Optimal Resource Configuration of Complex Services in the Cloud

Abhinandan S. Prasad*, David Koll*, Jesus Omana Iglesias†, Jordi Arjona Aroca†, Volker Hilt†, Xiaoming Fu*

*University of Goettingen, Germany
{asridha,dkoll,fu}@cs.uni-goettingen.de

†Nokia Bell Labs
{jesus.omana_iglesias,jordi.arjona_aroca,volker.hilt}@nokia-bell-labs.com

***Abstract*—Virtualization helps to deploy the functionality of expensive and rigid hardware appliances on scalable virtual resources running on commodity servers. However, optimal resource provisioning for non-trivial services is still an open problem. While there have been efforts to answer the questions of *when* to provision additional resources in a running service, and *how many* resources are needed, the question of *what* should be provisioned has not been investigated, in particular, for complex applications or services, which consist of a set of connected components, where each component in turn potentially consists of multiple component instances (e.g., VMs or containers). Each instance of a component can be run in different flavors (i.e., number of cores or amount of memory), while the service constructed by the combination of these component configurations must satisfy the customer Service Level Objective (SLO). In this work, we offer to service providers an answer to the *what* to deploy question by introducing `RConf`, a system that automatically chooses the optimal combination of component instances for non-trivial network services. In particular, we propose an analytical model based on *robust queuing theory* that is able to accurately model arbitrary components, and develop an algorithm that finds the combination of their instances, such that the overall utilization of the running instances is maximized while meeting SLO requirements.**

***Index Terms*—Robust queue; Resource configuration; Performance of complex services; Optimization**

I. Introduction

One major issue in automated resource provisioning is that of *optimally* configuring resources to applications or services[1] that are requested by customers in an ad-hoc fashion. In particular, when considering non-trivial services (e.g., multi-tier services or service chains) composed of several distributed *components* (e.g., load balancers, webservers, databases, etc.). Without loss of generality, we can define a complex service as a chain of virtual components that together process a certain fraction of the traffic passing through the provider's network, as for example shown in Figure 1. In the context of complex services, adequately provisioning resources while enforcing the Service Level Objectives (SLOs) expected by the customer becomes an arduous task.

There have been proposals, from both industry and academia, that address the problem of *when* to provision more resources (e.g., more component instances) for an application that is under high load. These solutions span from autoscaling strategies based on threshold policies, such as the ones proposed by the major cloud computing players like Amazon[2], Google[3], Microsoft[4], etc., to more elaborated solutions based on control theory [1], [2], [3] or empirical service modelling [4], [5]. In addition, some proposals leverage time series analysis to make workload predictions and decide *how many* resources to provision [6], [7]. These strategies are used jointly with other tools that facilitate the task of *how to* provision these resources, like Puppet, Openstack Heat, Ansible or Chef.

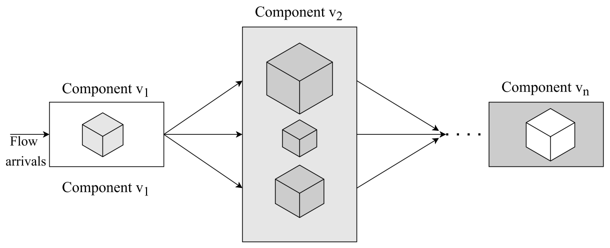


Figure 1: Model of a complex service consisting of many components, each of which can consist of several instances.

However, there is one overlooked problem which is usually not addressed, the question of *what* to provision. Generally, a requested service is deployed as a combination of different components, where each component is assigned a set of virtual instances and their resources. For instance, in Network Function Virtualization (NFV), a requested service chain is a combination of different Virtual Network Functions (VNFs), and each of these VNFs is assigned a set of virtual instances. Typically, each of these instances is tied to a single *flavor* (i.e., one VM size for a typical webserver) or set of resources that is then provisioned on demand. A component configuration can be thus defined, through its instances, as a number of cores and an amount of memory. However, using a single flavor results in several drawbacks. First, deciding which

[1]in the remainder of the paper we will refer to services and applications with the same "services" term.

[2]https://aws.amazon.com/ecs
[3]https://cloud.google.com/appengine/
[4]https://azure.microsoft.com/en-us/

flavor to use for a particular component is not a trivial task, as it requires expertise on that component. Second, if only one unique flavor is used, then it is likely that the flavor may not be suited for every amount of incoming load or type of traffic. Finally, evaluating a configuration without considering the other components in the service, and how such a configuration affects their performance might result in inefficient utilization or unexpected performance degradations. In multi-tier services, aspects such as the workload variation, or the composition of the service must be taken into account in order to provision resources appropriately. A one-fits-all solution results in deploying unneeded resources generating additional costs.

In this work we introduce `RConf`, a technique to answer the *what* to deploy problem. We denote this problem as the Virtual Configuration Problem (**VCP**), which consists of deciding, for a set of available component configurations and a workload input, the flavors and number of instances to be deployed for each component of a service, minimizing the resource wastage and ensuring the agreed SLOs requirements.

`RConf` divides the problem in two different stages. An initial profiling stage, where the different flavors available are modeled using robust queuing theory; and an online stage, where it proposes a configuration depending on the amount of workload arriving at the service. To propose this configuration, `RConf` solves a relaxed version of the integer linear programming (ILP) formulation of **VCP** and obtains an approximated solution. This solution is obtained through an algorithm that leverages the results of the initial profiling phase to subsequently find the optimal combination of different instance flavors for each service component, such that the end-to-end user requirements are satisfied.

To the best of our knowledge, this work describes the first analytical approach to find the optimal configuration for multi-tier services by approaching the problem from a robust queuing theory perspective. Concretely, our contributions are:

- A generic model for **VCP**, based on knapsack problems and its formulation as an ILP model. This model can be adapted to different performance metrics and profiling techniques.
- An algorithm able to find an approximated solution for a particular (complex) service, such that the overall utilization of all components is maximized and the SLOs enforced. This algorithm is based on the optimal solution for the relaxed version of the ILP formulation of **VCP**.
- A profiling implementation that gives accurate service rate and utilization estimations using an analytical model based on robust queuing. This model evaluates and estimates the impact on the service latency of different component configurations. Robust queuing uses uncertainty sets instead of randomization (as standard queuing theory does), in that way, it is possible to capture both: network traffic following arbitrary distributions and heterogeneous processing power of different components.

The remainder of this paper is structured as follows. In Section III, we formally describe the problem of finding the optimal configuration for multi-tier services consisting of different components, and propose an algorithm towards solving this problem in Section IV. Section V then specifies the system model that provides the input for our algorithm. We present the feasibility and practicality of `RConf` in our evaluation in Section VI and conclude our paper in Section VII.

II. Related Work

There have been several approaches that address the challenges of provisioning resources in the cloud (e.g. [2], [3], [5]). However, to the best of our knowledge, no previous work studied *what* resources should be provisioned in the context of complex services. The most similar work was recently published by Delimitrou *et al.* [8]. They proposed HCloud, a hybrid provisioning system that uses both reserved and on-demand resources (i.e., fits the business models of Amazon EC2). HCloud determines the best possible configuration by considering load-fluctuations, performance, and cost of resources. In contrast, this work focuses on describing a theoretical model based on robust queuing theory [9] that accurately describes the behaviour of a given component in a service regardless of the input traffic distribution. We are interested on understanding the exact behaviour of each component in the service, not in inferring it through micro-benchmarking as done by HCloud. Moreover, recently there have been some efforts on reducing the number of reconfigurations in cloud environments. For instance, Jiao, *et al.* [10] studied the problem of resource allocation and reconfiguration in the multi-tier resource pool from an online optimization perspective. However, they do not address the issue of what configuration should be used.

