H2020-ICT-2018-2-825040

Rational decomposition and orchestration for serverless computing

Deliverable 3.3

Decomposition Tool II

Version: 1.0

Publication Date: 30-October-2020

Deliverable Card

Deliverable	D3.3
Title:	Decomposition Tool II
Editor(s):	Lulai Zhu (IMP)
Contributor(s):	Giuliano Casale (IMP), Alim Gias (IMP), Lulai Zhu (IMP)
Reviewers:	Stefania D'Agostini (ENG), Pelle Jakovits (UTR)
Type:	Report
Version:	1.0
Date:	30-October-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

The decomposition tool aims to help RADON users in finding the optimal decomposition solution for an application based on the microservices architectural style and the serverless FaaS paradigm. This deliverable reports efforts that have been made in order to achieve the expected outcomes of the decomposition tool, including extension to RADON models in support of deployment optimization and architecture decomposition, initial approaches implemented to coarse-grained and fine-grained decompositions as well as innovative methods for enhancing the accuracy of performance predictions.

Glossary

AWS	Amazon Web Services
DECOMP_TOOL	Decomposition Tool
FaaS	Function as a Service
LQN	Layered Queueing Network
QoS	Quality of Service

Table of contents

1. Introduction

This deliverable describes final work carried out toward developing a tool for decomposing and optimally mapping nodes onto resources in RADON models. As described in deliverable *D2.2 Final requirements*, the following features are foreseen for the decomposition tool:

1. *Deployment optimization*. The capability to assign the concrete resources (e.g., memory, compute) of a RADON application taking into account quality requirements (e.g., service level agreements). This closes the gap between high-level abstractions used by the developers and concrete operational details needed to instantiate the application in the cloud.
2. *Architecture decomposition*. The capability to analyze a pre-existing RADON model and suggest possible changes based on known architectural patterns, with the aim of breaking down the complexity of a monolith into finer microservices or serverless functions. Reasoning about the consequences of a refactoring is also entailed.
3. *Accuracy enhancement*. The capability to improve the results of features 1 and 2 after correlating their outcomes with measurements in the operational environment of the application. In particular, it aims at both: a) increasing the accuracy of model predictions by grounding model parameters into concrete data that is representative of the application usage; b) successively refining decisions made for optimization and decomposition through statistical inference.

1.1 Deliverable Objectives

To reduce the complexity of delivering the above features, the development of the decomposition tool has been divided into two sequential milestones:

Tool Milestone 1. Introduce a method to reason about the service level of an application based on a RADON model that describes its architecture, exploring elementary techniques for accuracy enhancement and subsequently realizing an initial deployment optimization capability.

Tool Milestone 2. Build upon results achieved in the first milestone to further mature the deployment optimization feature, provide architecture decomposition functionalities, and generalize the accuracy enhancement methodology.

The outcomes of *Tool Milestone 1* can be found in deliverable *D3.2 Decomposition Tool I*. The objective of this deliverable is to present work done for *Tool Milestone 2*. The behavior of a given RADON application is modeled by a layered queueing network (LQN) in *Tool Milestone 1*. This modeling approach not only exhibits good accuracy in predicting the performance of the application but also embodies its execution logic as a layered hierarchy of interacting entities. Thus, the decomposition in *Tool Milestone 2* essentially reuses quality-of-service (QoS) models developed in *Tool Milestone 1*.

1.2 Main Achievements

The main achievements of this deliverable are as follows:

1) *Extension to RADON models for deployment optimization and architecture decomposition.* Additional TOSCA definitions are needed to support the features of the decomposition tool. We have integrated these definitions into the RADON modeling profile and applied them to several examples from the demo applications and use cases.
2) *Development of the tool prototype with an architecture decomposition capability.* A prototype of the decomposition tool is made available on a public access server. We have enabled this prototype to generate a decomposition solution to a monolithic or microservice application.
3) *Research on performance prediction techniques in support of accuracy enhancement.* We have developed specialized methods to increase the accuracy of the decomposition tool in performance prediction by means of advanced cold-start modeling, model parameter learning from trace data, and solution algorithms available in the LINE[1] backend used by the tool.

1.3 Tool Requirements

The following table summarizes RADON requirements for the decomposition tool defined in deliverable *D2.2 Final requirements* at M18.

Table 1.1: The RADON requirements for the decomposition tool.

Id	Requirement Title	Priority
R-T3.2-1	Given a monolithic RADON model, the DECOMP_TOOL should be able to generate a coarse-grained RADON model.	Should have
R-T3.2-2	Given a coarse-grained RADON model, the DECOMP_TOOL should be able to generate a fine-grained RADON model.	Should have
R-T3.2-4	Given a platform-specific RADON model, the DECOMP_TOOL must be able to obtain an optimal deployment scheme that minimizes the operating costs on the target cloud platform under the performance requirements.	Must have
R-T3.2-5	Given a deployable RADON model, the DECOMP_TOOL could be able to refine certain properties of the nodes and relationships using runtime monitoring data.	Could have
R-T3.2-6	The DECOMP_TOOL should be able to carry out architecture decomposition, deployment optimization and accuracy enhancement for a mixed-grained RADON model.	Should have

[1] http://line-solver.sourceforge.net/

R-T3.2-7	The DECOMP_TOOL should be able to carry out architecture decomposition, deployment optimization and accuracy enhancement for a RADON model with heterogeneous cloud technologies.	Should have
R-T3.2-8	The DECOMP_TOOL should be able to carry out architecture decomposition, deployment optimization and accuracy enhancement for a RADON model across multiple cloud platforms.	Should have
R-T3.2-9	The DECOMP_TOOL should be able to allow the option of specifying the granularity level for architecture decomposition and generate a grained RADON model at that level.	Should have
R-T3.2-10	The DECOMP_TOOL should be able to allow the option of specifying the solution method for deployment optimization and obtain the optimal deployment scheme with that method.	Could have
R-T3.2-11	The DECOMP_TOOL should be able to allow the option of specifying the time limit for deployment optimization and return a sub-optimal deployment scheme upon timeout.	Should have
R-T3.2-12	Given a space of possible RADON models, the tool could compute an optimal RADON model with respect to CDL constraints. The computation for this may take place offline.	Could have
R-T3.2-13	Given a compliant sub-optimal RADON model, the tool could provide suggestions, which would improve its score with respect to the CDL soft constraints, while keeping the change to the original RADON model as small as possible. This computation should be fast enough to be used by a user interactively.	Could have

1.4 Structure of the Document

The rest of this deliverable is structured as follows. **Section 2** describes TOSCA types extended to support deployment optimization and architecture decomposition along with examples that make use of them. **Section 3** presents the initial approaches to coarse-grained and fine-grained decompositions as well as the implementation of the architecture decomposition feature. **Section 4** discusses advanced methods proposed to predict the performance of a RADON application more accurately. **Section 5** concludes the deliverable and outlines future work. **Appendices** provide additional information on the work done in the earlier sections.

2. Extension to RADON Models

The decomposition tool implements model-driven approaches to deployment optimization and architecture decomposition based on LQNs, a canonical form of extended queueing networks developed for modeling systems with nested simultaneous resource possession. To apply these approaches, the RADON modeling profile is extended to support the following features:

- Incorporation of modeling constructs provided by LQNs;
- Specification of reference workloads, either open or closed;
- Specification of performance requirements, e.g. the maximum response time;
- Definition of third-party services, e.g. Google Maps Directions;
- Definition of monolithic and microservice applications.

2.1 Data Types

Extra data types are defined to incorporate LQN constructs, i.e. entries, activities, precedences and requests. They are essential for the tool-specific extension to node, relationship and policy types. An introduction to LQNs and details about the QoS models for RADON applications are available in deliverable *D3.2 Decomposition Tool I*. This section only focuses on the TOSCA definitions of the related data types.

