RDOF: Deployment Optimization for Function as a Service

Lulai Zhu
Department of Computing
Imperial College London
London, UK
lulai.zhu15@imperial.ac.uk

Giorgos Giotis
Innovation Lab
Athens Technology Center
Athens, Greece
g.giotis@atc.gr

Vasilis Tountopoulos
Innovation Lab
Athens Technology Center
Athens, Greece
v.tountopoulos@atc.gr

Giuliano Casale
Department of Computing
Imperial College London
London, UK
g.casale@imperial.ac.uk

***Abstract*—Function as a service (FaaS) simplifies the runtime resource management of cloud applications and enables fine-grained scaling and billing at the function level, thus becoming the most widespread serverless paradigm today. Cost-effective use of FaaS entails appropriately deploying individual functions. We propose in this paper RDOF[1], a model-driven approach to deployment optimization for FaaS. RDOF predicts the performance of a FaaS-based application by instantiating a layered queueing network and finds the optimal configuration of each function such that the total operating cost is minimized under the specified performance requirements. We have validated RDOF on Amazon Web Services (AWS) and implemented it in an online tool that operates on TOSCA metamodels.**

***Index Terms*—deployment optimization, function as a service, layered queueing networks, TOSCA**

I. Introduction

Serverless computing emerges as a novel resource provisioning solution for cloud applications. It aims at entirely bypassing user involvement in runtime resource management, which may radically evolve the landscape of today's cloud technologies [1]. As a result of the increasing trend for software vendors to migrate their products into the microservices architectural style, function as a service (FaaS) has become the most prominent paradigm of serverless computing. The basic idea behind FaaS is to deploy applications under development as individual functions and serve function invocations remotely and automatically from the cloud [2]. This leads to two significant advantages: fine-grained autoscaling and productivity gain [3].

Although FaaS can simplify runtime resource management and save operating costs through fine-grained scaling and billing, new challenges arise in practicing DevOps on FaaS-based applications [4]. One remaining to be addressed is how to find the optimal deployment scheme of a FaaS-based application that minimizes the total operating cost while satisfying the specified performance requirements, e.g. service-level agreements. In this paper, we propose a model-driven approach to deployment optimization for FaaS, RDOF[1], which uses a layered queueing network (LQN) [5] to predict performance measures and a genetic algorithm (GA) [6] to optimize deployment schemes. Evaluation results indicate that the relative error of the LQN in performance prediction is less than 10% on Amazon Web Services (AWS) and the saving rate of RDOF is greater than 15% against a baseline approach that enforces an identical configuration for all the functions. A tool implementing RDOF has been made available online. Its input and output are TOSCA metamodels that can be consumed by orchestrators for increased automation.

The rest of the paper is organized as follows. Section II uses a FaaS-based application to show the motivation behind RDOF. This application will act as a running example in other sections. Section III describes the LQN for performance prediction. Details about RDOF together with an introduction to the online tool are given in Section IV. The accuracy of the performance model and the utility of the optimization approach are evaluated in Section V. Section VI demonstrates the applicability of RDOF through a case study in industry. Section VII is dedicated to the conclusions.

II. Motivation

Most public cloud platforms offer FaaS compute services, among which AWS Lambda is in our experience the most mature and popular one. The runtime resource management of a Lambda function is fully automated. However, users can control the autoscaling of the function through configuration of *memory* and *concurrency*. These two parameters are critical to the performance and operating costs of a FaaS-based application.

When a function is invoked, Lambda assigns an event to an instance of that function. Each instance may only handle one event at a time. If all the instances of the function are currently busy, a new instance is created to process the event. Resources allocated to the new instance, including both memory and CPUs, are in proportion to the memory of the function, while the maximum number of instances that the function can have is specified by its concurrency.

Lambda works along with other AWS services such as Simple Queue Service (SQS), API Gateway and Simple Storage Service (S3) to invoke functions. S3 is an object storage service with scalability, high availability and low latency.

