H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable D4.4

RADON Models II

Version: 1.0

Publication Date: 26-June-2020

Deliverable Card

Deliverable	D4.4
Title:	RADON Models II
Editor(s):	Michael Wurster (UST) and Vladimir Yusupov (UST)
Contributor(s):	Michael Wurster (UST), Vladimir Yusupov (UST), Matija Cankar (XLB), Lulai Zhu (IMP), André van Hoorn (UST)
Reviewers:	Pelle Jakovits (UTR), Matija Cankar (XLB)
Type:	Report
Version:	1.0
Date:	26-June-2020
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 the second iteration of the RADON modeling profile that focuses on representation of serverless application topologies and data flows using a set of abstract and concrete modeling constructs defined using the TOSCA cloud modeling language.

The goal of this deliverable is twofold. Firstly, the document introduces a set of extensions to the modeling profile presented in deliverable D4.3 RADON Models I as means to address the requirements arising from the RADON use case providers and tool owners. Secondly, this deliverable elaborates on the aspects related to usage, extension, and maintenance of the modeling profile and provides the guidelines facilitating the usage of the RADON modeling profile.

The results presented in this document have been achieved as a part of the task T4.2. The modeling profile is publicly available online in the RADON Particles repository, which provides a comprehensive set of TOSCA-based modeling constructs supporting RADON's requirements.

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
TOSCA	Topology and Orchestration Specification for Cloud Applications
VCS	Version Control System
VM	Virtual Machine
TPS	Template Publishing Service

Table of contents

1. Introduction

An important part of the RADON project [Casale2019] and the underlying methodology is a standard-based modeling approach allowing designing microservice-based and serverless applications, as well as data flows using a set of reusable modeling constructs to represent the chosen component types, e.g., FaaS-hosted functions, cloud data stores. This document presents the second iteration of RADON modeling approach which focuses more on the aspects of profile's extensibility and maintainability and can serve as a guideline for employing the modeling profile.

One of the main benefits of the RADON Modeling Profile is the unified, DevOps-optimized modeling experience which it brings. With RADON Models, end users can easily represent technology-agnostic and technology-specific models of serverless and microservice-based architectures as well as data flows together with various non-functional requirements. The resulting application models can be automatically deployed, continuously tested, and analyzed using multiple tools of RADON methodology. Due to the flexibility of the TOSCA standard, the modeling profile can also leverage existing industrial deployment automation technologies for implementing reusable modeling constructs, which makes the modeling profile easily extensible. In the following, we elaborate more on the modeling profile's characteristics and highlight the benefits of using it as a part of the RADON Methodology.

1.1 Deliverable Objectives

This document presents the revised and extended view on the RADON Modeling Profile introduced in the D4.3 "RADON Models I" which enables modeling of serverless and microservice-based architectures, and data flows that can be processed by TOSCA-compliant orchestrators and other tools described in the RADON Methodology.

This deliverable reiterates on specifics of modeling using TOSCA [OASIS2020, Lipton2018] employed in RADON modeling profile and discusses the aspects of extensibility, maintainability and reusability of the introduced modeling constructs. The presented deliverable has three main objectives:

1. Demonstrate how arising requirements can be addressed by the modeling profile and highlight the required modifications to the modeling profile.
2. Elaborate on the aspects of persistence and reuse of RADON Models.
3. Provide the usage and extensibility guidelines and highlight the benefits of the RADON Modeling Profile with respect to other modeling options.

1.2 Overview of Main Achievements

This deliverable makes several contributions:

- Categorization of arising requirements targeting the modeling profile and definition of actions needed for addressing these requirements, including:
  - Serverless and general-purpose modeling challenges;
  - Data flow modeling challenges;
  - DevOps challenges.
- Detailed discussion of different persistence options for the RADON Modeling Profile.
- Presentation of different usage scenarios and target consumers of the RADON Modeling Profile together with the use case-specific model consumption details.
- General extensibility guidelines with respect to requirements related to TOSCA and target applications consuming RADON Models.

1.3 Structure of the Document

The remainder of the document is structured as follows: Section 2 reiterates on the requirements related to the RADON Modeling Profile. Section 3 elaborates on how arising requirements can be addressed by the profile and which kinds of modifications might be needed. In Section 4 we discuss the persistence options for RADON Models, and Section 5 focuses on the aspects of usage and extensibility of the profile. Finally, Section 6 concludes the deliverable and elaborates on the fulfillment of requirements.

2. Requirements

The initial version of RADON requirements was provided in the deliverable D2.1 “Initial Requirements and baselines”, whereas the deliverable D2.2 “Final Requirements” finalizes RADON requirements according to the emerging challenges arising over the course of the project. This section outlines the final RADON requirements relevant for the RADON Modeling Profile.

ID	R-T4.2-1
Title	Deployment types heterogeneity
Priority	Must have
Description	RADON models 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).

ID	R-T4.2-8
Title	Model API Gateway resources in AWS
Priority	Must have
Description	A RADON Model should contain an API Gateway resource with respective configuration to trigger a FaaS function hosted in AWS Lambda.

3. Addressing Requirements in RADON Modeling Profile

In this section, we elaborate on which types of modeling challenges and requirements have arisen from RADON use case providers and tool owners, and how the evolved version of the modeling approach initially introduced in D4.3 "RADON Models I" supports addressing them. In the following sections, we discuss general requirements related to modeling of serverless and FaaS-based application topologies, data flows, as well as RADON-specific tool requirements such as enabling support for continuous testing in the models.

3.1 Serverless and General-purpose Modeling Requirements

As demonstrated in the first iteration of the deliverable, serverless and FaaS-based architectures [Jonas2019], as well as traditional components, require a clearly-defined type system for modeling provider-specific component types that represent functions and event sources. Additionally, to connect these components in event-driven fashion, RADON models must provide a clear type system for modeling such trigger semantics. To achieve this, RADON Modeling Profile introduced a hierarchy of TOSCA Node Types representing platform-specific functions and event source types such as object storage, and a hierarchy or Relationship Types that represent event-driven function invocations. All type systems provide both abstract and technology-specific types that can be used in modeled Service Templates. As discussed in the first iteration of the deliverable, to model executable applications, RADON Modeling Profile employs the usage of concrete types, whereas abstract types can be used for model validation and refinement purposes. In the following subsections, we focus on specific requirements that arise or might arise in the future, and elaborate how the introduced modeling profile can address them and which adjustments might be needed starting with the most obvious one: how new component types can be introduced into the modeling profile.

3.1.1 Introducing New and Enriching Existing Modeling Constructs

The first deliverable introduced a comprehensive set of types covering multiple kinds of event sources such as file storage, relational and non-relational databases, or message-oriented middleware. Unsurprisingly, the first kind of arising requirements is related to introducing new modeling entities and extending existing ones, e.g., when a new FaaS-enabled cloud service is introduced or an existing event source's configuration requirements are changed by cloud providers. Due to a large amount of possible event source categories and the fast development pace of public cloud offerings and open source FaaS platforms [Yussupov2019], the need to introduce new types or modify existing modeling constructs is expected to be a regular part of the modeling profile maintenance.

