



H2020-ICT-2018-2-825040



Rational decomposition and orchestration for serverless computing

Deliverable 3.2

Decomposition Tool I

Version: 1.0

Publication Date: 23-December-2019

Disclaimer:

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

Deliverable Card

Deliverable	D3.2
Title:	Decomposition Tool I
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:	23-December-2019
Status:	Final
Dissemination level:	Public
Download page:	http://radon-h2020.eu/public-deliverables/
Copyright:	RADON consortium

The RADON project partners

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

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

Executive summary

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 the efforts that have been made in order to achieve the expected outcomes of the decomposition tool, including innovative research on QoS modeling of both microservices-based and FaaS-based applications. A prototype of the decomposition tool is implemented and demonstrated through an example for deployment optimization.

Glossary

AMVA	Approximate Mean Value Analysis
AWS	Amazon Web Services
DECOMP_TOOL	Decomposition Tool
FaaS	Function as a Service
LQN	Layered Queueing Network
LQNS	Layered Queueing Network Solver
LQSIM	Layered Queueing Network Simulator
QoS	Quality of Service
SOA	Service-Oriented Architecture
TPS	Transactions per Second

Table of contents

1. Introduction	7
1.1 Deliverable Objectives	7
1.2 Overview of Main Achievements	8
1.3 Decomposition Tool Requirements	8
1.4 Structure of the Document	10
2. QoS Modeling for Microservices-Based Architectures	11
2.1 Abstract Views	11
2.1.1 Sock Shop Application	11
2.2 Microservices QoS Modeling	12
2.3 Model Parameterization	14
2.4 Assessing QoS Prediction	15
2.5 Summary and Lessons Learned	16
3. QoS Modeling for FaaS-Based Applications	17
3.1 FaaS Compute Services	17
3.1.1 AWS Lambda and Google Cloud	17
3.1.2 Microsoft Azure	18
3.2 AWS Lambda Functions	18
3.2.1 Lambda Function Behavior	18
3.2.2 Event-Driven Invocation from S3 Buckets	19
3.3 Thumbnail Generation Example	19
3.4 FaaS QoS Modeling	20
3.5 Assessing QoS Prediction	23
3.6 Summary and Lessons Learned	25
4. Deployment Optimization	26
4.1 Problem Formulation	26
4.2 Solution Procedure	27
4.3 Tool Implementation	27
4.3.1 Overall Approach	28
4.3.2 RADON YAML Processor for MATLAB	28
4.3.3. Topology Graph and Optimization Program Generation	29
4.3.4 Customization of GA Solver	30
4.3.4.1 Background	30
4.3.4.2 Use of GA inside the Decomposition Tool	31
4.3.5 Customization of the LINE Engine	31

4.3.5.1 Extending LINE to Version 2	32
4.3.5.2 Application of LINE 2 to Serverless Modeling	34
4.4. Feature Demonstration	35
4.4.1 Benchmark Creation	35
4.4.2 Applicability to Different Workloads	36
5. Conclusion and Future Work	39
References	41
Appendix A: Layered Queueing Networks	42
Appendix B: QoS Measures Provided by LQNs	45
Appendix C: Optimization Program Notation	46
Appendix D: RADON Model for Thumbnail Generation	48
Appendix E: Validation Example Measurements	49

1. Introduction

This deliverable presents the initial work carried out towards establishing a tool for decomposing and optimally mapping nodes to resources in RADON models. As described in deliverable *D2.1 Initial requirements and baselines*, the following capabilities are foreseen for the tool:

1. *Deployment optimization*. The ability to assign the concrete resources (e.g., memory, compute) of a RADON application taking into account quality requirements (e.g., service level agreements). This capability closes the gap between the high-level abstractions used by the developers and the concrete operational details needed to instantiate the application in the cloud.
2. *Architecture decomposition*. This feature deals with analyzing a pre-existing RADON model and suggesting possible changes based on known architectural patterns, in particular with the aim of breaking down the complexity of a monolith into finer microservices or serverless functions. This entails the ability to reason about the consequences of a refactoring.
3. *Accuracy enhancement*. This capability deals with the ability to improve the decisions offered in features 1 and 2 after correlating their outcomes with measurement in the operational environment of the application. In particular, this step is aimed at both: a) increasing the accuracy of the model predictions by grounding the model parameters into concrete data that is representative of application usage; b) in successively refining the decisions taken for optimization and decomposition based on these estimates.

1.1 Deliverable Objectives

This deliverable reports on the initial measures taken to implement a tool that offers the above functionalities. To break down complexity in the delivery of the above features, the definition of the tool has been based on two sequential milestones:

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

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

The objective of this deliverable is to present the work done for tool *Tool Milestone 1*, while the outcomes of *Tool Milestone 2* will be covered in the next deliverable, which is due at M22. The need for this sequencing arises from the fact that without developing a modeling approach for the behavior of the system at hand it is difficult to undertake a decomposition exercise, thus the

quality-of-service (QoS) model developed in *Tool Milestone 1* are the baseline for the decomposition in *Tool Milestone 2*.

1.2 Overview of Main Achievements

The main achievements of this deliverable are as follows:

- 1) *Definition of a service level modeling approach for microservices-based applications.* Using one of the RADON demo applications, we have studied the problem of QoS modeling of a microservices-based application. The corresponding work is illustrated in Section 2.
- 2) *Definition of a service level modeling approach for FaaS-based applications.* Applications developed with RADON are allowed to use an architecture that incorporates both microservices and serverless FaaS, which requires technology-specific considerations (e.g., cold starts of serverless functions).
- 3) *A methodology for parameterizing the QoS model of a microservice-based application.* We have experimentally compared and validated state-of-the-art techniques for model parameter estimation and shown that remain effective in the context of a microservices architecture, thereby explaining how they can help in enhancing the accuracy of QoS models for microservices-based applications.
- 4) *Proof of concept for parameterizing the QoS model of a FaaS-based application.* We have carried out a measurement and validation exercise on the QoS model of a RADON demo application built upon serverless FaaS. Our initial results indicate that a model parameterized by measurement is adequately applicable to QoS prediction for FaaS-based applications.
- 5) *A prototype of the tool featuring a deployment optimization capability.* We have implemented a prototype of the decomposition tool that can be used to obtain the optimal deployment scheme of an application composed of Lambda functions and S3 buckets. A demonstration of this feature is provided at the end of this deliverable.

1.3 Decomposition Tool Requirements

The following table summarizes RADON requirements defined in deliverable *D2.1 Initial requirements and baselines* at M6.

Table 1.1: 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	Should have

	generate a fine-grained RADON model.	
R-T3.2-3	Given a platform-independent RADON model, the DECOMP_TOOL must be able to obtain an optimal deployment scheme that minimizes the operating costs on a specific cloud platform under the performance requirements.	Must 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 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.	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

1.4 Structure of the Document

The rest of this deliverable is structured as follows. **Section 2** reviews the work carried out towards understanding how to model and parameterize microservices-based architectures in order to compare quality characteristics arising from alternative architectural choices. **Section 3** carries out a similar analysis in the context of serverless FaaS, explaining the importance of explicitly modeling parameters such as function *concurrency* and *memory*. **Section 4** presents the initial results in terms of the optimization feature and evidence that measurement-based parameterization provides an effective way to optimize the resources assigned to a RADON demo application. **Section 5** concludes the deliverable and outlines future work. **Appendices** provide additional information on the work done in the earlier sections.

2. QoS Modeling for Microservices-Based Architectures

In this section, we describe the work carried out in RADON towards establishing the applicability of QoS modeling formalisms based on stochastic modeling to microservices architectures. For illustration purposes, we center the description of this part on *Sock Shop*¹, one of RADON's demo applications introduced in deliverable *D6.1 Validation plan*. The work presented in this section is developed in full detail in our publication [GiaC19a].

2.1 Abstract Views

Microservices define a cloud-native architecture that is increasingly accepted in the software industry due to its synergy with DevOps. A microservices architecture intends to deliver a highly scalable application by decentralizing business logic among fine-grained services. This property results, among other benefits, in greater control of performance since the scaling of an application can be addressed by adding capacity only to the sections that actually need to be scaled. In addition, using containers for deploying microservices inherently improves the underpinning resource management, thanks to fast start-up times and ease of replication and reconfiguration.

2.1.1 Sock Shop Application

The reference microservices application, *Sock Shop*, is an ecommerce website that allows one to view and buy different types of socks, see deliverable D6.1. To develop this study, we install and deploy this application on Docker² swarm nodes, as in Figure 2.1.

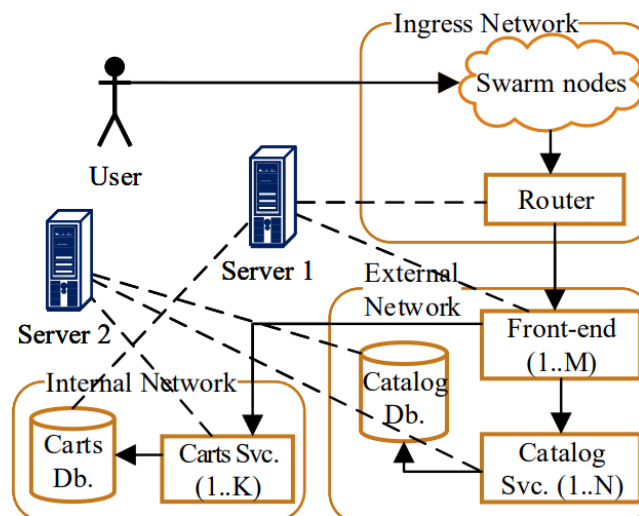


Figure 2.1: The architecture of the *Sock Shop* application [GiaC19a].

¹ <https://microservices-demo.github.io/>

² <https://www.docker.com/>

The front-end, catalog and carts services can have multiple replicas, respectively, which are load balanced by a router service. In addition, the catalog and carts services persist data on private database services. These database services and the router service are stateful, whereas the other services are stateless. To study QoS in microservices architectures, we have designed a browsing workload in this application, which creates a scenario where a user can visit the website and add or remove items in the cart. In this scenario, we have investigated two cases where the front-end microservice is the bottleneck and requires additional capacity. The workload and system setup used in the two cases are summarized in Table 2.1.

Table 2.1: Two cases where the front-end microservice is the bottleneck.

Case	System Workload					Front-End Configuration	
	Request Distribution			Concurrent Users	Think Time	CPU Share	Replica
	Home	Catalog	Carts				
A	57%	29%	14%	1000	7 s	0.2	1
B				4000		1.0	

The workload is specified by the mix of requests, i.e., the number and types of concurrent users issuing synchronous requests, and their think times, the times in-between a request completion and issue of the next request. The system configuration includes the CPU share and the replica number of the front-end microservice. A CPU share of 0.2 means that the microservice can at most utilize 20% of a CPU core, even if the rest of the CPU remains idle. The cases, A and B, represent a light and heavy workload respectively.

2.2 Microservices QoS Modeling

This section illustrates the use of Layered Queueing Network (LQNs) for QoS modeling of microservices, together with service demand estimation methods required to instantiate such models in concrete systems. We point to **Appendix A** for a general presentation of LQNs as a class of QoS models.

The objective of this section is to illustrate how LQNs can be applied in practice to a concrete microservices architecture, which then yields the general approach we have followed in the decomposition tool to translate RADON models to corresponding QoS models **automatically**.

LQN models can naturally abstract scenarios where a microservice can sometimes also work as a client to another microservice. This property often leads to the use of LQN in modeling distributed systems and can motivate the adoption of these models in microservice autoscalers.

