SQLR: Short-Term Memory Q-Learning for Elastic Provisioning

Constantine Ayimba, Paolo Casari, *Senior Member, IEEE*, Vincenzo Mancuso *Member, IEEE*

***Abstract*—As a growing number of service and application providers choose cloud networks to deliver their services on a software-as-a-service (SaaS) basis, cloud providers need to make their provisioning systems agile enough to meet service level agreements (SLAs). At the same time, they should guard against over-provisioning, which limits their capacity to accommodate more tenants. To this end, we propose Short-term memory Q-Learning pRovisioning (SQLR, pronounced as "scaler"), a system employing a customized variant of the model-free reinforcement learning algorithm. It can reuse contextual knowledge learned from one workload to optimize the number of virtual machines (resources) allocated to serve other workload patterns. With minimal overhead, SQLR achieves comparable results to systems where resources are unconstrained.**

Our experiments show that we can reduce the amount of provisioned resources by about 20% with less than 1% overall service unavailability (due to blocking), while delivering similar response times to those of an over-provisioned system.

***Index Terms*—Provisioning; Service management; Optimization; Q-Learning; SLA;**

I. Introduction

A growing tendency among application service providers (ASPs) is to leverage cloud networks to deliver services to consumers [1]. Cloud networks help reduce capital expenditure (CAPEX) as ASPs can host their services by leasing computing infrastructure from cloud service providers (CSPs), rather than owning it. ASPs also reduce operating expenditure (OPEX), because they only pay for the resources they use, and do not incur maintenance costs.

As a consequence, CSPs face increasing demands on finite resources from ASPs that typically have high expectations on performance [2]. CSPs must therefore balance the need for high quality of service (QoS) guarantees with just the right amount of resources. In this way, cloud services can be delivered cost-effectively, without violating service level objectives (SLOs). Two such SLOs are service availability and response time. These are usually stipulated in a service level agreement (SLA) between a CSP and an ASP.

SLO violations can have grave consequences for an ASP, including loss of users and revenue [3]. Depending on the SLA contract signed, the ASP may even seek compensation from the CSP for such violations [4]. For CSPs, this creates the need for dynamic provisioning/scaling tools. Such tools increase the resources allocated to an application when sudden increases in demand occur (or are foreseen), and release resources when they are no longer needed. The adaptations of allocated resources (i) save costs for ASPs and (ii) free capacity for other CSP tenants.

Most state-of-the-art solutions to this problem guide scaling by continuously monitoring application and system metrics [5]. Other recent proposals addressing similar resource allocation problems, such as those presented in [6], leverage model-based machine learning. Though computationally efficient, they rely on extensive modelling and simulations, which may not always correspond to real-world demand and cloud application dynamics. Significantly different applications may require very different scaling configurations. Therefore, the challenge is to minimize the resources assigned to any application, while guaranteeing service quality in the face of variable demand.

To address this challenge, we developed an application-agnostic Short-term memory Q-Learning pRovisioning

TABLE I
List of key abbreviations employed in this paper

5G	Fifth-Generation Cellular Networks
AC	Admission Control
ASP	Application Service Provider
BTU	Billing Time Unit
CAPEX	Capital Expenditure
CSP	Cloud Service Provider
EKF	Extended Kalman Filter
LB	Load Balancer
LSTM	Long-Short Term Memory
MDP	Markov Decision Process
MEC	Multi-access Edge Computing
NUMA	Non-Uniform Memory Allocation
OPEX	Operating Expenditure
QoS	Quality of Service
RL	Reinforcement Learning
RLPAS	Reinforcement Learning-based Proactive Auto-Scaler
SaaS	Software-as-a-Service
SLA	Service Level Agreement
SLO	Service Level Objective
SQLR	Short-term memory Q-Learning pRovisioning
VM	Virtual Machine

Manuscript received June 27, 2020; revised January 08 and April 15, 2021; accepted April 21, 2021. The associate editor coordinating the review of this paper and approving it for publication was D. Pezaros. (*Corresponding author: C. Ayimba.*)

C. Ayimba (e-mail: constantine.ayimba@imdea.org) and V. Mancuso (e-mail: vincenzo.mancuso@imdea.org) are with the IMDEA Networks Institute, 28918 Leganés, Madrid, Spain.

C. Ayimba is also with the University Carlos III of Madrid, 28918 Leganés, Madrid, Spain.

P. Casari (e-mail: paolo.casari@unitn.it) is with the Department of Information Engineering and Computer Science, University of Trento, 38123 Povo (TN), Italy.

This work received support from the Spanish Ministry of Science and Innovation grant PID2019-109805RB-I00 (ECID), and from the Italian Ministry for University and Research (MIUR) under the initiative "Departments of Excellence" (Law 232/2016).