[1]RDOF stands for RADON deployment optimization for FaaS. RADON is a project funded by the European Union's Horizon 2020 research and innovation program under Grant No. 825040. This work is considered as an contribution of the RADON project. The referenced data has been published at https://zenodo.org/ with DOIs 10.5281/zenodo.3894501, 10.5281/zenodo.3894566 and 10.5281/zenodo.5043700

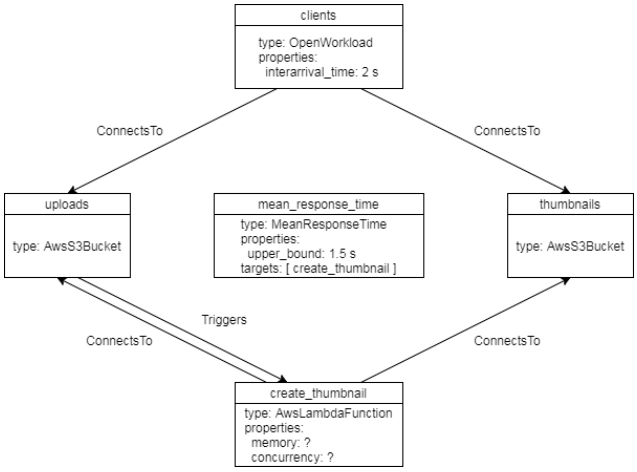

Fig. 1. The TOSCA metamodel of the thumbnail generation application.

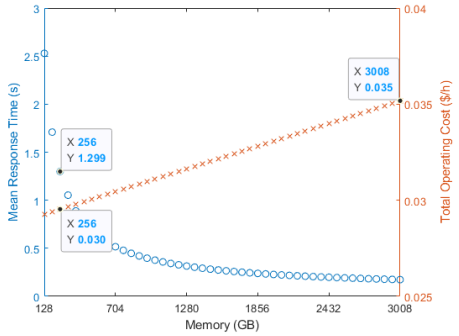

Fig. 2. The mean response time of the `create_thumbnail` function versus the total operating cost of the thumbnail generation application.

It allows AWS users to store untyped binary objects in a hierarchical, file-system-like structure. Each object in S3 is identified by a unique key and located in a virtual component called the bucket.

Topology and Orchestration Specification for Cloud Applications (TOSCA) [7] is an OASIS standard language to describe cloud applications and their orchestration workflows. In TOSCA, a cloud application is represented as a topology containing nodes, relationships and policies attached to them. RDOF complies with the definitions of node, relationship and policy types in a public repository called RADON Particles[2], which extends the TOSCA YAML profile[3] with support for FaaS.

Figure 1 illustrates the TOSCA metamodel of a FaaS-based application, thumbnail generation, which is composed of two S3 buckets named `uploads` and `thumbnails` as well as a Lambda function named `create_thumbnail`. When an image is put into the `uploads` bucket, S3 produces an event to invoke the `create_thumbnail` function asynchronously. The `create_thumbnail` function then gets the image from the `uploads` bucket, resizes it into a thumbnail and puts the result into the `thumbnails` bucket.

As defined by the `clients` node, the reference workload of the thumbnail generation application is an infinite stream of clients with a mean interarrival time of 2 s. We assume that every client carries out a sequence of activities: (i) put an image into the `uploads` bucket; (ii) wait 10 s for the thumbnail to be ready; (iii) get the result from the `thumbnails` bucket; (iv) delete the image from the `uploads` bucket; (v) delete the result from the `thumbnails` bucket. The mean response time of the `create_thumbnail` function is expected to be less than 1.5 s. This performance requirement is specified through the `mean_response_time` policy.

To show the motivation behind RDOF, we use the performance and cost models present in Sections III-A and IV-A respectively to predict the mean response time of the `create_thumbnail` function and to calculate the total operating cost of the thumbnail generation application for different memories but an infinite concurrency under the reference workload. As shown in Figure 2, the performance requirement can be satisfied with a minimum total operating cost of 0.030 $/h when the memory of the function is set to 256 MB. Although a memory configuration of 3008 MB is also feasible, the total operating cost increases by around 17%. The goal of RDOF is to solve this kind of deployment optimization problem.

III. Performance Prediction

A. Layered Queueing Network

LQNs are a canonical form of extended queueing networks developed for modeling systems with nested simultaneous resource possession [5]. An LQN can be thought of as a directed graph, where each vertex represents either a software or a hardware entity and each edge represents a dependency of one entity on another. In LQNs, software entities are termed tasks while hardware entities are termed processors. Every task is hosted on a single processor and contains a set of entries, which denotes the interface of the software entity. Entries are defined as consisting of phases or activities. Each phase or activity mirrors an execution step and is allowed to send either synchronous or asynchronous requests to the entries of other tasks. We refer the reader to [8] for the notation and features of LQNs.