2.1.1 Random Variable

LQNs use non-negative random variables to express temporal information. To represent a non-negative random variable, we define the *RandomVariable* data type as shown in Listing 2.1. One can specify the *mean* and *scv* of a *RandomVariable*, i.e. its expectation and squared coefficient of variation (the variance divided by the squared mean). The second property is only reserved for now due to the limitation of the underlying LQN solver invoked by the decomposition tool. Notably, a zero *mean* implies a zero *scv* and therefore should be also avoided. A workaround is to replace zero with a small value like 0.001.

Listing 2.1: The data type for representing a non-negative random variable.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.RandomVariable:
    derived_from: tosca.datatypes.Root
    properties:
      mean:
        type: float
        required: true
        default: 1.0
        constraints:
```

```
        - greater_or_equal: 0.0
    scv:
      type: float
      required: false
      constraints:
        - greater_or_equal: 0.0
```

2.1.2 Entry

We model each node of a RADON application as a task, which contains a set of entries that describe the behavior of the node. The *Entry* data type shown in Listing 2.2 is defined for representing an entry of a node. This data type allows the specification of *activities* comprising the entry and *precedences* connecting them. The former is a map of *Activities* (see Section 2.1.3) while the latter is a list of *Precedences* (see Section 2.1.4).

Listing 2.2: The data type for representing an entry of a node.

```
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
```

2.1.3 Activity

An entry consists of an activity graph. Listing 2.3 shows the *Activity* data type defined for representing an activity of an entry. The properties of an *Activity* are as follows:

- *service_time*, a *RandomVariable* specifying the service time of the activity;
- *bound_to_entry*, a boolean indicating whether the activity is the first one to be executed by the entry upon receipt of a request;
- *replies_to_entry*, a boolean indicating whether the entry sends back a response upon completion of the activity (used only when the entry accepts synchronous requests);

- *request_pattern*, a string specifying the request pattern of the activity.

Listing 2.3: The data type for representing an activity of an entry.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Activity:
    derived_from: tosca.datatypes.Root
    properties:
      service_time:
        type: radon.datatypes.RandomVariable
        required: false
      bound_to_entry:
        type: boolean
        required: false
      replies_to_entry:
        type: boolean
        required: false
      request_pattern:
        type: string
        required: false
        constraints:
          - valid_values: [ stochastic, deterministic ]
```

2.1.4 Precedence

The activities of an entry are connected by precedences. To represent a precedence among activities, we define the *Precedence* data type as shown in Listing 2.4. A *Precedence* has the following properties:

- *type*, a string specifying the type of the precedence;
- *pre_activities*, a list of strings specifying the pre-activities of the precedence;
- *post_activities*, a list of strings specifying the post-activities of the precedence;
- *parameters*, a list of floats specifying the parameters of the precedence.

Table 2.1 summarizes the valid settings for these properties.

Listing 2.4: The data type for representing a precedence among activities.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Precedence:
    derived_from: tosca.datatypes.Root
    properties:
```

```
    type:
      type: string
      required: true
      default: sequence
      constraints:
        - valid_values: [ sequence, and-fork, and-join, or-fork, or-join, loop ]
    pre_activities:
      type: list
      required: true
      entry_schema:
        type: string
    post_activities:
      type: list
      required: true
      entry_schema:
        type: string
    parameters:
      type: list
      required: false
      entry_schema:
        type: float
        constraints:
          - greater_or_equal: 0.0
```

Table 2.1: The valid settings for the properties of a *Precedence*.

Type	Pre-Activities	Post-Activities	Parameter Number	Parameter Definition
sequence	single	single	0	NA
and-fork	single	multiple	0	NA
and-join	multiple	single	1	Number of pre-activities that must be completed before the join.
or-fork	single	multiple	*n*	Probability of each post-activity being selected the next one to execute
or-join	multiple	single	0	NA
loop	single	multiple	*n*−1	Number of times that each of the first *n*−1 post-activities is executed

Note: *n* denotes the number of post-activities.

2.1.5 Interaction

Two nodes may interact with each other based on their relationships. LQNs allow an activity of a task to send requests to an entry of another task. We describe interactions between nodes by applying the concept of requests to relationships. The *Interaction* data type shown in Listing 2.5 is defined for representing an interaction of a relationship. The properties of an *Interaction* are as follows:

- *type*, a string specifying the type of the interaction;
- *source_activity*, a string specifying the source activity of the interaction;
- *target_entry*, a string specifying the target entry of the interaction;
- *number_of_requests*, a float specifying the number of requests through the interaction;
- *network_delay*, a *RandomVariable* specifying the network delay of the interaction.

Listing 2.5: The data type for representing an interaction of a relationship.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.Interaction:
    derived_from: tosca.datatypes.Root
    properties:
      type:
        type: string
        required: true
        default: synchronous
        constraints:
          - valid_values: [ synchronous, asynchronous ]
      source_activity:
        type: string
        required: true
      target_entry:
        type: string
        required: true
      number_of_requests:
        type: float
        required: true
        default: 1.0
        constraints:
          - greater_or_equal: 0.0
      network_delay:
        type: radon.datatypes.RandomVariable
        required: false
```

2.2 Node Types

To generate an LQN from a RADON model, the decomposition tool transforms each node that composes the application into a task. The behavior of the node is described by a set of *entries*, which can be specified through an optional property. Since nodes of the same abstract type behave similarly across different cloud platforms, the *entries* property is introduced at the abstract level.

2.2.1 Reference Workload

Listing 2.6 shows the *Workload* node type defined for representing a generic workload. The *entries* of a *Workload* are of special data type *workload.Entries*. As shown in Listing 2.7, this data type restricts a *Workload* to a single *Entry* named *start*. It is possible for a *Workload* to *Triggers* any *Invocable* node, e.g. a *Function*, or *ConnectsTo* any *Endpoint* node, e.g. an *ObjectStorage*. To represent a specific workload, we derive two child node types, *OpenWorkload* and *ClosedWorkload*, as shown in Listings 2.8 and 2.9. One can specify the *interarrival_time* of an *OpenWorkload* while the *population* and *think_time* of a *ClosedWorkload*. In particular, the *interarrival_time* and the *think_time* are both *RandomVariables*.

Listing 2.6: The abstract node type for representing a generic workload.

```
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.Triggers
          occurrences: [ 0, UNBOUNDED ]
      - endpoint:
          capability: tosca.capabilities.Endpoint
          relationship: radon.relationships.ConnectsTo
          occurrences: [ 0, UNBOUNDED ]
```

Listing 2.7: The data type for representing the entries of a generic workload.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.workload.Entries:
    derived_from: tosca.datatypes.Root
    properties:
      start:
        type: radon.datatypes.Entry
        required: true
```

Listing 2.8: The abstract node type for representing an open 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 2.9: The abstract node type for representing a closed 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
```

2.2.2 Serverless Function

The extended *Function* node type for representing a serverless function is shown in Listing 2.10. To restrict a *Function* to a single *Entry* named *execute*, we define the *entries* of the *Function* to be of special data type *function.Entries*, as shown in Listing 2.11.

Listing 2.10: The abstract node type for representing a serverless function.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
```

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

Listing 2.11: The data type for representing the entries of a serverless function.

```
tosca_definitions_version: tosca_simple_yaml_1_3
data_types:
  radon.datatypes.function.Entries:
    derived_from: tosca.datatypes.Root
    properties:
      execute:
        type: radon.datatypes.Entry
        required: true
```

2.2.3 Object Storage

Listing 2.12 shows the extended *ObjectStorage* node type for representing an object storage. The *entries* of an *ObjectStorage* are defined to be a map of *Entries*. However, the name of each *Entry* must be prefixed with the name of the operation that it is associated with, e.g. “get”, “get_image”

and “getImage”. This enables the decomposition tool to compute the operating cost of the *ObjectStorage* for deployment optimization. The supported operations currently include GET, PUT and DELETE.

Listing 2.12: The abstract node type for representing an object storage.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.ObjectStorage:
    derived_from: tosca.nodes.Storage.ObjectStorage
    properties:
      entries:
        type: map
        required: false
        entry_schema:
          type: radon.datatypes.Entry
    requirements:
      - host:
          capability: tosca.capabilities.Container
          node: radon.nodes.abstract.CloudPlatform
          relationship: tosca.relationships.HostedOn
          occurrences: [ 1, 1 ]
      - invoker:
          capability: radon.capabilities.Invocable
          node: radon.nodes.abstract.Function
          relationship: radon.relationships.Triggers
          occurrences: [ 0, UNBOUNDED ]
```

2.2.4 Third-Party Service

To represent a third-party service, we define a new node type called *Service*. This node type is inspired by the ATC use case, where an AWS Lambda function needs to access Google Maps Directions. As shown in Listing 2.13, a *Service* can have an arbitrary map of *Entry* and is an *Endpoint* node that other nodes, e.g. a *Function*, may *ConnectsTo*.