Many queuing theory based approaches have been proposed in the past. For instance, Urgaonkar *et al.* [11] models servers at each tier as a G/G/1 queue for representing arbitrary arrival and service time distributions. They presented both reactive and predictive provisioning algorithms, and assumed uniform processing power. However, if the servers are allocated with different cores or processors, their model fails to capture this aspect due to intractability of the G/G/m queue. Furthermore, Gandhi *et.al* [12] employed Kalman filters and queueing networks to proactively scale a given application. However, the biggest drawback is the assumption of Markovian arrivals and departures in a multi-tier architecture, which in reality is not always the case [11]. Tsoumakos *et.al* [13] propose the TIRAMOLA framework for performing automatic resizing of NoSQL clusters based on a Markov Decision Process (MDP). The resizing operation is performed based on the user policies and current system state. Naskos *et.al* [14] extend the TIRAMOLA model to resizing clusters of a single generic application hosted on virtual machines. However, both approaches cannot be applied if a service comprises multiple *different* components. In this work, we propose `RConf`, a solution that determines the optimal configuration for a service consisting of multiple different components and both markovian and non-markovian arrivals/departures.

III. Problem Statement

The composition of a complex service can be achieved in multiple ways. For instance, every instance from every component in the service can be instantiated with the same flavor. However, instantiating different types of components with a one-fits-all flavor may result in severe over-provisioning and thus resource wastage, simply because all components do not consume resources in the same way. Instead, in this work we propose to leverage the usual variety of instance flavors when creating a configuration for each component of the service. That is, the same (virtual) functionality can be provided with varying resource capacities in terms of CPU cores and memory. For instance, a load balancer might run in a "small" instance (e.g., 1 core, 2GB memory and 1Gbps bandwidth), instead of a "big" instance (e.g., 4 cores, 4GB memory) when the traffic load is low.

Let us consider a hypothetical service configuration for which a customer has expressed a SLO latency requirement of $u(l) = 300ms$. In this example, our service consists of four components, a firewall (FW), a load balancer (LB), deep packet inspection (DPI) and a webserver (WS). Table I shows the effect of using different instance flavors (e.g., the smallest instance of a LB will take 80ms to process the traffic, while medium instances combined are able to do so in 40ms), as well as their total utilization, and the end-to-end service latency. Each row of Table I represents a feasible service configuration, that is, it satisfies the customer SLO. The first five columns of the table represent the configuration name and the latency for the FW, LB, DPI and WS respectively, while the sixth and seventh column represent the service utilization and service latency. While all instance combinations satisfy $u(l)$, they differ in their total utilization. Configuration A may represent the traditional way of one-fits-all (over-provisioning) approach, which meets the SLO but results in low resource utilization. Moreover, even though both E and G exhibit the lowest latency, F represents the best configuration, as it results in a greater cumulative utilization of the components than in any other service configurations.

The challenge is then to find the combinations of configuration for each component in the service such that the complete service satisfies both service provider and customer constraints. While one particular problem may yield a large number of feasible service configurations, our goal is further to find the one single configuration of a service that maximizes the resource utilization of all service components. We define the problem of finding the optimal configuration for a complex service as the *Virtual Configuration Problem* (**VCP**)

A. Definitions

We start by providing some formal definitions that will be used throughout the paper. We denote a service composed of n virtual components as $\mathcal{S} = \{v_1, v_2, \ldots, v_n\}$. Flows passing through $\mathcal{S}$ are serviced by the components in order, i.e, flows enter $\mathcal{S}$ through v_1 and leave through v_n. To capture the order in which flows are serviced we assume $\mathcal{S}$ to be a partially ordered set (poset).

Config	FW	LB	DPI	WS	Utilization	Latency
A	40	40	40	40	0.20	160
B	60	60	100	60	0.1	280
C	80	60	120	40	0.4	300
D	40	80	100	40	0.6	260
E	60	60	60	40	0.7	220
F	60	60	60	60	0.8	240
G	40	40	100	40	0.5	220

Table I: A summary of feasible service configurations for an SLO requirement $u(l) = 300ms$. The utilization is given as $\sum_{i=1}^{n} \upsilon(\pi_i)$. All values are given in milliseconds.

A component can consist of one or more instances. We call this set of instances as *component configuration*. Then, we define the set of feasible configurations c of the i^{th} component as the array $c_i = \{c_{i1}, \ldots, c_{i|c_i|}\}$. Similarly, each of these configurations can be defined as $c_{ij} = \{x_{j1}, \ldots x_{j|\theta|}\}$, where $x_{j\theta}$ represents the number of instances of a given flavor θ, and θ is the set of available flavors $\theta = \{\theta_1, \ldots, \theta_{|\theta|}\}$. Each flavor θ has an associated tuple of m dimensions describing its allocated resources (e.g. CPU cores, memory, ...). We denote the resources for the k^{th} flavor as $\theta_k = \langle \theta_k^1, \ldots, \theta_k^m \rangle$. The set of configurations c_i is sorted in non-decreasing order of resource requirements. Finally, we denote the set of all possible configurations for the components in a service $\mathcal{S}$ as $\mathcal{C} = \{c_i | \forall i \leq n\}$. Note that there is no restriction about the choice of instance flavors within the configuration. For instance, one configuration for a load balancer might contain multiple instances of same type, while a different one might contain instances of different flavors.

Regarding resources, we denote the aggregated required resources of type l within configuration c_{ij} as $r_{ij}^l = \sum_{k=1}^{|\theta|} x_{jk}\theta_k^l$. The required resources are upper bounded by u^l, that represents the maximum resources of type l available for service $\mathcal{S}$. Finally, with respect to the service, we define the utilization of c_{ij} as υ_{ij} as the ratio between the workload being served and the aggregated capacity of a component. In addition to utilization, we define the cost in terms of SLO of a configuration as g_{ij}. Table II summarizes the notation used in this section and throughout the paper.