The need to constantly adapt to changes in the cloud landscape imposes requirements not only on the underlying cloud modeling language, but also on the ways executable deployment logic is

implemented. Therefore, it is important to consider the following two aspects: (i) extensibility of the type hierarchy and the underlying modeling language, and (ii) the extensibility of the deployment logic implementation. The former aspect is directly related to the RADON Modeling Profile and in the following paragraphs we provide examples of modifications to the modeling profile that were performed to address such kinds of emerging requirements. Conversely, the latter aspect is rather related to the capabilities provided by the deployment orchestrator. We will discuss the deployability aspects of RADON models in the Subsection 3.1.2.

As RADON Modeling Profile uses TOSCA CML for representing application topologies, introducing new modeling entities basically requires creating new TOSCA constructs such as Node, Relationship, or Policy Types in a chosen namespace, which represent a desired application component, behavior, or a non-functional requirement. Similarly, the task of extending existing modeling entities is related to modifying property and attribute sets, defining new operations, etc.

An important point here is also to follow the modeling profile's type system. For example, when introducing a component representing a new object storage service, the respective type must inherit from *radon.nodes.abstract.ObjectStorage* Node Type to ensure the compatibility with RADON tools that rely on the type system. Moreover, since all types in RADON are derived from normative TOSCA types, inheriting from RADON types does not actually result in additional requirements for introduced constructs. For example, while RADON uses the custom Relationship Type *radon.realtionships.abstract.Triggers* to model the event-driven invocation of FaaS-hosted functions, it is derived from the normative DependsOn relationship type. This allows TOSCA orchestrators to understand the fact that an event source depends on a function and the event binding can only be established when the function is present, i.e., already deployed. Hence, inheriting from RADON types allows relying on the semantics of normative types.

As discussed in the first iteration of the deliverable, for representing serverless component types, i.e., functions and event sources, RADON Modeling Profile's type system organizes entities following the *provider.component-type* ordering in the namespace. For instance, nodes.aws.AwsLambdaFunction and nodes.aws.AwsS3Bucket represent AWS-specific component types for deploying functions on AWS Lambda and object storage instances on AWS S3. Typically, every serverless-specific modeling construct also has an abstract parent, e.g., nodes.abstract.Function and nodes.abstract.ObjectStorage respectively. While in its current state, the RADON Particles[1] repository already covers multiple types of event-sources, it might be possible that certain event sources are not represented in the namespace, e.g., API Gateway services were not covered in the first iteration of the deliverable. In the following subsections, we demonstrate which changes to the RADON Modeling Profile (type system and entities) were needed to add support for modeling components representing provider-specific API Gateway services. Additionally, we provide an example of enriching an existing AWS Lambda Function

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

Node Type to support the deployment-specific requirement stemming from the RADON Decomposition tool that required adding new properties and attributes to the type.

Introducing New Modeling Constructs: API Gateway example

One common event binding 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 modeling profile requires having a new 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. Similar to other abstract node types, the generic ApiGateway entity is stored using the same namespace schema: *radon.nodes.abstract.**. The TOSCA model shown in Listing 1 defines an abstract API Gateway node type, which can be hosted on an abstract Cloud Platform and can trigger an abstract Function using the dedicated abstract Relationship Type.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.ApiGateway:
    derived_from: tosca.nodes.Root
    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.abstract.Triggers
          occurrences: [ 0, UNBOUNDED ]
```

Listing 1 - An excerpt from the abstract Node Type for modeling API Gateways.

After extending the namespace with a dedicated category for the required event sources and introducing an abstract type, one or more concrete Node Types representing provider-specific API Gateways can be introduced. In this particular example, we discuss the details and design decision for the *radon.nodes.aws.ApiGateway* Node Type.

Essentially, to support deploying applications that rely on the API Gateway, the designed Node Type must reflect the properties of the underlying provider-specific technology. For example, such information as AWS region and Amazon Resource Name (ARN) of a role allowing to trigger functions must be defined on the type level and instantiated on the level of Node Templates.

Moreover, to avoid having redundancy, e.g., duplicate but conflicting property values, the designed Node Type must leverage the type system and TOSCA features to reuse properties from other Node Types involved in the topology. For example, since all AWS-specific nodes are modeled as hosted on radon.nodes.aws.AwsPlatform Node Type the region value can be set only in the platform node and then reused by every other component that is hosted on this platform. As a result, the set of properties in the introduced type can be simplified. Listing 2 shows an excerpt of the TOSCA definition for the Node Type radon.nodes.aws.ApiGateway. Apart from properties, it also contains an overlapping set of attributes which are used for storing the actual property values after templates are instantiated. For instance, the value of the *role_name* property is stored in the attribute when the AWS role is created and this actual value can be used by other nodes of the application. Additionally, two TOSCA requirements are defined for this Node Type, namely *host* and *invoker*. The former allows attaching this Node Type to the *AwsPlatform* Node Type using *HostedOn* Relationship Type. The latter allows modeling the function triggering semantics using *AwsTriggers* Relationship Type. As discussed in the first iteration of the deliverable, having a matching set of requirements and capabilities is required for defining a clear relationship semantics among nodes.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.aws.AwsApiGateway:
    derived_from: radon.nodes.abstract.ApiGateway
    attributes: [role_name, aws_region, ...]
    properties: [role_name, api_gateway_title, ... ]
    requirements:
      - host:
          capability: tosca.capabilities.Container
          node: radon.nodes.aws.AwsPlatform
          relationship: tosca.relationships.HostedOn
          occurrences: [ 1, 1 ]
      - invoker:
          capability: radon.capabilities.Invocable
          node: radon.nodes.aws.AwsLambdaFunction
          relationship: radon.relationships.aws.AwsTriggers
          occurrences: [ 0, UNBOUNDED ]
```

Listing 2 - An excerpt from the AWS API Gateway Node Type for modeling properties, attributes, and requirements.

Another important part of the Node Type definitions is a set of interfaces and respective operations that will be invoked by the TOSCA orchestrator, e.g., for enacting the deployment of the application, running configuration scripts, or undeploying it. Listing XX shows a simplified excerpt of the interface definition for the *radon.nodes.aws.ApiGateway* Node Type. To enable deploying instances of this type, TOSCA's standard lifecycle interface has to be defined with the *create* operation providing a suitable implementation artifact, e.g., an Ansible script (we provide

more details about the deployability aspects in the next subsection). Note that interface also defines the inputs required by implementation artifacts, e.g., the *create* operation requires such information as AWS region and function's Amazon Resource Name (ARN). Using TOSCA's intrinsic functions one might use properties of related types in the topology as inputs to the implementation artifacts. For example, the aforementioned AWS region information is accessed from the AwsPlatform node using the following specification: *{ get_property: [SELF, host, region] }*. Firstly, the keyword "SELF" refers to the AwsApiGateway Node Type itself, meaning that the starting point for accessing the desired property value is the Node Type itself. Next part of the specification is a reference to the hosting type, i.e., the AwsPlatform acts as a *host* for the AwsApiGateway Node Type as defined in the requirements section shown in Listing 2. Therefore, the final part of the specification references a property named *region* of the AwsPlatform Node Type. In addition, some components might require additional files for deployment operations. For example, to create endpoints for respective functions, AWS APIGateway might use a Swagger specification file as an input. One way to implement the deployment logic is to use a template of Swagger specification and substitute it based on the properties of the corresponding Node Type, i.e., substitute function endpoint names, HTTP methods used, etc. To allow such behavior, an interface operation might define a *swagger* dependency file as shown in Listing 3.

```
interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        inputs:
          lambda_function_arn:
            default: { get_attribute: [ SELF, invoker, arn ] }
          aws_region:
            default: { get_property: [ SELF, host, region ] }
          function_name:
            default: { get_property: [ SELF, invoker, name ] } ...
        operations:
          create:
            implementation:
              primary: create
              dependencies: [ swagger ]
    artifacts:
      create:
        file: create.yml
      swagger:
        file: swagger_template.yaml
```

Listing 3 - A simplified excerpt from the Node Type for modeling interfaces and operations of AWS API Gateway.