The LQN model we have derived for the running case is shown in Figure 2.2.

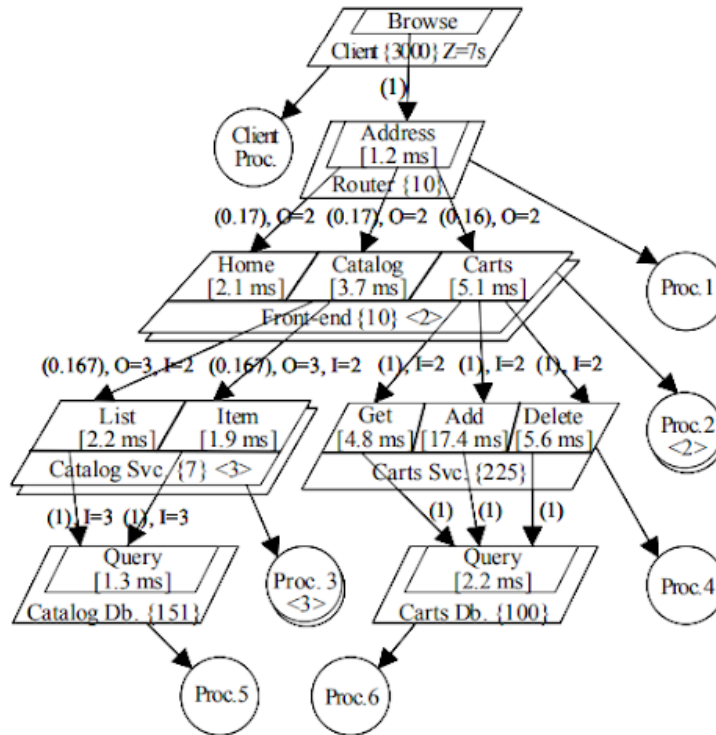


Figure 2.2: An LQN model of the *Sock Shop* application [GiaC19a].

In the LQN model, microservices are abstracted by tasks:

- Each **task** (e.g., Catalog, Carts, ...) consists of **entries** (e.g., Get, Add, Delete), which are equivalent to service classes in a queueing network and model the features or APIs exposed by the microservices to their clients. The **reference task** (Client) represents the system workload, consisting in this case of 3000 users.
- The tasks run on a **processor** (e.g., Proc. 6), which in this case represents the CPUs in the host servers. In an LQN model, the performance impact of vertical or horizontal scaling can then be assessed either by scaling the service demands of the activities in a processor according to the CPU share (vertical scaling) or, for horizontal scaling, by increasing the level of replication of a task. This inherently increases the number of service queues for that task.
- The tasks of an LQN model can include one or more **activities**, i.e., each entry can lead to a sequence of operations. An activity consumes a CPU time (e.g., 17.4 ms) which is represented by its service demand, which needs to be estimated to parameterize an LQN.

Our decomposition tool first maps a RADON model to an LQN, then reasons on expected performance by analyzing the LQN with standard solvers and obtaining corresponding metrics such as throughput, utilization and response times expected for each microservice after performing the scaling action. **Appendix B** describes the list of performance metrics available for QoS analysis of the microservices architecture using the LQN formalism.

2.3 Model Parameterization

We then move to the critical question of how to use data obtained by a monitoring infrastructure to obtain a **concrete parameterization** of the QoS model at hand. A number of established techniques exist for monoliths and service-oriented architecture (SOA), however to our knowledge we have been the first project to systematically investigate the applicability of statistical inference methods for model parameterization to microservices as well.

The main conceptual difference compared to the same problem in a monolith or SOA setting is that, since microservices run in independent containers that can be easily monitored both in terms of network traffic (e.g., via packet sniffers) and in terms of local computational behavior, there is an expectation of achieving higher accuracy in **demand estimation** for microservices compared to monoliths or SOA. By demand estimation, we mean the estimation of the CPU time that activities or entries in a LQN take on the underpinning compute resources (e.g., values such as 17.4ms in *Figure 2.2*).

For example, in a monolith it is difficult to resolve the individual functions of the software that carry out a particular activity once we see CPU utilization from running process, therefore demand estimation poses an ill-posed inverse problem in this context. As the monolith can feature hundreds of different services, this inverse problem can be practically too hard to solve, with the typical situation being that the demands of the most frequently used functions is estimated accurately by regression methods, whereas the other demands can incur large errors. As the erroneous estimates can apply to tens of functions, the resulting models tend to incur significant prediction errors when used to analyse what-if scenarios.

Conversely, a microservice exposes a handful functions, and often just one, considerably reducing the challenge of the associated demand estimation problem. Therefore, to prove that this is the case, we have carried out a systematic comparison of two established methods, whose performance was not well understood by the scientific community in the context of microservices. The first method is the common approach of using a **linear regression model** based on the utilization law to estimate the service demand parameters (the accumulated service requirements of a call to a microservice or resource by a request) [Pér17]. To get the demand, initially some utilization and throughput samples are collected from system measurements. The utilization function is then fitted by ordinary least square regression, possibly with non-negativity constraints, to obtain the service demands. The regression method assumes some variability in the observed throughputs, which is not always present in observations on microservices.

To illustrate this, in *Figure 2.3a*, we have plotted the measured utilization and response times associated to view item requests of the *Sock Shop* application. The data does not show strong correlation between the variables, which in turn means that accurate estimates of demand cannot be found this way. The latter issues can occur frequently for microservices as they are finely grained

and contain simple business logic. This makes microservices less resource intensive than traditional applications. Thus estimating their service demands based on utilization becomes more difficult.

More accurate estimates can be found using more fine-grained observations on the system, such as the response time of individual operations or groups of operations versus as a function of the queue length at their arrival. What makes **response time based estimation** viable here is that microservices architectures are highly decomposed, so it is easy from network traffic monitoring to determine the execution time and traffic towards an individual function exposed by a microservice. This technique is based on the formula for estimating response time using the mean-value analysis arrival theorem [Kra09e], which requires knowing queue-length values seen by arriving requests to a service, which we estimate from packet sniffing. When applied to the running case, the results are shown in Figure 2.3b. From the figure, it is seen that the variability in data (y axis) is significantly higher than the data for utilization samples, simplifying the estimate of demands, numerical results are given later. It is also evident that the estimate will be less sensitive to anomalies in the data, since the number of samples is greater than with the utilization technique since we collect samples in each individual call to the microservice, thus at higher frequency than utilization samples.

The conclusion of the study is therefore that tracking the response time of requests at a microservice over time is the most effective way to infer its service demand. This can be done with relative ease, for example by sniffing the request arrivals and request completions around a microservice, either during pre-production testing against a reference workload, or periodically in production against end-user workloads.

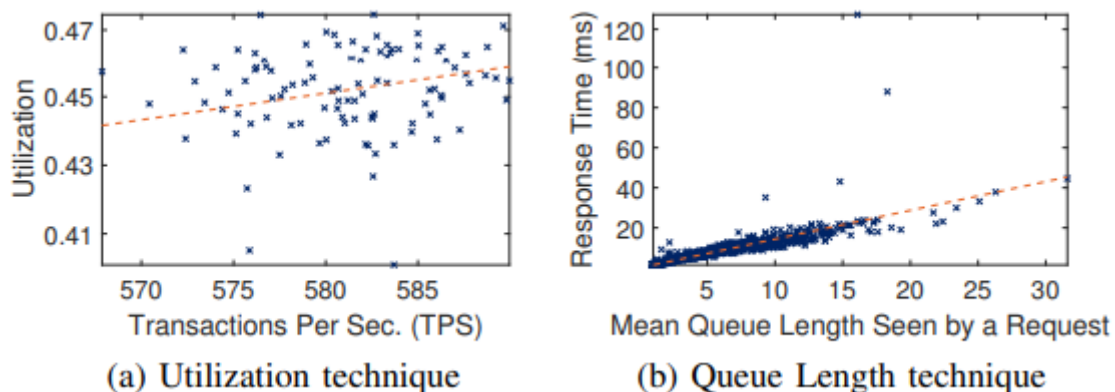


Figure 2.3: Service demand estimation using two different techniques [GiaC19a].

2.4 Assessing QoS Prediction

To validate the LQN predictions against real system measurements, we have performed multiple experiments. We have considered a subset of the *Sock Shop* application from Figure 2.1 that contains business logic. The workloads for the experiments are presented in Table 2.2. The request mix and the number of concurrent users have been selected such that they create light, normal and

heavy load in the system. The think times are set to 7s as frequent in the literature [Men02t], [Rol09p]. For workload 2 and 4, we have deployed the microservices in Docker compose mode, creating a scenario with a single host server. For workload 1 and 3, we have deployed the microservices in two Docker swarm nodes, creating a scenario with multiple host servers, where server 1 hosts the front-end and cart services and server 2 hosts the other three services. Within these servers, to avoid the approximation error of multi-server queueing nodes, we have kept a single CPU online. For workload generation, we have used the Locust³ tool with a distributed load testing configuration. The models are solved using the LQN simulator (LQSIM) [Woo13t].

The percent errors, considering the TPS and utilization, between the model and system measurement are presented in [GiaC19a], where it is shown that all the **average errors are at 5.05%**, and even the maximum error, observed in the utilization of front-end microservice, is only 9.98%. These errors are well accepted for QoS modeling purposes. We point to [GiaC19a] for detailed numerical evidence of these results.

2.5 Summary and Lessons Learned

The above exercise conducted on a RADON demo application, *Sock Shop*, has featured a methodology that is general, i.e., it does not leverage characteristics unique to the running application to deliver its results. Stemming from this experience, we have therefore defined, as described in the next sections, a general model-driven approach for automated QoS model generation based on the RADON model specification.

Key takeaways are that:

- The modularity of the microservices architecture is practically in a one-to-one mapping with the abstractions offered by the LQN formalism, easing debugging and parameterization from the real system.
- The ability to monitor network traffic and unfinished work within a microservice in a transparent manner enables the effective use of service demand inference methods to algorithmically parameterize these models from data.
- The general level of prediction error is low. The error in prediction of performance metrics in SOA or monoliths is often around above 15%, as opposed to the 5%-10% observed for the LQN in our study.

Based on the above observations, we have concluded that LQNs provide a practical way to describe software architecture with a fine granularity in their structure and we have established this approach within the decomposition tool implementation, as described in detail in Section 4.

³ <https://locust.io/>

3. QoS Modeling for FaaS-Based Applications

Unlike applications using a microservices architecture, FaaS-based applications are composed of stateless functions instead of individual services deployed on a cloud platform. These functions are executed in response to incoming events and can be activated and deactivated dynamically depending on the workload characteristics. The FaaS implementations currently differ across cloud providers, posing a need for an analysis of their unique features in the implementations.

We therefore begin this section by comparing FaaS compute services offered by Amazon Web Services (AWS), Google Cloud and Microsoft Azure. Our focus is then particularly put on the behavior of Lambda functions and S3 buckets defined by AWS. We use another RADON demo application, *Thumbnail Generation*, as an example to introduce an approach for QoS modeling of FaaS-based applications. In addition, a validation is presented to show the applicability.

3.1 FaaS Compute Services

FaaS compute services, such as AWS Lambda, Google Cloud Functions and Microsoft Azure Functions, allow developers to register functions in the cloud and **declare events that trigger function calls**. The underlying infrastructure monitors incoming events to each function and creates a runtime environment for that function to process them. Resources allocated to the function are automatically scale up and down on demand. The customer is charged only for resources and time spent in invoking the function. Since each function execution is isolated and ephemeral, building FaaS-based applications often requires managing shared data in separate storages, additionally to implementing business logic in an event-driven manner.