Digital Object Identifier XX.XXXX/XXXXXXXXXXX

(SQLR) system. Our scheme leverages two distinct and co-operating model-free reinforcement learning (RL) agents. The first agent uses conventional Q-Learning to make admission control decisions. Specifically, it learns the utilization threshold of compute resources that would make further task admissions inconvenient. The second RL agent uses a customized, context-aware Q-Learning algorithm to make resource scaling decisions when the system is exposed to dynamic and stochastic workloads. Since RL works by learning from experience rather than training on static data-sets, RL offers a practical and adaptive solution to the problem. Concretely, our main contributions are:

1) A configuration-agnostic admission control agent based on Q-Learning, that learns the most appropriate action to take given the level of resource utilization reported by a virtual machine (VM) instance.
2) A flexible RL scaling agent that
   - is horizontal, i.e., it adapts resources in terms of the number of VMs allocated to a service, without resizing them by changing, e.g., the number of virtual CPUs, or the amount of memory allotted;
   - given high-level objectives, learns and enforces the best tradeoff between service availability and resource costs, even in the presence of challenging workloads;
   - quickly adapts to previously unexplored workloads by learning from *observed* resource utilization patterns rather than from the workloads *per se*;
   - progressively improves with every scaling decision, resulting in better service reliability and availability as time passes by.
3) A weighted fair learning mechanism to scale resources horizontally. This scheme encourages the exploration of new system states, while consistently exploiting better known states; this increases the likelihood of selecting near-optimal actions prior to completing the exploration of all possible states, i.e., much before reaching full policy convergence.

Our work is also relevant for modern cloud deployments such as multi-access edge computing (MEC) for fifth-generation (5G) networks. Here, vertical scaling (which entails adjusting the capacity of running VMs) is not preferred given the constraints on MEC deployments, and horizontal scaling represents a more viable solution [7], [8].

The remainder of this article is organized as follows: Section II examines prior work on the scaling of cloud resources; in Section III, we discuss the II conventional Q-Learning algorithm and some ancillary modifications we made to it. In Section IV, we present the design of our system. In Section V, we outline the experiments made to test the scaling and admission control algorithms. We discuss the results of our approach compared to other methods in Section VI. In Section VII, we finally provide the main conclusions and discuss future directions along this line of research. A listing of the abbreviations used in this work is given in Table I.

II. Related Work

In Table II, we present a summary of recent research on auto-scaling. The most common approaches for automated scaling decisions are rule-based, and rely on fixed resource utilization thresholds. This is the case for such commercial tools as Rightscale [9] and Amazon's EC2 [10]. The fuzzy logic variants in [11]–[13], among others, rely instead on loosely defined thresholds. These methods require sufficient knowledge of the cloud application in order to define the operating bounds correctly. Recent work on communication security [14] has shown that RL can be exploited to learn threshold values for attack detection. Our work demonstrates that it is similarly applicable to resource provisioning. In addition, the authors of [15] demonstrate that deep RL can be leveraged for resource scheduling in a cloud network consisting of Internet of Things (IoT) devices.

The authors of [16] propose a theoretical, model based, RL approach to cloud resource allocation which factors in both SLO violations and net gains for the CSP. Their model, however, assumes high predictability in arrival rates and system responses, both of which are highly stochastic.

In [17], the authors propose an on-policy reinforcement learning-based proactive auto-scaler (RLPAS). Their technique leverages a one-step temporal difference scheme with multiple coordinating agents. Application-specific targets for throughput and response time drive reward functions. However, this method requires to approximate the action-values, and tends to bias action selection. This makes it ill-suited to highly dynamic workloads.

The authors of [18] implement a vertical scaling agent based on Q-learning. Distributed RL agents adjust the CPU, memory and bandwidth allocations to a set of active VMs handling different applications. The effect on the applications' response time and throughput act as inputs for the rewards fed back to the agent. Application agents maintain fine-grained SLA metrics for each application.

The work in [25] proposes an RL-based agent that triggers the migration of VMs from under-utilised servers, in order to power them off. Utilization bounds that trigger decisions in this scheme are predetermined. This implies that the response of the agent will be compromised, should the system configuration change in a significant way.

The authors of [19] use a queue model and an extended Kalman filter (EKF) for horizontal scaling. They use a 3-tier cloud application with 3 classes of requests to generate the measurement model. The model, enhanced by EKF, estimates the response times given the workload as input. These estimates trigger an appropriate horizontal scaling operation.