After defining all the required information and providing the implementation artifact with zero or more dependency files, the resulting Node Type can be used as a part of the RADON model that can be processed by the TOSCA orchestrator to deploy it.

Enriching Existing Modeling Constructs: AWS Lambda function's concurrency example

An initial version of the Node Type representing a function hosted on AWS Lambda was introduced in the previous iteration of the deliverable. While it allowed representing the core properties and attributes of an AWS Lambda function, it was also not possible to fine-tune. For example, since AWS also allows configuring the concurrency settings of a function and this was needed by RADON's Decomposition Tool, this information had to be added to the existing Node Type. Listing 4 demonstrates an excerpt from the *radon.nodes.aws.AwsLambdaFunction* Node Type with the added property definition for concurrency configuration.

```
tosca_definitions_version: tosca_simple_yaml_1_3


node_types:
  radon.nodes.aws.AwsLambdaFunction:
    derived_from: radon.nodes.abstract.Function
    attributes: ...
    properties:
      ...
      concurrency:
        type: integer
        description: Function concurrency
        required: false
        status: supported
        constraints:
          - in_range: [ 1, UNBOUNDED ]
```

Listing 4 - An excerpt from the AwsLambdaFunction Node Type with concurrency-related modifications.

Essentially, this is an integer value that defines a maximum number of simultaneous function executions. As discussed previously, apart from modifying the respective TOSCA definition, the underlying implementation artifact has to be enriched to use this newly-introduced property. For example, this property value could be used as a part of a command executed using AWS CLI as an Ansible task.

3.1.2 Deployability of Modeling Constructs

To be able to deploy created RADON Models, each non-abstract modeling entity contained in the model must also provide implementation artifacts for deploying this entity. RADON Orchestrator is one of the core tools in RADON methodology, which was described in D5.1 "Runtime Environment" and is responsible for enacting the deployment of supplied RADON Models. One of the current requirements of the RADON Orchestrator is to provide implementation artifacts in a

form of Ansible scripts. Internally, the orchestrator parses given RADON Models, determines the deployment order, and executes the implementation artifacts following this order. As a result, the deployability aspect of entities in the RADON Modeling Profile is defined by the presence of valid and functional Ansible scripts. At the moment of writing, not all types in the RADON Particles repository introduced in D4.3 "RADON Models I" have implementation artifacts introduced, as it is an ongoing development process with contributions being introduced by the member of RADON consortium and the community growing around the Particles repository. Considering the various persistence options described in more detail in Section 4, the finalized and deployable modeling entities are intended to be published for reuse as parts of the RADON's Template Library described in the deliverable D5.3 "Technology Library".

Essentially, it is not trivial to provide general recommendations for Ansible-based implementation artifacts development as Ansible is a quickly-evolving deployment automation technology with multiple new modules being introduced, hence changing the overall implementation workflow. Some issues can arise in the future as Ansible changed the way of delivering the tool. Modules will not be available with the Ansible engine in the future and users will need to rely on Ansible collections. However, there are several points on artifacts development we want to highlight.

Firstly, as was shown in the previous examples, implementation artifacts might require specifying file dependencies, e.g., for a template-based generation of Swagger or AWS policy specifications. These file dependencies can be specified using TOSCA's *dependencies* property in operation definition, hence making the designed modeling entity self-contained and fostering its reuse. Not surprisingly, such dependency handling can also be implemented differently, e.g., wrapping existing third-party tools and APIs as Ansible tasks that solve the given problem, e.g., generate a Swagger specification. Moreover, Ansible provides a constantly-growing number of collections that might help to solve the given task. As a result of such flexibility, implementation artifacts for the same modeling entity might be developed differently leading to a need to either separate such modeling entities or wrap multiple implementations and execute them conditionally based on the supplied user input. As a simple example, consider the previously-discussed AwsLambdaFunction's implementation with and without the concurrency property configuration. For simple deployment use cases, setting this property might not be needed, leading to a redundant Ansible task in the implementation artifact. For such a simple case, the implementation artifact can simply skip the concurrency configuration step if the property is empty. However, for cases introducing multiple additional properties only for special cases which rarely happen, the resulting modeling entity will be unnecessarily cluttered. One solution in this case is to have two different implementation artifacts and *separate versions* of the modeling entity, hence supporting simplified and complex modeling use cases.

Another important issue is the implementation guidelines for cases when there are no Ansible modules suitable for solving the deployment task. For example, at the moment of writing, Ansible

modules[2] for Azure Functions are not as powerful as, e.g., using Azure's REST API directly. Due to flexibility provided by TOSCA, implementation artifacts can then wrap the REST API calls or even existing third-party tools to accomplish the deployment task. However, the main risk here is the additional coupling that has to be maintained as well as the need to implement a proper error handling logic in respective Ansible tasks. Such external dependencies, therefore, have to be carefully documented in respective entity's README files.

3.1.3 Heterogeneity of Modeling Constructs

The rapid evolution of cloud technologies and the resulting heterogeneity of components and features results in another interesting design decision that a modeler needs to make: to what extent implementation artifacts must cover the deployment heterogeneity. For example, the introduced abstract Triggers Relationship Type defines the function triggering semantics for a given event source. However, establishing the event binding at deployment time requires different steps for different providers and event source types, e.g., configuring an AWS API Gateway and AWS S3 bucket to invoke a function based on certain event kinds is different from the same event triggering in Microsoft Azure due to provider- and technology-specific requirements [Yussupov2019, Yussupov2020]. As a result, when dealing with such heterogeneity, modelers might choose between two modeling approaches, namely coarse-grained, generic types with complex implementations or finer-grained, concrete types with limited implementation scope.

In the first iteration of RADON Modeling Profile, for modeling the function triggering semantics a set of provider-specific Relationship Types was introduced, e.g., AwsTriggers Relationship Type. However, at this moment, this modeling construct combines together multiple possible event sources such object storage or API gateways, which eventually might clutter the model with event source-specific information in case it must be modeled as a part of the Relationship Type. While there is no clear recommendation for the tradeoff between the complexity of implementation artifacts and how generic types must be, in RADON Modeling Profile we try to combine type's semantics as much as possible to avoid having a quadratic number of types that represent, e.g., interaction between a function and each possible event source type. However, in some cases it might be needed to have more specific modeling entities describing a particular type of interaction such as a NoSQL database triggering a function. However, maintaining the proper type inheritance for such cases allows abstracting away the model, e.g., for more general-purpose model analysis tasks. In such cases, the underlying tooling might also support substitution of generic types to specific depending on the source and target nodes, hence eliminating the need to manage multiple related types and reducing the complexity of type implementations.