As a paradigm of serverless computing, FaaS compute services hide the resource management of functions. However, the developer can control the autoscaling of a function by adjusting two parameters, **memory** and **concurrency**. These are therefore both critical parameters that need to be captured by a QoS model for serverless FaaS. To further understand how to carry out QoS modeling, we therefore need to discuss these aspects in the specific context of the public cloud provider implementations. Table 3.1 summarizes maximum memory and concurrency imposed by the three FaaS compute services.

3.1.1 AWS Lambda and Google Cloud

When a function is invoked, AWS Lambda and Google Cloud assign the event to an instance of the function. Each function instance may only handle one event at a time. It is not possible for a second event to be routed to a busy instance. The current invocation can use all the resources of the instance, which are allocated in proportion to the value of the memory parameter. If all the instances of the function are currently busy, a new instance is created to process the event. The concurrency parameter specifies the maximum number of instances that the function can have.

3.1.2 Microsoft Azure

Differently on Microsoft Azure, functions that belong to the same application are bundled together. A component, called the scale controller, monitors the arrival rate of incoming events to a function application and determines whether to scale up or down the number of instances hosting that application. Each host instance may handle more than one event at a time. Thus, there is no limit on the number of concurrent executions.

Table 3.1: Maximum memory and concurrency imposed by AWS Lambda, Google Cloud Functions and Microsoft Azure Functions.

Parameter Limit	AWS Lambda	Google Cloud Functions	Microsoft Azure Functions
Max Memory	3008 MB per function instance	2048 MB per function instance	1536 MB per host instance
Max Concurrency	1000 per account (upgradable as requested)	1000 per function	Unlimited

3.2 AWS Lambda Functions

At this stage, we have focused our efforts on defining the modeling approach on a particular target platform, namely AWS Lambda, which is the market leader in serverless computing. However, several of the notions and concepts applied in our methodology can be easily translated to other providers.

3.2.1 Lambda Function Behavior

AWS Lambda works along with other AWS services to invoke functions. By specifying the appropriate trigger, a Lambda function may be enabled to handle resource lifecycle events, respond to incoming HTTP requests, consume events from a queue or run on a predefined schedule. Some services, for example Amazon Simple Queue Service (SQS), produce a queue or data stream. One can define an event source mapping for Lambda to read data from these services and create events to invoke functions. Other services may be configured to generate events that trigger the executions of Lambda functions directly. Depending on the service, the invocation can be synchronous or asynchronous. For synchronous invocation, the other service waits for the response from the function and might retry on errors. Amazon API Gateway is a typical service that invokes Lambda functions synchronously. For asynchronous invocation, Lambda queues each event before passing it to the function. The other service receives a success response immediately after the event is queued. When an error occurs, Lambda retries the invocation and sends failed events to a dead-letter queue if configured. Services that invoke Lambda functions asynchronously include Amazon Simple Storage Service (S3), Amazon Simple Notification Service (SNS) and Amazon Simple Email Service (SES).

3.2.2 Event-Driven Invocation from S3 Buckets

Since the *Thumbnail Generation* example involves the interactions between a Lambda function and S3 buckets, we are particularly interested in how Lambda deals with **asynchronous invocation**. When a function is invoked asynchronously, Lambda first sends the event to a queue. A separate process reads events from the queue and forwards them to the function. Once an event is added to the queue, Lambda returns a success response without additional information. Lambda manages the event queue of each function and retries failed events automatically. If the function returns an error, Lambda attempts to run it two more times, with one minute between the first two attempts and two minutes between the second and third attempts. If the function does not have enough concurrency available to process all events, additional requests are throttled. For throttling errors and system errors, Lambda returns the event to the queue and attempts to run the function again for up to six hours. The retry interval increases exponentially with base two from one second after the first attempt to a maximum of five minutes.

3.3 Thumbnail Generation Example

We now describe a refined *Thumbnail Generation* from *D6.1 Validation Plan* that includes QoS parameters we need. This version also highlights parameters critical to deployment optimization. We use it as a running example of a FaaS-based application in this deliverable.

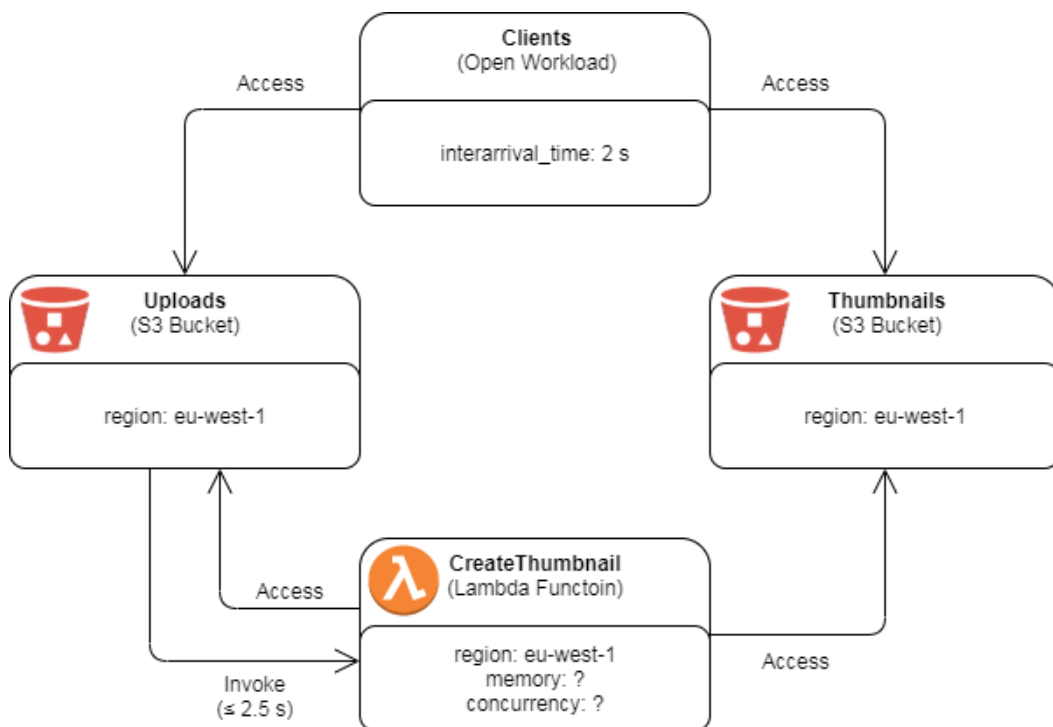


Figure 3.1: A more practical version of the *Thumbnail Generation* example.

As illustrated in Figure 3.1, the *Thumbnail Generation* application consists of two S3 buckets named *Uploads* and *Thumbnails* respectively as well as a Lambda function named *CreateThumbnail*. When a client puts an image into the *Uploads* bucket, S3 generates an event to invoke the *CreateThumbnail* function asynchronously. The *CreateThumbnail* function then gets the image from the *Uploads* bucket, resizes it into a thumbnail and puts the result into the *Thumbnails* bucket.

We consider the reference workload of the application to be an infinite stream of external clients that arrive one after another. The interarrival times of the clients are exponentially distributed with a mean of 2 s. Each client carries out the following sequence of activities and then leaves:

- Put an image to the *Uploads* bucket;
- Wait 10 s for the thumbnail to be ready;
- Get the thumbnail from the *Thumbnails* bucket;
- Delete the image in the *Uploads* bucket;
- Delete the thumbnail in the *Thumbnails* bucket.

A new abstract node called *Clients* is introduced to represent such an open workload in the example. We further require the average response time of invocations to the *CreateThumbnail* function to be less than 2.5 s when the application is running under the reference workload.

To minimize the effects of network delays, it is straightforward to deploy all the components of the application in a region that the majority of the clients come from. We choose eu-west-1 (Ireland) as the deployment region in the example. Resources available for a busy instance of a Lambda function are directly proportional to the function's memory, and a Lambda function can have at most the number of instances corresponding to its concurrency. These are the two critical parameters that determine whether the average response time of the *CreateThumbnail* function is less than 2.5 s. Indeed, one could find by experiment a configuration of memory and concurrency that satisfies the performance requirement under the reference workload. It is difficult to obtain the optimal configuration that can also lead to the minimum operating cost in the same way, especially for a more complex application. One goal of the decomposition tool is to help in solving this kind of deployment optimization problem.

3.4 FaaS QoS Modeling

The LQN structures for modeling various nodes that compose the *Thumbnail Generation* example as well as two common types of remote interactions are elaborated below. Each task in these structures is hosted on a dedicated processor with the same multiplicity, and can thus be conceptually considered as a pure queueing system. For conciseness, we drop the depiction of processors in both text and figures. Some LQN structures contain immediate activities that demand zero service from the hosting processors. Since LQNs with this kind of activity are not analytically solvable, every immediate activity is assigned a small host demand of δ_D to approximate its actual behavior.

As illustrated in Figure 3.2, an open workload can be modeled using an LQN structure where a reference task sends asynchronous requests to a normal task with an infinite multiplicity. Each request represents the arrival of a client from outside, and the think time at the reference task corresponds to the interarrival time. The behavior of the clients is incorporated into the *start* entry as an activity graph. Figure 3.3 shows the LQN structure for modeling a closed workload, which is simply a reference task whose multiplicity specifies the population of clients comprising the workload.

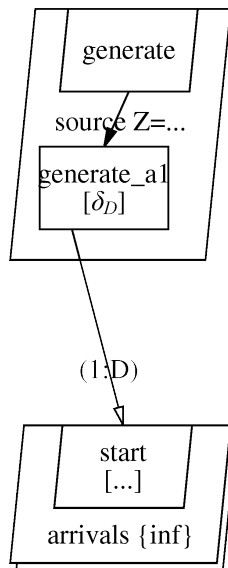


Figure 3.2: The LQN structure for modeling an open workload.

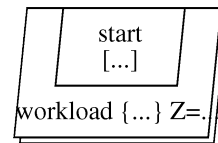


Figure 3.3: The LQN structure for modeling a closed workload.

S3 automatically scales a bucket to support a high request rate. Each bucket can handle at least 3500 PUT/COPY/POST/DELETE and 5500 GET/HEAD requests per second per prefix. There is no limit on the number of prefixes in a bucket. Also, the access throughput from a client to a bucket may be increased by spreading requests over separate connections. S3 does not restrict the bandwidth available for a bucket or the number of concurrent connections acceptable by a bucket. We model an S3 bucket in LQNs as a task with an infinite multiplicity and are particularly concerned about the three operations involved in the *Thumbnail Generation* example, i.e. GET, PUT and DELETE. As illustrated in Figure 3.4, each type of operation is represented by an individual entry of the task. Because AWS currently does not provide information about how long an S3 bucket spends in replying to a request, the first activity of all the entries is assigned a negligible host demand. We take this into account in the network delay of the corresponding remote interaction. Once the bucket replies to a PUT request, the new object may not be immediately ready for access. The *put_a2* activity is introduced to exactly capture this second-phase behavior. The *put_a3* activity is present only when the bucket is configured to invoke a Lambda function after a PUT operation.