Listing 2.13: The abstract node type for representing a third-party service.

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

```
        type: string
        required: false
      entries:
        type: map
        required: false
        entry_schema:
          type: radon.datatypes.Entry
    capabilities:
      endpoint_service:
        type: tosca.capabilities.Endpoint
        occurrences: [ 1, UNBOUNDED ]
```

2.2.5 Monolithic Application

Listing 2.14 shows the node type for representing a monolithic application, which is the normative *WebApplication* node type extended with two properties: *granularity* and *entries*. The former specifies the decomposition granularity while the latter allows an arbitrary map of *Entry*. To represent the server of monolithic applications, we also extend the normative *WebServer* node type, as shown in Listing 2.15.

Listing 2.14: The abstract node type for representing a monolithic application.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.WebApplication:
    derived_from: tosca.nodes.WebApplication
    properties:
      name:
        type: string
        required: false
      granularity:
        type: string
        required: false
        constraints:
          - valid_values: [ monolithic, coarse-grained, fine-grained ]
      entries:
        type: map
        required: false
        entry_schema:
          type: radon.datatypes.Entry
```

Listing 2.15: The abstract node type for representing the server of monolithic applications.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.WebServer:
    derived_from: tosca.nodes.WebServer
    properties:
      name:
        type: string
        required: false
```

2.2.6 Microservice Application

The *ContainerApplication* and *ContainerRuntime* node types shown in Listings 2.16 and 2.17 are defined for representing a microservice application and the runtime of microservice applications respectively. They are derived from the normative *Container.Application* and *Container.Runtime* node types. Similarly to the case of a *WebApplication*, two additional properties, *granularity* and *entries*, are available for a *ContainerApplication*.

Listing 2.16: The abstract node type for representing a microservice application.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.ContainerApplication:
    derived_from: tosca.nodes.Container.Application
    properties:
      name:
        type: string
        required: false
      granularity:
        type: string
        required: false
        constraints:
          - valid_values: [ coarse-grained, fine-grained ]
      entries:
        type: map
        required: false
        entry_schema:
          type: radon.datatypes.Entry
```

Listing 2.17: The abstract node type for representing the runtime of microservice applications.

```
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  radon.nodes.abstract.ContainerRuntime:
```

```
    derived_from: tosca.nodes.Container.Runtime
    properties:
      name:
        type: string
        required: false
```

2.3 Relationship Types

A relationship may realize interactions between its source and target nodes. When generating an LQN from a RADON model, the decomposition tool transforms such a relationship into a task that forwards requests between those resulting from the source and target nodes of the relationship. Once can specify the *interactions* of a relationship through an optional property. As shown in Listings 2.18 and 2.19, we introduce the *interactions* property into two relationship types, *Triggers* and *ConnectsTo*, which are used for representing a function trigger and a network connection respectively at either abstract or concrete level. The *interactions* of a *Triggers* or *ConnectsTo* are defined to be a list of *Interaction*s.

Listing 2.18: The relationship type for representing a function trigger.

```
tosca_definitions_version: tosca_simple_yaml_1_3
relationship_types:
  radon.relationships.Triggers:
    derived_from: tosca.relationships.ConnectsTo
    properties:
      events:
        type: list
        required: false
        entry_schema:
          type: radon.datatypes.Event
      interactions:
        type: list
        required: false
        entry_schema:
          type: radon.datatypes.Interaction
    valid_target_types: [ radon.capabilities.Invocable ]
```

Listing 2.19: The relationship type for representing a network connection.

```
tosca_definitions_version: tosca_simple_yaml_1_3
relationship_types:
  radon.relationships.ConnectsTo:
    derived_from: tosca.relationships.ConnectsTo
```

```
    properties:
      interactions:
        type: list
        required: false
        entry_schema:
          type: radon.datatypes.Interaction
```

2.4 Policy Types

To represent a generic performance requirement, we extend the normative *Performance* policy type as shown in Listing 2.20. This policy type enables the specification of *target_entries* that the policy applies to as well as *upper_bound* and *lower_bound* that the performance measure should satisfy. Listings 2.21 and 2.22 show two child policy types, *MeanResponseTime* and *MeanTotalResponseTime*, which are derived for representing performance requirements on the mean response time and the mean total response time respectively. The main difference between these two policy types is that the former applies to each target node or entry individually while the latter applies to all the target nodes or entries together. 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 considered to be the most important performance measure of an application.

Listing 2.20: The policy type for representing a generic performance requirement.

```
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 2.21: The policy type for representing a performance requirement on the mean response time.

```
tosca_definitions_version: tosca_simple_yaml_1_3
```

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

Listing 2.22: The policy type for representing a performance requirement on the mean total response time.

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

2.5 Examples

Appendices A and B give two examples of RADON models, *Thumbnail Generation* and *Routes Calculator*, which apply the aforementioned extension to deployment optimization. Particularly, the *AwsLambdaFunction* and *AwsS3Bucket* node types used in these two examples are the childs of the *Function* and *ObjectStorage* node types respectively. The *Thumbnail Generation* example represents a RADON demo application that creates a thumbnail for each uploaded image. The *Routes Calculator* example arises from the ATC use case. It represents a typical FaaS-based application that invokes Google Map Directions to find the feasible routes from one position to another.

The two examples of RADON models given in Appendices C and D apply the aforementioned extension to architecture decomposition. They represent a monolithic and a microservice application respectively. According to the *granularity* and *entries* properties specified in the first example, the monolithic application will be decomposed into microservices that together provide the same set of entries. As for the second example, the decomposition tool will refactor the microservice application at the function level and meanwhile preserve the execution logic of each entry.

3. Architecture Decomposition

The initial approaches to coarse-grained and fine-grained decompositions are described in this section. We have implemented these approaches in the prototype of the decomposition tool to provide the architecture decomposition feature. This section also reports details about the implementation of the new feature.

3.1 Coarse-Grained Decomposition

Coarse-grained decomposition aims at refactoring a monolithic application into microservices. To reuse the available quality-of-service models, our approach to coarse-grained decomposition identifies microservices by analyzing the interface of the monolithic application.

3.1.1 Architecture Refactoring Pattern

The architecture refactoring pattern for coarse-grained decomposition is shown in Figure 3.1. We consider the interface of a monolithic application as defining a set of operations, e.g. {op_*i*}, which can be divided into subsets of correlated operations, e.g. {op_1, op_2, op_3}, {op_4, op_5, op_6, op_7}, etc. Correlated operations often share data and code. To reduce communications between the resultant microservices and increase their maintainability, the monolithic application can be decomposed in such a way that each microservice implements a subset of correlated operations. This also preserves the entire set of operations, thus avoiding any modifications on the client side after the decomposition.