3.2 Data Flow Modeling Challenges

Another important capability that must be provided by the RADON Modeling Profile is the modeling of data pipelines using various processing component types. For example, apart from the

[2]We use the term *module* to refer to plain Ansible modules and Ansible collection modules.

event-driven data flow brought by serverless, FaaS-based applications, data processing components might be hosted in a *serverful* or *hybrid* fashion, e.g., using dedicated software components such as Apache NiFi hosted on virtual machines that are also modeled as parts of the application topology which might also use FaaS-hosted functions as processing blocks. The first iteration of the deliverable presented an initial version of Apache NiFi-based data pipeline models using dedicated TOSCA Node Types. In the following, we elaborate on the extension of this approach and respective changes made to the RADON Modeling Profile.

As discussed in the initial version of RADON Modeling Profile, Apache NiFi-based data pipelines can be modeled with a varying degree of details: from self-contained Node Types representing composite data pipelines to finer-grained types representing particular pipeline steps such as uploading the data to an AWS S3 bucket. Since the initial set of modeling constructs focused on representing multiple pipeline actions as a single node, the overall direction of addressing data flow modeling challenges in the RADON Modeling Profile is related to design of finer-grained modeling entities. While the core concepts behind the finer-grained data pipeline modeling constructs and the corresponding type hierarchy are described in-detail in "Data pipeline orchestration I" deliverable [D5.5], in this section we focus more on technical aspects and design decisions related to the model extensions contributed by the RADON consortium members.

As described in Section 3.1.1, the overall workflow of modeling profile modification involves (i) enriching the RADON types system, (ii) introducing new or updating existing modeling constructs, and (iii) adding the implementation artifacts make these modeling constructs deployable. Additionally, as a minimal requirement, the updated version of the Modeling Profile has to be merged into an existing community-driven RADON Particles repository. We provide more details on the respective repository layout and persistence options in Section 4.

Essentially, the finer-grained data pipelines modeling approach distinguishes three main types of pipeline components, namely source blocks, middle processing blocks, and target blocks, which are interconnected by means of NiFi-specific Relationship Types with ConnectsTo semantics. All respective modeling constructs reside in a dedicated RADON namespace **.datapipeline*, e.g., *radon.nodes.datapipeline* and *radon.relationships.datapipeline*. The introduced node types are derived from *radon.nodes.abstract.DataPipeline* Node Type shown in Listing 5, and are connected with a NiFi-specific Relationship Type that is derived from the normative TOSCA ConnectsTo Relationship Type. The relationship type includes a configuration implementation for initiating the data transfer between separately deployed pipeline blocks and it defines the direction of the flow. Target block provides the capability for other block types to connect to it. Source block requires to be connected to another block type and the middle processing block has both requirement and capability for connecting to other blocks, including additional middle blocks. In terms of TOSCA features, the introduced models actively use properties in combination with configurable NiFi XML-based implementation scripts attached on the Node Type level. The specifics of pipeline block's implementation depends on the actual type of represented component, e.g., source block representing the consumption of local files or files stored in the cloud object

storage service. Therefore, the extensions for data pipeline modeling cover multiple specific component types relevant for RADON use cases.

```
node_types:
  radon.nodes.abstract.DataPipeline:
    derived_from: tosca.nodes.Root
    properties: [...]
    requirements:
    - host:
        capability: tosca.capabilities.Container
        relationship: tosca.relationships.HostedOn
        occurrences: [ 1, 1 ]
```

Listing 5 - An abstract Node Type for representing generic data pipeline blocks.

Due to specifics of Apache NiFi's XML-based implementation scripts, it is sufficient to attach implementation artifacts on the level of Node Types which makes the modeled extension easily configurable and, hence, reusable. Listing 6 shows a simplified RADON Model representing an Apache NiFi-based data pipeline that emulates a thumbnail generation flow using AWS infrastructure. More specifically, a NiFi pipeline comprising three blocks is hosted on OpenStack. The source block is responsible for getting an image from an S3 bucket, a processing block generates a thumbnail by invoking an AWS Lambda function, and the destination block stores the thumbnail in a target bucket.

```
tosca_definitions_version: tosca_simple_yaml_1_3
topology_template:
  node_templates:
    AWSLambda:
      type: radon.nodes.datapipeline.process.AWSLambda
      properties: [...]
      requirements:
        - host: Nifi
        - ConnectToPipeline: PubsS3Bucket
    PubsS3Bucket:
      type: radon.nodes.datapipeline.destination.PubsS3Bucket
      properties: [...]
      requirements:
         - host: Nifi
    ConsS3Bucket:
      type: radon.nodes.datapipeline.source.ConsS3Bucket
      properties: [...]
      requirements:
        - host: Nifi
        - ConnectToPipeline: AWSLambda
```

```
  Nifi:
    type: radon.nodes.nifi.Nifi
    properties: [...]
    requirements:
      - host: OpenStack
  OpenStack:
    type: radon.nodes.VM.OpenStack
    properties: [...]
relationship_templates: [...]
```

Listing 6 - An example thumbnail generation data flow using Apache NiFi and AWS infrastructure.

3.3 DevOps Challenges

In this section we elaborate on addressing the challenges motivated by DevOps-related use cases. This includes requirements arising from RADON tools such as Continuous Testing Tool and Decomposition Tool as well as requirements related to the overall application development flow according to RADON Methodology, in particular the issue related to decoupling models from the application business logic.

3.3.1 Heterogeneity of Modeling Constructs in the Context of Infrastructure-as-Code

Although the main purpose of RADON Models is to enable deployment automation using the RADON Orchestrator, resulting application models serve multiple purposes. For instance, to support the continuous testing of modeled applications, a set of complementary modeling entities needs to be introduced as discussed in the deliverable covering RADON Continuous Testing Tool [D3.4]. Another prominent part of the RADON Methodology is the support for deployment optimization, e.g., to minimize the operating costs of a FaaS-based application under the performance requirements by configuring appropriate memory and concurrency for each serveless function. In the following, we discuss these DevOps-motivated examples and how their respective requirements can be addressed by the RADON Modeling Profile.

Use Case: Continuous Testing

To support continuous testing of RADON Models, the RADON Modeling Profile must provide the possibility to define test-related requirements apart from representing the application's components. Moreover, the modeling of testing infrastructure which enables running tests based on defined requirements has to be supported too, e.g., running load tests targeting specific application components using Apache JMeter. Therefore, one extension to the modeling profile contributed by the members of RADON consortium enables (i) modeling test infrastructures, and (ii) defining test requirements for modeled applications. In the following, we highlight some examples and discuss the design decisions behind the introduced TOSCA constructs.

As the first requirement, the RADON Modeling Profile must support modeling test infrastructures. In most cases, a dedicated software has to be used for running tests, e.g., load tests using Apache JMeter or Apache Bench. Thus, a dedicated type hierarchy for *testing agents* was introduced covering both abstract and concrete entity modeling layers (AEML & DEML).