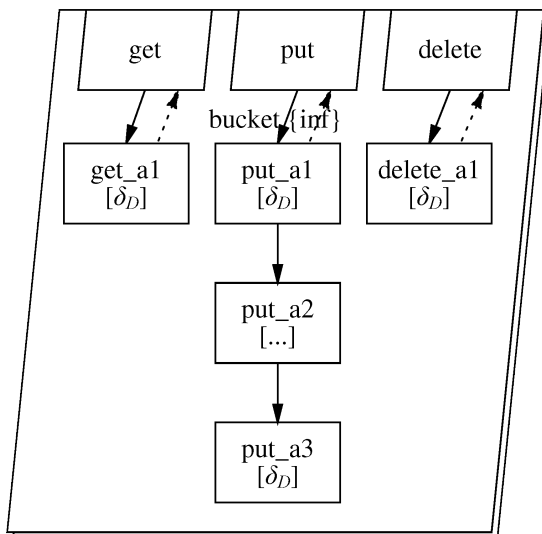


Figure 3.4: The LQN structure for modeling an S3 bucket.

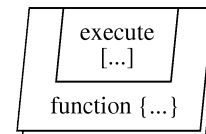


Figure 3.5: The LQN structure for modeling a Lambda function.

Lambda allocates dedicated resources to a busy instance of a function. Each function instance can handle at most one request at a time, and the maximum number of instances that a function can have is specified by the function’s concurrency. Figure 3.5 shows the LQN structure for modeling a Lambda function, which is a normal task with a multiplicity equal to the concurrency of the function. A single entry called *execute* is defined in the task to depict the execution logic of the function as an activity graph. Two points are worthy of notice in using this LQN structure to model a Lambda function. Since the memory of a function determines the amount of resources available for its instances, the host demand of every activity in the *execute* entry is inversely proportional to the function’s memory. This reveals how the optimization procedure should modify the LQN model with respect to the memory configuration of functions. Although Lambda maintains an event queue for each function, throttled invocations are retried after a delay instead of being processed once a function instance becomes idle. It is also difficult to measure the network delays between servers managing the event queue and those hosting the instances of the function. Therefore, the proposed LQN structure is valid only if invocations to the function are barely throttled. This condition is provably equivalent to that requests to the tasks are barely queued.

Cloud services such as S3 and Lambda are built upon distributed infrastructures. On a cloud platform, a request from one node to another and its corresponding response constitutes a remote interaction that introduces a network delay. We group all the remote interactions between a pair of source and target nodes and model them as a normal task with an infinite multiplicity, as illustrated in Figure 3.6. Each entry of the task represents a single interaction, which may be synchronous or asynchronous. Figure 3.7 shows the entry for capturing the behavior of synchronous interactions. Both requests and responses between the source activity and the target entry are forwarded by this entry. The network delay of an interaction is incorporated into the activity of the entry as a host

demand, which results in a balanced division of the network delay between the request and the response. This is probably not true in reality, but the analytical result does not change due to its invariability to the arrangement of pure delays. Figure 3.8 shows the entry for capturing the behavior of asynchronous interactions. Only requests from the source activity to the target entry are forwarded by this entry. The first activity of the entry incorporates the network delay while the second one is present to forward the request so that the network delay is fully attached to the request.

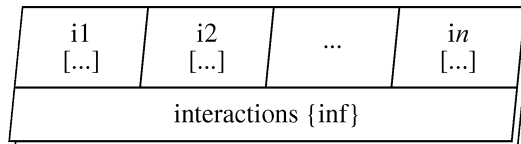


Figure 3.6: The LQN structure for modeling a group of remote interactions.

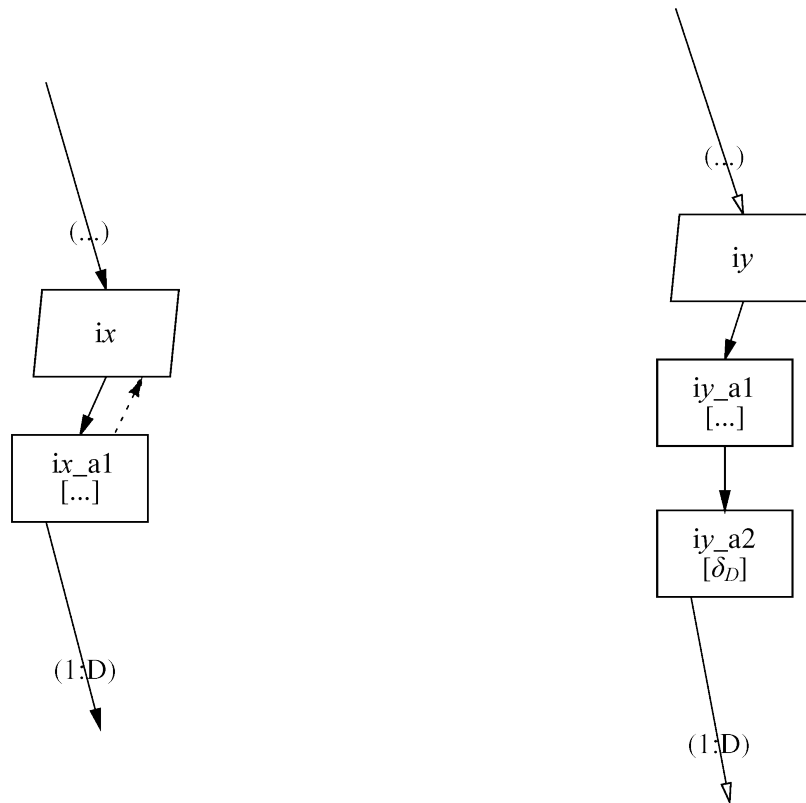


Figure 3.7: The LQN structure for modeling a synchronous remote interaction.

Figure 3.8: The LQN structure for modeling an asynchronous remote interaction.

3.5 Assessing QoS Prediction

We incorporate the above LQN structures into the definitions of the corresponding node and relationship types in TOSCA and create a RADON model for the *Thumbnail Generation* example,

as illustrated in **Appendix D**. A new policy type is also defined to express the requirement on the average response time of the *CreateThumbnail* function. We present here a validation of the LQN embedded in the RADON model concerning its accuracy in predicting this QoS measure.

To validate the LQN model, three groups of experiments are designed by varying the memory and concurrency of the function as well as the interarrival time of the workload. The experiments are based on the benchmark reported in **Appendix E**. For simplicity, this benchmark is created by uploading the logo of the RADON project as the image to be processed and setting the memory of the function to be 128 MB. Host demands of the function activities for other memory configurations are obtained by applying the inversely proportional relation. In particular, the small host demand assigned to each immediate activity is chosen to be 0.001 s. We use the Locust tool to simulate experiment workloads with clients that behave exactly as described in Section 3.3. Results for different groups of experiments are reported in Tables 3.2, 3.3 and 3.4. Each measured value is estimated as an average of 200 samples without cold starts. It can be seen that the error of the LQN model in predicting the average response time of the *CreateThumbnail* function is around 10% when rare invocations to the function is throttled. This indicates the applicability of the model to QoS prediction. In addition, whether invocations to a function are rarely throttled can be verified by checking if requests to the task representing that function are rarely queued.

Table 3.2: Experiment results for configurations of different memory but the same concurrency under the reference workload.

Memory	Concurrency	Predicted Value	Measured Value	Relative Error	Task Queueing	Function Throttling
192 MB	3	3.023 s	2.810 s	7.6%	Often	Often
256 MB		2.199 s	2.005 s	9.7%	Rare	Rare
320 MB		1.765 s	1.578 s	11.9%	Rare	Rare

Table 3.3: Experiment results for configurations of the same memory but different concurrency under the reference workload.

Memory	Concurrency	Predicted Value	Measured Value	Relative Error	Task Queueing	Function Throttling
256 MB	2	3.114 s	2.421 s	28.6%	Often	Often
	3	2.199 s	2.005 s	9.7%	Rare	Rare
	4	2.069 s	1.936 s	6.8%	Rare	Rare

Table 3.4: Experiment results for different interarrival times when the memory and concurrency of the function are set to 256 MB and 3 respectively.

Interarrival Time	Predicted Value	Measured Value	Relative Error	Task Queueing	Function Throttling
1 s	3.154 s	2.520 s	25.2%	Often	Often
2 s	2.199 s	2.005 s	9.7%	Rare	Rare
3 s	2.106 s	1.952 s	7.9%	Rare	Rare

3.6 Summary and Lessons Learned

The above exercise demonstrates, through the *Thumbnail Generation* example, a general approach to building QoS models for FaaS-based applications from predefined LQN structures. The demonstrated approach allows us to make use of node and relationship templates readily available in a RADON model to embed an LQN that can predict the performance of the target application. As shown in the following sections, the prototype of the decomposition tool implements this approach to automatically generate an LQN from a given RADON model, which is a preliminary step to complete a deployment optimization task.

Key takeaways are that:

- The proposed LQN structures are designed in a one-to-one mapping to a specific node or relationship type in TOSCA. This principle eases further extensions with new node and relationship types.
- The error of the LQN model in predicting the performance of the *Thumbnail Generation* example is within an acceptable range of 5%-15%, which illustrates the applicability of our approach to QoS modeling of FaaS-based applications.
- The validation results additionally implies that the QoS model of a FaaS-based application can be well parameterized using direct measurement in place of advanced model parameter estimation techniques that are often not suitable for this case.

We conclude from the above observations that LQNs are also a particularly useful formalism for RADON to model the performance of FaaS-based applications. As presented in the next sections, this modeling approach is adopted in the implementation of a prototype for the decomposition tool that features an initial optimization deployment capacity.

4. Deployment Optimization

Our approach to optimizing the deployment scheme of an application consisting of Lambda functions and S3 buckets is described in this section. We come up with a formulation of the optimization problem and then show how to divide it into two subproblems that can be solved efficiently. Particularly, the values of optimization constraints arising from performance requirements are predicted using the QoS model introduced in Section 3. We have implemented a prototype of the decomposition tool that adopts our approach for deployment optimization. This section also reports details about the implemented prototype together with a demonstration of its feature through the *Thumbnail Generation* example.

4.1 Problem Formulation

In order to decide on how to optimally allocate resources (e.g., memory, concurrency) to a Lambda function, we define a nonlinear optimization program with constraints on performance measures that can be obtained from LQNs. This program is built upon quantitative relations listed below (see **Appendix C** for notation):

$$M_i^P = \begin{cases} c_i & \text{if } i \in \mathcal{F}, \\ \infty & \text{if } i \in \mathcal{S}. \end{cases} \quad (1)$$

$$M_i^T = \begin{cases} c_i & \text{if } i \in \mathcal{F}, \\ \infty & \text{if } i \in \mathcal{S}. \end{cases} \quad (2)$$

$$D_{i,k,r} = \frac{m'_i}{m_i} D'_{i,k,r} \quad \text{for } i \in \mathcal{F}. \quad (3)$$

$$B_{i,k} = \sum_{j,l,s} Y_{(j,l,s),(i,k)} X_{j,l,s} W_{(j,l,s),(i,k)}. \quad (4)$$

$$R_{i,k} = \frac{B_{i,k} + U_{i,k}^{\text{PH1}}}{X_{i,k}}. \quad (5)$$

$$C_i = \begin{cases} (P^{\text{FI}} + P^{\text{FE}} m_i (S_{i,1}^{\text{PH1}} + S_{i,1}^{\text{PH2}})) X_{i,1} & \text{if } i \in \mathcal{F}, \\ \sum_k P_k^{\text{SO}} X_{i,k} & \text{if } i \in \mathcal{S}. \end{cases} \quad (6)$$

These relations are obtained by applying *operational analysis*, i.e., the fundamental laws for performance evaluation such as Little's law, flow balance and other classic principles established in the area. In particular, (6) computes the cost of operating a certain node i , either a function or a storage, in the RADON model topology. The overall objective of the optimization program is to minimize the summation of operating costs over all the nodes.