B. Problem Formulation and Reduction

A valid solution for **VCP** is the service configuration that maximizes the usage of resources while enforcing an SLO. Note that before trying to find such a configurations we need to define the set of possible configurations for a component and for the entire service. Assume that we have such a method that allow us to find the set feasible configurations c_i for a component, estimate their utilization υ_{ij}, required usage r_{ij}^l and cost in performance g_{ij} given an input workload. Having these estimations we can formulate **VCP** as the following integer linear program (ILP):

Symbol	Description
$\mathcal{S}$	Service
v_i	i^{th} virtual component
c_i	Set of configurations for v_i
c_{ij}	Configuration j for v_i
θ	Set of available flavors
λ_i	Traffic arrival rate at v_i
μ_{ik}	Service rate for flavor type θ_k, and v_i
ρ_{ik}	Traffic intensity for flavor type θ_k, and v_i
τ_{ik}	System time for flavor type θ_k, component i
$\mathcal{U}_i^a, \mathcal{U}_{ik}^s$	Uncertainty set for arrival and departure processes
$\Gamma_i^a, \Gamma_{ik}^s$	Arrival and departure process variability.
$\sigma_i^a, \sigma_{ik}^s$	Standard deviation of arrival and departure processes
r_{ij}^l	Aggregated required resources of type l for c_{ij}
υ_{ij}	Utilization of c_{ij}
x_{jk}	Number of instances of a given flavor θ_k for c_{ij}
u^l	Upper bound of resource type l
g_{ij}	Performace cost for c_{ij}
w_{ij}	Binary indicator of configuration selection

Table II: A list of notations used in this paper.

$$
\begin{aligned}
&(ILP1)\ \text{maximize} \sum_{i=1}^{n}\sum_{j=1}^{|c_i|} \upsilon_{ij} w_{ij} \\
&\text{subject to} \\
&(C1) \quad \sum_{i=1}^{n}\sum_{j=1}^{|c_i|} r_{ij}^k w_{ij} \le u^l(\mathcal{S}), \forall k = 1, 2, \ldots, m \\
&(C2) \quad \sum_{i=1}^{n}\sum_{j=1}^{|c_i|} g_{ij} w_{ij} \le SLO(\mathcal{S}), \forall k = 1, 2, \ldots, m \\
&(C3) \quad \sum_{j=1}^{|c_i|} w_{ij} = 1; \forall i = 1, 2, \ldots, n \\
&(C4) \quad w_{ij} = \{0, 1\}
\end{aligned}
\tag{1}
$$

where w_{ij} is a binary indicator that represents whether a given configuration is selected or not (C4). If $w_{ij} = 1$ then configuration c_{ij} is included, otherwise it is excluded. The first constraint (C1) implies that the resources required by a configuration cannot exceed any of the budgeted resources. The second constraint (C2) controls that the performance costs are not exceeded (e.g., latency). Finally, the last constraint (C3) limits the number of selectable configurations for a certain component to 1.

The above formulation can be reduced to a multiple-choice multidimentional knapsack problem (MCMKP) [15]. In MCMKP we have a set of items divided in n classes. Only one item per class can be selected. In addition, each item has multiple dimensions. The knapsack has as many constraints as dimensions. Finally, each item has a profit and the goal is to maximize the total profit while respecting the knapsack constraints. In our problem, the knapsack constraints are the m budgeted resources u, each item is a configuration, divided into categories which are the different components. Only one configuration for each component is to be chosen. Similarly, each configuration consumes a set of resources (the weight), while the utilization can be seen as the profit.

Algorithm 1 VCP approximation algorithm.

```
Require: $\lambda$
 1: $\{\mathcal{C}, \sum_{l=1}^{m} r_{ij}^l, \upsilon_{ij}, g_{ij}\} \leftarrow$ get_init_params($\lambda$)
 2: $w_{ij} \leftarrow$ solution of relaxed VCP
 3: if No valid $w_{ij}$ then
 4:     exit()
 5: $P, R \leftarrow \emptyset$
 6: for $i \le n$ do
 7:     $w'ij' \leftarrow 1$ for $j' : j' : w_{ij'} \ge wij\ \forall\ j \ne j'$
 8:     $w'ij \leftarrow 0$
 9: while C1 or C2 are not met do
10:     if C2 not met then
11:         $i' : w'_{i'j} \cdot gi'j > w'_{ij} \cdot g_{ij},\ \forall\ i \ne i', j$
12:         $j'' : w'_{i'j''} == 1$
13:         $j' : w_{i'j'} > w_{ij}$, for $j'' < j' \le |c'_i|$
14:         if $i' \notin R$ or $(i' \in P\ and\ P[i'] < j')$ then
15:             $w'_{i'j'} \leftarrow 1$, $w'_{i \ne i', j \ne j'} \leftarrow 0$
16:             $P[i'] = j'$
17:         else if $w'_{i'j} \cdot g_{i'j}\ \not{m}in(w'_{ij}) \cdot g_{ij})$ then
18:             try next $i'$
19:         else
20:             No valid configuration found; Exit
21:     if C1 not met then
22:         $i' : w'_{i'j} \cdot (r_{i'j}^k / \upsilon^k) \ge w'_{ij} \cdot (r_{ij}^k / \upsilon^k),\ \forall\ i \ne i', j, k$
23:         $j'' : w'_{i'j''} == 1$
24:         $j' : w_{i'j'} > w_{ij}$, for $0 \le j' < j''$
25:         if $i' \notin P$ or $(i' \in RandR[i'] > j')$ then
26:             $w'_{i'j'} \leftarrow 1$, $w'_{i \ne i', j \ne j'} \leftarrow 0$
27:             $R[i'] = j'$
28:         else if $w'_{i'j} \cdot (r_{i'j}^k / \upsilon^k)\ /min(w'ij) \cdot gij)$ then
29:             try next $i'$
30:         else
31:             No valid configuration found; exit()
    Return $w'_{ij}$
```

IV. Algorithm

In this section we present an approximation algorithm for **VCP** based on the relaxation of the formulation presented in Section III, and a deterministic rounding process. A pseudocode version can be found in Algorithm 1. The algorithm takes an expected workload λ as input and computes the set of configurations $\mathcal{C}$ that can serve it. From $\mathcal{C}$ we can compute the set of resources of each type l required by each c_{ij}, r_{ij}^l, its utilization υ_{ij}, and its cost in terms of performance g_{ij}. The set $\mathcal{C}$ can be trivially computed with greedy approaches. For instance, for each v_i, take the minimum number of instances of the flavor with the less resources as first configuration. From there, we start replacing a number of these instances by equivalent superior instances iteratively. A configuration is finally given by a set of instances, which is sufficient to serve λ but that would not be if we remove any of the selected instances. Knowing $\mathcal{C}$, their resource requirements are computed as $r_{ij}^l = \sum_{k=1}^{|\theta|} x_{jk}\theta_k^l$. Details about how to estimate υ_{ij} and g_{ij} are provided in Section V.

We start by computing the set of initial parameters (line 1). Then, we relax condition C4, converting our ILP into a linear problem. Using the formulation from Equation 1 with solvers such as CPLEX, we can obtain a fractional optimal configuration, obtaining the set of coefficients w_{ij} (line 2). If there is no feasible configuration, the algorithm terminates (line 4). If there is a solution, we initialize variables P and R, that we will use to store our intermediate rounding steps.

`RConf` first evaluates the trivial rounding making the highest coefficient for each component equal to 1 and zeroing the rest (lines 6-8). This configuration is returned if it meets C1 and C2. If not, `RConf` will tackle them alternatively in an iterative process. First, if performance constraints are not met (C2), we select the component contributing the most to the performance costs and try to select a configuration with more resources in order to reduce its cost. As c_i is sorted in non-decreasing order of resources, we will choose the highest coefficient between the current and the configuraton with most resources for that component, make it 1 and zero the rest (lines 11-13). Before updating the coefficients w'_{ij}, we check if the selected component has not been previously pushed in the other direction, i.e., we have not reduced its resources. If we have not, it will not be present in set R. Similarly, we also check $P[i']$ to see whether we have already increased the resources of this component configuration, but not as much as now. If the condition is met, we update the coefficients $w'ij$ and add the duple (i', j') to P. If not met, we try with the next component in terms of performance cost. If no component can be updated, `RConf` fails to find a valid configuration in terms of performance (lines 14-20).