Other methods seek to characterize the workload and make resource allocations accordingly. In [20], Vasic *et al*. classify workloads based on recurring patterns. They derive optimized resource allocations for these patterns, and re-use them every time the same patterns appear in a new workload.

Ibidunmoye *et al*. [21] use a modified Q-Learning scheme in order to carry out vertical scaling, defined as the addition of system resources, e.g., virtual CPUs on VMs, while they are running. Their state space is based on a fuzzy logic

TABLE II
SUMMARY OF RELATED WORK

Reference	Parameters Considered	Scaling type	Technique used	Limitations
Rightscale [9]	Con gurable triggers (e.g., memory, utilization, etc.)	Horizontal	Application dependent rule-based threshold setting	Requires knowledge of cloud application and underlying con guration. Our scheme is application and con guration agnostic.
Alsarhan et al. [16]	Theoretical: arrival rates, service times	Horizontal	Theoretical model-based reinforcement learning.	Dependent on workload pro ling. Our scheme learns how to adapt to different workloads, and reuses knowledge for similarities between them.
Benifa et al. [17]	Measured arrival rates, response times, throughput	Horizontal	Q-Learning with function approximation.	Too sensitive to transients in workload and CPU utilization. Given the use of global average metrics, our scheme does not react to transients but to trends instead.
Rao et al. [18]	CPU, memory and I/O utilization	Vertical	Model-free reinforcement learning with distributed agents.	VMs require autonomous control of host resources. Our scheme employs horizontal scaling, and executes it centrally as is common practice for most CSPs.
Gandhi et al. [19]	Measured arrival rates, response times	Vertical and Horizontal	Application modeling and extended Kalman lter (EKF).	Produces stiff scaling response, requires knowledge of expected response times. Our system is more responsive to workload trends and does not require any knowledge of the cloud application.
Vasić et al. [20]	Measured arrival rate	Horizontal	Workload pro ling, classi cation and pattern matching.	Unexpected workloads may cause erratic behavior. Our scheme does not require workload pro ling and can learn how to respond to unexpected workloads.
Ibidunmoye et al. [21]	CPU utilization, response times	Vertical	Model-free reinforcement learning based on fuzzy logic with multiple agents.	Fuzzy state classi cation requires knowledge of cloud application and con guration. Our scheme is both application and con guration agnostic.
Liu et al. [22]	CPU and memory utilization	Horizontal	Standard reinforcement learning with aggressive rewards for over-provisioning.	Prone to wasteful over-provisioning. Our learning objective optimizes the use of resources to a necessary minimum.
Fernandez et al. [23]	Measured arrival rates, CPU utilization and throughput	Horizontal	Threshold-based technique based on short-term capacity forecasts.	Vulnerable to over/under provisioning when faced with unpredictable workloads. Our provisioning scheme progressively learns on new workloads and as such over time is less vulnerable to unpredictable workloads.
Xu et al. [24]	Pricing and availability of transient servers	Horizontal	Long-short term memory price prediction of transient servers.	Relies on short-lived virtual instances, possibly yielding inconsistent application performance. In contrast, our learning approach centers on application performance, correlated to CPU utilization.

combination of response times and utilization levels. They employ several cooperating agents to simultaneously explore the state space, in order to speed up policy convergence.

The authors of [22] propose a scheme based on Q-Learning and heuristics that immediately over-provisions resources when it detects an increase in the workload. It then gradually de-allocates extra resources. The objective of this scheme is to reduce SLO violations that occur when the workload increases suddenly, but resources are added conservatively.

In [23], the authors propose a system which pro les resource capacities, predicts the subsequent workload pattern over a monitoring window, and scales the system accordingly based on a tradeoff between scaling costs and SLOs.

The authors of [24] develop a long-short term memory (LSTM) algorithm to of oad big data analytics to Amazon EC2 spot instances. The scheme trades off the very cheap prices of these transient servers with their unreliability due to revocation. Given its focus on ASP costs, this work is largely orthogonal to our approach, as we rather consider CSP costs.

Leitner et al. [26] consider the provisioning problem in terms of costs for the provider. The authors propose a system that schedules resources in a bid to minimize the costs incurred due to SLO violations and those resulting from leasing cloud resources. This work assumes that such costs are known well in advance, and that billing time units (BTUs) for leasing resources are both coarse-grained (in the order of hours) and xed. However, recent proposals on cloud brokerage [27] promise greater exibility by making BTUs much more ne-grained. This would be much more cost-effective for tenants with short-lived requirements.

In summary, whereas rule-based schemes are simple and easy to implement, setting the correct thresholds requires both specialized cloud application domain knowledge and awareness of resource con gurations. Other state-of-the-art scaling methods require to continuously measure and monitor the service response times, the workload, or both. In many cases, obtaining such data requires real-time analysis of logs, which may lead to signi cant overhead in large-scale systems. Workloads, in particular, may also exhibit unpredictable behavior [28], resulting in premature scaling directives.