4.2 Solution Procedure

Based on the aforementioned quantitative relations, a numerical optimization approach is proposed to solve the resource management problem for Lambda functions as two subproblems (see again **Appendix C** for notation):

- **Memory allocation subproblem.** Solve for \mathbf{m}^* such that

$$\begin{aligned}
 \min_{\mathbf{m}} \quad & C(\mathbf{m}, \mathbf{c}), \\
 \text{s.t.} \quad & \mathbf{m} \in \{\mathbf{m} \in \mathbb{Z}_{>0}^{|\mathcal{F}|} \mid \forall i \in \mathcal{F}, (m_i \mid 64) \wedge (2 \leq m_i/64 \leq 47)\}, \\
 & \mathbf{c} = \infty, \\
 & \mathbf{a} \leq \mathbf{G}(\mathbf{m}, \mathbf{c}) \leq \mathbf{b},
 \end{aligned} \tag{7}$$

where ∞ is a vector whose entries are all infinite, and \mathbf{G} denotes the constrained quantities, e.g. the average response time R_i of a request at a function node i , which are bounded between \mathbf{a} and \mathbf{b} .

- **Concurrency determination subproblem.** For each function node i , solve for c_i^* such that

$$\begin{aligned}
 \min_{c_i} \quad & c_i, \\
 \text{s.t.} \quad & \mathbf{m} = \mathbf{m}^*, \\
 & \mathbf{c} \in \{\mathbf{c} \in \mathbb{Z}_{>0}^{|\mathcal{F}|} \mid \forall (j \in \mathcal{F} \wedge j \neq i), c_j = \infty\}, \\
 & B_i(\mathbf{m}, \mathbf{c}) \leq \delta_B,
 \end{aligned} \tag{8}$$

where δ_B is a small value of the mean buffer length. The last constraint in (8) uses the task state probabilities of the LQN, not detailed here, to limit the probability of an invocation to a function being throttled.

A heuristic strategy is available to speed up the above solution procedure for a large-scaled RADON model. To this end, we relax the integer variable \mathbf{m} of the memory allocation subproblem into a continuous one. The resulting continuous relaxation can be solved rapidly using a derivative-based optimization algorithm. With the continuous optimal solution \mathbf{m}^* , we may provide an initial point or range for a mixed-integer nonlinear solver to search more effectively for a feasible discrete solution.

4.3 Tool Implementation

This section elaborates the implementation of a prototype for the decomposition tool, which adopts the previous procedure to obtain the optimal deployment scheme of an application comprising Lambda functions and S3 buckets. The overall approach to deployment optimization is introduced.

We also discuss how the LINE⁴ engine and the GA⁵ solver have been customized for this usage scenario.

4.3.1 Overall Approach

The implementation of the decomposition tool is based on a chain of tools and data structures illustrated in Figure 4.1. Given a RADON model, the tool uses a built-in YAML processor to import the service template into MATLAB and generates the topology graph and the embedded LQN automatically through model-to-model transformation. An optimization problem is then created from the topology graph and solved by invoking the GA solver and the LINE engine. When the optimal solution is found, the tool writes the result back into the original service template. Although MATLAB is a commercial product, the tool does not require ownership of a MATLAB license as it can be compiled and released as an executable running on MATLAB Runtime⁶, which is royalty free.

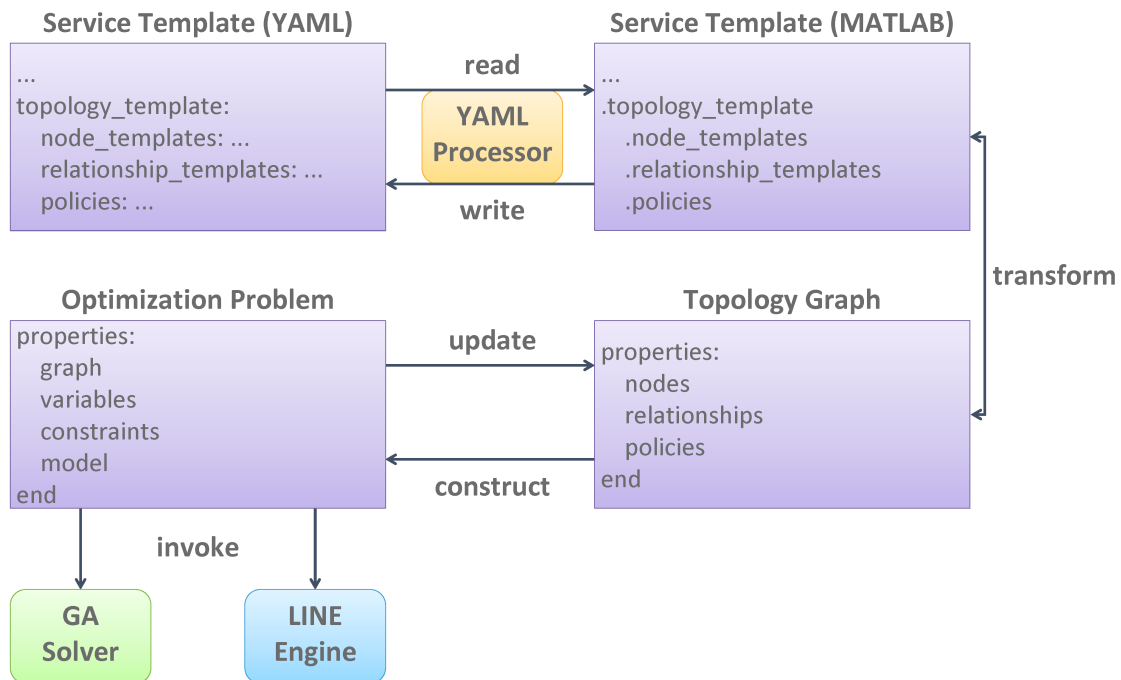


Figure 4.1: The overall approach to deployment optimization.

4.3.2 RADON YAML Processor for MATLAB

The TOSCA simple profile in YAML is defined based on the YAML 1.2⁷ specification. There are currently no official or third-party utilities available in MATLAB for processing YAML 1.2. To

⁴ <http://line-solver.sourceforge.net/>

⁵ <https://uk.mathworks.com/help/gads/genetic-algorithm.html>

⁶ <https://uk.mathworks.com/products/compiler/matlab-runtime.html>

⁷ <https://yaml.org/spec/1.2/spec.html>

enable the decomposition tool to import and export RADON models, we have implemented a MATLAB YAML processor by wrapping SnakeYAML Engine, an open-source Java YAML processor that supports the YAML 1.2 specification. With this built-in YAML processor, the tool can read/write the service template of a given RADON model into/from a MATLAB data structure consisting of structs and cell arrays, which represent two basic primitives in YAML: mapping (an unordered set of key-value pairs) and sequence (an ordered set of elements). In addition, Table 4.1 summarizes the casting between YAML and MATLAB types for scalars.

Table 4.1: The casting between YAML and MATLAB types for scalars.

YAML Type (URI)	MATLAB Type
tag:yaml.org,2002:null	double (empty)
tag:yaml.org,2002:bool	logical
tag:yaml.org,2002:int	int32, int64
tag:yaml.org,2002:float	double
tag:yaml.org,2002:str	char

4.3.3. Topology Graph and Optimization Program Generation

As illustrated in **Appendix D**, we have made an extension to the definitions of certain node and relationship types in TOSCA, which enables RADON users to model application behavior with LQN constructs and to specify performance requirements such as the average response time of a serverless function. LQN constructs are embedded into the properties of node and relationship templates. Each performance requirement corresponds to a performance policy template. All these templates are essential components that comprise the topology template of a service template.

To extract necessary information for deployment optimization, the decomposition tool will further transform the *topology_template* field of the MATLAB data structure into an extended directed graph incorporating nodes, relationships and policies, which we call the topology graph. A MATLAB class is created for each type of node, relationship and policy. This design follows the open/closed principle, allowing the tool to easily support new node, relationship and policy types. The topology graph could also be useful for architecture decomposition, which requires the manipulation of the topology template.

To find the optimal deployment scheme, the decomposition tool will construct for the topology graph an optimization problem that contains variables, constraints and the embedded LQN. Variables are found from various nodes comprising the topology graph, while constraints are found from performance policies incorporated in the graph. Additional constraints may arise from

assumptions under which the LQN is valid. Due to the complexity of the pseudocode, we do not report here in detail.

As a case in point, the mean buffer length of each serverless function has to be negligible so that invocations to the function are barely throttled. The tool invokes the GA solver in MATLAB to solve the optimization problem, aiming to minimize the total operating cost of all the nodes under all the constraints found in the topology graph. When the objective or constraint function of the optimization problem is called, the LQN is reparameterized with new variable values and then solved at most once by invoking the LINE engine, which subsequently invokes the LQN solver (LQNS) [Woo13t].

Upon completion of the optimization procedure, the tool will feed the optimal deployment solution back to the original RADON model. This is realized by updating the topology graph with the solution, transforming the graph into a MATLAB data structure, and write that data structure into a service template using the built-in YAML processor.

4.3.4 Customization of GA Solver

4.3.4.1 Background

The GA solver in MATLAB is a genetic algorithm solver for constrained or unconstrained optimization with mixed-integer variables. Genetic algorithms are a class of evolutionary algorithms based on a natural selection process that mimics biological evolution. They can be used to solve a variety of optimization problems that are not well suited for classical optimization algorithms, including those where some variables are restricted to be integer-valued or the objective function is nonlinear, nondifferentiable, discontinuous or stochastic.

The genetic algorithm implemented by the GA solver is based on the state of the art and may be summarized as follows:

- An initial population of individuals is created at random.
- An iterative procedure is conducted to obtain a sequence of generations toward the optimal solution. Each iteration consists of the following steps:
 - Evaluate the fitness value of each individual in the current generation;
 - Convert the fitness values into a more usable range of expectation values;
 - Select parents from the current generations according to the expectation values;
 - Find individuals with best expectation values in the current generation as elite;
 - Produce children from the parents. Children are produced either by combining the genes of two parents (crossover) or by randomly changing the genes of one parent (mutation);
 - Replace individuals in the current generation with the elite and the children to form the next generation.
- The iterative procedure terminates when one of the stopping criteria is met.

In general, one can control the behavior of the algorithm by adjusting selection, crossover and mutation rules that govern the production of children for the next generation. Several modifications of the basic algorithm have been made in the GA solver for integer programming, including the adoption of special creation, crossover and mutation functions that enforce variables to be integer. These modifications disable many options of the GA solver, greatly reducing its flexibility in dealing with integer problems.

4.3.4.2 Use of GA inside the Decomposition Tool

Two measures are carried out by the decomposition tool to accelerate deployment optimization of an application that consists of Lambda functions and S3 buckets. Section 4.2 presents a heuristic strategy for finding a possible range of the optimal solution to the memory allocation subproblem. The tool implements this heuristic strategy to provide a range for the GA solver to create the initial population, which could make it easier for the solver to find the optimal solution.

When individuals in the current generation are close to the optimal solution, the next generation created by the GA solver is very likely to overlap the current generation. Therefore, a dynamic cache for buffering the results of individuals in recent generations has been developed to solve the memory allocation subproblem efficiently. As for the concurrency determination subproblem, we use a static cache that stores the results of all the individuals that have been computed. This is because a single bounded variable is involved in determining the concurrency of each function.