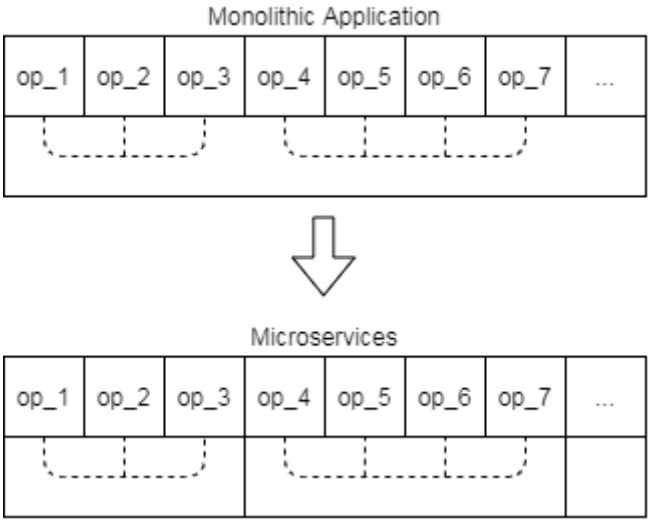


Figure 3.1: The architecture refactoring pattern for coarse-grained decomposition.

3.1.2 Model-to-Model Transformation

As mentioned earlier in Sections 2.2.5 and 2.2.6, the *WebApplication* and *ContainerApplication* node types are used for representing monolithic and microservice applications respectively. The *entries* of these node types refer to operations defined by the application interface. Figure 3.2 shows the model-to-model transformation obtained by applying the architecture refactoring pattern for coarse-grained decomposition.

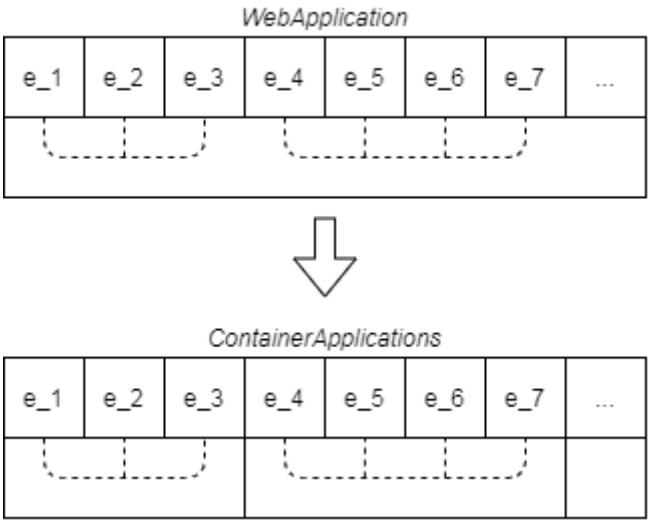


Figure 3.2: The model-to-model transformation for coarse-grained decomposition.

3.1.3 Clustering of Correlated Entries

The key to our coarse-grained decomposition approach is the clustering algorithm for correlated entries, which originates from the idea proposed in [Bar17m]. As shown in Algorithm 3.1, the clustering of correlated entries consists of five steps:

- lines 1–4, for each entry extract terms from the entry name and put them into *entryTermsMap*;
- lines 5–8, find all the concepts no lower than *BASE_CONCEPT_LEVEL* in *vocabularyTree* and add them into *conceptList*;
- lines 9–18, for each entry find the concept in *conceptList* that has the maximum similarity and put it into *closestConceptMap*;
- lines 19–31, for each entry find the concept at *BASE_CONCEPT_LEVEL* that corresponds to the most similar one and put it into *closestConceptMap*;
- lines 32–40, for each concept in *closestConceptMap* add matching entries into a list and put that list into *entryListMap*.

A list in *entryListMap* collects entries that are semantically similar to the same concept and therefore can be considered as a subset of correlated entries. In particular, the granularity of the clustering is determined by *BASE_CONCEPT_LEVEL*.

Algorithm 3.1: The clustering algorithm for correlated entries.

```
1:  entryTermsMap = createEmptyMap()
2:  for entry in webApplication.entryList
3:      entryTerms = extractNameTerms(entry.name)
4:      entryTermsMap.put(entry.name, entryTerms)
5:  conceptList = createEmptyList()
6:  for concept in vocabularyTree.conceptList
7:      if concept.level >= BASE_CONCEPT_LEVEL
8:          conceptList.append(concept)
9:  closestConceptMap = createEmptyMap()
10: for entry in webApplication.entryList
11:     entryTerms = entryTermsMap.get(entry.name)
12:     maxSimilarity = -1.0;
13:     for concept in conceptList
14:         conceptTerms = concept.terms
15:         similarity = similarityAnalyzer.compare(entryTerms, conceptTerms)
16:         if similarity > maxSimilarity
17:             maxSimilarity = similarity
18:             closestConceptMap.put(entry.name, concept)
19: for entry in webApplication.entryList
20:     entryTerms = entryTermsMap.get(entry.name)
21:     concept = closestConceptMap.get(entry.name)
22:     while concept.level > BASE_CONCEPT_LEVEL
23:         superConceptList = vocabularyTree.getPredecessors(concept)
24:         maxSimilarity = -1.0;
25:         for concept in superConceptList
26:             conceptTerms = concept.terms
27:             similarity = similarityAnalyzer.compare(entryTerms, conceptTerms)
28:             if similarity > maxSimilarity
29:                 maxSimilarity = similarity
30:                 closestConceptMap.put(entry.name, concept)
31:         concept = closestConceptMap.get(entry.name)
32: entryListMap = createEmptyMap()
33: for entry in webApplication.entryList
34:     concept = closestConceptMap.get(entry.name)
35:     if !entryListMap.isKey(concept.name)
36:         entryList = createEmptyList()
37:         entryListMap.put(concept.name, entryList)
38:     else
39:         entryList = entryListMap.get(concept.name)
40:     entryList.append(entry)
```

3.2 Fine-Grained Decomposition

Fine-grained decomposition in this section means refactoring a microservice application into serverless functions. However, our approaches to coares-fined and fine-grained decompositions can be combined to carry out the complete decomposition of a monolithic application in two steps.

3.2.1 Architecture Refactoring Pattern

We adopt the architecture refactoring pattern shown in Figure 3.3 for fine-grained decomposition. The interface of a microservice application is considered as defining a set of correlated operations, e.g. {op_*i*}. An intuitive way to decompose the microservice application is to implement each of these operations using a serverless function. This is in fact not enough due to the inability of serverless functions to store data shared by correlated operations. We thus introduce an extra object storage, which acts as a network file system for the resultant serverless functions to access.

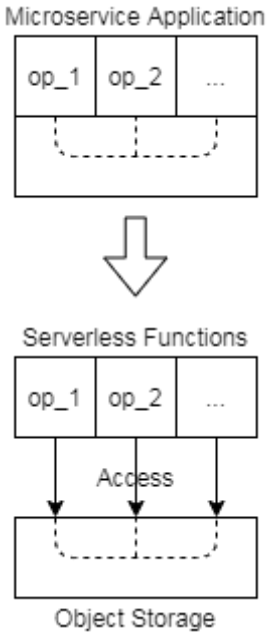


Figure 3.3: The architecture refactoring pattern for fine-grained decomposition.

3.2.2 Model-to-Model Transformation

The model-to-model transformation shown in Figure 3.4 is obtained by applying the architecture refactoring pattern for fine-grained decomposition. We use the *Function*, *ObjectStorage* and *ContainerApplication* node types present in Sections 2.1.2, 2.1.3 and 2.1.6 for representing serverless functions, object storages and microservice applications respectively. Again, the *entries* of these node types correspond to operations defined by the application interface.

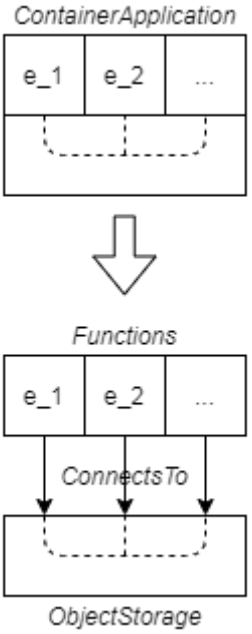


Figure 3.4: The model-to-model transformation for fine-grained decomposition.