The performance of the thumbnail generation application is essentially determined by the behavior of the `clients` node, the `uploads` and `thumbnails` buckets, the `create_thumbnail` function as well as the remote interactions between them. Workloads, either open or closed, can be represented in LQNs using a primitive construct called the reference task. We elaborate below the LQN structures for modeling S3 buckets, Lambda functions and remote interactions. Each task in these structures is hosted on a dedicated processor with the same multiplicity, and can thus be simply regarded as a queueing station. For conciseness, we omit both textual and graphical depiction of processors. Some

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

[3]https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html

LQN structures contain immediate activities that demand zero service time from the hosting processor. Since LQNs with deterministic service times are not analytically tractable, every immediate activity is assigned a small mean service time of δS as an approximation.

1) Modeling S3 Buckets: We model an S3 bucket in LQNs as a task with an infinite multiplicity and are particularly concerned about three operations involved in the thumbnail generation application, i.e. GET, PUT and DELETE. As illustrated in Figure 3, each operation is represented by an individual entry of the task. Because how long an S3 bucket spends in responding to a request is opaque, the first activity of all the entries is defined to be an immediate activity. We take this into account in the network delay of the associated interaction. Upon completion of a PUT operation, the new object may not be instantly accessible from an S3 bucket. A second activity is introduced to exactly capture this two-phase behavior. The third activity is needed only when the bucket is configured to invoke a Lambda function afterward.

2) Modeling Lambda Functions: Figure 4 shows the LQN structure for modeling a Lambda function, which is a task with a multiplicity equal to the concurrency of the function. A single entry is defined in the task to depict the execution logic of the function as an activity graph. A few points are worthy of notice in using such an LQN structure to model a Lambda function:

- *Cold start.* A function instance is deleted by Lambda after it has been idle or existent for a long time [9]. Then, a new instance may be created to process an incoming event, and an extra time is needed to establish the runtime environment of that instance. This is known as cold start. We ignore the effects of cold start in the proposed LQN structure by assuming the reference workload of the application is intensive such that the interarrival times of events to any instance almost never exceed the idle time of Lambda functions.
- *Service times.* Since the memory of a function specifies the amount of memory and CPU resources available for its instances, the mean service times of activities in the entry are inversely proportional to the configured memory when running an instance of the function does not require more memory resources. This relation not only enables us to accurately estimate the mean service times of function activities through linear regression across different memories but also tells how the optimization procedure should modify the LQN with respect to the memory configuration of functions.
- *Retrial.* If a function does not have enough concurrency to process all the incoming events, additional invocations are throttled. Lambda manages an event queue for each function and retries throttled invocations after a delay. The proposed LQN structure does not capture the retrial behavior of a function and therefore is valid only if invocations to the function are rarely throttled. This condition is equivalent to that requests to the task representing the function are almost never queued.

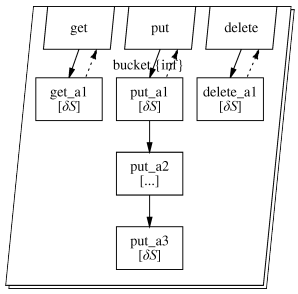

Fig. 3. The LQN structure for modeling an S3 bucket.

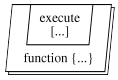

Fig. 4. The LQN structure for modeling a Lambda function.

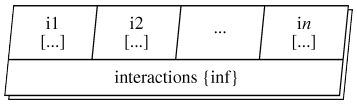

Fig. 5. The LQN structure for modeling a group of remote interactions.

3) Modeling Remote Interactions: Remote interactions between a pair of source and target nodes are modeled by a task with an infinite multiplicity, as illustrated in Figure 5. Each entry of the task represents a single interaction, which may be either synchronous or asynchronous. Figure 6 shows the entry for modeling a synchronous interaction. Both requests and responses between the source activity and the target entry are forwarded by this entry. The network delay is incorporated into the activity of the entry as a service time, which results in it being divided in a balanced manner between any pair of request and response. Figure 7 shows the entry for modeling an asynchronous interaction. 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 requests. Therefore, the network delay is fully pinned to any request.

B. Service Time Estimation

Two methods are commonly used for service time estimation, namely utilization-based [10] and response-time-based [11]. As mentioned in the last section, we model a Lambda function as a multi-server queueing station where each server is corresponding to a function instance. Because the instances of a Lambda function are distributed among anonymous hosts that may change over time, it is difficult to measure quantities required by these methods. Lambda allocates dedicated resources to every instance of a function. The mean service times of activities performed by Lambda functions are invariant to the intensity of the workload. Thus, we can in fact estimate them through direct measurement. This simple method is also applicable to S3 buckets and remote interactions, which are modeled as infinite-server queueing stations that only introduce pure delays.