4.3.5 Customization of the LINE Engine

LINE is a MATLAB toolbox for performance and reliability analysis of software applications, business processes and computer networks that can be described by queueing models. At the beginning of the RADON project, the tool consists of a set of MATLAB scripts to analyze a subset of LQNs with 2 layers, using a particular algorithm (fluid method), which allowed to solve these models efficiently when only mean performance metrics are required.

Upon developing the decomposition tool, it was recognized that this capability was insufficient to realize the tool. For example, the computation of buffer length occupancy measures in the concurrency determination optimization program is critical to establish if a function will be throttled. Moreover, the *cold start* behavior implies that some elements of a model may carry a binary state (cold/hot), and this was not possible to model using the fluid method, which is not meant to handle binary state variables. Moreover, the definition of LQN models did not have the required expressiveness to model complex chains of asynchronous requests due to the event-driven nature of the serverless paradigm.

Due to the above and other requirements and prospective benefits for the decomposition tool, as part of this deliverable we have introduced a new major release of our LINE engine, which provides the essential quantitative solver needed to compare alternative performance and reliability during decomposition and optimization analysis.

4.3.5.1 Extending LINE to Version 2

In the initial months of the project, we have produced a new release of LINE that essentially rewrites the 1.0.x nearly entirely. The features available to the modeler are extensive, and described in a new manual, over 100 pages long, that we have produced and released at

<https://github.com/line-solver/LINE/raw/master/doc/LINE.pdf>

This new extension has been documented in a published demonstration paper [Cas19a], we here highlight the key elements. We point the reader to that work for further details about this line of work within the RADON project.

The new version 2.0.x provides a major tool overhaul and refactoring, whereby the tool introduces a distinction between the modeling language used to represent a system - in our case a software architecture - and the solution methods used for performance and reliability analysis. In particular, a new object-oriented language has been introduced to model extended and layered queueing networks. The next example it is meant to illustrate the fact that the RADON decomposition tool, building on top of LINE 2.0.x, can now feature programmatic specification of QoS models after TOSCA YAML parsing. To create a basic layered network, we can instantiate a new model as

```
model = LayeredNetwork('myLayeredModel');
P1 = Processor(model, 'P1', 1, SchedStrategy.PS);
P2 = Processor(model, 'P2', 1, SchedStrategy.PS);
T1 = Task(model, 'T1', 5, SchedStrategy.REF).on(P1);
T2 = Task(model, 'T2', Inf, SchedStrategy.INF).on(P2);
E1 = Entry(model, 'E1').on(T1);
E2 = Entry(model, 'E2').on(T2);
```

Here, the *on* method specifies the associations between the elements, e.g., task T1 runs on processor P1, and accepts calls to entry E1. Furthermore, the multiplicity of T1 is 5, meaning that up to 5 calls can be simultaneously served by this element (i.e., 5 is the number of servers in the underpinning queueing system for T1). Both processors and tasks can be associated to the standard LINE scheduling strategies. For instance, T2 will process incoming requests in parallel according as an infinite server node. An exception is that SchedStrategy.REF should be used to denote the reference task (e.g. a node representing the clients of the models).

We can then elicit the possible services exposed by the above architecture and their performance characteristics by declaring them as LQN activities:

```
A10 = Activity(model, 'A10', Exp(1.0)).on(T1).boundTo(E1).synchCall(E2,3.5);
A11 = Activity(model, 'A11', Exp(1.0)).on(T1).boundTo(E1).asynchCall(E2,1);
T1.addPrecedence(ActivityPrecedence.Serial(A10, A11));
A20 = Activity(model, 'A20', Exp(1.0)).on(T2).boundTo(E2);
A21 = Activity(model, 'A21', Erlang.fitMeanAndOrder(1.0,2)).on(T2);
A22 = Activity(model, 'A22', Exp(1.0)).on(T2).repliesTo(E2);
T2.addPrecedence(ActivityPrecedence.Serial(A20, A21, A22));
...
```

In mapped to particular entry points on the tasks (*boundTo*), and their reply (*repliesTo*) and precedence patterns (*Serial*), additional language primitives are available to characterize both

synchronous calls (*syncCall*) and asynchronous calls (*asynccall*). An example of layered topology defined by the LINE engine within MATLAB is shown in the figure below.

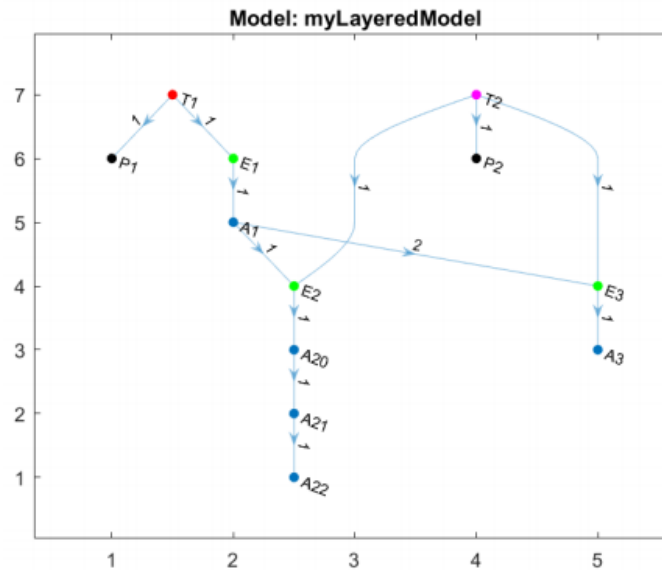


Figure 4.2: A layered topology defined by the LINE engine within MATLAB.

After the LQN is declared, a fixed-point iteration algorithm is used to determine its solution. A difference that makes LINE 2 unique compared to other state-of-the-art solvers for LQNs is that the particular algorithm used to solve a particular layer of the software architecture is customizable. For example, we can choose to use simulation in a layer where we wish to employ a measured trace to describe request arrivals, and instead analytical methods in all other layers. Among the supported methods is also the LQNS solver, which covers a broad spectrum of LQNs, but focusing primarily on mean-value analysis. Conversely, LINE offers some analytical solvers that return probabilistic measures as well, which are useful for SLA and buffer occupancy analyses.

In numbers, extensions released during RADON for the LINE engine are as follows:

- 40+ solution algorithms;
- 13 general types of analyses;
- 13 supported scheduling and routing strategies;
- A rich LQN language consisting of 100+ object-oriented classes;
- 14 node types supported in the models;
- 4 reference steady-state metrics (Utilization, Throughput, Response Time, Queue-Length) and many other transient and probabilistic measures (e.g., node marginal probabilities, joint state probabilities in the network, etc).

The above extension has provided the necessary backend to realize the higher-level functionalities of the decomposition tool.

4.3.5.2 Application of LINE 2 to Serverless Modeling

The LINE engine offers a programmatic interface to create LQNs and solve them by invoking LQNS. We have extended this interface to enable more features of LQNs, which are found useful for modeling FaaS-based applications. These features have been then integrated in the open source release of LINE. Table 4.2 reports LQN features supported by the LINE engine. To make all these features work properly and reduce time spent in solving an LQN, we have further specified additional pragmas to control LQNS. Table 4.3 reports LQNS pragmas adopted by the decomposition tool.

Table 4.2: LQN features supported by the LINE engine.

Feature	Support
Task Scheduling	FIFO
Processor Scheduling	FIFO, PS
Server Multiplicity	Finite, Infinite
Request Pattern	Stochastic, Deterministic
Reply Mode	Synchronous, Asynchronous
Phase Number	1, 2
Activity Precedence	Sequence, And-Fork, And-Join, Or-Fork, Or-join, Loop

Table 4.3: LQNS pragmas adopted by the decomposition tool.

Pragma	Remark
stop-on-message-loss=false	Do not stop the solver on queue overflow for asynchronous requests
severity-level=run-time	Only display runtime error messages
layering=batched	Solve layers composed of as many servers as possible from top to bottom
mva=one-step	Perform one step of the Bard-Schweitzer AMVA for each iteration of a submodel
multiserver=conway	Use the Conway approximation for all multi-server queues

4.4. Feature Demonstration

This section presents a demonstration of the prototype for the decomposition tool in terms of deployment optimization. We take the *Thumbnail Generation* application as an example to show how to create a benchmark for an application consisting of Lambda functions and S3 buckets. Then, we use the tool to optimize the deployment scheme of this example application and carry out a load test to validate the result for both open and closed workloads.

4.4.1 Benchmark Creation

There are two common techniques for creating a benchmark for a system under a given workload. By convention, they are called the utilization-based and the response-time-based technique respectively. The former applies the utilization law:

$$U_i = \sum_k D_{i,k} X_k \quad (9)$$

where U_i is the average utilization of resource i , $D_{i,k}$ is the service demand of a class- k job at resource i , and X_k is the average system throughput of class- k jobs. To estimate the service demands, we can collect utilization and throughput samples for the system and fit (12) to the collected samples through ordinary least square regression with non-negative constraints. The latter applies the following equation:

$$R_{i,k} = T_{i|k} + \sum_l S_{i,l} B_{i,l|k} + S_{i,k} \quad (10)$$

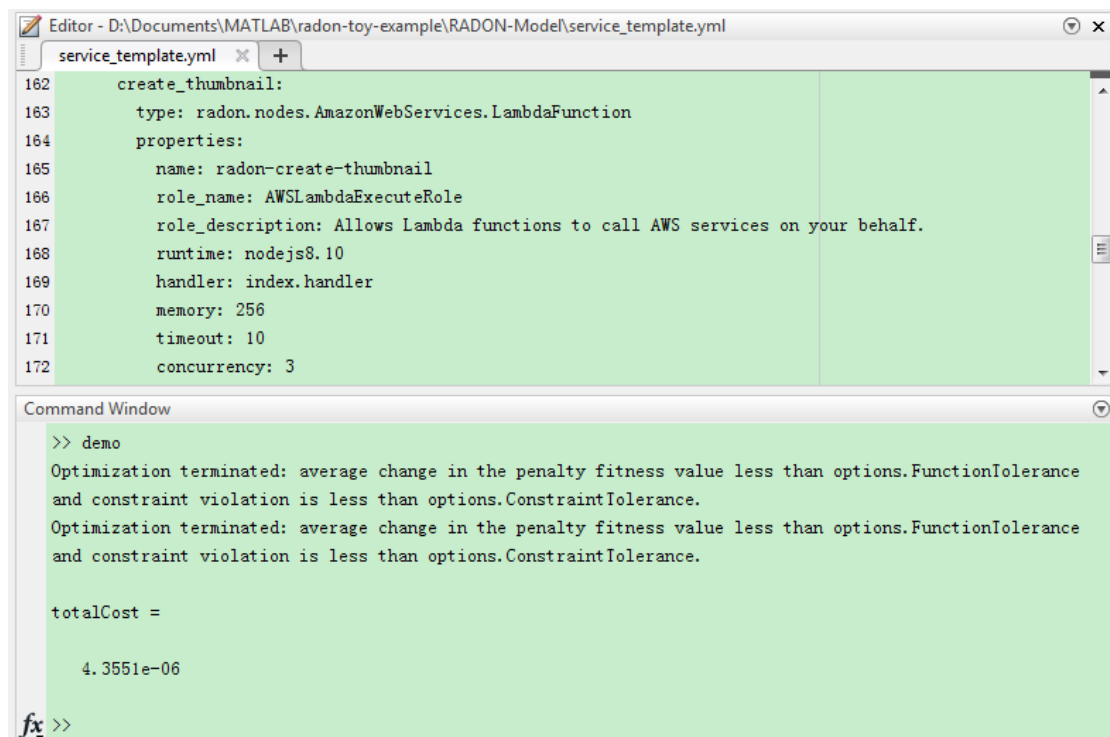
where $R_{i,k}$ is the average response time of a class- k job at resource i , $T_{i|k}$ is the average residual time of the job in service upon arrival of a class- k job at resource i , $S_{i,l}$ (or $S_{i,k}$) is the service requirement of a class- l (or class- k) job at resource i , and $B_{i,l|k}$ is the mean buffer length of class- l jobs upon arrival of a class- k job at resource i . The service requirements can thus be estimated using linear regression to fit (13) to response time and buffer length samples collected for each resource. Particularly, the response-time-based technique assumes resources as single-server queues but is known generally more accurate than the utilization-based technique.