3.2.3 Restructuring of Activity Graphs

One issue of the architecture refactoring pattern for fine-grained decomposition lies in the difficulty of maintaining code shared between correlated operations. Any modification to the shared code needs to be mirrored across all serverless functions implementing the operations. Given that each operation is modeled as an entry, which consists of an activity graph for representing the execution logic of that operation, a possible solution could be to introduce extra annotation for specifying activities that execute the same code and create an auxiliary function to implement that code. However, this will increase the total operating cost of the decomposition solution as the execution time of auxiliary functions is counted twice in the computation of the total operating cost.

3.3 Feature Implementation

This section elaborates the implementation of the architecture decomposition feature in the prototype of the decomposition tool. We first introduce the overall approach to deployment optimization and then discuss essential techniques applied in the implementation.

3.3.1 Overall Approach

The implementation of the architecture decomposition feature is based on external tools and data structures shown in Figure 3.5. Given a RADON model, the tool uses a built-in YAML processor to import the service template into MATLAB and generates a topology graph automatically through model-to-model transformation. A decomposition problem is then created from the

topology graph and solved by invoking the Schema.org[2] dataset and the DKPro Similarity[3] tool. When the decomposition solution is found, the tool writes the result back into the original service template.

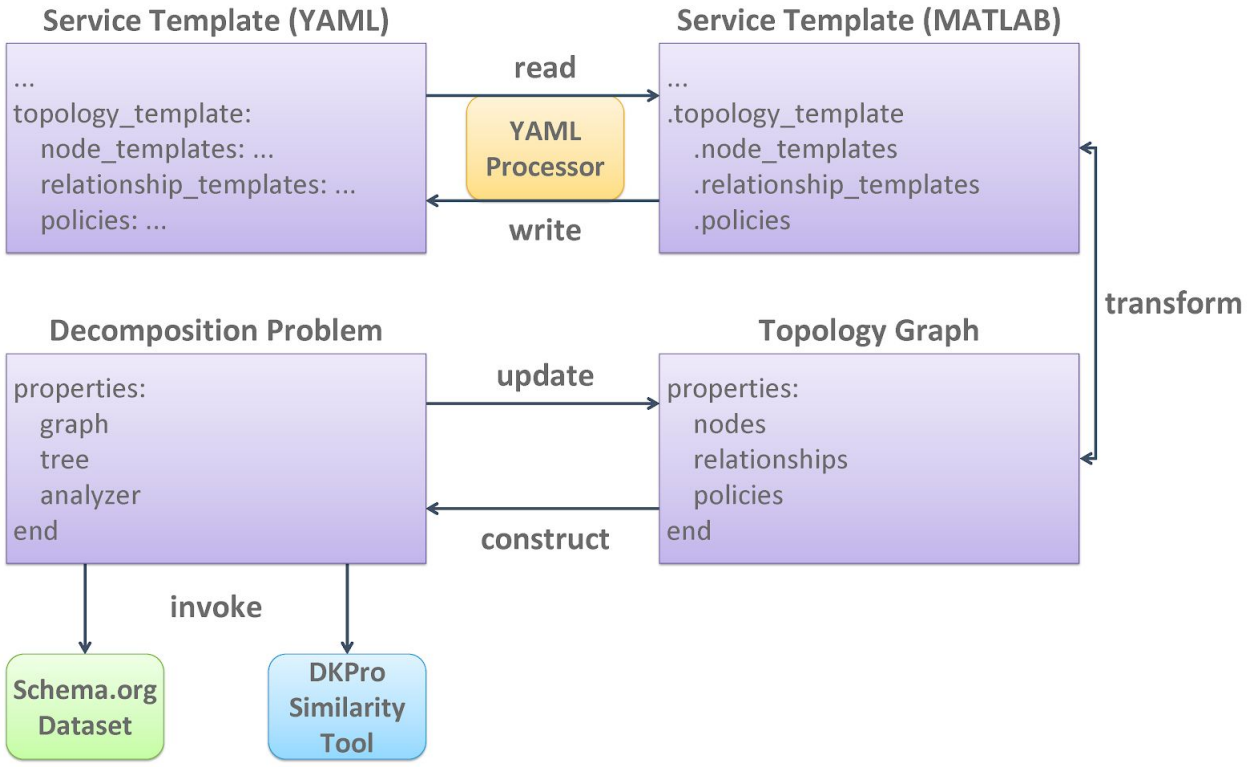


Figure 3.5: The overall approach to architecture decomposition.

3.3.2 Term Extraction with MATLAB

Four steps are performed to extract terms from an entry or concept name: word separation, digit word removal, stop word removal and word lemmatization. We consider names to be a sequence of letters, digits and underscores starting with a letter and implement a simple procedure to separate words in a given name. Table 3.1 summarizes the rules applied by this word separation procedure. The rest three steps are realized using standard methods provided by the Text Analytics Toolbox[4] of MATLAB. Particularly, lemmatization reduces inflected words to their dictionary form for the ease of further analysis.

Table 3.1: The rules applied by the word separation procedure.

[2] https://schema.org/
[3] https://dkpro.github.io/dkpro-similarity/
[4] https://uk.mathworks.com/products/text-analytics.html

Id	Rule	Input	Output
1	Split when changing the case of letters	getUserName	get, User, Name
2	Split when switching between letters and digits	get0user0name	get, 0, user, 0, name
3	Split when encountering underscores	get_user_name	get, user, name

3.3.3 Construction of Vocabulary Tree

Vocabulary trees provide a hierarchy of concepts available in a vocabulary. To find correlated entries, we construct a vocabulary tree from Schema.org, a well-known vocabulary for creating, maintaining and promoting the schemas of structured data on the Internet, on web pages, in email messages and beyond. The structure of the vocabulary tree is discovered by traversing concepts in Schema.org through the breadth-first search. Meanwhile, we also record the level and extract the terms for each concept, which helps to reduce the computational overhead of the entry clustering algorithm.

3.3.4 Development of Similarity Analyzer

A similarity analyzer is developed by wrapping DKPro Similarity [Bär13d]. It aims to evaluate the similarity between terms extracted from a pair of entry and concept names. As part of the DKPro project, DKPro Similarity is an open-source tool for comparing words and segments. A number of state-of-the-art lexical and semantic similarity measures are implemented by DKPro Similarity. The similarity analyzer of the decomposition tool uses a similarity measure computed through the explicit semantic analysis of word occurrences on Wikipedia.

3.3.5 Manipulation of Topology Graph

The transformation from the original model to the decomposed one requires a heavy manipulation of the topology graph. Thus, we implement a rich set of methods for manipulating a topology graph. These methods are summarized in Table 3.2.

Table 3.2: The methods for manipulating a topology graph.

Id	Method	Description
1	nodes = findNodes(self, className)	Find the nodes with the given class name in this graph
2	relationships = findIngressRelationships(self, node)	Find the ingress relationships of a node in this graph
3	relationships = findEgressRelationships(self, node)	Find the egress relationships of a node in this graph

4	policies = findAttachedPolicies(self, node)	Find the attached policies of a node in this graph
5	addNode(self, node)	Add a node to this graph
6	removeNode(self, node)	Remove a node from this graph
7	addRelationship(self, relationship, sourceNode, targetNode)	Add a relationship to this graph
8	removeRelationship(self, relationship)	Remove a relationship from this graph
9	addPolicy(self, policy)	Add a policy to this graph
10	removePolicy(self, policy)	Remove a policy from this graph

4. Accuracy Enhancement

In this section, we outline three developments in period 2 that support the third and final feature of the decomposition tool, i.e., the ability to enhance the accuracy of performance predictions. This feature has been achieved along three dimensions:

- Offering more expensive, but also more accurate, predictive models capable of taking into account advanced features of serverless platforms such as cold-starts.
- Offering advanced methods to analyze the models provided by the decomposition tool to its backend, by means of rigorous queueing theoretic approximations.
- Offering the possibility to integrate in a microservices architecture model on-the-fly estimation of service demands from performance data obtained through monitoring.