However, the service time of a function activity may not always directly measurable. Take the `create_thumbnail` function for example. This function carries out three sequential activities: (i) get the image from the `uploads` bucket; (ii)

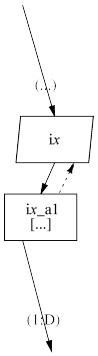

Fig. 6. The entry for modeling a synchronous interaction.

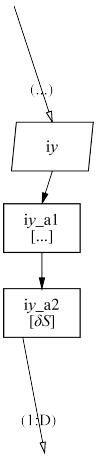

Fig. 7. The entry for modeling an asynchronous interaction.

resize the image into a thumbnail; (iii) put the result into the `thumbnails` bucket. The execution time of the first activity includes the service time spent in completing the activity itself and the waiting time spent in accessing the `uploads` bucket, which cannot be measured separately. Let $E_{i,k,r}$, $S_{i,k,r}$ and $W_{i,k,r}$ be the mean execution, service and waiting times of activity r in entry k at node i respectively. In general, we have

$$E_{i,k,r} = S_{i,k,r} + W_{i,k,r}. \tag{1}$$

The mean service time of activity r at a function node i is in inverse proportion to the configured memory:

$$S_{i,1,r} = \frac{K_{i,1,r}}{m_i} \quad \text{for } i \in \mathcal{F}, \tag{2}$$

where $K_{i,1,r}$ is the corresponding proportionality constant, m_i is the memory of the function node i, and $\mathcal{F}$ is the set of function nodes. Applying (2) to (1) yields

$$E_{i,1,r} = \frac{K_{i,1,r}}{m_i} + W_{i,1,k} \quad \text{for } i \in \mathcal{F}. \tag{3}$$

(3) and (2) provides a two-step method for service time estimation of a Lambda function. One can estimate the proportionality constant and the mean waiting time of an activity by fitting (3) through linear regression across different memories and then calculate the mean service time of that activity for a particular memory from (2).

IV. Deployment Optimization

A. Problem Formulation

The operating cost of a Lambda function is split into the costs of invocation requests, execution duration and data transfer. Lambda counts a request every time the function is invoked. The duration of a function execution is rounded up to the nearest multiple of 1 ms, and the price depends on the memory configuration. Data transferred to or from outside the AWS region where the function resides is charged as well.

Let $E_{i,k}$ and $X_{i,k}$ be the mean execution time and throughput of entry k at node i respectively. We calculate the operating cost of a function node i as

$$C_i = (P^{\text{FI}} + P^{\text{FE}} m_i (E_{i,1} + 0.001)) X_{i,1} \quad \text{for } i \in \mathcal{F}, \tag{4}$$

where P^{FI} and P^{FE} are the prices for function invocation and execution. (4) does not take into account the cost of data transfer, which is often negligible compared to the others. Apart from that, the cost of execution duration given in (4) is an upper bound since the expectation of a rounded random variable is generally undecidable.

Object storage like S3 buckets is charged according to operation requests, space usage and data transfer. The operating cost of a storage node i is given by

$$C_i = \sum_k P_k^{\text{SO}} X_{i,k} \quad \text{for } i \in \mathcal{S}, \tag{5}$$

where P_k^{SO} is the price for storage operation k, and $\mathcal{S}$ is the set of storage nodes. Assuming that objects will not be kept for a long time, we drop the costs of space usage and data transfer in (5). This assumption is true for the thumbnail generation application, as described in the Section II.

Let $C = \sum_i C_i$ be the total operating cost of the application under development, and let $\boldsymbol{m} = (m_i)$ and $\boldsymbol{c} = (c_i)$ be the vectors of the configured memory and concurrency at each node[4]. We formulate the deployment optimization problem for an application composed of serverless functions and object storage as follows:

$$\begin{aligned} \min_{\boldsymbol{m},\boldsymbol{c}} \quad & C(\boldsymbol{m}, \boldsymbol{c}), \\ \text{s.t.} \quad & \boldsymbol{m} \in \{\forall i \in \mathcal{F}, m_i \in M\}, \\ & \boldsymbol{c} \in \{\forall i \in \mathcal{F}, c_i \in \mathbb{Z}_{>0}\}, \\ & \boldsymbol{a} \le \boldsymbol{G}(\boldsymbol{m}, \boldsymbol{c}) \le \boldsymbol{b}, \end{aligned} \tag{6}$$

where M is the set of memory options for function nodes, $\mathbb{Z}_{>0}$ is the set of positive integers, and $\boldsymbol{G}$ is the vector of quantities, e.g. the mean response time of a function node i, that are expected to be bounded between $\boldsymbol{a}$ and $\boldsymbol{b}$. (6) considers the concurrency of a function to be unlimited as the maximum concurrency imposed by a FaaS compute service is normally upgradeable or large enough.

