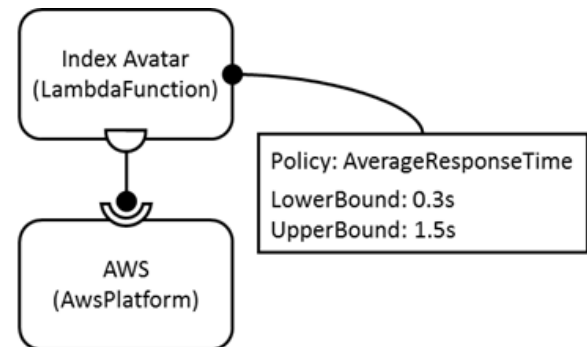


Figure 10 - Performance requirement policy

```

tosca_definitions_version: tosca_simple_yaml_1_0
topology_template:
  groups:
    nifi_group:
      type: tosca.groups.Root
      members: [ EventProcessing, Nifi, Ubuntu ]
  policies:
    - nifi_autoscaling_policy:
      type: radon.policies.scaling.AutoScale
      targets: [ nifi_group ]
      properties:
        min_size: 1
        max_size: 5
  
```

Listing 6 - Auto-scaling policy for a group of nodes

4.4 RADON Template Library

The RADON Template Library is a repository containing TOSCA blueprints, reusable definitions and extensions to deploy and manage RADON applications. The template library provides reusable TOSCA types of application runtimes, computing resources, and FaaS platforms in the form of abstract (AEML) as well as deployable modeling entities (DEML). The repository also comprises RADON's FaaS abstraction layer that provides TOSCA definitions to deploy particular FaaS application component to AWS, Azure, and Google as well as to on-premise serverless platforms such as OpenFaaS and OpenWhisk.

A. RADON Particles

The template library is a central element in RADON's envisioned development workflow. It is mainly used by the Graphical Modeling Tool (GMT) as its central repository of reusable types and existing deployable blueprints, but it is also shared and used by other RADON toolchain tools such as the IDE. The template library serves three main scenarios:

- (1) the GMT uses the template library to understand which blueprints and types exist and could be reused for a specific application deployment,
- (2) a user of the GMT utilizes the template library to create a new application blueprint based on the existing types inside the repository,
- (3) the template library provides the necessary type definitions that can be exported into a CSAR required by the orchestrator for the deployment. On top of the GMT, the template library is also used by other RADON tools.

For example, the respective application blueprints are inputs for the Defect Prediction Tool to highlight code smells or bad practices introduced during application modeling. Further, RADON's constraint definition language (CDL) uses the blueprints as an input in order to check if all described hard constraints are satisfied by the modeled application.

In general, the template library is a file-based repository and is managed by a version control system. The RADON consortium decided to use GitHub to maintain the template library. Therefore, we created a publicly available repository:

<https://github.com/radon-h2020/radon-particles>

The repository is called "RADON Particles" and is a cryptonym for the template library as it contains small, reusable entities to model modern, serverless, and data-driven applications.

The repository's structure is organized by the main TOSCA elements on the root level, such as there is a directory name "nodetypes" containing all TOSCA node types relevant for RADON. Below the root level, the RADON namespace of an entity is used for structuring. We apply a similar file system layout like the directory structure for Java packages. Considering the RADON namespace from above, "radon", "[entity-type]", "[purpose-identifier]", and "[entity]" become directories and form the repository's structure. RADON's conventions is that below each "[entity]" directory is a

“definitions.yaml” and “README.md” file. The following listing shows an excerpt of the file structure:

```

radon-particles
|-artifacttypes
|-capabilitytypes
| |-radon
| | |-capabilities
| | | |-Invocable
| | | | |-...
|-datatypes
|-docs
|-nodetypes
| |-radon
| | |-nodes
| | | |-abstract
| | | | |-...
| | | |-apache
| | | |-aws
| | | | |-AwsPlatform
| | | | |-LambdaFunction
| | | | |-S3Bucket
| | | |-google
| | | | |-...
|-policytypes
|-relationshiptypes
| |-radon
| | |-relationships
| | | |-abstract
| | | | |-Triggers
| | | | |-...

```

Listing 7 - RADON Template Library structure

The “definitions.yaml” file contains the actual TOSCA syntax describing the respective type. The “README.md” file, on the other hand, contains a user-friendly documentation and highlights the most important facts for each corresponding type. Furthermore, to make the repository self-contained, TOSCA implementation artifacts that belong to a node or relationship type are maintained in subdirectories of the respective type. With all implementation artifacts in place, service templates representing desired applications can be automatically deployed using the chosen, TOSCA-compliant orchestrator.

B. RADON Contribution Model

To control the contributions to the shared template library, the RADON consortium agreed to apply a Feature-Branch-Workflow model to update the blueprints and type definitions inside the repository.

The template library is RADON’s central repository and, therefore, we assume the `master` to represent the official project history and latest development state. Whenever a TOSCA entity needs to be adapted or added, the respective project partner creates a so-called “feature branch” (using a descriptive name) based on the current `master`. This feature branch is used to commit and push the required changes or additions. By regularly pushing the feature branch to the central repository, teammates or other project partners can comment or collaborate on this feature. To merge the changes into the repository’s `master` branch, a pull request has to be created. This gives other project partners the opportunity to review the changes before they become a part of the main codebase. Once UST (responsible for WP4) approved the PR, it is merged into the `master`.