At this stage, these three capabilities have been implemented in the LINE backend. We will move in the final period to integrate these options in the framework feature as part of the general integration activities of the project.

4.1 Enhancing performance models for serverless functions

The earlier version of this deliverable (D3.2) proposed a methodology to optimize compute and memory capacity allocation for serverless FaaS function. The proposed approach relies on layered queueing modelling and queueing theory models. However, a limitation of the proposed approach lied in its ability to capture a function performance only after the initial invocations, once its performance has settled around the expected value.

We here present new developments that address this shortcoming, described in our work published in [GiaC20]. Specifically, the *cold-start* effect represents an inherent feature of serverless platforms. A cold start occurs when a function is called in a period during which the corresponding container is not yet loaded in memory. This adds a delay in sending the response needed to spin-up the container and to load the function runtime dependencies. The cold start issue can pose a significant trade-off between latency and memory allocation optimization. Despite hurting performance, the cold-start mechanism aims at reducing memory consumption by deallocating functions that are idle for a sufficiently long time, therefore it is advantageous from the provider point of view.

In the reporting period, we have investigated how to model by means of stochastic analysis techniques the notion of cold-start. They are particularly suitable to model hosted platforms, but they can also be applied to size a single application with minor modifications. We have investigated in particular two modelling approaches:

- *TTL-type modelling*. We can draw parallels between a FaaS platform and a Time to Live (TTL) cache. Similar to a FaaS platform, a TTL caching system also periodically offloads its cached objects. Thus, the cold start issue resembles the hit rate improvement problem of

a TTL cache, where one needs to decide on the optimal time to keep objects in the cache. Thus, analysis methods from TTL cache research may be in principle applicable also to FaaS sizing in order to estimate the required memory capacity. However, from our study in [GiaC20] we have identified two limitations of such an approach. First, contrary to TTL cache misses, the latency incurred by function cold start times can vary widely. Next, while a large fraction of TTL research considers equal-sized objects, a function consumes different amounts of memory depending on whether it is idle or not.

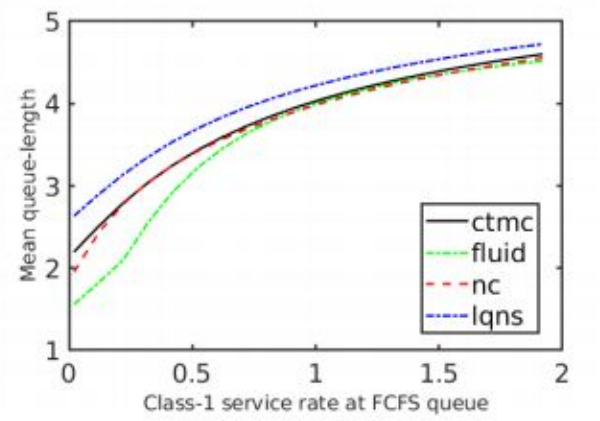
- *Setup modelling approach*. To address the lack of information on function latency that arises in TTL-type modelling, we have studied the application of so-called M/M/1/setup/delayedoff models, which are variant of multiserver queueing systems in which the servers that process the arriving jobs (in our case, the functions) require a setup time before being ready to execute. Such models are well understood analytically, as they can be solved using a class of numerical methods for Markov models called matrix-analytic methods. More details and examples of the Markov model state space are given in [GiaC20]. Overall, setup models can approximate the cold start probability for a function, taking into account cold starts. Our investigation reveals that this model can incur very little modelling errors, around 2.38% on average in capturing the latency of functions with cold-starts [GiaC20].

Summarizing, building on the work described in D3.2, we have developed novel techniques and methods to predict performance for serverless functions, demonstrating very high accuracy in challenging scenarios with cold starts. We point the interested reader to [GiaC20] for technical details and further examples of applicability of the technique to serverless provisioning.

4.2 Enhancing the accuracy of the decomposition tool backend

Among the enhancements achieved in year 2, we have improved the analysis techniques that our backend solver LINE adopts to analyze the queueing models created by the decomposition tool. Our recent work in this area is described in [Cas20]. In particular, we have now achieved approximations for queueing analysis of systems with FCFS buffers that improve the predictive capabilities of the state-of-the-art tool LQNS, which was earlier used to analyze microservices architectures and serverless functions. The new methodology relies on an approximate iterative method that combines a light-load and a heavy-load queueing approximation in a single formula to approximate queueing performance in arbitrary distributed systems.

Compared to an exact solution obtained with Continuous-time Markov chains (CTMCs), the proposed method integrated in LINE's NC solver is very accurate, frequently improving over the best existing techniques in the area such as LQNS or fluid-approximations. This is illustrated in the figure below, which shows the approximation accuracy in predicting the unfinished work at a resource as the speed of the resource is linearly increased.



For more details on this example and the overall technique approach we point the interested reader to our recent work in [Cas20].

4.3 Enhancing the model parameterization

In the earlier deliverable D3.2 we have presented a methodology based on utilization-based and response-time based linear regression that was found effective to parameterize microservices-based architectures. However, in that earlier deliverable the method was presented as-is, outside the concrete implementation as part of the decomposition tool backend. As demand estimation is the essential third feature of the decomposition tool in period 2 we have developed a complete integration of the method in the LINE backend. We here illustrate the extension.

Specifically, we have augmented the LINE backend with a new engine, called *ServiceEstimator*, which is able to automatically infer parameters required by the decomposition tool from an empirical dataset. For example, the following LINE code produces an automated initialization of the engine with a utilization-based regression estimator (ubr).

```
options = ServiceEstimator.defaultOptions;
options.method = 'ubr';
se = ServiceEstimator(model, options);
```

The *ServiceEstimator*, after initialization, is supplied a set of sample metrics, such as arrival rate (in LINE, mapped to a data-type *Metric.ArvR*) of requests and measure utilization of a particular microservice. The node and class of the requests that are characterized by this dataset can also be specified within our new extension. For example, the following snippet loads arrival rates and utilization data for two classes of requests processed at node 2:

```
lambda1 = SampledMetric(Metric.ArvR, ts, arvr1_samples, node{2}, jobclass{1});
lambda2 = SampledMetric(Metric.ArvR, ts2, arvr2_samples, node{2}, jobclass{2});
```

```
util1 = SampledMetric(Metric.Util, ts, util1_samples, node{2}, jobclass{1});
util = SampledMetric(Metric.Util, ts, util_samples, node{2});
se.addSamples(lambda1); se.addSamples(lambda2); se.addSamples(util); se.addSamples(util1);
```

Lastly, the novel *ServiceEstimator* is able to interpolate for missing data, ensuring that the input data is consistently available at all timestamps, and subsequently obtain the estimate of the resource demands at node 2 for the considered classes of requests. In code:

```
se.interpolate();
estVal = se.estimateAt(node{2})
```

Since the above backend extensions can be applied generically to any node in a microservices- or serverless-based architecture, it provides an easy-to-integrate capability in RADON.

Additional example of the above capabilities are available in the LINE example folder, in particular the *example_scvEstimation_*.m* files at: https://github.com/line-solver/line/tree/master/examples.

5. Conclusion and Future Work

In this deliverable, we describe the extension to RADON modeling profile in support of deployment optimization and architecture decomposition. This extension has been applied to several examples from the demo applications and use cases. We also present the initial approaches to coarse-grained and fine-grained decompositions, which have been implemented in the prototype of the decomposition tool to provide the architecture decomposition feature. In addition, three developments to enable the accuracy enhancement feature are outlined, involving the QoS models for serverless functions, the backend of the decomposition tool and the model parameterization methodology. Table 5.1 reports the achieved level of compliance to RADON requirements defined in deliverable *D2.2 Final requirements* at M18.

Table 5.1: The achieved level of compliance to RADON requirements (✘=left for next months, ✔= preliminary work, ✔✔= addressed but not yet completed, ✔✔✔=completed).