Then, we evaluate constraint C1. If it is not met, we follow a similar procedure, but this time taking the component consuming the most resources (line 22). We will try to switch to a configuration with less resources by choosing the largest coefficient between w_{i1} and our current $w'ij$, making it 1 and zeroing the others (lines 23-24). We then check whether this component has been updated before or not by checking sets P and R. If conditions are met, we update $w'ij$ and add the duple (i', j') to R (lines 25-27). Otherwise, we try with the next component in terms of resource requirements (line 29) or fail to return a configuration (line 30). When both constraints C1 and C2 are met, `RConf` returns the current w'_{ij} that indicates the required configuration for each component (line 31).

V. Modeling a Complex Service

This section describes how to model complex services with robust queuing theory. As described in section III, we model $\mathcal{S}$ as a partially ordered set (poset).

A. Robust queue Motivation

The basic idea is to model each combination of VM flavor θ_k and component v_i as a queue q_{ik} using one or multiple servers (i.e. CPU cores) with either limited or unlimited buffer capacity. Arrival processes enter the system (i.e. VM), and either are processed or wait in a queue (if the system is busy). Once a process is served and leaves the system, it is marked as a departure process.

Queues are represented using the standard Kendall's notation [16]: $A/B/m/$, where A is the probability distribution of the arrival process, B is the probability distribution of the departure process, and m is the number of servers assigned to the system. Moreover, *system time* is defined as the sum of *waiting time* (i.e. time a process waits to be served) and *service time* (i.e. time spent on serving an arrival process).

Applying traditional queuing theory to model complex services is either not possible or highly inaccurate. This is mostly due to the assumption that process's arrivals and departures are Markovian or deterministically distributed, which is not the case for many cloud or networks services [11], [17], [18], [19], [20]. For instance, Ethernet traffic is heavy-tailed and self similar [18], thereby does not follow the aforementioned assumption.

Complex services could be modeled as networks of queues, where each component of the service is modeled as a G/G/m queue [16]. However this translates to making the system analysis computationally intractable [9], [21]. To overcome these limitations, Bandi *et.al* [9] proposed robust optimization for analyzing a single, or network of, G/G/m queues, and subsequently derives a closed-form expression for the *system time*.

Robust queue analysis does not consider arrival and departure processes to be arbitrarily distributed, as traditionally done in G/G/m queue models. Instead, robust optimization is performed on uncertainty sets (constraints are allowed to vary within this set) without affecting the optimality of the solution [22], and then determines the expected system time based on those uncertainty sets. In this work we leverage the work of Bandi *et al.* and apply it to model complex cloud services. Furthermore, `RConf` uses this model as a decision making tool when evaluating different component configurations.

B. Service's Component Modeling

In this work we model each instance flavor type of a given service's component as a robust queue q_{ik}. We assume that each flavor θ_k is a duple of assigned CPU cores (θ_k^1) and memory (θ_k^2). Our goal is to obtain an estimation of the latency of a component configuration c_{ij}. This latency will be the performance cost g_{ij} defined in Section III.

The system time for q_{ik} is based on the network traffic flow instead of on the traffic request. Limiting the solution to a specific request type, especially in application layer, would affect the applicability of the solution [12]. For instance, in a typical data center, there are hundreds or thousands of heterogeneous services. Therefore it would be beneficial not to model each request of each service as a unique process, but instead to monitor the transport layer, where one can capture traffic either at *packet* or *flow* level. Packets give more information compared to flows, but require significant processing overhead, especially in environments where the number of packets grows exponentially (e.g., data centers). Robust queue models require only the arrival and departure times of the requests, which can be obtained at the flow level.

Flows enter a given component with an arrival rate λ_i and are processed at rate μ_{ik} (service rate). If there are available servers (CPU cores) in the system, the flows are processed, otherwise flows have to wait. Using standard definitions from queueing theory [16] we define ρ_{ik} as the utilization of an instance with flavor k for the i^{th} component:

$$\rho_{ik} = \frac{\lambda_i}{\theta_k^1 \mu_{ik}} \tag{2}$$

where λ_i is the arrival rate of component i, μ_{ik} the service rate and θ_k^1 the total number of CPU cores of an instance θ_k.

Moreover, as indicated in the previous subsection, robust queue models require the construction of uncertainty sets for both arrival and departure processes. Before presenting the closed form expression for the system time, it is necessary to construct these sets. The uncertainty sets are based on a stable distribution. According to the central limit theorem, any non-heavy tailed distribution converges to a normal distribution as $n \to \infty$ [9]. The normal distribution is a special case of the stable distribution [23], [9]. Nolan [23] proves that by setting the tail coefficient $\alpha = 2$, the stable distribution behaves like a normal distribution. In this way, a non-heavy tailed distribution can be converted to a stable distribution. Heavy-tailed distributions are those whose tails are not exponentially bounded like in pareto or log normal distributions (e.g., Ethernet traffic is heavy tailed and self similar [18]). In our model, these kind of heavy tailed distributions are converted to stable distributions by setting the tail co-efficient α to appropriate values using Hill's estimator [24] with order statistics approximately $< 0.13\%$ of sample size [25]. Hence, our model can handle both heavy-tailed and non-heavy tailed traffic.

1) Uncertainty set for arrivals: In this work, robust queue models are created per flavor type θ_k and component v_i. However, we assume that the arrival traffic requests are same for all the instances. Therefore, at least for arrivals, one uncertainty set is sufficient. This is not the case for departures, where the service rate of an instance depends on the instance attributes (i.e. number of processors, memory allocated, and SLO requirements). The uncertainty set for inter arrivals of component i is denoted by:

$$\mathcal{U}_i^a = \left\{ (T_{i,1}^a, T_{i,2}^a, \dots T_{i,t}^a) \quad \left| \frac{\sum_{j=t'+1}^{t} T_{i,j}^a - \frac{(t-t')}{\lambda_i}}{(t-t')^{\frac{1}{\alpha_i}}} \geq \Gamma_i^a \right. \quad \forall 0 \leq t' \leq t-1 \right\} \tag{3}$$

where $T_{i,1}^a, T_{i,2}^a, \dots T_{i,t}^a$ denotes the inter-arrival times of the flows at component i with an expected arrival rate $\frac{1}{\lambda_i}$. Moreover, $t - t'$ is the number of arrivals considered while constructing the uncertainty set. If t' is equal to 0, it means that all arrivals are considered for uncertainty set construction. In our case, we set $t' = 0$ in order to validate the robust queue model with measured data. In the above equation Γ_i^a captures the variability in the inter-arrival times. Since the standard deviation measures the spread of data around the mean, we set Γ_i^a as the standard deviation of inter-arrival times i.e, $\Gamma_i^a = \sigma_i^a$.