B. Complexity Reduction

As shown in the previous section, the deployment optimization of a FaaS-based application is an integer nonlinear programming problem, whose search space is extremely large even if the application comprises a few functions. Unlike the memory, the concurrency of a function does not contribute to its operating cost. An under-resourced concurrency configuration however can significantly affect the performance of the function due to extra delays arising from throttling and retrial of invocations. The optimal deployment scheme should maximize the concurrency of each function as necessary. Based on this argument, we divide the overall optimization

[4] m_i and c_i are undefined when node i is not a function.

problem into two subproblems to reduce the computational complexity of searching for the optimal solution:

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

$$\begin{aligned} \min_{\boldsymbol{m}} \quad & C(\boldsymbol{m}, \boldsymbol{c}), \\ \text{s.t.} \quad & \boldsymbol{m} \in \{\forall i \in \mathcal{F}, m_i \in M\}, \\ & \boldsymbol{c} \in \{\forall i \in \mathcal{F}, c_i = \infty\}, \\ & \boldsymbol{a} \leq \boldsymbol{G}(\boldsymbol{m}, \boldsymbol{c}) \leq \boldsymbol{b}. \end{aligned} \tag{7}$$

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

$$\begin{aligned} \min_{c_i} \quad & c_i, \\ \text{s.t.} \quad & \boldsymbol{m} \in \{\forall j \in \mathcal{F}, m_j = m_j^*\}, \\ & \boldsymbol{c} \in \{\forall (j \in \mathcal{F} \wedge j \neq i), c_j = \infty\}, \\ & B_i(\boldsymbol{m}, \boldsymbol{c}) \leq \delta B, \end{aligned} \tag{8}$$

  where B_i is the mean buffer length of the function node i, and δB is a small upper bound for the mean buffer lengths of function nodes.

We obtain from (7) the optimal memory allocation by setting the concurrencies of all the functions to infinity. The solution to (7) is then used in (8) to determine the optimal concurrency of each function individually while the concurrencies of the others are fixed at infinity. The last constraint in (8) is to limit the probability that an invocation to the function is throttled. Let b_i be the buffer length of a function node i. The probability of this random variable being zero can be written as

$$\mathbb{P}(b_i = 0) = 1 - \mathbb{P}(b_i \geq 1). \tag{9}$$

Notice that

$$B_i = \mathbb{E}(b_i) = \sum_{n=1}^{\infty} n\mathbb{P}(b_i = n) \geq \sum_{n=1}^{\infty} \mathbb{P}(b_i = n) = \mathbb{P}(b_i \geq 1). \tag{10}$$

By applying (9), (10) and (8), we get

$$\mathbb{P}(b_i = 0) = 1 - \mathbb{P}(b_i \geq 1) \geq 1 - B_i \geq 1 - \delta B \approx 1. \tag{11}$$

(11) indicates that invocations to the function node i would be rarely throttled.

Both memory allocation and concurrency determination subproblems are integer nonlinear programming problems. GAs are a powerful method to deal with this kind of optimization problem. The search space of the memory allocation subproblem grows exponentially with respect to the number of functions. To speed up the optimization procedure, we first use a derivative-based algorithm to find the continuous optimal solution, from which an initial population range is decided for GAs to search more effectively for the discrete optimal solution.

C. Tool Support

We have developed the so-called RADON decomposition tool[5], which implements RDOF for deployment optimization of FaaS-based applications and is accessible online through a RESTful API. This tool invokes the GA solver[6] and the LINE engine[7] to solve the aforementioned subproblems and LQN respectively. Details about the RADON decomposition tool can be found in [12], [13].