Id	Requirement Title	Priority	Level of Compliance
R-T3.2-1	Given a monolithic RADON model, the DECOMP_TOOL should be able to generate a coarse-grained RADON model.	Should have	✔✔
R-T3.2-2	Given a coarse-grained RADON model, the DECOMP_TOOL should be able to generate a fine-grained RADON model.	Should have	✔✔
R-T3.2-4	Given a platform-specific RADON model, the DECOMP_TOOL must be able to obtain an optimal deployment scheme that minimizes the operating costs on the target cloud platform under the performance requirements.	Must have	✔✔✔
R-T3.2-5	Given a deployable RADON model, the DECOMP_TOOL could be able to refine certain properties of the nodes and relationships using runtime monitoring data.	Could have	✔
R-T3.2-6	The DECOMP_TOOL should be able to carry out architecture decomposition, deployment optimization and accuracy enhancement for a mixed-grained RADON model.	Should have	✔✔
R-T3.2-7	The DECOMP_TOOL should be able to carry out architecture decomposition, deployment optimization and accuracy enhancement for a RADON model with heterogeneous cloud technologies.	Should have	✔✔✔
R-T3.2-8	The DECOMP_TOOL should be able to carry out architecture decomposition, deployment optimization and accuracy enhancement for a RADON model across multiple cloud platforms.	Should have	✔✔
R-T3.2-9	The DECOMP_TOOL should be able to allow the option of	Should have	✔✔✔

	specifying the granularity level for architecture decomposition and generate a grained RADON model at that level.		
R-T3.2-10	The DECOMP_TOOL should be able to allow the option of specifying the solution method for deployment optimization and obtain the optimal deployment scheme with that method.	Should have	✔✔✔
R-T3.2-11	The DECOMP_TOOL should be able to allow the option of specifying the time limit for deployment optimization and return a sub-optimal deployment scheme upon timeout.	Should have	✔✔
R-T3.2-12	Given a space of possible RADON models, the tool could compute an optimal RADON model with respect to CDL constraints. The computation for this may take place offline.	Could have	✘
R-T3.2-13	Given a compliant sub-optimal RADON model, the tool could provide suggestions, which would improve its score with respect to the CDL soft constraints, while keeping the change to the original RADON model as small as possible. This computation should be fast enough to be used by a user interactively.	Could have	✘

The final development of the decomposition tool is due at M27, by which all the features of the tool including architecture decomposition, deployment optimization and accuracy enhancement will be delivered. To this end, we will be primarily focusing on the following work:

- Extend the decomposition tool to support serverless functions and object storages on Google Cloud, and enable deployment optimization of a RADON model across AWS and Google Cloud.
- Improve the approaches to coarse-grained and fine-grained decompositions, and validate the architecture decomposition feature by applying it to RADON demo applications and use cases.
- Conduct collaboration with RADON partners to afford the decomposition tool the capability of invoking the continuous testing tool and the monitoring system to enhance the accuracy of performance annotations in a given RADON model.

References

[Bar17m] L. Baresi, M. Garriga and A. De Renzis. Microservices Identification through Interface Analysis. Proceedings of ESOCC 2017, Springer, September 2017.

[Bär13d] D. Bär, T. Zesch and I. Gurevych. DKPro Similarity: An Open Source Framework for Text Similarity. Proceedings of ACL 2013, ACL, August 2013.

[GiaC20] A. Gias, G. Casale. COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms. Proceedings of IEEE MASCOTS 2020, IEEE, November 2020.

[Cas20] G. Casale. Integrated Performance Evaluation of Extended Queueing Network Models with LINE. Proceedings of the 2020 Winter Simulation Conference, ACM, December 2020.

Appendix A: Example - Thumbnail Generation

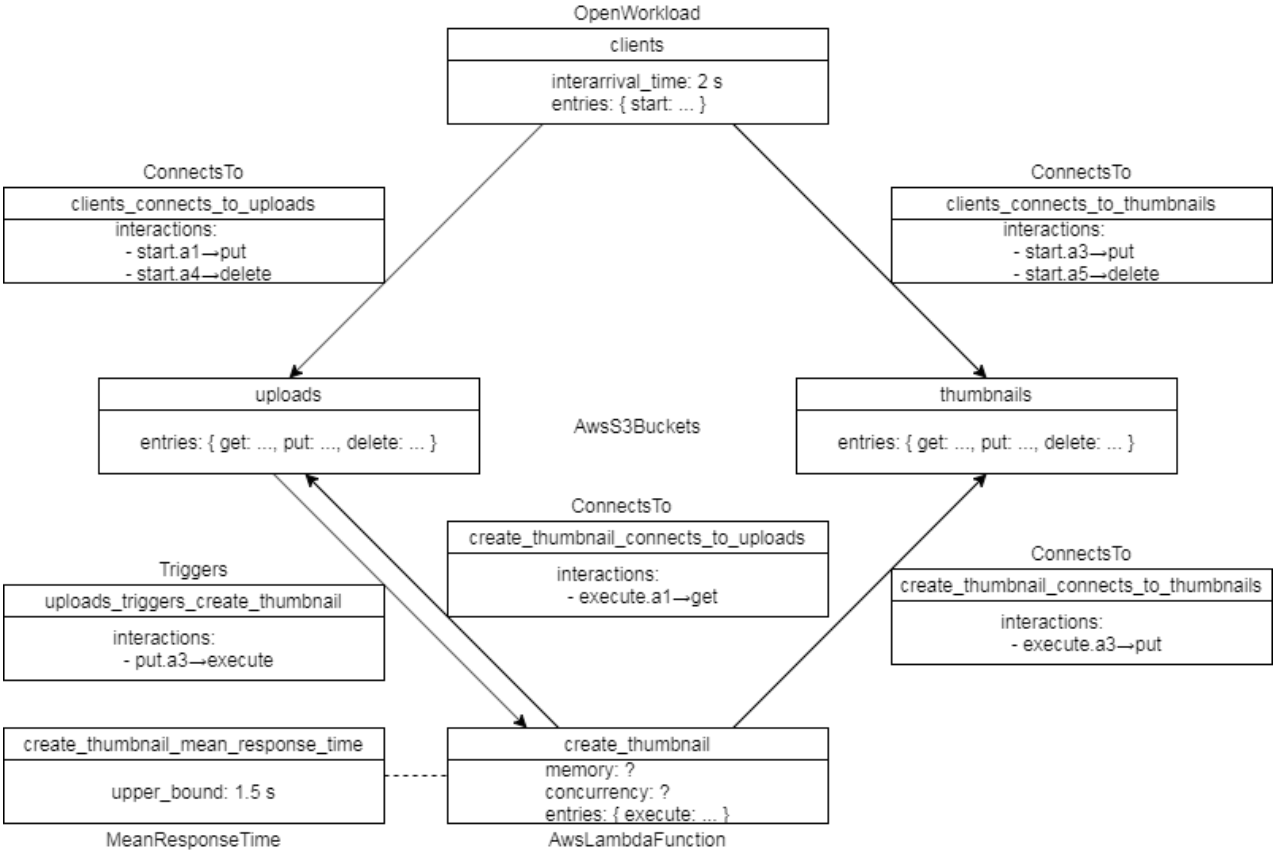


Figure A.1: The RADON model for *Thumbnail Generation*.