As mentioned in Section 3.4, we model a Lambda function as a multi-server queue that can serve as many invocations as specified by its concurrency in parallel. Consequently, the response-time-based technique cannot be used to estimate service requirements at Lambda functions. Since the instances of a Lambda function are distributed among anonymous servers that may change over time, it is also difficult to collect utilization samples for Lambda functions. In fact, Lambda allocates dedicated CPU and memory resources to each busy instance of a function. Service requirements at Lambda functions are invariant for workloads that result in invocations with jobs of the same size. This is also true for S3 buckets and remote connections as they are essentially infinite-server queues that only introduce pure delays. To create a benchmark for the *Thumbnail Generation* example, we simply simulate a closed workload that comprises a single

client with zero think time and record time points immediately before or after critical operations at appropriate nodes. The host demands of activities as well as the network delays of requests can be estimated straightforwardly as measures based on these time points. Table E.1 lists time points recorded for creating the benchmark. Tables E.2 and E.3 report measures and results for the host demands of activities and the network delays of requests. The results are obtained when the logo of the RADON project is uploaded as the image to be processed and the memory of the `create_thumbnail` function is set to 128 MB.

4.4.2 Applicability to Different Workloads

After creation of the benchmark, the deployment scheme of the *Thumbnail Generation* application is optimized under the specified requirement for two different workloads to showcase the applicability of the decomposition tool. One is an open workload with external clients whose interarrival times follow an exponential distribution with a mean of 2 s, each performing a sequence of activities as described in Section 3.3. The other is a closed workload with a fixed population of 12 clients who spend an exponentially distributed think time of 10 s between two consecutive activity sequences. All the clients upload the RADON logo as the image to be processed, which complies with the created benchmark. Figure 4.3 shows the interactive execution of the decomposition tool for deployment optimization of the *Thumbnail Generation* example. Optimal deployment schemes and operating costs obtained for the two workloads are reported in Table 4.4.



```

Editor - D:\Documents\MATLAB\radon-toy-example\RADON-Model\service_template.yml
service_template.yml x +
162   create_thumbnail:
163     type: radon.nodes.AmazonWebServices.LambdaFunction
164     properties:
165       name: radon-create-thumbnail
166       role_name: AWSLambdaExecuteRole
167       role_description: Allows Lambda functions to call AWS services on your behalf.
168       runtime: nodejs8.10
169       handler: index.handler
170       memory: 256
171       timeout: 10
172       concurrency: 3

Command Window
>> demo
Optimization terminated: average change in the penalty fitness value less than options.FunctionTolerance
and constraint violation is less than options.ConstraintTolerance.
Optimization terminated: average change in the penalty fitness value less than options.FunctionTolerance
and constraint violation is less than options.ConstraintTolerance.

totalCost =

    4.3551e-06

fx >>
  
```

Figure 4.3: The interactive execution of the decomposition tool for deployment optimization of the *Thumbnail Generation* example.

Table 4.4: Optimal deployment schemes and minimum operating costs obtained for the two workloads.

Workload	Deployment Scheme	Operating Cost
R-T3.2-1 interarrival_time: 2 s	create_thumbnail: memory: 256 MB concurrency: 3	4.3551×10^{-6} \$/s
clients: population: 12 think_time: 10 s	create_thumbnail: memory: 256 MB concurrency: 4	5.1303×10^{-6} \$/s

To validate the resulting optimal solutions, we use Locust to conduct a load test on the *Thumbnail Generation* application deployed for each workload. Locust is a simple Python-based tool for load testing of websites or other systems. It can simulate a workload with thousands of concurrent clients on a single machine. A web UI is also offered by Locust for monitoring the interactions between the mock clients and the target system in real time. Figures 4.4 and 4.5 show the load testing results for the two workloads. The response time of the PROCESS_OBJECT request corresponds to the period from the end time of putting the image to the end time of listing the thumbnail, which is on average 2.487 s and 2.437 s for the open and the closed workload respectively. According to measures and results reported in Tables E.2 and E.3, the actual average response time of the *create_thumbnail* function can be calculated as 2.005 s and 1.955 s for the two workloads, thus satisfying the specified requirement.



Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)
DELETE_OBJECT_1	delete_object	200	0	100	106	77.41951942443848	190.05584716796875
DELETE_OBJECT_2	delete_object	200	0	35	41	25.38299560546875	203.43947410583496
GET_OBJECT	download_file	200	0	120	127	52.77419090270996	245.6514835357666
PROCESS_OBJECT	list_objects	200	0	2400	2487	1909.4288349151611	5806.118726730347
PUT_OBJECT	upload_file	200	0	140	151	115.43893814086914	366.2846088409424
Total		1000	0	120	582	25.38299560546875	5806.118726730347

Figure 4.4: The load testing result for the open workload.

Statistics Charts Failures Exceptions Download Data							
Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)
DELETE_OBJECT_1	delete_object	200	0	100	106	68.81237030029297	336.4064693450928
DELETE_OBJECT_2	delete_object	200	0	33	43	25.429487228393555	535.1309776306152
GET_OBJECT	download_file	200	0	120	130	87.42356300354004	359.9052429199219
PROCESS_OBJECT	list_objects	200	0	2400	2437	1911.7958545684814	3900.9335041046143
PUT_OBJECT	upload_file	200	0	140	162	110.00180244445801	1315.2811527252197
Total		1000	0	120	576	25.429487228393555	3900.9335041046143

Figure 4.5: The load testing result for the closed workload.

5. Conclusion and Future Work

In this deliverable, we have described our approaches for QoS modeling of both microservices-based and FaaS-based applications. The applicability of these approaches has been validated using two RADON demo applications: *Sock Shop* and *Thumbnail Generation*. As part of this exercise, we explored and compared model parameterization techniques in the context of microservices and serverless FaaS. A prototype of the decomposition tool has also been developed, featuring an initial deployment optimization capability. Table 5.1 reports the achieved level of compliance to RADON requirements defined in deliverable *D2.1 Initial requirements and baselines* at M6.

Table 5.1: The achieved level of compliance to RADON requirements (✘=left for next iteration, ✓= 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-T3.2-3	Given a platform-independent RADON model, the DECOMP_TOOL must be able to obtain an optimal deployment scheme that minimizes the operating costs on a specific cloud platform under the performance requirements.	Must 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 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.	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 version of this deliverable *D3.3 Decomposition tool II*, which is due at M22, will present efforts to deliver the complete functionalities of the decomposition tool. Details about the full implementation of the tool will also be included in the final deliverable. To this end, we will be primarily focusing on the following work:

- Synchronize with the template library definitions of TOSCA node, relationship and policy types customized for architecture decomposition, deployment optimization and accuracy enhancement.
- Extend the decomposition tool to support serverless functions and object storages on Google Cloud, and enable deployment optimization of platform-independent RADON models across AWS and Google Cloud.
- Investigate approaches for automatically decompose a monolithic RADON model at the level of microservices and serverless functions, and implement these approaches in the decomposition tool to provide the architecture decomposition feature.
- Conduct collaboration with UST and ATC 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

- [Cas19a] G. Casale. Automated Multi-Paradigm Analysis of Extended and Layered Queueing Models with LINE, Proc. of ACM/SPEC ICPE 2019.
- [GiaC19a] A. Gias, G. Casale, M. Woodside. ATOM: Model-Driven Autoscaling for Microservices, Proc. of IEEE ICDCS 2019.
- [Kra09e] S. Kraft, S. Pacheco-Sanchez, G. Casale, S. Dawson. Estimating Service Resource Consumption From Response Time Measurements, Proc. of VALUETOOLS 2009.
- [Men02t] D. A. Menascé. TPC-W: A Benchmark for E-Commerce, IEEE Internet Computing, 2002.
- [Pér17l] J. F. Pérez, G. Casale. LINE: Evaluating Software Applications in Unreliable Environments, IEEE Transactions on Reliability, 2017.
- [Rol09p] J. Rolia, G. Casale, D. Krishnamurthy, S. Dawson, S. Kraft. Predictive Modelling of SAP ERP Applications: Challenges and Solutions, Proc. of VALUETOOLS 2009.
- [Woo13t] M. Woodside, G. Franks, Tutorial Introduction to Layered Modeling of Software Performance, Edition 4.0, 2013.

Appendix A: Layered Queueing Networks

Layered queueing networks (LQNs) are a canonical form of extended queueing networks developed for modeling systems with nested simultaneous resource possession. In ‘ordinary’ queueing networks, hardware resources such as CPUs and disks are represented as stations while the software structure is implicitly taken into account in the service demand at each station. LQNs instead abstract a system as a layered hierarchy of interacting software entities hosted on hardware entities. This layered hierarchy arises from client-server interactions commonly found in software structures. The capabilities to separate software and hardware entities as well as to identify the layers of the software structure make LQNs a modeling formalism particularly suitable to describe modern software systems.

An LQN can be thought of as a directed graph, where each vertex represents either a software or a hardware entity and each edge denotes a dependency of one entity on another. In LQNs, software entities are called tasks while hardware entities are named processors. Every task is hosted on a single processor and contains a set of entries, which represent different services. Entries are defined as consisting of phases or activities. Each phase or activity mirrors an execution step and is allowed to send requests to entries of other tasks.

By default, a task serves only one request at a time and buffers extra incoming requests in a queue. It is possible to assign a task a multiplicity, which enables the task to serve the number of requests corresponding to the multiplicity in parallel. Tasks may schedule requests according to three disciplines:

- First-in first-out. Requests are served in the order of their arrival.
- Preemptive-resume priority. Requests are served in the order of their priority. Incoming requests with higher priority will preempt lower-priority requests currently in service.
- Head-of-line priority. Requests are served in the order of their priority. Incoming requests with higher priority will not preempt lower-priority requests currently in service.

The top layer of every LQN consists of a special type of task, termed the reference task, which typically has a single entry and never accepts requests from others. A reference task represents a group of clients with statistically identical behavior. Apart from a multiplicity, a think time needs to be specified for each reference task. The multiplicity determines the population of clients while the think time indicates how long a client is expected to spend between two consecutive interactions. Thus, reference tasks naturally model closed workloads. Open workloads can also be easily obtained by letting a reference task send asynchronous requests to a normal task.

Despite being hardware entities, processors behave similarly to normal tasks except that they do not require service from others. Each processor has a multiplicity and can serve the number of requests up to its multiplicity at the same time. Extra incoming requests are buffered in a queue. In addition

to first-in first-out, preemptive-resume priority and head-of-line priority, three more scheduling disciplines are available for processors:

- Random. Requests are served in a random order.
- Processor sharing. All the requests are served simultaneously, each receiving an equal fraction of the service capacity.
- Generalized processor sharing. Requests are divided into different groups. Each group is assigned a positive weight and receives a weighted fraction of the service capacity, which is shared equally among requests within the group.