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.testing.CTTAgent:
 derived_from: radon.nodes.docker.DockerApplication
```

Listing 7 - An abstract Node Type representing a generic testing agent.

All Node Types representing testing infrastructure components are derived from the abstract radon.nodes.testing.CTTAgent Node Type shown in Listing 7, which in its turn is derived from a Node Type representing a Docker Application which is required to run specialized testing software. While it is not obligatory to always inherit from Docker Application Node Type, in many cases testing infrastructure is deployed in a more traditional fashion, requiring additional general-purpose Node Types representing various types of hosts, e.g., a virtual machine, a Docker Engine, or a Kubernetes cluster. The majority of testing infrastructure use cases employed in RADON are sufficient to model using container-based deployments as discussed in more details in the RADON CTT deliverable [D3.4]. However, additional kinds of testing infrastructure Node Types can easily be introduced as discussed in Section 3.1.1. Essentially, a testing infrastructure is then modeled as a separate RADON Model that complements the actual application and is deployed with it to enable running defined tests. To facilitate this task, the introduced modeling extensions also contain a set of predefined test infrastructures enabling particular test types, e.g., load testing using Apache JMeter. These predefined infrastructure blueprints reside in *radon.blueprints.testing.** namespace and can be used to enable continuous testing without the need to model the general-purpose test infrastructure, whereas custom test infrastructure models can be produced by combining or extending contributed testing Node Types.

The next requirement related to continuous testing is the ability to specify tests directly in RADON Models, e.g., tests targeting an AWS Lambda function or a microservice modeled as a Docker container. To enable this, TOSCA Policy constructs were used as discussed briefly in the first iteration of the RADON Modeling Profile deliverable. Similar to other modeling extensions, the policy-based tests hierarchy is introduced in a dedicated namespace *radon.policies.testing.**, with all test types originating from an abstract *radon.policies.testing.Test* Policy Type, which represents a generic test case that relies on a specific test infrastructure as shown in Listing 8.

```
policy_types:
  radon.policies.testing.Test:
  properties:
    ti_blueprint:
      type: string
      description: Reference to a RADON test infrastructure blueprint
```

```
    test_id:
      type: string
      description: Identifier for this test case
```

Listing 8 - An abstract Policy Type for representing a generic test case.

Examples of concrete test types include JMeter- and Locust-based load tests, endpoint accessibility tests, e.g., HttpEndpoint and TcpPing Policy Types. These testing policies can be attached to desired Node templates in the chosen Service Template and exported as a part of the deployable application package, i.e., TOSCA CSAR, to support processing by the Continuous Testing Tools as well as RADON Orchestrator. As a result, using these extensions, modelers are able to design and export required test infrastructures and attach test cases to particular nodes (or the application as a whole), enabling the deployment of the given infrastructure and running modeled test w.r.t. the deployed application. For the sake of conciseness, similar to the first iteration of the deliverable, we provide all introduced modeling extensions in the companion document described in Section 6.

Use Case: Deployment Optimization

The RADON decomposition tool implements a model-driven approach to deployment optimization based on layered queueing networks (LQNs), a canonical form of extended queueing networks developed for modeling systems with nested simultaneous resource possession [D3.2]. To apply this approach to a TOSCA model, the RADON Modeling Profile needs to be extended to support the following features:

- Specification of reference workloads, either open or closed;
- Specification of performance requirements, e.g. the maximum response time;
- Incorporation of LQN structures into nodes and relationships.

As shown in Listing 9, an abstract node type called *Workload* is defined for representing a generic workload. The behavior of a workload can be described through the *entries* property, which is of special data type *workload.Entries*. It is possible that a workload *Triggers* any *Invocable* node, e.g. an *AwsLambdaFunction*, or *ConnectsTo* any *Endpoint* node, e.g. an *AwsS3Bucket*. To represent a specific workload, we derive two child abstract node types, *workload.OpenWorkload* and *workload.ClosedWorkload*, as shown in Listings 10 and 11. One can specify the *interarrival_time* of an open workload with the former and the *population* and *think_time* of a closed workload with the latter. In particular, the *interarrival_time* and the *think_time* are both *RandomVariable*.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.Workload:
    derived_from: tosca.nodes.Root
    properties:
      name:
```

```
        type: string
        required: false
      entries:
        type: radon.datatypes.workload.Entries
        required: false
    requirements:
      - invoker:
          capability: radon.capabilities.Invocable
          relationship: radon.relationships.abstract.Triggers
          occurrences: [ 0, UNBOUNDED ]
      - endpoint:
          capability: tosca.capabilities.Endpoint
          relationship: radon.relationships.abstract.ConnectsTo
          occurrences: [ 0, UNBOUNDED ]
```

Listing 9 - An abstract node type for representing a generic workload.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.workload.OpenWorkload:
    derived_from: radon.nodes.abstract.Workload
    properties:
      interarrival_time:
        type: radon.datatypes.RandomVariable
        required: true
```

Listing 10 - An abstract node type for representing an open workload.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.workload.ClosedWorkload:
    derived_from: radon.nodes.abstract.Workload
    properties:
      population:
        type: integer
        required: true
      think_time:
        type: radon.datatypes.RandomVariable
        required: true
```

Listing 11 - An abstract node type for representing a closed workload.

A root policy type is defined for representing performance requirements. As shown in Listing 12, it allows the specification of *target_entries* that the requirements apply to as well as the expected *upper_bound* and *lower_bound* of the performance measure. Listings 13 and 14 show the definitions of two child policy types, *MeanResponseTime* and *MeanTotalResponseTime*, which are

derived to represent requirements on the mean response time and the mean total response time respectively. The main difference between these two policy types is that the former takes effect on each specified node or entry individually while the latter operates on all specified nodes or entries as a whole. It is worth pointing out that in certain cases, the mean total response time is equivalent to the mean system response time, which is often the most important performance measure.

```
tosca_definitions_version: tosca_simple_yaml_1_3
policy_types:
  radon.policies.Performance:
    derived_from: tosca.policies.Performance
    properties:
      target_entries:
        type: list
        required: false
        entry_schema:
          type: string
      lower_bound:
        type: float
        required: false
      upper_bound:
        type: float
        required: false
```

Listing 12 - A root policy type for representing performance requirements.

```
tosca_definitions_version: tosca_simple_yaml_1_3
policy_types:
  radon.policies.performance.MeanResponseTime:
    derived_from: radon.policies.Performance
```

Listing 13 - A child policy type for representing requirements on the mean response time.

```
tosca_definitions_version: tosca_simple_yaml_1_3
policy_types:
  radon.policies.performance.MeanTotalResponseTime:
    derived_from: radon.policies.Performance
```

Listing 14 - A child policy type for representing requirements on the mean total response time.

To describe the behavior of a node, we typically add an optional property named *entries* to the abstract node type. This property is used to represent services provided by the node and defined as a map of *Entry* elements. Listing 15 shows the definition of the *Entry* data type. As for a relationship, an optional property named *interactions* is normally added to the abstract relationship type. This property is used to represent communications raising from the relationship and defined

as a list of *Interaction* elements. Listing 16 shows the definition of the *Interaction* data type. The properties of *Entry* and *Interaction* conceptually map onto the essential LQN constructs [D3.2].

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Entry:
    derived_from: tosca.datatypes.Root
    properties:
      activities:
        type: map
        required: true
        entry_schema:
          type: radon.datatypes.Activity
      precedences:
        type: list
        required: false
        entry_schema:
          type: radon.datatypes.Precedence
```

Listing 15 - A data type for representing an entry at a node.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Interaction:
    derived_from: tosca.datatypes.Root
    properties:
      type:
        type: string
        required: true
      source_activity:
        type: string
        required: true
      target_entry:
        type: string
        required: true
      number_of_requests:
        type: float
        required: true
      network_delay:
        type: radon.datatypes.RandomVariable
        required: false
```

Listing 16 - A data type for representing an interaction on a relationship.