Our scheme, instead, infers the changes in workloads by monitoring how the system responds to them. This provides a more robust basis for decision-making, even in large deployments. Different workloads trigger distinct system state transition sequences, resulting in the learning of new policies, in addition to those already learned. Hence, the system will scale for future workloads that exhibit combinations of the already observed patterns, without need for further training.

III. RL Modeling and Modified Q-Learning Approximation

A. Key idea

We assume a resource scaling system that allocates tasks to available VMs in a cloud computing environment, and that horizontally scales the number of VMs in face of time-varying workload. The scaling system is composed of three agents: a load balancer (LB) agent, an admission control (AC) agent, and a scaling agent. The LB assigns each incoming tasks to a VM, then the AC decides whether or not to admit the task to that VM. Periodically, the scaler oversees the utilization of active VMs and adapts their number to match workload requirements.

By design, our system is oblivious to the particular application that runs in the cloud: therefore, we only rely on system-level metrics (e.g., CPU utilization and time evolution thereof) in order to make admission and scaling decisions. We tackle the complexity of this scenario while keeping the system responsive to changes in stochastic workload patterns. To do so, we design our AC and scaling agents as sequential decision processes, where optimal decisions are identified thanks to a Q-learning approach.

In the following, we detail our system model for decision making (Section III-B), discuss the admissible actions and state space of the AC and scaling agents (Section III-C), explain our decoupled learning mechanisms (Sections III-D and III-E) and how we drive the exploration of new decisions against the exploitation of best decisions found up to a certain time (Section III-F).

B. System model for decision making

Like in [16], we treat AC and horizontal scaling as sequential decisions, and assume that load balancing policies exist to evenly share workload among active VMs in the long run. Specifically, we consider a simple load balancer that dispatches incoming tasks to VMs by privileging the least utilized VM. We can then approximate the admission control and scaling processes as two separate classes of Markov decision processes (MDPs) with distinct sets of actions A, states S, state transition probabilities T and reward functions R.

We consider separate admission control and scaling decisions because: (i) the AC needs to operate at a much shorter time scale than resource scaling, and (ii) resource scaling cannot instantaneously obviate a lack of resources, because starting VMs up takes a non-negligible time. Moreover, the AC dropping rate for a given workload offered to a VM depends on the utilization of that VM and not on the number of deployed VMs. Therefore we can make AC decisions locally at each VM, in contrast with the necessarily global scope of scaling decisions.

We remark that treating our decision processes as MDPs requires a necessary approximation. The transition probabilities of a standard MDP are well defined: by way of contrast, our transition probabilities depend on the input workload, which is not necessarily stationary or known. Moreover, instantaneous fluctuations in the workload patterns are possible, which may lead to unexpected transitions. For such events, a plain memory-less decision process is inadequate [29, §17.3].

Yet, approximating a decision process through the MDP framework is still feasible, because we can track stochastic workload fluctuations by incorporating some memory of the past in the state definition, as will become clear later.

C. RL short-memory decision agents

Our system encompasses a load balancing component, a scaling component, and an AC component. We implement AC and scaling as reinforcement learning agents, whereas the load balancing component does not require learning in our setup.

A learning step[1], or epoch, consists of the agent observing its state, taking some action allowed in that state, and monitoring the environment to compute and accrue some positive or negative reward as the environment transitions to a new state.

For an AC agent, the permissible actions are to either (i) pass an incoming request on to the VM that will serve the request, or (ii) refuse service to an incoming request. Either action results in a VM transitioning from one level of utilization to another with some probability. By defining the state as the utilization level of the VM handling the request, we are able to formulate this process as a per-VM reinforcement learning problem. We structure the reward values of the AC agent based on the predictability of job service times, which in turn relates to VM utilization level. For a given machine, the service time increases exponentially with its load [30]. The role of the AC agent is therefore to infer the next VM utilization level after an AC decision.

For the horizontal scaling agent, the actions are: (i) increasing, (ii) decreasing, or (iii) maintaining the same number of VMs. The state, in this case, is defined by three values:

1) the average system-wide utilization over the previous epoch;
2) the average system-wide utilization in the current epoch;
3) and the number of active VMs.

The action of the scaling agent changes this set of values, hence the system state. Therefore, unlike conventional RL, decisions depend not only on the current state, but also on the transition that led to the current state. This means that decisions are not memory-less. Rather, the agents have to embed some short-term memory in their decisions. To make this setting compatible with an MDP model, we define the system state such that it includes the average utilization level in the previous state, along with the previous number of active VMs. Thus, our modified RL algorithm retains a memory of the immediate past in the present state.