V. Evaluation

A. Accuracy of Performance Model

A benchmark of the thumbnail generation application is created to evaluate the accuracy of the LQN. We collect a set of 50 images and use the Locust framework[8] to simulate a single client that repeatedly picks an image at random and puts it to the `uploads` bucket. The memory and concurrency of the `create_thumbnail` function are set to 128 MB and 1. Service time estimation is conducted through direct measurement within a relative error of 5% at the 99% confidence level. For activities whose service times are not directly measurable, i.e. the first and third activities of the `create_thumbnail` function, we estimate their mean execution and waiting times under two memory configurations, 128 MB and 320 MB, and then compute their mean service times by applying (3) and (2).

The accuracy of the LQN is evaluated in terms of its relative error in predicting the mean response time of the `create_thumbnail` function. To this end, three groups of experiments are designed by varying the memory and concurrency of the `create_thumbnail` function as well as the interarrival time of the reference workload. We parameterize the LQN based on the created benchmark and use (2) to calculate the mean service times of function activities for different memories. In particular, the small mean service time δS assigned to every immediate activity is chosen to be 0.001 s. Accuracy evaluation results for the three groups of experiments are reported in Tables I, II and III. The measured values are obtained under the same statistical criterion as for the benchmark.

The relative error of the LQN in predicting the mean response time of the `create_thumbnail` function is observed to be less than 10% in all the cases. Additionally, to what extent invocations to a function would be throttled is well anticipated by checking whether requests to the task representing that function are queued. We fix the small upper bound δB for the mean buffer lengths of functions at 0.01 in the implementation of RDOF. As implied by (11), the probability of an invocation to any function being throttled would be less than or equal to 1%, which is sufficiently small to guarantee the accuracy of the LQN.

B. Utility of Optimization Approach

We evaluate the utility of RDOF in terms of its saving rate against a baseline approach that can easily find a feasible solution. Similar to RDOF, this approach also divides the

[5]https://github.com/radon-h2020/radon-decomposition-tool

[6]https://www.mathworks.com/help/gads/ga.html

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

[8]https://locust.io

TABLE I
ACCURACY EVALUATION RESULTS FOR DIFFERENT MEMORIES BUT THE SAME CONCURRENCY UNDER THE REFERENCE WORKLOAD.

Memory	Concurrency	Predicted Value	Measured Value	Relative Error	Task Queueing	Function Throttling
192 MB	4	1.720 s	1.605 s	7.2%	$\leq$ 3.0%	$\leq$ 3.0%
256 MB		1.304 s	1.218 s	7.1%	$\leq$ 1.0%	$\leq$ 1.0%
320 MB		1.054 s	1.008 s	4.6%	$\leq$ 0.3%	$\leq$ 0.3%

TABLE II
ACCURACY EVALUATION RESULTS FOR THE SAME MEMORY BUT DIFFERENT CONCURRENCIES UNDER THE REFERENCE WORKLOAD.

Memory	Concurrency	Predicted Value	Measured Value	Relative Error	Task Queueing	Function Throttling
256 MB	3	1.335 s	1.219 s	9.5%	$\leq$ 3.0%	$\leq$ 3.0%
	4	1.304 s	1.218 s	7.1%	$\leq$ 1.0%	$\leq$ 1.0%
	5	1.300 s	1.210 s	7.4%	$\leq$ 0.3%	$\leq$ 0.3%

TABLE III
ACCURACY EVALUATION RESULTS FOR DIFFERENT INTERARRIVAL TIMES UNDER THE OPTIMAL CONFIGURATION (256 MB AND 4).

Interarrival Time	Predicted Value	Measured Value	Relative Error	Task Queueing	Function Throttling
1 s	1.332 s	1.247 s	6.8%	$\leq$ 3.0%	$\leq$ 3.0%
2 s	1.304 s	1.218 s	7.1%	$\leq$ 1.0%	$\leq$ 1.0%
3 s	1.301 s	1.214 s	7.2%	$\leq$ 0.3%	$\leq$ 0.3%

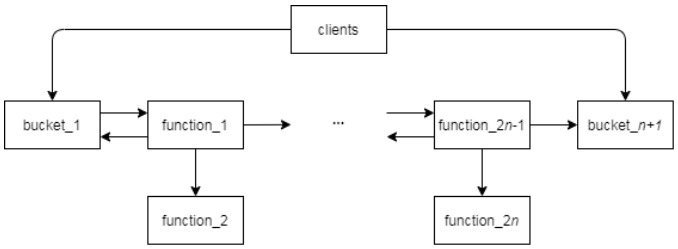

Fig. 8. The topology of the derived TOSCA metamodels.

overall problem into two subproblems, memory allocation and concurrency determination. It sets the concurrencies of all the functions to infinity and increases their memories from the minimum until the performance requirements are satisfied. The concurrency configuration is then obtained by incrementing the concurrencies of the functions from 1 until the mean buffer length of every function is no more than the small upper bound δB.