2) Uncertainty set for departures: The uncertainty set for θ_{ik} is denoted by:

$$\mathcal{U}_{ik}^s = \left\{ (T_{ik,1}^s, T_{ik,2}^s, \dots T_{ik,t}^s) \quad \left| \frac{\sum_{l=t'}^{t} T_{ik,l}^s - \frac{(t-t'+1)}{\mu_{ij}}}{(t-t'+1)^{\frac{1}{\alpha_i}}} \geq \Gamma_{ik}^s \right. \quad \forall 0 \leq t' \leq t \right\} \tag{4}$$

Where $T_{ik,1}^s, T_{ik,2}^s, \dots T_{ik,t}^s$ denotes the inter-arrival times of the flows at the i-th component with an expected service rate $\frac{1}{\mu_{ik}}$. Furthermore, Γ_{ik}^s captures the variability of departure process. Since the service rate depends on instance type θ_k, component utilization, arrival process variability and standard deviation of the departure process, we thereby define as $\Gamma_{ik}^s = f(\theta_k, \rho_{ik}, \Gamma_i^a, \sigma_i^s, \alpha_i)$, and given that each system is different and the function f is not known, we perform general linear regression to determine Γ_i^s.

$$\Gamma_{ik}^s = \beta_1 \theta_{ik} + \beta_2 \rho_{ik} + \beta_3 \Gamma_i^a + \beta_4 \sigma_i^s + \beta_5 \alpha_i + \epsilon \tag{5}$$

where ϵ is the difference between measured and estimated Γ_{ik}^s. We use the closed form expression of linear regression to compute the values $\beta_1, \dots, \beta_5$.

Then the estimated system time τ_{ik} is given by:

$$\tau_{ik} = \frac{(\alpha_i - 1)}{\alpha_i^{\frac{\alpha_i}{(\alpha_i-1)}}} \frac{\lambda_i^{\frac{1}{(\alpha_i-1)}} \Gamma_i^a + \left(\frac{\Gamma_{ik}^s}{(\theta_{ij}^1)^{\frac{1}{\alpha_i}}} \right)^{\frac{\alpha_i}{(\alpha_i-1)}}}{(1 - \rho_{ik})} + \frac{\theta_k^1}{\lambda_i} \tag{6}$$

Initially, we calculate output variability Γ_{ik}^s by substituting the observed average system time in Equation (6). We also consider the standard deviation of arrivals as input variability Γ_i^a. Once we have set Γ_{ik}^s, we perform a linear regression using Equation (5) to predict the next Γ_{ik}^s values and use these values again in Equation (6) to estimate the system time . Even though Bandi *et.al* [9] derive the system time expression for the worst case, we train the model using an average flow processing times.

C. Evaluation

We now describe and formalize the metrics required for solving the **VCP** problem as introduced in Section III.

Utilization: the utilization of jth configuration of an ith component υ_{ij} is defined as:

$$0 \leq \upsilon_{ij} = \frac{\sum_{k=1}^{|\theta|} \rho_{ik} x_{jk}}{\sum_{k=1}^{|\theta|} x_{jk}} \leq 1, \tag{7}$$

where x_{jk} represents the number of instances of θ_k for configuration c_{ij}. In order to compute ρ_{ik} we need to compute the service rate μ_{ik} (Equation (2)) as follows.

In a queueing model, system parameters are derived based on arrival and departure time epoch, i.e., $T_{ik,t}^s, T_{i,t}^a$. Hence, only the system time is observed. The system time is the sum of waiting time and processing time. The service rate measures how fast a server serving requests. Therefore, we compute the processing time from system time based on *Lindley's recursion* [26] which represents the waiting time of a current request as a recursive relation between system and processing times of the arrival and departure processes. On the other hand, it is non-trivial in the case of m multiserver queues since m servers are servicing in parallel and a flow can be serviced by any one among them. Krivulin [27] extends the idea of *Lindley's recursion* to a G/G/m and derives a recursive relationship between system, waiting and processing time. The basic idea of Kirvulin is to sort the departure time epochs and then compute the difference between arrival and departure time epochs. Also, in a m server queue, the first m departure processes have zero waiting time. Hence, once we know the processing time $T^s_{ik,t}$ we can compute $\mu_{ik} = 1/\mathbb{E}(X_{ik,t})$.

Resource usage: In our case, the resource requirements function is trivial for the number of cores and memory, $r^l_{ij} = \sum_{k=1}^{|c_i|} \theta^l_k x_{jk},$ for l=1,2.

Performance cost: Our performance metric for a configuration is latency. We compute it using Equation (6). Hence, $g_{ij} = \max_{\forall k \in \theta_k} \tau_{ik}$

Intuitively, in a scenario with multiple instances, the traffic is split between component instances since different instances work in parallel. Therefore, we define the latency of a component as that of the instance with the highest system time.

Capacity: We define the capacity as the quantity of available resources or type l, i.e., cores or memory. We denote it as u^l

SLO: As we use latency as our performance metric, our SLO is the maximum acceptable latency for the service.

VI. Experimental Evaluation

In this section, we evaluate RConf in a real-world deployment. In particular, we present two key results. First, we profile a variety of VM instance flavors with regards to their capabilities for each of the service's components and show that our robust queue model is able to predict these capabilities accurately, while traditional modeling approaches fail in doing so. Second, based on this profiling, we show that RConf, constrained by a customer SLO, does indeed find optimal configurations that increase resource utilization when compared to one-fits-all approaches.

A. Profiling

1) Methodology: Our goal in profiling instance flavors is to determine how well our robust queue model can predict the capabilities of each component run on different flavors. For this purpose, we first measure these capabilities (i.e., how many requests a component run on a specific flavor can handle) in a real-world deployment, and then compare our findings with the predictions obtained by employing RConf and its robust queue model, as well as with the predictions obtained by employing a traditional G/G/m queue. To measure the real-world capabilities, we evaluate user requests to a three-tier web-service as depicted in Figure 2. Web requests generated by httperf [28] arrive at a load balancer (LB), for which we use HAProxy[5]. The LB forwards user requests to a web server (WS) running Elgg[6], a widely used social networking engine. Finally, the WS queries a MySQL database (DB) to fetch requested content.

The key concept of RConf is to allow each of these components (LB, WS, DB) to be run in different instance flavors. To obtain representative flavors, we use a subset of the set of flavors offered by Amazon EC2 [7] as listed in Table III. This subset offers a variety of combinations of processors and memory, and each flavor has a different preferred use case. In general, flavors of the M3 category are intended to be used in memory-hungry components such as the DB in our case, and C3 instances are well-suited for compute-intensive components such as the WS in our three-tiered service. We also use some general purpose instances that are not designed for a particular type of component.

We then deploy our service as introduced above on Amazon EC2 such that each component is running on a different instance and profile each component running on each flavor. To be able to profile each component of the service individually, we isolate that component, in the sense that we intentionally over-provision all other components in the service with the goal of finding the limits of the component under investigation. For example, to profile a DB on a T2.small instance, we intentionally deploy both LB and WS on C3.2xlarge instances, so that the DB is the bottleneck of the complete chain.