Thanks to this definition of state, our scaling agent exploits the rate of change in global average utilization levels, and can thus distinguish different workload patterns. This allows the agent to cumulatively learn different VM scaling policies for different workload profiles without overriding previously learned policies. We structure the reward function of the scaling agent to include both the blocking rate resulting from AC and the number of VMs used.

[1] Also called time-step in RL literature.

D. Decoupled learning of agents

Given that the AC decision is local to each VM, the AC agent must learn admission policies before the scaling agent can refine scaling policies. AC policies can be learned offline for a single VM, and do not need later refinements. Once AC policies have stabilized, the scaling agent can continuously learn its policies as new workloads are observed. For this purpose, the scaling agent simply needs to use a reward function that includes the blocking rate resulting from the use of the AC.

E. Modified Q-learning approximation

Since our RLs can be described by MDPs, the optimal AC and scaling decision policies could be determined by evaluating the corresponding MDPs. This entails tracking how much reward an action receives and obtaining the state transition probabilities that yield the highest accumulated reward given a particular system state. Then, we can program an agent via either value or policy iteration, in order to execute the resulting policies.

However, policy evaluations of the MDPs are impractical given that transition probabilities can vary widely depending on the workload and configuration of the system. A practical solution that applies to this case is Q-Learning [31]. Here, the agent develops a mapping of states to actions (known as the Q function) by tracking the accumulated reward (or "Q-value") for each state-action pair. With reference to Table III, which summarizes the key notations used in this paper, we now explain the design principles and behavior of the scaling and admission control agents. From [29], at learning step t, the optimal action-value function q_* is approximated as:

$$Q(S^{(t)}, A^{(t)}) \leftarrow \alpha R' + (1-\alpha) Q(S^{(t)}, A^{(t)}), \quad (1)$$

where

$$R' = R^{(t+1)} + \gamma \max_a Q(S^{(t+1)}, a), \quad (2)$$

$Q(S^{(t)}, A^{(t)})$ is the action-value[2], and $R^{(t+1)}$ is the immediate reward the agent receives after taking action a and ending up in state $S^{(t+1)}$, whose action-value is $Q(S^{(t+1)}, a)$. The factor γ anticipates the contribution of future rewards towards the immediate reward [31].

However, the use of a fixed learning rate α in (1) assumes that all states are visited evenly during training [29]. Given the formulation of the state space, this may not always be the case for our AC and scaling agents. Further, the update process in (1) typically leads to a stochastic policy, with Q function values oscillating slightly about an estimated expected value. To ameliorate this effect, we employ a modified reward mechanism, which takes into account the number of times that the agent visited the given state. This method follows closely the algorithm for the online computation of the mean:

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k}(x_j) = \frac{1}{k}(x_k + (k-1)\mu_{k-1}). \quad (3)$$

[2] In this paper, we use the terms action-value and Q-value interchangeably.

The Q function update then becomes:

$$Q(S^{(t)}, A^{(t)}) \leftarrow \frac{1}{k}\left[\delta + (k-1) Q(S^{(t)}, A^{(t)})\right], \quad (4)$$

where $\delta = \gamma R' - Q(S^{(t)}, A^{(t)})$, and k is the number of learning steps (prior to the current action) when the agent found itself in state $S^{(t)}$ and acted with action $A^{(t)}$.

Note that we modify the commonly adopted update mechanism by using the discounted reward instead of the immediate reward R' in (4). This reduces the chance of wrongly estimating the mean action value at the initial learning phases. The update equation in (4) also guarantees that the policy will eventually converge, since the update value on the right-hand side of (4) becomes progressively smaller as the number of learning steps increases.

F. Exploration/exploitation tradeoff mechanism

For both AC and resource scaling, we train the agents by initially encouraging random actions (exploration phase). As the agent develops a policy, it progressively acts less randomly, i.e., it chooses those actions that are known to yield the highest reward (exploitation phase). We accomplish this by employing ϵ-greedy action selection [29]. In this scheme, the agent selects the action that yields the highest reward with probability $1-\epsilon(s)$, and a random action with probability $\epsilon(s)$.