The utility evaluation of RDOF is carried out on a set of 90 TOSCA metamodels derived from that of the thumbnail generation application. Figure 8 illustrates the topology of these TOSCA metamodels, each of which defines a chain of main functions with a synchronous invocation to their auxiliary function. A bucket is added in between two adjacent main functions to connect the output of the former to the input of the latter. The reference workload is the same as before except that the time spent by a client in waiting for the result is scaled with respect to the number of functions. We reuse the benchmark of the thumbnail generation application to parametrize the LQN. However, the mean time spent by a function in processing an object is randomly generated. We also expect the mean total response time of the main functions to be bounded from above. Table IV reports parameter settings of the experiments.

In each experiment, RDOF and the baseline approach are used to respectively provide an optimal and a feasible deployment scheme for a derived TOSCA metamodel. We compute the total operating costs of the two solutions and then the saving rate of the optimal to the feasible solution. Table V reports utility evaluation results for different numbers of functions under the reference workload. It can be seen that the deployment scheme found by RDOF saves on average more than 15% operating costs compared to the one given by the baseline approach, which is significant in practice.

VI. CASE STUDY

To demonstrate its applicability, RDOF has been adopted to deployment optimization of a natural language processing (NLP) pipeline from an industrial use case. This application consists of a chain of two Lambda functions, `ner` and `pos`, connected through three S3 buckets, `pipeline_in`, `pipeline_mid` and `pipeline_out`. Figure 9 illustrates the TOSCA metamodel of the NLP pipeline application. The reference workload of the application is defined by the `clients` node, which puts posts as JSON files into the `pipeline_in` bucket with a mean interarrival time of 0.1 s. Once a new JSON file is available, the `ner` function is invoked to get that file from the `pipeline_in` bucket and recognizes various named entities in the post. The JSON file is then updated and put into the `pipeline_mid` bucket. Following a similar procedure, the `pos` function parses the post and enriches the JSON file with part-of-speech tags. The final result will be archived in the `pipeline_out` for future analysis. The `ner` and `pos` functions are also responsible for deleting the original JSON files from their input buckets. As specified by the `total_mean_response_time` policy, the total mean response time of the `ner` and `pos` functions is expected to be no more than 5 s.

We first create a benchmark of the NLP pipeline application to parametrize the LQN. A set of 850 posts is collected, and the Locust framework is used to simulate a synchronized client that randomly picks a post and puts it as a JSON file into the `pipeline_in` bucket only after the result of the previous post is ready in the `pipeline_out` bucket. We set the memory and concurrency of both `ner` and `pos` functions to be 256 MB and 1, because Lambda needs at least 256 MB memory to run an instance of the two functions. This introduces an extra constraint in the memory allocation subproblem. We estimate the mean service times of the activities through direct measurement within a relative error of 5% at the 99% confidence level. In particular, an additional measurement under a memory configuration of 768 MB is carried out to compute the mean service times of function activities by applying (3) and (2). The small mean service time δS of every immediate activity is chosen to be 0.001 s.

The RADON decomposition tool is then used to optimize

TABLE IV
PARAMETER SETTINGS OF THE UTILITY EVALUATION EXPERIMENTS.

Parameter	Optional Values
Number of functions	2, 4, 8
Number of buckets	$\|\mathcal{F}\|+1$
Time spent by a client in waiting for the result	$10\|\mathcal{F}\|$ s
Mean time spent by a function in processing an object	$[1, 4]$ s*
Upper bound for the mean total response time	$1.5\|\mathcal{F}\|$ s
Random seed	1, 2, …, 30

* Given a derived TOSCA metamodel, we generate the mean times spent by the functions in processing an object through the MATLAB command `round(3*gallery('uniformdata',[scale,1],seed)+1,3)`, where `scale` is the number of functions and `seed` is the random seed in use.

TABLE V
UTILITY EVALUATION RESULTS FOR DIFFERENT NUMBERS OF FUNCTIONS UNDER THE REFERENCE WORKLOAD.