Thereafter, we perform profiling by increasing the system load at a constant rate and determine the system time (i.e., the total time spent in the component between arrival and departure) of each flow for the instance at that load. In order to determine the load of a component, it is further important to calculate the service rate of that component, which requires deriving the processing time from the system time. This is

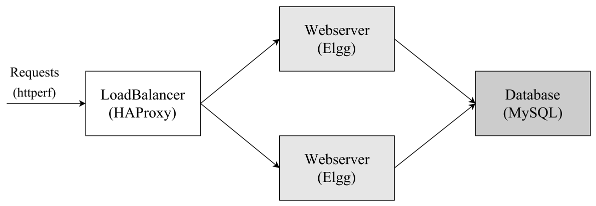


Figure 2: Experiment Setup

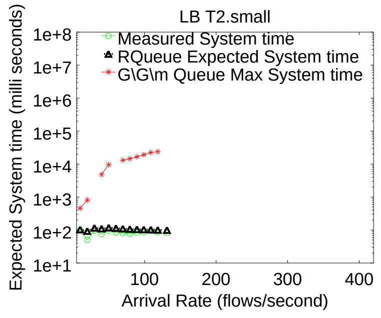
Instance	vCPU	Mem (GiB)	Usecase
T2.small	1	2	General purpose
T2.medium	2	4	General purpose
T2.large	2	8	General purpose
M3.medium	1	3.75	Enterprise applications
M3.large	2	7.5	Backend servers
M3.xlarge	4	15	Cluster computing
C3.large	2	3.75	Webservers
C3.xlarge	4	7.5	Batch processing
C3.2xlarge	8	15	Distributed analytics

Table III: A summary of the instance flavors used in our experiments.

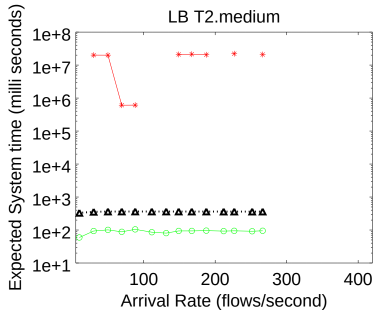
[5]http://haproxy.org

[6]http://elgg.org/

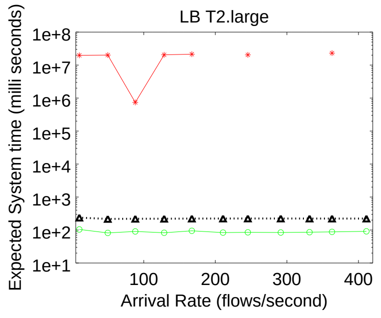
[7]https://aws.amazon.com/ecs



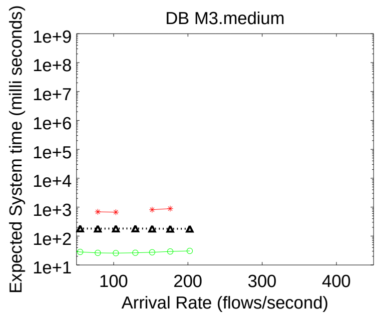
(a) LB @ T2.small



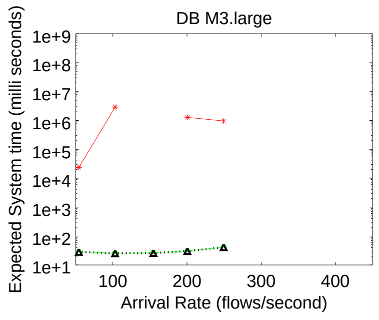
(b) LB @ T2.medium



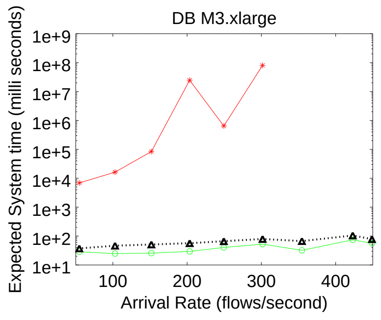
(c) LB @ T2.large



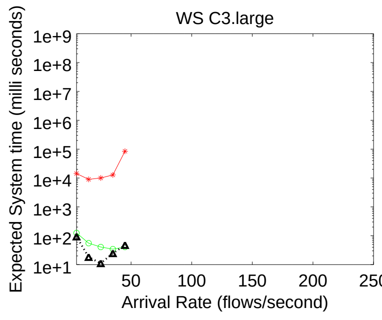
(d) DB @ M3.medium



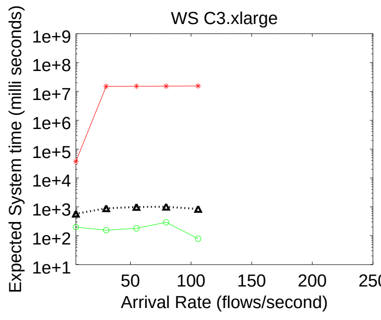
(e) DB @ M3.large



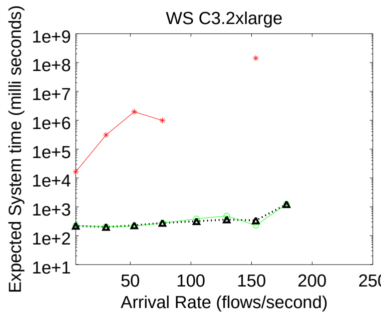
(f) DB @ M3.xlarge



(g) WS @ C3.large



(h) WS @ C3.xlarge



(i) WS @ C3.2xlarge

Figure 3: Measured (profiled) system time vs `RConf` prediction vs theoretical G/G/m system time (maximum).

not trivial in the case of multi-server queues since multiple servers are servicing in parallel and the flow can be serviced by anyone among them. Krivulin [27] extends the idea of *Lindley's recursion* to a G/G/m queue and derives a recursive relationship between system, waiting and processing time. We thus implement Krivulin's equations for computing the processing time and service rate of the flows.

2) Results: Figure 3 shows the system time of i) the measured system time in profiling, ii) the predicted system time by `RConf`, and iii) the maximum system time of a G/G/m queue for all types of components we consider in our service and their most reasonable instance flavors. Our key observations are as follows:

Components scale well on different flavors. Each row in Figure 3 (e.g., (a) to (c)) shows one scaled type of instance for a certain component. Each step in one row roughly represents a doubling of resources, and the right-most plotted point for each measure or prediction marks the breaking point of an instance, i.e., the point at which the instance cannot handle more requests. Here, we clearly see that more capable instance flavors are able to handle higher arrival rates, which is not surprising. One interesting find is however that some instances scale very well with an increasing number of requests. For instance, we can observe that the latency for running a DB on the memory optimized instance flavors (see (d) to (f)) actually *decreases* with an increasing arrival rate, which may seem counter-intuitive at first. The reason for this is that the DB with increasing load also starts more worker threads internally and is thus able to handle the load appropriately. We can observe this effect up to a certain point, at which the DB begins to struggle with the load and finally reaches a break point, at which it can not handle any more requests (e.g., at around 220 requests for a M3.medium instance).

`RConf` is accurate. More importantly, our model can accu-

Flavor	Largest	Smallest	Expert	`RConf`
Load Bal.				
C3.2XLarge	1	1	1	1
Webserver				
T2.small		5		5
C3.xlarge			2	
C3.2xlarge	2			
Database				
T2.small		2		
M3.medium				1
M3.Large			1	
C3.2xlarge	1			
SLO (ms)	300	300	300	300
Latency (ms)	299.28	303.93	293.28	296.3
CPU/Ram used	32 / 60	15 / 30	18 / 37.5	14 / 29
Valid (20/40)	No (Budget)	No (SLO)	Yes	Yes
Valid (15/30)	No (Budget)	No (SLO)	No (Budget)	Yes
ARU (%)	53.423	48.864	63.638	73.503

Table IV: Comparison of resource allocation and resulting utilization among different approaches under the specified budgets and latency SLO. Instance flavors not shown were not chosen by any approach.

rately predict the real-world system time of each component. We can see that `RConf` predicts the capabilities of most instances well. We see the most accurate predictions for a LB running on T2.small, a DB running on M3.large and a WS running on C3.2xlarge. In these cases `RConf` is at most a few milliseconds off target, and in some cases (e.g., DB @ M3.large, Figure 3(e)) yields a perfect prediction. For the remaining six depicted flavors we also receive encouraging results, which deviate in the scale of 10s of milliseconds for most cases. We have profiled all three components on all nine instance flavors and this result generalizes to the combinations not shown in Figure 3.