We remark that the agent does not visit all states with the same frequency. Therefore, a global assignment and decrease of ϵ may bias the learned policy towards the most visited states. To avoid this, we employ a scheme that reduces ϵ independently for each state, proportional to the number of times $i(s)$ that state s is visited. This accelerates the learning process by encouraging exploration for the least visited states, while exploiting optimal actions for the most visited states. Specifically, we set:

$$\epsilon(s) = \begin{cases} 1 - \frac{i(s)}{M}, & \text{if } i(s) < M \\ \epsilon_{min}, & \text{if } i(s) \geq M, \end{cases} \quad (5)$$

where M is a design parameter representing the number of statistically significant visits that should result in convergence to a stable policy. We consider that state s has achieved convergence when $\epsilon(s)$ equals $\epsilon_{min} > 0$. For clarity, in what follow we drop the dependence of ϵ and i on the state s.

In order for the system to perform satisfactorily prior to convergence, we devise a *weighted fair guided exploration scheme*. Consider learning instance i. If the most rewarding action is not chosen (which occurs with probability ϵ), the conditional probability $P_a^{(i)}$ of selecting any of the L possible actions depends on its present action-value $Q^{(i)}(s, a)$, and on the number of times $k^{(i)}$ that action a has previously been selected when the system was in state s:

$$P_a^{(i)} = \begin{cases} \frac{1}{L}, & \text{for } \rho_a^{(i)} = 0 \\ \frac{\rho_a^{(i)}(1-\tanh \rho_a^{(i)})}{\sum_{j=1}^{L}\rho_j^{(i)}(1-\tanh \rho_j^{(i)})}, & \text{for } \rho_a^{(i)} > 0 \end{cases} \quad (6)$$

where

$$ {}_a^{(i)} = Q^{(i)}(s;a) + \sum_{j=1} Q_j^{(i)}(s;a) \; ; \qquad (7)$$

$$ {}_a^{(i)} = \frac{k^{(i)}}{i} : $$

Note that, in (6), $^{(i)} > 0$ is used in place of the action value $Q^{(i)}(s;a)$ which, if negative, would result in unfeasible probabilities. We choose the hyperbolic tangent as a suitable weighting function since $0 \leqslant \tanh(\;) \leqslant 1$ for > 0.

The above strategy achieves a tradeoff between exploration and exploitation, and curtails the detrimental effects of unguided exploration on performance.

IV. SQLR Design

A. Key idea

In Section III, we have described a system that performs load balancing, admission control and horizontal scaling. The load balancer is simple and fair to active VMs, and optimizes resource utilization while minimizing AC blocking events [32]. The AC agent runs based on a QoS consideration: accepted tasks should receive a reasonably predictable service time. Another learning agent runs the third and most central component of our algorithm: horizontal resource scaling. This agent decides based on a cost-bene t tradeoff: utilize the least number of VMs so that the AC blocking rate be under a target threshold. With the above, we can formalize an optimization problem and design a framework to dynamically optimize the resources allotted to service a workload, as shown hereon.

B. Problem formalization

With the learning model described in Section III, the resource adaptation problem we tackle in this paper can be stated as follows: *Maximize the number of served tasks by horizontally scaling the number of VMs as workload evolves over time. Minimize the number of instantiated VMs that run a given service, under the constraint that the probability to block a service request remains below a prede ned threshold.*

Formally, let $X_{ji}(t) = 1$ if task j arrives at time t and is assigned to VM i, and $X_{ji}(t) = 0$ otherwise. Also, let $Y_{ji}(t) = 1$ if task j is running on VM i at time t, and 0 otherwise. Call $V(t)$ the number of VMs activated at time t, V_{max} the maximum number of VMs reserved for an ASP, $A(t)$ the set of tasks arriving at time t, and $J(t)$ the set of tasks to be served at time t. Let P_{blk} be the ideal blocking probability set out in the SLA, and $_j$ be the contribution of task j to the utilization level of a given VM. Finally, call x_{lim} the utilization level above which response times become unpredictable. We can express our problem formally as:

$$\min \frac{1}{T}\int_0^T V(t)\,dt \qquad (8a)$$

$$\text{s.t.:} \; \frac{\int_0^T \sum_{i=1}^{V(t)} \sum_{j=1}^{A(t)} X_{ji}(t)\,dt}{\int_0^T A(t)\,dt} \geqslant 1 \quad P_{blk} \; ; \qquad (8b)$$

$$1 \leqslant V(t) \leqslant V_{max} \quad \forall t \; ; \qquad (8c)$$

$$\sum_{i=1}^{V(t)} X_{ji}(t) \leqslant 1 \; ; \qquad (8d)$$

$$\sum_{i=1}^{V(t)} Y_{ji}(t) \leqslant 1 \; ; \qquad (8e)$$

$$\sum_{j \in J(t)} {}_j Y_{ji}(t) \leqslant x_{lim} \; ; \quad 1 \leqslant i \leqslant V(t) : \qquad (8f)$$