3.3.2 Decoupling artifacts from models

Another DevOps-related requirement stemming from the RADON's CI/CD pipeline and the overall development workflow using RADON IDE is related to decoupling the models from business logic to facilitate making changes to the source code without modifying the models. As discussed previously, a RADON Model represents the desired application topology that comprises various component types such as FaaS-hosted functions, Database-as-a-Service components such as AWS DynamoDB, Docker-based applications and test infrastructures, and data processing components. Some of these components are intended to be developed using the RADON IDE and afterwards attached to RADON Models to be processed, e.g., function source code that allows generating thumbnails is developed in JavaScript, packaged, and attached as a deployment artifact to a Node Template of type AwsLambdaFunction. Two issues arise in this workflow if artifacts are attached as files, namely (i) need to modify application models each time after changes are introduced to the component's source code, and (ii) increased size of the TOSCA CSAR that packages the given RADON Model.

Enabling both options in RADON Modeling Profile, i.e., modeling of attached files and file references only, does not require any modifications to the modeling constructs since TOSCA allows specification of file references for artifacts. As a result, tackling this issue is a responsibility of the corresponding RADON tooling. For example, RADON's Graphical Modeling Tool (GMT) might introduce support for referencing files in the artifacts and also enable, e.g., materializing the references at CSAR export time to support self-contained CSARs export. Listing 17 shows how artifacts can reference local files attached as well as usage of URLs as file references in RADON Modeling Constructs.

```
...
artifacts:
  create:
    type: radon.artifacts.Ansible
    file: create.yml
  thumbnail-generator-dev:
    type: radon.artifacts.archive.JAR
    file: https://my-file-archive.radon/files/thumbnail-generator-dev.jar
```

Listing 17 - Artifact files referencing alternatives in RADON modeling entities

4. Persistence and Reuse of RADON Modeling Entities

RADON Models are reusable entities and, therefore, must be stored in a shareable and reusable way to be used by RADON users as well as consortium partners. In a DevOps-enabled environment, RADON Models need to be versioned and changes need to be tracked and managed as infrastructure as code (IaC). Therefore, we need approaches commonly used in modern, DevOps-enabled software engineering that are appropriate to store and share such modeling entities. In RADON, we foresee three options to persist and share RADON Models, each of which applies to different stages in the DevOps lifecycle:

(1) Local filesystem: In the early stages of the lifecycle, design and development, RADON Models are usually stored locally on the developers workstations. At this stage, the developer composes a RADON Model based on reusable building blocks in the form TOSCA entity types, e.g., to instantiate TOSCA node types to create a TOSCA service template expressing the application structure.
(2) Version control system: Working in (distributed) teams is natural in a DevOps-enabled environment. Created RADON Models by one developer needs to be shared with the rest of its team. Therefore, for team-based development and testing, we need the means of version control systems (VCS) to version, track, and manage RADON Models.
(3) Publishing service: For deploying RADON Models in production you may want an additional publishing service layer on top of VCS. This option lets software development teams publish a deployable RADON Model in a certain version such that the content can no longer be changed. This approach persuades users to organize each modification in a new version, similar to package managers such as “pypi” (Python) work.

In RADON, we unify these options with the notion of the RADON Template Library (see deliverable D5.3 “Technology Library” for further technical details [D5.3]). Conceptually, as shown in Figure 1, RADON users use the RADON IDE or the Graphical Modeling Tool (GMT) to create and develop RADON Models. The models can then be versioned and managed using VCS such as Git and stored and shared in public or private remote repositories (e.g., by using GitHub). For example, the RADON consortium publishes its developed and reusable RADON Models as open-source to a public GitHub repository, the so-called RADON Particles repository, which essentially is an example of how to publish RADON Models to a remotely managed VCS. Once a RADON Model has been finalized, it can be published into the RADON Template Publishing Service (TPS). This service is a standalone, enterprise-ready service to store, share, and publish RADON Models [D5.3]. This service puts additional value on top, e.g., by a tight integration with the RADON Orchestrator as well as enterprise-ready features such as user management to share RADON models either publicly or privately.

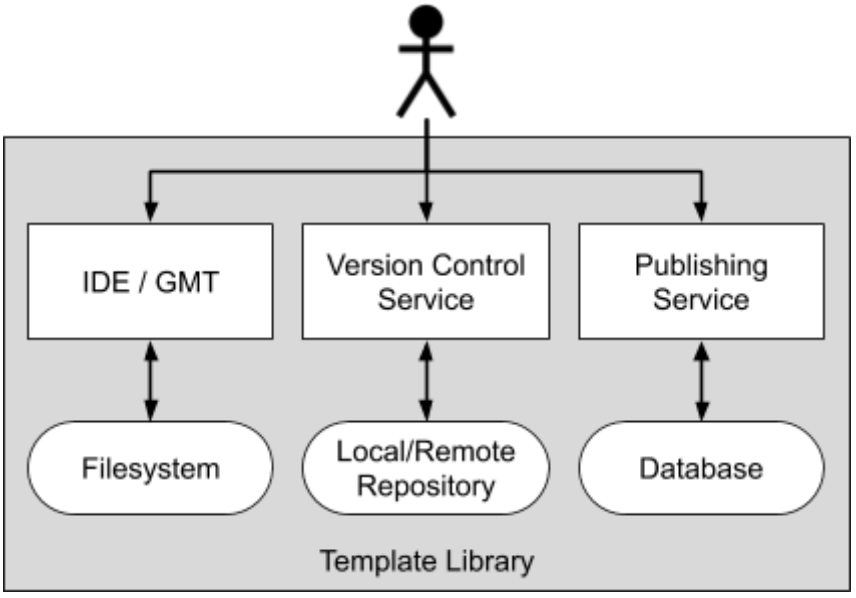

Figure 1 - Different options to store and maintain RADON Models.

Independently of the persistence option, concerning the RADON Models we need to persist the following information:

- 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[3].
- RADON Models itself, i.e., application blueprints or TOSCA service templates, which are composed of reusable TOSCA entity types.

To store these information unified in the Template Library, we propose a layout that can be applied to a file-based structure (local filesystem or VCS) or onto a database. The file-based structure has already been introduced by the first RADON Model deliverable [D4.3]. However, we slightly changed and generalized this layout to be used as the baseline for the RADON TPS as well as the file-based Template Libraries in the form of the RADON Particles.

Generally, the Template Library is structured by the `<TOSCA entity type>`, a `<namespace>`, and an `<identifier>` for the respective TOSCA entity, plus additional versioning and metadata information. For example, Listing 18 shows an example how this translates to a file-based directory structure. 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, and service templates (RADON Models). One level below, we employ a namespace to ensure that all maintained entities have unique names so that they can be easily identified, e.g.,

[3] Types without TOSCA implementations are called “abstract types”.

`radon.nodes.aws` to group all AWS-related entity types. The third level identities the entity itself. It's 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, in the RADON Particles we have `NodeType.tosca` files that contain the actual TOSCA syntax to define the respective TOSCA node type. Beside 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.

```
<root>
  |-<entity type>
  |  |-<namespace>
  |  |  |-<identifier>
  |  |  |  |-files
  |  |  |  |  |-<implementation>
  |  |  |  |  |  |-<name>.yml
  |  |  |  |-<entity type>.tosca
  |  |  |  |-README.md
  |  |  |  |-LICENSE
```

Listing 18 - File-based persistence example of the Template Library.