5. Conclusions

In this document, we described the current version of RADON Models and the details on the corresponding RADON template repository that supports them. Further, the definitions of these reusable types are specified in the respective **companion** document covering the reference application level technologies agreed upon the consortium.

Table 3 shows an overview of the level of fulfillment for each of the agreed requirements. The labels specifying the “Level of fulfillment” are defined as follows:

- (i) **X** (unsupported): the requirement is not fulfilled by the current version
- (ii) **✓** (partially-low supported): a few of the aspects of the requirement is fulfilled by the current version
- (iii) **✓✓** (partially-high supported): most of the aspects of the requirement is fulfilled by the current version
- (iv) **✓✓✓** (fully supported): the requirement is fulfilled by the current version). Afterwards, we discuss for each requirement briefly how it has been addressed by publishing this deliverable

Table 3 - Overview of requirement compliance level

Id	Requirement Title	Priority	Level of fulfillment
R-T4.2-1	Deployment types heterogeneity	MUST_HAVE	✓✓
R-T4.2-2	Reusable types and blueprints	SHOULD_HAVE	✓✓
R-T4.2-3	Data processing modeling	MUST_HAVE	✓✓
R-T4.2-4	Preconditions for data processing	SHOULD_HAVE	X

R-T4.2-5	Scaling of computing resources	SHOULD_HAVE	✓✓✓
R-T4.2-6	Data processing compression	COULD_HAVE	✗
R-T4.2-7	Test case specification	MUST_HAVE	✓

Future work

In the next iteration of the deliverable *RADON Models - final version* at month 18, we will further detail the type hierarchy to address the appearing use cases requirements. Generally, the requirements related to specification of NFRs will need to be revised and refined based on the arising requirements collected during the iterative feedback sessions with the use case providers. The usage of introduced modeling constructs in practical settings defined by the use cases will help to highlight the shortcomings and overlooked areas. Moreover, new requirements can arise and be collected from the feedback loops with the tool owners. For example, the constraints definition language and its relation to RADON modeling constructs will be further enhanced with more details. One possible direction would be defining CDL-specific TOSCA data types that will facilitate representing CDL statements and using these introduced types for specification of properties in CDL-specific policy types. In a similar fashion, specification of test annotations and other NFRs will be supported by more detailed RADON modeling entities.

Moreover, with respect to the requirements described in Section 2, we still need to address the following issues:

- **Requirement R-T4.2-1:** the type hierarchy currently does not address the challenge of modeling cross-cloud FaaS use cases, where sources and targets do not belong to the same provider. In addition, the node and relationship types might become finer-grained in case required by the use cases.
- **Requirement R-T4.2-2:** the template repository structure will be further refined and revised based on the fulfillment of remaining requirements.
- **Requirement R-T4.2-3:** depending on the desired granularity of processing nodes, the data processing modeling will be revised and refined with additional entities if needed by arising use case requirements.
- **Requirement R-T4.2-4:** this requirement has to be addressed
- **Requirement R-T4.2-6:** this requirement has to be addressed
- **Requirement R-T4.2-7:** current test annotations will be enhanced with additional details according to the arising requirements from the tool owners

References

- [Binz2012] Binz, T.; Breiter, G.; Leymann, F. & Spatzier, T.: “Portable Cloud Services Using TOSCA”, IEEE Internet Computing, IEEE, 2012, 16, 80-85
- [OASIS2019] OASIS, TOSCA Simple Profile in YAML Version 1.2, 2019
- [Lipton2018] Lipton, P.; Palma, D.; Rutkowski, M. & Tamburri, D. A.: “TOSCA Solves Big Problems in the Cloud and Beyond!”, IEEE Cloud Computing, 2018
- [Wurster2018] Wurster, M.; Breitenbücher, U.; Képes, K.; Leymann, F. & Yussupov, V.: “Modeling and Automated Deployment of Serverless Applications using TOSCA”, Proceedings of the IEEE 11th International Conference on Service-Oriented Computing and Applications (SOCA), IEEE Computer Society, 2018, 73-80
- [Casale2019] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza A. Russo, S.N. Srirama, D.A. Tamburri, M. Wurster, L. Zhu, “Rational Decomposition and Orchestration for Serverless Computing”, The Symposium and Summer School on Service-Oriented Computing (SummerSoc), 2019, (accepted for publication)
- [Jonas2019] Jonas, E.; Schleier-Smith, J.; Sreekanti, V.; Tsai, C.-C.; Khandelwal, A.; Pu, Q.; Shankar, V.; Carreira, J. M.; Krauth, K.; Yadwadkar, N.; Gonzalez, J.; Popa, R. A.; Stoica, I. & Patterson, D. A.: “Cloud Programming Simplified: A Berkeley View on Serverless Computing”, Electrical Engineering and Computer Sciences, University of California at Berkeley, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2019
- [Baldini2017] Baldini I. et al. (2017) Serverless Computing: Current Trends and Open Problems. In: Chaudhary S., Somani G., Buyya R. (eds) Research Advances in Cloud Computing. Springer, Singapore
- [Newman2015] Newman, S.: “Building Microservices: Designing Fine-Grained Systems” O'Reilly, 2015
- [Bergmayr2018] Bergmayr, A.; Breitenbücher, U.; Ferry, N.; Rossini, A.; Solberg, A.; Wimmer, M. & Kappel, G.: “A Systematic Review of Cloud Modeling Languages”, ACM Computing Surveys (CSUR), ACM, 2018, 51, 1-38