Constraint (8b) ensures that the number of tasks dropped remain within SLA bounds for service unavailability. Constraint (8c) ensures that the number of VMs reserved for an ASP is bounded. Constraint (8d) mandates that each task be assigned to a single VM, and (8e) indicates that each task can only be running on one VM at a time. Constraint (8f) avoids driving the utilization of the VM above an allowable level x_{lim}, which is the threshold learned by the AC agents. This ensures that tasks admitted to a VM will not suffer from unpredictable response times. Fig. 1, extracted from the extensive analysis in [30], illustrates what AC agents typically observe. Section IV-D details how the AC agents detect and learn the knee of the curve, so as not to drive a VM's CPU utilization beyond limits that would make the response time unpredictably high.

Besides $A(t)$ and $_j$ being unknown functions, the problem presented in (8a) is a variant of the knapsack problem, which

TABLE III
Key notation employed in the definition of SQLR

Variable	Meaning	Description
$Q(S^{(t)};A^{(t)})$	Action-value	The value of the Q function at time t.
$R^{(t+1)}$	Immediate reward	The reward the agent receives after taking action a and ending up in state $S^{(t+1)}$.
	Learning rate	A fraction that modi es the reward update and in uences the speed of convergence.
	Randomness factor	The probability of selecting an exploratory action prior to convergence.
$_{min}$	Minimum randomness factor	The minimum probability of selecting an off-policy action after convergence. We set this at 0.
k	State visits/action counter	The state- and action-dependent number of times the system was in state s and selected action a.
M	Visits to states after which $(s) = {}_{min}$	A statistically signi cant number of visits to achieve a stable policy for a given state. For the AC, we set $M = 1000$. For the scaling agent, we set it at ten times the number of actions allowed in that state.
	Discount rate $\in (0;1]$	Expresses the current value of a future reward due to the present action. We set $= 0.8$.
x_{bnd}	Utilization upper bound	The utilization level above which response times become unpredictable [30]. We set $x_{bnd} = 60$ %.
x_n	Highest quantized utilization	The quantized utilization level closest to x_{bnd} used in the AC policy (see Fig. 4). We set $n = 3$ in (9).
x_{lim}	Utilization admission limit	The practical limit of resource utilization obtained after training the AC.
	Resource cost modi er	Multiplier that weighs the cost of deploying resources in the reward function.
	Blocking probability modi er	Multiplier that weighs the blocking rate in the reward function.
P_{blk}	Target blocking probability	We set $P_{blk} = 0.001$, corresponding to service availability of 99.9%.
R_{min}	Minimum reward	Small, positive reward for maintaining the blocking probability lower than P_{blk}. We set $R_{min} = 0.001$.

Fig. 1. Response time variation with load based on queuing theory results [30]. The variation is approximately linear just below the "knee" of the curve. If utilization levels remain below this value, response times are highly likely to be predictable and reliable.

Fig. 2. SQLR block diagram. "LB" is the Load Balancer agent and "AC" is the Admission Control agent.

is NP-hard and cannot be solved exactly in polynomial time. Furthermore, the analytical modelling of ${}_j$ is impractical in real environments, because of the excessively high number of concurrent factors that affect it. These include the complexity of an incoming task, the architecture of computing hardware, operating system scheduling and thread handling, the presence of ongoing background processes, etc.

Instead, the Q-Learning approximation described in Section III solves the NP-hard problem near-optimally, under the uncertainty of operational system conditions. Indeed, Q-Learning is known for its versatility in finding near optimal solutions in uncertain settings [29]. We remark that problem (8a) optimizes VM provisioning. This means that the horizontal scaler should learn the behavior of the AC agent so as to block the least number of tasks with the least possible number of VMs. Therefore, we chain two separate learning processes: first the AC agent learns how to accept or reject workload to avoid unpredictable service times; then the scaler learns how to act around the AC's behavior to avoid blocking with the least number of VMs.

For a practical implementation of the optimization, we use the block diagram shown in Fig. 2. We call the resulting system SQLR, read as "scaler," because we have crafted a definition of state that embeds short-term memory, as described in Section III-C, for the scaling agent, and because SQLR can be classified as a dynamic resource-provisioning scheme. SQLR comprises: a LB, AC and a scaling agent. We describe each component in the following subsections.

C. Load balancer

As this component does not constitute a contribution of our work, we only mention it briefly here. We log CPU utilization at 1 s intervals. Our LB works by delivering an incoming task to the VM with the lowest, most recently logged CPU utilization at the time such task arrives. This policy is similar to server state-based strategies used for classic web traffic [33], and yields a high probability that the available resources are evenly loaded in the long run. Hence, the disparity in overall response times is also reduced.