An instantiated example of such a file-based template library is the RADON Particles[4] 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 and will be developed by the RADON consortium. Moreover, it is used by the Graphical Modeling Tool (GMT) to initialize and provide a modeling baseline.

Similar to a file-based persistence, we can apply the general structure also to a database to offer an advanced and enterprise-ready publishing service such as the RADON TPS. It can be applied to a relational database, e.g., where TOSCA entity types are stored in separate tables while having respective relations to attached file objects, as well as to NoSQL databases, e.g., where anything related to TOSCA entity type can be stored as separate "documents" (in case of a document-oriented database such as MongoDB). However, independent of the underlying technical implementation, the RADON TPS will provide a public API that follows exactly this layout [D5.3].

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

5. Usage and Extensibility Guidelines

In this section, we focus on scenarios in which produced RADON Models can be used and the advantages they bring. In addition, we highlight some important aspects related to profile's extensibility as a short set of recommendations.

5.1 RADON Models Packaging as the Main Point of Integration

As discussed in the first deliverable on RADON Modeling Profile, TOSCA defines so-called Cloud Service Archives (CSAR) as a way to package modeled applications with all respective type definitions and referenced artifacts. Basically, the specification describes how files have to be stored inside the CSAR and how the main model definition, i.e., the application's Service Template, can be found inside the CSAR. While the overall folders structure might vary, the CSAR must provide a so-called TOSCA.meta file which lists all archive entries with their MIME types. Moreover, this meta file also specifies the main entry point in the standardized header field. Listing 19 shows an example of the main block in the TOSCA.meta file which specifies the main definition files using the standardized *Entry-Definitions* field.

```
TOSCA-Meta-Version: 1.0
CSAR-Version: 1.1
Created-By: Winery 2.0.0-SNAPSHOT
Entry-Definitions: _definitions/radonblueprintsexamples__EC2_on_AWS.tosca
```

Listing 19 - The main block of the TOSCA.meta file which specifies the application's entry point.

Therefore, the accessibility of the main definition files can be guaranteed by enforcing a specific CSAR structure at CSAR export time, e.g., generated automatically by RADON GMT. The overall way to automate CSAR processing by multiple target consumers is then to simply open the TOSCA.meta file and get the file path (referencing a file inside CSAR) to the entry definition. This entry point can then be used, e.g., to automate the deployment using RADON Orchestrator or verify the model using RADON Verification Tool.

From a conceptual point of view, the application models packaged in a form of CSAR serve as one of the main integration points for the tools employed in the RADON methodology. GMT as a way to graphically model applications allows exporting them as self-contained CSARs (also including modeled test specifications), which can then be used as an input for deployment automation using the RADON Orchestrator. In addition, this applies to CSARs of modeled test infrastructures required for continuous testing. Furthermore, such RADON tools as the Decomposition Tool, Verification Tool, and Defect Prediction Tool require the model and/or artifacts data as an input. Since tools are loosely-coupled, CSAR can be used as a common, standardized input format for processing. For example, if attached Ansible implementation scripts have to be analyzed for possible faults or anti-patterns, the Defect Prediction tool can consume CSARs as an input and use the application model to access all attached artifacts of type

radon.artifacts.Ansible, eliminating the need to search for Ansible scripts among other available .yml files. Similar with the Decomposition Tool: CSARs can be consumed as an input and the main Topology Template can be accessed for analysis via the entry point specified in the TOSCA.meta file. RADON's Verification Tool also relies on the application model as a part of the input, hence the overall processing mechanics is similar. Additionally, in case the analyzed model has to be enriched, e.g., annotate nodes with faulty Ansible scripts, optimize properties of the modeled entities, the resulting modified CSARs can be imported into GMT to view the enrichment following the same loosely-coupled workflow. Finally, since this is a packaging format, the CSAR export event can also be used to trigger CI/CD pipelines.

While TOSCA does not impose any restrictions on the way implementation artifacts are developed, the deployability of modeling constructs is defined by the capabilities of the employed TOSCA orchestrator. One current requirement related to the RADON Particles repository is to use Ansible for implementing the deployment logic, as it is used by the RADON Orchestrator. Ansible is an extensible, open-source deployment automation technology with a large number of available modules and constantly-growing community. Therefore, using Ansible for implementing the deployment logic is rather a design decision than a technical limitation. Moreover, in case future versions of the RADON Orchestrator will also support other deployment automation technologies, the RADON Modeling Profile structure will not change, since the overall artifact attachment workflow remains the same. The only modifications that might be needed is introducing a corresponding Artifact Type in the *radon.artifacts.** namespace, e.g., *radon.artifacts.Chef*.

5.2 Target Consumers of RADON Models

RADON Modeling Profile serves as one of the key enablers of the RADON's DevOps-motivated transition from the design to runtime phase of serverless and microservice-based applications and data flows. While there are multiple provider-specific and provider-agnostic deployment automation technologies such as Azure Resource Manager, AWS CloudFormation and AWS Serverless Application Model (SAM) for modeling serverless applications, or Serverless Framework, these technologies cannot be used directly to suffice all RADON requirements. For example, serverless-specific modeling using AWS SAM and Serverless Framework will not satisfy the requirement to also support modeling traditional component types, data flows, or representing test specifications in models. Additionally, these technologies focus mainly on the deployment aspect, which is not the case for RADON Modeling Profile as it also can be used for analysis and verification purposes before the deployment stage. Another important requirement is to support technology-agnostic modeling, which limits a set of possible options to custom DSLs provided by provider-agnostic deployment technologies such as Terraform or Ansible. However, these languages lock the entire RADON toolchain into a particular deployment orchestrator, e.g., Terraform. On the other hand, TOSCA is a vendor-agnostic standard and TOSCA-compliant orchestrators are not bound to a particular technology, making reuse of the RADON Modeling Profile easier. In addition, such modeling constructs as TOSCA policies facilitate representing

additional requirements such as test case specification in a standardized manner. Moreover, the RADON Modeling Profile supports might leveraging existing state-of-the-art industrial tooling such as Terraform, Ansible, and Serverless Framework by introducing new modeling constructs with the deployment logic implemented using the most suitable tool for the task, e.g., Serverless Framework for deploying serverless components, Ansible for configuration management tasks.

Based on the aforementioned scenarios, RADON Models can be consumed for the following purposes: (i) deployment automation, (ii) continuous testing, and (ii) model and code validation and optimization. In the next paragraphs, we briefly cover the key points related to how target consumers, i.e., respective tooling in the context of RADON, use RADON Models as an input.

The deployment automation in RADON is a responsibility of the RADON Orchestrator, which consumes RADON Models in a form of CSARs, processes them and enacts the deployment following the specified interfaces or operations and invoking the corresponding implementation artifacts. Most concepts and design decisions discussed in both iterations of this deliverable target mainly the deployment automation, including the deployability aspects discussed in Section 3.1.2. In addition, the technical details about RADON Orchestrator are provided in D5.1 “Runtime Environment I”.

With respect to continuous testing, RADON Models are also used after the application is deployed by the Continuous Testing Tool [D3.4]. Firstly, RADON Models here represent the test infrastructure, which has to be deployed by the RADON Orchestrator. Additionally, RADON Model representing the to-be-tested application with test specifications is used as an input by the CTT to run modeled test cases on the deployed test infrastructure.