An entry can accept either synchronous or asynchronous requests, but not both. In the former case, it must reply to each request or forward the request to another task. The behavior of an entry may be specified through phases or activities. Phases are essentially activities that are connected sequentially. Activities are used to depict a sophisticated control flow. Each phase or activity needs to be assigned a host demand, which is the average service time that it requires from the hosting processor.

Regardless of the specification approach, an entry is conceptually considered as comprising two phases. The first phase, known as the service phase, spans all the activities necessary to reply to the request. The second phase, known as the autonomous phase, covers the remaining activities needed to complete the request and is executed in parallel with the request sender.

A phase or activity can send more than one requests. The pattern of these requests may be either stochastic or deterministic. If the request pattern of a phase or activity is stochastic, a random number of requests are sent to each target entry. The number of requests follows a geometric distribution with the specified mean. A phase or activity with a deterministic request pattern sends exactly the specified number of requests to each target entry.

Activities in an entry are connected by precedences to form a control flow graph. There are two classes of precedences, namely pre-precedences and post-precedences. The former are used to merge multiple executions while the latter are used to split a single execution. To connect two activities, one needs to define a pair of pre-precedence and post-precedence, and then associates the source and target activities with the pre-precedence and post-precedence respectively. Table A.1 summarizes precedences support by LQNs.

Table B.1: Precedences supported by LQNs.

Class	Name	Description
Pre-precedences	Sequence	Transfer control from an activity
	And-Join	Merge all the concurrent executions
	Quorum-Join	Merge some concurrent executions

	Or-Join	Merge branch executions
Post-precedences	Sequence	Transfer control to an activity
	And-Fork	Split into concurrent executions
	Or-Fork	Split into branch executions
	Loop	Repeat subsequent executions

Appendix B: QoS Measures Provided by LQNs

Table B.1: Entity measures provided by LQNs.

Measure	Entity	Definition
Utilization	Task, Entry, Activity	Mean number of requests at the node
Phase 1 Utilization	Task, Entry	Mean number of requests in phase 1 at the node
Phase 2 Utilization	Task, Entry	Mean number of requests in phase 2 at the node
Service Time	Activity	Average service time of a request at the node
Phase 1 Service Time	Entry	Average service time of a request in phase 1 at the node
Phase 2 Service Time	Entry	Average service time of a request in phase 2 at the node
Throughput	Task, Entry, Activity	Average completion rate of requests at the node
Processor Waiting Time	Activity	Average queueing time of a request from the node at its processor
Processor Utilization	Task, Entry, Activity	Mean number of requests from the node at its processor

Table B.2: Request measures provided by LQNs.

Measure	Request	Definition
Waiting Time	Synchronous, Asynchronous	Average queueing time of a request on the edge

Appendix C: Optimization Program Notation

Table C.1: Main notation used to formulate deployment optimization.

Symbol	Definition
\mathcal{F}	Set of function nodes
\mathcal{S}	Set of storage nodes
m_i	Memory of a function node i
c_i	Concurrency of a function node i
P^{FI}	Price for function invocation
P^{FE}	Price for function execution
P_k^{SO}	Price for storage operation k
M_i^{P}	Multiplicity of the processor for node i
M_i^{T}	Multiplicity of the task for node i
$D_{i,k,r}$	Host demand of a request in activity r of entry k at node i
$Y_{(i,k,r),(j,l)}$	Number of requests from activity r of entry k at node i to entry l at node j
$B_{i,k}$	Mean buffer length of requests to entry k at node i
B_i	Mean buffer length of requests at node i
$U_{i,k}^{\text{PH1}}$	Average phase-1 utilization of requests in entry k at node i
$U_{i,k}^{\text{PH2}}$	Average phase-2 utilization of requests in entry k at node i
$X_{i,k,r}$	Average throughput of requests in activity r of entry k at node i
$X_{i,k}$	Average throughput of requests in entry k at node i
$W_{(i,k,r),(j,l)}$	Average waiting time of a request from activity r of entry k at node i to entry l at node j

$S_{i,k}^{PH1}$	Average phase-1 service time of a request to entry k at node i
$S_{i,k}^{PH2}$	Average phase-2 service time of a request to entry k at node i
$R_{i,k}$	Average response time of a request to entry k at node i
R_i	Average response time of a request at node i
C_i	Operating cost of node i
C	Total operating cost of all the nodes

Appendix D: RADON Model for Thumbnail Generation

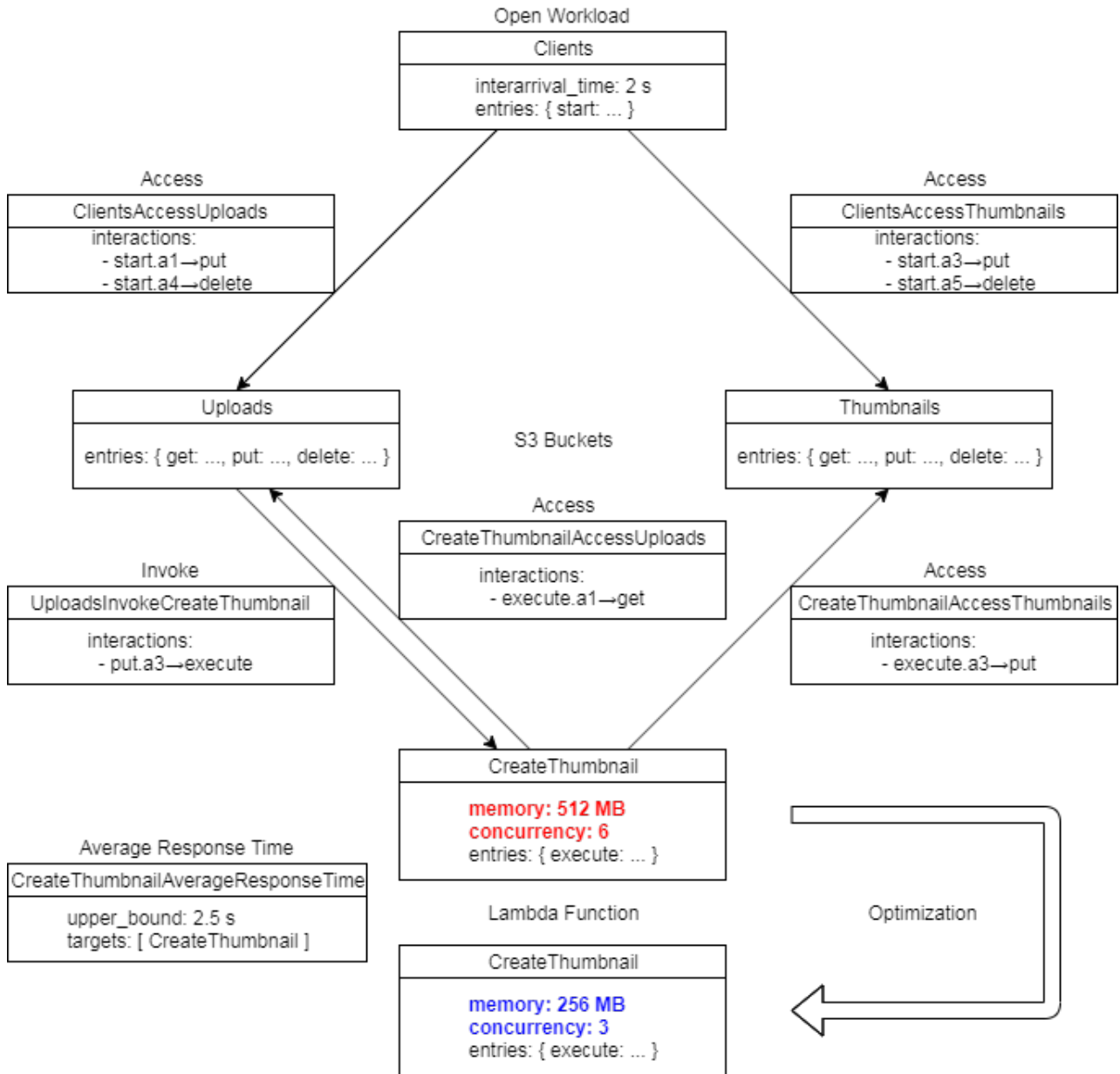


Figure A.1: The RADON model of the *Thumbnail Generation* example.

Appendix E: Validation Example Measurements

Table E.1: Time points recorded for creating the benchmark.

Symbol	Node	Description
t_{start}^{PI}	clients	Start time of putting the image
t_{end}^{PI}	clients	End time of putting the image
t_{end}^{LI}	clients	End time of listing the image
t_{end}^{LT}	clients	End time of listing the thumbnail
t_{start}^{GT}	clients	Start time of getting the thumbnail
t_{end}^{GT}	clients	End time of getting the thumbnail
t_{start}^{DI}	clients	Start time of deleting the image
t_{end}^{DI}	clients	End time of deleting the image
t_{start}^{DT}	clients	Start time of deleting the thumbnail
t_{end}^{DT}	clients	End time of deleting the thumbnail
t_{start}^{GI}	create_thumbnail	Start time of getting the image
t_{end}^{GI}	create_thumbnail	End time of getting the image
t_{start}^{RZ}	create_thumbnail	Start time of resizing the image into the thumbnail
t_{end}^{RZ}	create_thumbnail	End time of resizing the image into the thumbnail
t_{start}^{PT}	create_thumbnail	Start time of putting the thumbnail
t_{end}^{PT}	create_thumbnail	End time of putting the thumbnail

Table E.2: Measures and results for the host demands of activities.

Activity	Measure	Result
uploads.put.a2	$t_{\text{end}}^{LI} - t_{\text{end}}^{PI} *$	0.109 s
thumbnails.put.a2	$t_{\text{end}}^{LT} - t_{\text{end}}^{PT} *$	0.109 s
create_thumbnail.execute.a2	$t_{\text{end}}^{RZ} - t_{\text{end}}^{RZ}$	3.607 s

* These two measures are collected in a separate experiment.

Table E.3: Measures and results for the network delays of requests.

Source Activity	Target Entry	Measure	Result
clients.start.a1	uploads.put	$t_{\text{end}}^{PI} - t_{\text{start}}^{PI}$	0.153 s
clients.start.a3	thumbnails.get	$t_{\text{end}}^{GT} - t_{\text{start}}^{GT}$	0.076 s
clients.start.a4	uploads.delete	$t_{\text{end}}^{DI} - t_{\text{start}}^{DI}$	0.108 s
clients.start.a5	thumbnails.delete	$t_{\text{end}}^{DT} - t_{\text{start}}^{DT}$	0.035 s
uploads.put.a3	create_thumbnail.execute	$(t_{\text{end}}^{LT} - t_{\text{end}}^{PI}) - (t_{\text{end}}^{LI} - t_{\text{end}}^{PI})$ $-(t_{\text{end}}^{PT} - t_{\text{start}}^{GI}) - (t_{\text{end}}^{LT} - t_{\text{end}}^{PT})$	0.264 s
create_thumbnail.execute.a1	uploads.get	$t_{\text{end}}^{GI} - t_{\text{start}}^{GI}$	0.099 s
create_thumbnail.execute.a3	thumbnails.put	$t_{\text{end}}^{PT} - t_{\text{start}}^{PT}$	0.136 s