If `RConf` is wrong, it does the right thing. It is further worth noting that if `RConf` deviates from the actual measured values, the deviation goes in the right direction. `RConf` is a conservative approach that *over-predicts* the time required to process a certain load with a certain combination. While over-predictions are not desirable, their downside is that `RConf` might occasionally miss a combination that is able to do the job with less resources. In contrast, if it would systematically *under-predict* the latency, it would run in the much more severe danger of violating customer SLOs by selecting instance flavors too optimistically. The only case in which we see under-prediction is for a WS running on C3.large, and there only for a few intermediate request rates. This rare under-prediction is balanced by over-predicting in other cases.

`RConf` will adapt to dynamic load. Our model is also able to adapt to dynamic (e.g., increasing) load. As we now know the maximum service rate for each combination (indicated by the breaking point), `RConf` will know at which point it will need to scale up a specific component. For instance, consider a service that is dealing with an arrival rate of 100 requests per second (rps) and has thus deployed a single M3.medium instance for the DB. If the load increases to 200 rps, `RConf` will not scale up the DB, as it knows that the M3.medium instance can handle this load well (Figure 3(d)). In contrast, if the load further increases to 250 rps, `RConf` will add one or more additional instance(s). Note that these instances do not necessarily have to be M3.medium again, as now `RConf` only needs to provide resources for an additional 50 rps.

G/G/m queues are unsuitable for our purpose. Employing a G/G/m queue results in significantly over-predicting the system time, often by several orders of magnitude. The reason is that we can only use the maximum system time in a standard G/G/m queue, while we can not compute the average system time as done in `RConf` [16]. As a consequence, employing G/G/m queues for resource provisioning would result in a prohibitive resource over-provisioning. Concretely, deploying a DB component that matches a 30ms SLO can be done with one M3.large instance in `RConf`, while a G/G/m queue would deploy 1344 such instances (see Figure 3(e), note the log scale of the y-axis for the expected delay of a single instance predicted by a G/G/m queue).

Additionally, when using a G/G/m queue we can not find a solution at all times (indicated by breaks in the respective curve). The reason is that for a G/G/m queue, the maximum system time is computed as follows:

$$\tau \leq \frac{\lambda(\sigma_a^2 + \sigma_s^2/m + (1/m - 1/m^2)/\mu^2)}{2(1-\rho)} + \frac{1}{\mu} \qquad (8)$$

From the above equation we observe that τ is undefined when $\rho \geq 1$. That is, when the system is fully loaded, the system time is undefined for G/G/m.

B. Finding Optimal Configurations

In our second step, based on the results obtained from the profiling, we evaluate `RConf` with regards to its real-world feasibility. Here, we first let `RConf` find the optimal solution for the deployed service, and show that `RConf` improves utilization over one-size-fits-all solutions. In a second step, we show based on real experiments that the solution `RConf` found in the first step actually satisfies the customer SLO.

1) Methodology: In our experiment a customer wants to set up a three-tiered web service as introduced above. Here, the customer demands that each request towards that service is handled in less than 300ms, in accordance with typical SLOs seen in virtual infrastructures [29]. The customer also indicates that the service will experience a load of 200 rps, and that she is not willing to pay for more than 20 CPU cores and 40 GB memory. To show the adaptability of `RConf`, we also assume that this load is not static, but increases over time, and will thus evaluate the configurations chosen by `RConf` for varying request rates. Note that we fix the LB to the largest instance in the experiment for all approaches, as all traffic enters our experimental deployment through that node.

We compare the solution found by `RConf` to several one-size-fits-all solutions. In particular, we evaluate whether it is possible to find a solution that matches the customer SLO and budget in any of these approaches (including `RConf`), and how well each of the approaches utilizes the deployed resources. More concretely, we compare RConf against configurations that only choose the largest available instances (i.e., over-provisioning), configurations that only choose the smallest available instances, and an expert configuration. In the latter, we scale up a component by adding instances that best match that component's requirements. For instance, the DB component will be scaled by adding memory-dominated instances. Here, the expert determines a flavor that is matches component's requirements in general, but it is important to note that the expert can not know the resource requirements of particular services. Thus, this allocation is a better choice than the largest-only approach (which is not fit to any component), but is still likely to result in over-provisioning to some extent.

2) Results: Table IV shows the solutions found by each approach for 200 rps. We can observe that only RConf and the expert choice can find solutions in this case, and further that RConf outperforms all other approaches in terms of average resource utilization (ARU) of each component. In particular, we increase utilization by 38% over over-provisioning, 50% over choosing only small instances, and 16% over the expert choice, which uses computationally powerful C3.xlarge instances for the WS and M3.large as a memory-optimized instance for the DB. Note that this utilization gain is achieved even though RConf over-predicts the latency for some configurations as indicated above.

In detail, selecting large instances results in over-provisioning with only approximately half of the resources provisioned being used. Also, the solution found in over-provisioning is violating the customer budget and thus not valid. At the same time we are not able to match the SLO by selecting small instances only. The reason for this is that here we are limiting the DB component to T2.small instances (which are not a good match for the DB requirements) results in an increased latency for this component. As a result, no matter how many T2.small instances we deploy, we can not get below the SLO of 300ms. In contrast, the expert choice solution avoids this dilemma and selects an appropriate instance type for the DB component. We also see that with this allocation, resource utilization is increased over over-provisioning and that the solution is valid with respect to the customer SLO and budget.

Still, RConf outperforms the expert choice. In order to find the configuration that maximizes utilization, it finds a configuration that is a mixture of smallest only (web servers) and the expert choice (database). Here, the strength of RConf is that it is not bound to a single instance flavor, but can instead choose from all available flavors and thus yield significant utilization gain. This utilization gain is also reflected in saving resources: RConf also deploys 22% less resources. As a consequence, RConf can even meet stricter customer budgets. As shown in Table IV, in contrast to RConf, the expert choice approach can not find a valid solution if we decrease the budget to 15 cores and 30 GB ram.

Finally, we show that the configuration found by RConf for 200 rps is actually meeting the SLO set by the customer. For that, we deploy our three-tiered system under this configuration, and measure the end-to-end latency of the complete service as configured by RConf. To demonstrate the ability to dynamically scale-up a service on demand, we increase the load towards the service sequentially. In Table V, we indeed see that even for dynamic request rates, RConf is able to comply with the SLO at all times. Moreover, RConf scales up instances and runs with different combinations over time. In this case, RConf provides additional web servers and databases for each increase in the request rates. Here, once RConf receives the information about the new (increased) load, it finds the optimal configuration for the difference between the new load and the previous maximum service rate, and deploys instances accordingly. For instance, the five WS running on T2.small are (almost) completely utilized with 200 rps (each having a maximum service rate of 40 rps), and thus RConf finds an optimal solution for the remaining 50 requests when configuring the service for the new load of 250 rps. It then deploys the found solution *in addition* to already running instances to avoid complete reconfiguration of the service. For the opposite way (decreasing load), RConf can find the new configuration in a similar way. In that case, the set of possible configurations to choose from for provisioning the new (reduced) load would be the set of combinations of the currently deployed instances. That way, the required instances will be kept running, while RConf can shut down instances that are not used anymore. Note however that, for stateful applications like a DB or a firewall, this requires the transfer of application state to the remaining instances, which is out of the scope of this paper.