Finally, multiple RADON tools use RADON Models as an input for model and code validation purposes. For example, defects such as code smells or anti-patterns that might be present in Ansible-based implementation scripts have to be identified prior to deployment. To support this, RADON Defect Prediction Tool can consume the exported CSAR for validation purposes. In case the defects are present in the topology, several options can be employed to notify RADON users. Firstly, the notification is possible by means of the interfaces provided by the Defect Prediction Tool. Due to extensibility of TOSCA, model-based notifications can also be introduced, e.g., the Service Template can be annotated with policies signifying the presence of defects or additional meta-data properties can be added to the model, to allow visualizing this information in the GMT. The similar workflow is applicable to both Decomposition and Verification Tools. Essentially, these tools use application Service Templates for special-purpose analysis, e.g., to identify whether the application complies with the defined constraints, or can be optimized w.r.t. Resource usage-related property values such as memory limits for a FaaS-hosted function. In both cases, an exported RADON Model in the form of a CSAR can be consumed by the respective RADON tool, validated and modified if needed. The result notification flow might also rely on custom TOSCA-based annotations in a form of Policy Types or metadata properties. As an optional step,

such annotated RADON Models can further be imported into GMT to display the results of validation/optimization.

5.3 General Profile Extensibility Guidelines

As discussed in Section 3.1, the constant need to evolve the RADON Modeling Profile is motivated by the fast pace of modifications introduced to the cloud service offerings and open-source solutions. The extensibility of TOSCA is indeed a key enabler of RADON Modeling Profile enrichment. In most cases, when modeling constructs have to be introduced or enhanced, the modification flow is identical to guidelines described in Section 3.1.1. However, there are points worth highlighting that are related to support of TOSCA features by the respective tooling.

Firstly, the modification guidelines in Section 3.1.1 comply with the normative TOSCA modeling to guarantee that modeled entities would be deployed by the RADON Orchestrator. The only additional requirement that has to be taken into consideration is the usage of Ansible-based implementation artifacts. Since Ansible is also easily-extensible and can be even used to wrap any other deployment logic, e.g., wrapping REST API calls or an existing CLI tool, the extensibility of implementation artifacts can be achieved by following the Ansible documentation.

For cases when new behavior needs to be modeled using TOSCA, there might be cases when normative types are not sufficient. For example, introducing custom interfaces and operations might be required to represent a particular operation on the Node Type not supported by normative interfaces, e.g. *test* operation to verify that the deployment was successful. Not surprisingly, in terms of TOSCA-related modifications, RADON Modeling Profile has to provide these new interface types. However, another requirement is that the RADON orchestrator also understands such custom operations. Therefore, this extension would require referring to the extensibility guidelines of the RADON Orchestrator.

Another extensibility use case is related to RADON Model consumers that rely on custom Policy Types, Data Types, etc. In case new modeling constructs have to be introduced to represent new functional or non-functional requirements, e.g., new Policy Types to specify more test case types, it is important to ensure that the target consumer also can process new constructs. Hence, the extensibility of tool-specific behavior, as with the RADON Orchestrator, must also be ensured on the level of the tool.

6. Conclusions

In this document, we described the second revision of RADON Models and focused on the aspects of maintainability and extensibility of the RADON Modeling Profile. We elaborated on how models can be introduced or enhanced, what is required to make them deployable and how they can be persistent. In addition, we discussed the advantages of using the same modeling profile for multiple purposes including deployment automation, continuous testing, model verification and optimization. Finally, we outlined the key aspects of profile extensibility. The extended version of the profile containing newly-defined modeling constructs are provided in a **companion** document.

Table 1 shows an overview of the level of fulfillment for each of the agreed requirements. The labels specifying the "Level of fulfillment" are defined as follows:

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

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

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

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

Table 1 - 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	✔
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	✔✔✔
R-T4.2-8	Model API Gateway resources in AWS	MUST_HAVE	✔

We hereafter discuss for each requirement briefly how it has been addressed:

- **Requirement R-T4.2-1:** the RADON Modeling Profile was extended with new modeling constructs that address requirements from RADON use case and tool providers. Moreover, this deliverable provided the guidelines for extending the modeling profile.

- **Requirement R-T4.2-2:** the aspects of persistence and reuse of the modeling profile were clearly defined including different stages of models i.e., community-driven repository RADON Particles and final version published using RADON TPS.
- **Requirement R-T4.2-3:** data processing modeling was refined to support finer-grained processing components using the dedicated types hierarchy.
- **Requirement R-T4.2-4:** the dedicated types hierarchy was introduced to support modeling various intermediary data pipeline blocks that can represent preconditions.
- **Requirement R-T4.2-5:** This requirement has already been fulfilled in the first iteration of the deliverable at M6 [D4.3].
- **Requirement R-T4.2-6:** the dedicated types hierarchy was introduced to support modeling various intermediary data pipeline blocks that can represent data compression requirements.
- **Requirement R-T4.2-7:** modeling constructs for representing testing infrastructures and test cases were introduced in a dedicated types hierarchy enabling continuous testing of modeled RADON applications.
- **Requirement R-T4.2-8:** This requirement has been added during discussions with use case partners after Y1. We initially implemented a few aspects of this requirement and populated an alpha version of a respective TOSCA node type that can be used to provision an API Gateway configuration to AWS.

Future work. For the remaining project period we will focus on two main aspects: we will (i) finalize and maintain the existing modeling entities, and (ii) work on the modeling profile extensions. To finalize partially-completed requirements (R-T4.2-4, R-T4.2-6, R-T4.2-8) we need

- to finalize the API Gateway types hierarchy which spans multiple cloud providers in addition to AWS API Gateway, and
- introduce a set of data processing Node Types representing preconditions as intermediary pipeline blocks.

Additionally, we plan to extend the types repository with the emerging cloud service offerings and enrich the existing modeling constructs with Ansible-based implementation artifacts to support a broader set of use cases. Additionally, we will work on a better integration of the modeling profile with the RADON GMT and other related tools.

References

[D3.2] RADON Consortium, “Deliverable D3.2 “Decomposition Tool I”, 2019

[D3.4] RADON Consortium, “Deliverable D3.4: Continuous Testing Tool I”, 2020

[D4.3] RADON Consortium, “Deliverable D4.3: RADON Models I”, 2019

[D4.5] RADON Consortium, “Deliverable D4.5: Graphical Modelling Tool I”, 2019

[D5.3] RADON Consortium, “Deliverable D5.3: Technology Library”, 2020

[D5.5] RADON Consortium, “Deliverable D5.5: Data pipeline orchestration I”, 2019

[OASIS2020] OASIS, TOSCA Simple Profile in YAML Version 1.3, 2020

[Lipton2018] Lipton et al.: “TOSCA Solves Big Problems in the Cloud and Beyond!”, IEEE Cloud Computing, 2018

[Casale2019] G. Casale et al.: “Rational Decomposition and Orchestration for Serverless Computing”, The Symposium and Summer School on Service-Oriented Computing (SummerSoc), 2019, (accepted for publication)

[Jonas2019] Jonas et al.: “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

[Yussupov2019] Yussupov et al.: Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends. In: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2019

[Yussupov2020] Yussupov et al.: SEAPORT: Assessing the Portability of Serverless Applications. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER), 2020