Number of Functions	Optimization Approach		Baseline Approach		Average Saving Rate
	Average Total Memory	*Average Total Concurrency*	*Average Total Memory*	*Average Total Concurrency*	
2	2970 MB	6	717 MB	9	17.5%
4	6306 MB	12	1451 MB	19	20.4%
8	12582 MB	25	2987 MB	40	21.6%

the deployment of the NLP pipeline application. The optimal memory and concurrency configurations of the `ner` and `pos` functions are found to be 512 MB, 41 and 576 MB, 37 respectively. To validate this solution, we configure the actual deployment of the application on AWS accordingly and carry out a load test using again the Locust framework to simulate the reference workload. The total mean response time of the two functions is measured to be 4.529 s under the same statistical criterion as for the benchmark, thus satisfying the specified performance requirement. In addition, the value predicted by the LQN is 4.960 s with a relative error of 9.5%.

VII. CONCLUSION

A model-driven approach to deployment optimization for FaaS, named RDOF, is proposed in this paper. RDOF can find the optimal deployment scheme of a FaaS-based application that minimizes the total operating cost under the specified performance requirements. We model the performance of FaaS-based applications using an LQN, which exhibits a relative error of less than 10% on AWS. To reduce computational complexity, the deployment optimization problem is divided into two subproblems so that the memory and concurrency configurations of functions are decided separately. RDOF has been compared with a baseline approach that obtains a feasible deployment scheme of a FaaS-based application by treating all the functions identically. It achieves a saving rate of greater than 15% against this simple approach.

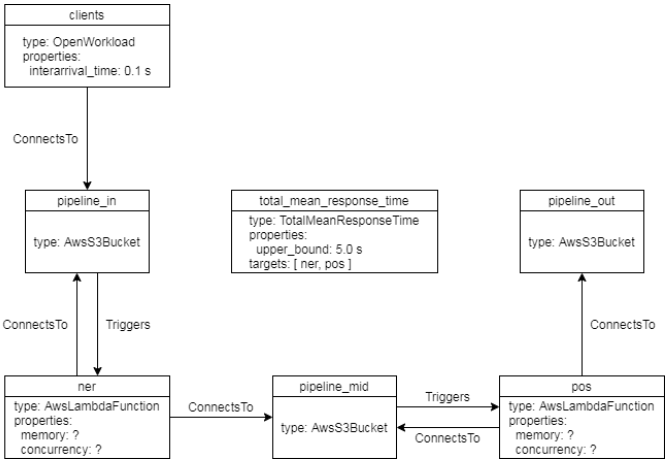

Fig. 9. The TOSCA metamodel of the NLP pipeline application.

[1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[2] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research," *arXiv preprint arXiv:1708.08028*, 2017.

[3] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, S. N. Srirama, D. A. Tamburri, M. Wurster, and L. Zhu, "RADON: Rational Decomposition and Orchestration for Serverless Computing," *Software-Intensive Cyber-Physical Systems*, pp. 1–11, 2019.

[4] V. Yussupov, U. Breitenbücher, F. Leymann, and M. Wurster, "A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 229–240.

[5] M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 20–34, 1995.

[6] T. M. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[7] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: Portable Automated Deployment and Management of Cloud Applications," in *Advanced Web Services*. Springer, 2014, pp. 527–549.

[8] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz, *Layered Queueing Network Solver and Simulator User Manual*, Department of Systems and Computer Engineering, Carleton University, 2013.

[9] L. Muller, C. Chrysoulas, N. Pitropakis, and P. J. Barclay, "A Traffic Analysis on Serverless Computing Based on the Example of a File Upload Stream on AWS Lambda," *Big Data and Cognitive Computing*, vol. 4, no. 4, p. 38, 2020.

[10] S. Spinner, G. Casale, F. Brosig, and S. Kounev, "Evaluating Approaches to Resource Demand Estimation," *Performance Evaluation*, vol. 92, pp. 51–71, 2015.

[11] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson, "Estimating Service Resource Consumption from Response Time Measurements," in *Proceedings of the 4th International ICST Conference on Performance Evaluation Methodologies and Tools*, 2009, pp. 1–10.

[12] G. Casale, A. Gias, and L. Zhu, "Deliverable 3.2 - Decomposition Tool I," RADON Consortium, Tech. Rep., 2019.

[13] G. Casale, A. Gias, and L. Zhu, "Deliverable 3.3 - Decomposition Tool II," RADON Consortium, Tech. Rep., 2020.

REFERENCES