VII. Conclusion

In this paper, we have introduced RConf, the first answer to the *what* to deploy question in the context of resource provisioning for complex services in the cloud. While previous work have mainly dealt with the questions of *when* and *how* to provision resources, RConf is the first approach that departs from a one-size-fits-all solution and instead intelligently selects the resources needed to provision a service from many different possible combinations of VM instance flavors. To do so, RConf employs an analytical model based on robust queuing theory that is able to accurately predict the capabilities of a specific instance flavor when used for a given component in the service. RConf then uses the output of this model as a basis for selecting the optimal service configuration that maximizes resource utilization while meeting the customer SLO. Our real-world experiments show that RConf can increase utilization by 16-50% over one-size-fits-all solutions and deploys 22% less resources than the best of the studied approaches. Further, RConf manages to find valid configurations where the one-size-fits-all approaches either violate the customer SLO or the budget constraints, and thus gives the service provider additional opportunities to satisfy incoming customer requests.

Requests/second	200	250	350	450
Configuration DB	1x M3.medium	1x M3.medium + 1x T2.small	1x M3.medium + 2x T2.small	1x M3.medium + 3x T2.small
Configuration WS	5x T2.small	5x T2.small + 2x M3.medium	5x T2.small + 2x M3.medium + 1x C3.xlarge	5x T2.small + 2x M3.medium + 1x C4.xlarge + 4x T2.medium
Latency (ms)	217.1	213.0	207.6	215.3

Table V: `RConf` meets the pre-defined SLO requirements and scales up components as required.

Acknowledgment

The research leading to these results has received funding from the EU FP7 Marie Curie Actions by the EC Seventh Framework Programme (FP7/2007-2013) Grant Agreement No. 607584 (the Cleansky project). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the CleanSky project or the European Commission.

References

[1] H. C. Lim, S. Babu, and J. S. Chase, "Automated Control for Elastic Storage," in *Proceedings of the 7th International Conference on Autonomic Computing*, June 2010, pp. 1–10.

[2] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic Resource Provisioning for Cloud-based Software," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, June 2014, pp. 95–104.

[3] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann, "Automated Control for Elastic N-tier Workloads Based on Empirical Modeling," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, June 2011, pp. 131–140.

[4] R. Han, L. Guo, M. Ghanem, and Y. Guo, "Lightweight Resource Scaling for Cloud Applications," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2012, pp. 644–651.

[5] M. Hasan, E. Magana, A. Clemm, L. Tucker, and S. Gudreddi, "Integrated and Autonomic Cloud Resource Scaling," in *2012 IEEE Network Operations and Management Symposium*, April 2012, pp. 1327–1334.

[6] S. Di, D. Kondo, and W. Cirne, "Google hostload prediction based on Bayesian model with optimized feature combination," in *Journal of Parallel and Distributed Computing*, vol. 74, no. 1. Elsevier, 2014, pp. 1820–1832.

[7] D. Jacobson, D. Yuan, and J. Neeraj, "Scryer: Netflix's Predictive Auto Scaling Engine." The Netflix Tech Blog., retrieved on: April, 2016.

[8] C. Delimitrou and C. Kozyrakis, "HCloud: Resource-Efficient Provisioning in Shared Cloud Systems," *SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 473–488, May 2016.

[9] C. Bandi, D. Bertsimas, and N. Youssef, "Robust Queueing Theory," *Operations Research*, vol. 63, no. 3, pp. 676–700, 2015.

[10] L. Jiao, A. Tulino, J. Llorca, Y. Jin, and A. Sala, "Smoothed Online Resource Allocation in Multi-tier Distributed Cloud Networks," in *2016 IEEE International Parallel and Distributed Processing Symposium*, May 2016, pp. 333–342.

[11] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile Dynamic Provisioning of Multi-tier Internet Applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 3, no. 1, pp. 1:1–1:39, March 2008.

[12] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, Model-driven Autoscaling for Cloud Applications," in *11th International Conference on Autonomic Computing*, June 2014, pp. 57–64.

[13] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 34–41.

[14] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, "Dependable Horizontal Scaling Based on Probabilistic Model Checking," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 31–40.

[15] S. A. Plotkin, D. B. Shmoys, and Éva Tardos, "Fast Approximation Algorithms for Fractional Packing and Covering Problems," *Mathematics of Operations Research*, vol. 20, no. 2, pp. 257–301, 1995.

[16] R. B. Cooper, *Introduction to Queueing Theory*, 2nd ed. New York, NY: North-Holland, 1981.

[17] K. Park, G. Kim, and M. Crovella, "On the relationship between file sizes, transport protocols, and self-similar network traffic," in *Proceedings of 1996 International Conference on Network Protocols*, October 1996, pp. 171–180.

[18] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the Self-similar Nature of Ethernet Traffic," *IEEE/ACM Transactions on Networking*, vol. 2, no. 1, pp. 1–15, February 1994.

[19] P. R. Jelenkovic, A. A. Lazar, and N. Semret, "The Effect of Multiple Time Scales and Subexponentiality in MPEG Video Streams on Queueing Behavior," *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 6, pp. 1052–1071, Aug 1997.

[20] W. Willinger, V. Paxson, and M. S. Taqqu, *A Practical Guide to Heavy Tails*, R. J. Adler, R. E. Feldman, and M. S. Taqqu, Eds. Birkhauser Boston Inc., 1998.

[21] F. Pollaczek, *Problèmes stochastiques posés par le phénomène de formation d'une queue d'attente à un guichet et par des phénomènes apparentés*. Gauthier-Villars, 1957.

[22] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski, *Robust Optimization*, ser. Princeton Series in Applied Mathematics. Princeton University Press, October 2009.

[23] J. P. Nolan, *Stable Distributions - Models for Heavy Tailed Data*. Boston: Birkhauser, 2015.

[24] B. M. Hill, "A Simple General Approach to Inference About the Tail of a Distribution," *The Annals of Statistics*, vol. 3, no. 5, pp. 1163–1174, 09 1975.

[25] P. Cízek, W. Härdlem, and R. Weron, "Statistical Tools for Finance and Insurance," 2016. [Online]. Available: http://fedc.wiwi.hu-berlin.de/xplore/tutorials/stfhtmlnode8.html

[26] D. V. Lindley, "The Theory Of Queues With A Single Server," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 48, no. 2, p. 277–289, October 2008.

[27] N. Krivulin, "A Recursive Equations Based Representation for the g/g/m Queue," *Applied Mathematics Letters*, vol. 7, no. 3, pp. 73 – 77, May 1994.

[28] D. Mosberger and T. Jin, "Httperf—a tool for measuring web server performance," *SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, December 1998.

[29] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing: Principles and Paradigms*. John Wiley & Sons, 2010, vol. 87.