Appendix B: Example - Routes Calculator

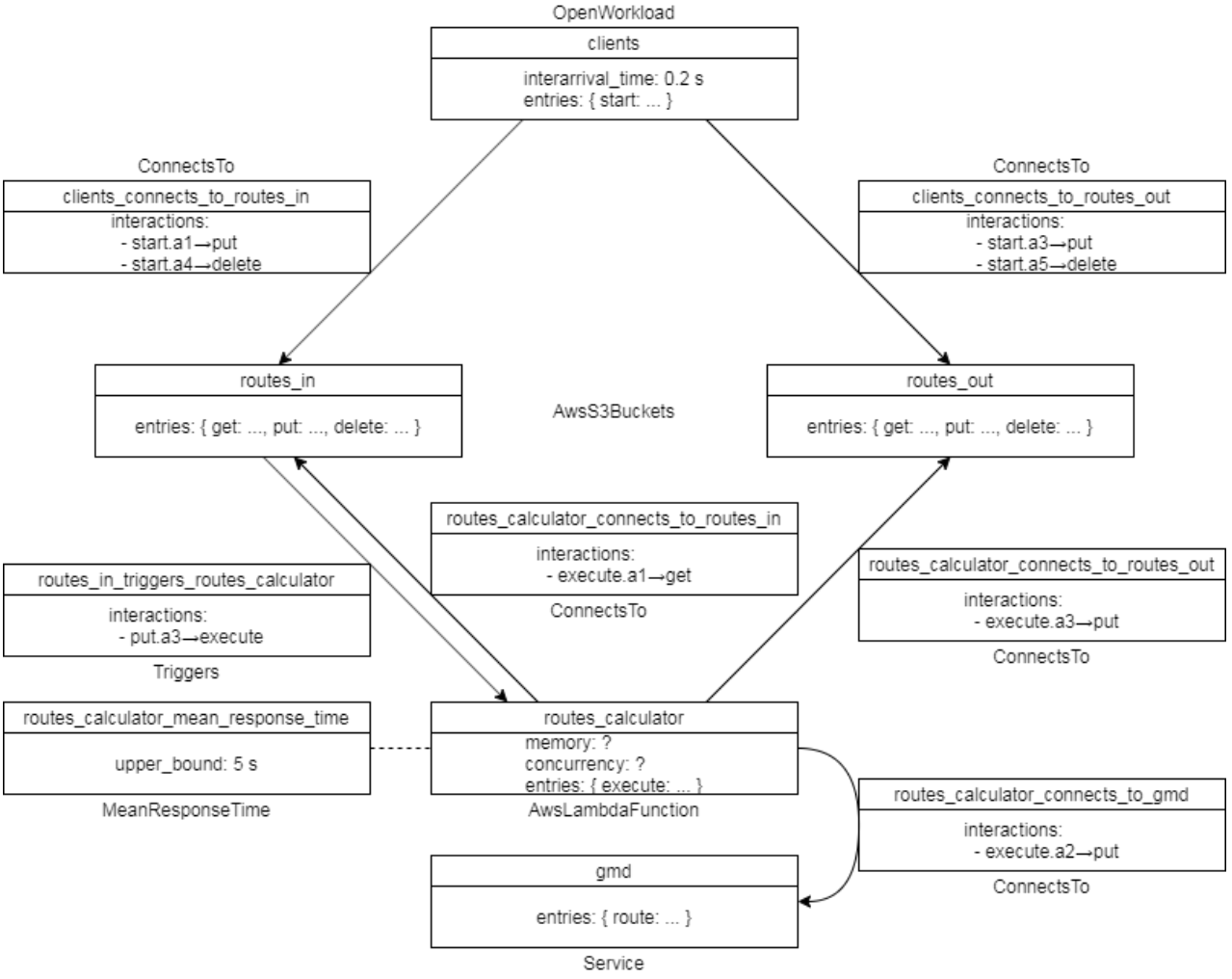


Figure B.1: The RADON model for *Routes Calculator*.

Appendix C: Example - Monolithic Application

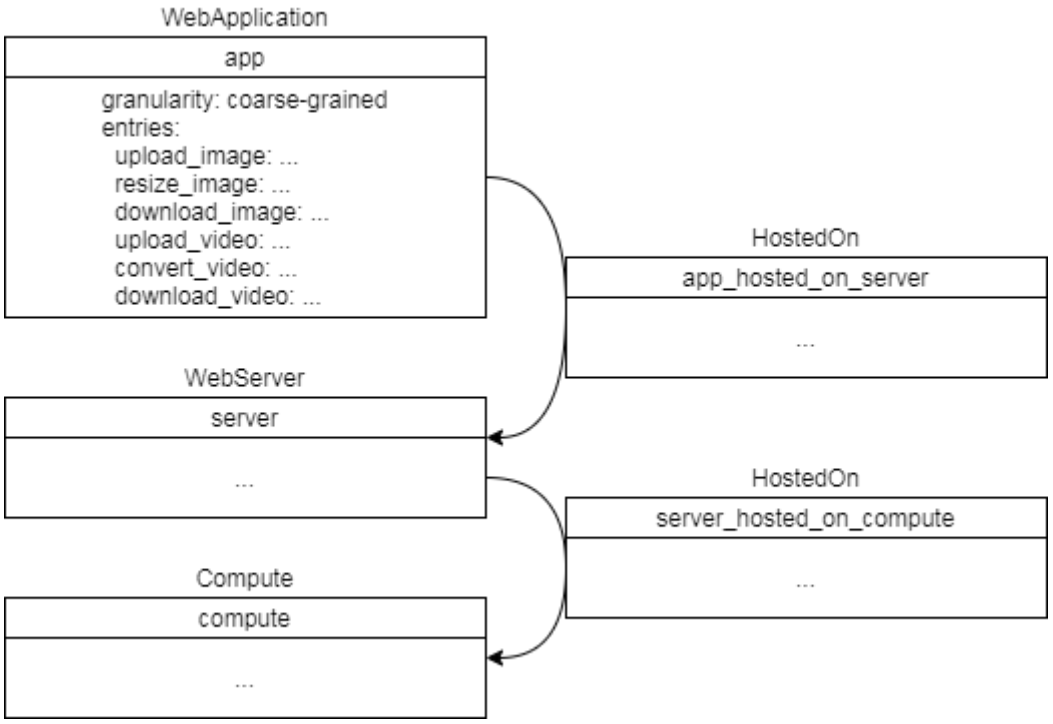


Figure C.1: The RADON model of a monolithic application.

Appendix D: Example - Microservice Application

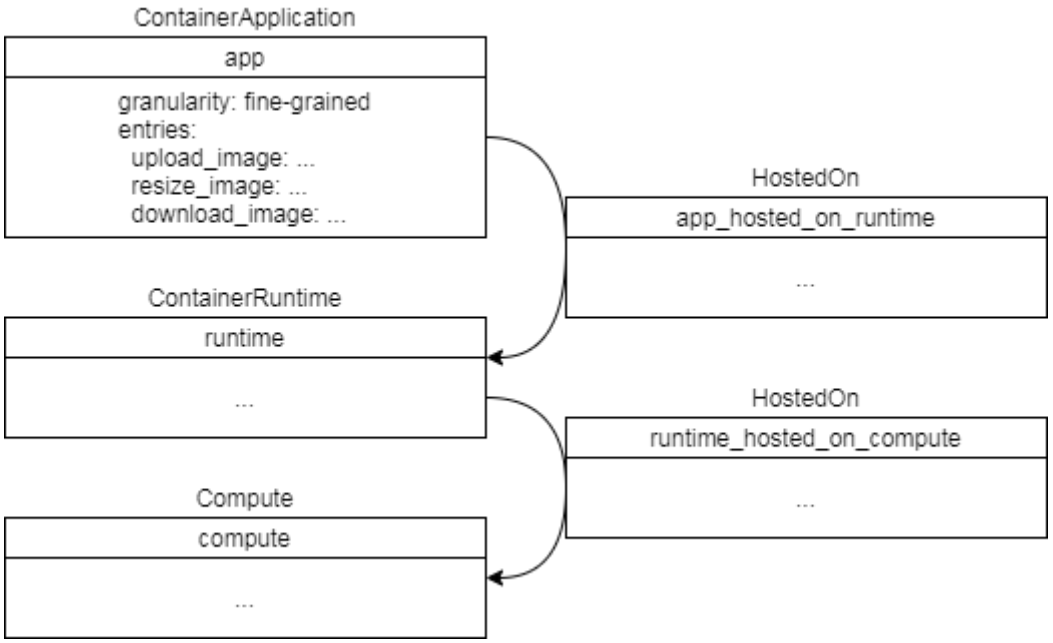


Figure D.1: The RADON model of a microservice application.