D. Admission control

In a resource-constrained system, admission control ensures that the system does not take on more tasks than it can satisfactorily handle. According to [30], it is possible to use the theoretical utilization bound (x_{bnd}) to make the admission decision. However, to obtain the best results for an actual system, the admission control agent must learn an appropriate admission limit (x_{lim}) that takes the system configuration into account. As mentioned in Section III, we do this by treating admission control as an MDP.

AC action and state spaces—The action space of the AC agent consists of the mutually exclusive options:

1) ADMIT: allow an incoming task to be served by a VM;
2) DROP: refuse service to an incoming task.

The state space derives from the quantized levels of resource utilization on the VM serving the task. The resource we consider in this work is CPU utilization. This low-level metric correlates well with the workload, and does not require any domain-specific knowledge of the deployed application [22]. Bearing in mind that CPU utilization greatly impacts response times, we choose the upper utilization threshold as the one beyond which the service times will likely violate the agreed SLO. We use this threshold as a target to determine the rewards/penalties the admission controller will accrue as it builds a policy using Q-Learning. Building an AC policy therefore consists in identifying the most rewarding action (ADMIT or DROP) for each state.

Discretized AC state space—In order to obtain a discretized state space, we partition the utilization values corresponding to predictable response times into regions. To this end, we employ the geometric quantizing function:

$$x_j = \left\lfloor \left(1 - \left(\frac{1}{2}\right)^j\right) x_{bnd} \right\rfloor; \quad j = 0, 1, \ldots, n; \qquad (9)$$

All values above x_n (which correspond to a CPU utilization greater than $(1 - (1/2)^n)\, x_{bnd}$) can be regarded as a single, undesired state, and need no further quantization. Note that x_n is the quantization level closest to the ideal utilization. Therefore, operating a VM beyond x_n likely leads to service times that violate SLOs.

Fig. 3. In uence of CPU utilization on service response times.

Fig. 4. Q function table to train the AC. The gray area represents the ideal operating region at which resources are highly utilized and the service times are within SLOs. The red-shaded area on the right represents the region where VM operation is likely to cause SLO violations.

By using the geometric quantizer provided in (9), we achieve both coarse and ne adjustment. The quantizer is coarse and reduces the state space (and hence the time needed to train the agent) by sparsely quantizing the load levels just below x_{bnd}. At loads closer to x_{bnd}, the quantizer becomes ne-grained, Therefore, the AC agent learns how the VM responds to such high loads with a small quantization error. Indeed, it is fundamental to accurately learn which load value $x_{lim} < x_{bnd}$ ensures predictable service times in a real system. In fact, the value of x_{bnd} is inferred from ideal theoretical analysis, hence it is practically too high and may lead to undesirable service times. Therefore the AC agent employs $x_{lim} < x_{bnd}$ to make admission choices.

We now explain the details of load limit calculations in theory (x_{bnd}) and in practice (x_{lim}).

Theoretical AC admission bound—We choose the CPU utilization threshold x_{bnd} based on the analytical results described in [30], and relating response times to occupancy in a processor sharing queue. The time T, taken by a processor with capacity C operations per second to serve a request requiring ℓ operations is given by:

$$T(\ ) = \frac{\ell}{C}; \qquad = -; \tag{10}$$

where is the arrival rate (i.e., the workload) and is the departure rate. The occupancy of the processor is here considered as its utilization level.

We refer the reader back to Fig. 1, which plots Eq. (10). The point at which the gradient of the curve changes from an almost constant value to an exponential rise is chosen as the threshold beyond which service times become unpredictable and unreliable. We compute this value by taking the intersection of the tangent to the curve at the initial point with 0% utilization with the tangent at the point where the gradient is approximately 0.5s per 1% rise in utilization. This queuing theory result for x_{bnd} assumes Poisson arrivals, but it ts well our experimental observations. An example of such an observations is depicted in Fig. 3, for the hardware/software con guration of a VM in our testbed. Considering tasks that require about 1.2 s to complete, we observe that service times are relatively constant around 1.2 s for utilization values lower than 62%. Instead, service times vary wildly for higher utilization levels. Accordingly, we set x_{bnd} = 60% to ensure a safety margin when building the discretized state space.

AC admission bound based on learned rewards—The immediate reward R for the action taken by the AC agent

Fig. 5. SQLR action and state space. K is the current number of active VMs, N is the number of VMs that can be added and N_r is the number of VMs that can be removed. In general, $N \neq N_r$. The state space, whose parameters are pre xed by (*), comprises the number of active VMs and the quantized values of the average CPU utilization for the set of active VMs.

is the load x, discretized to the nearest quantized level boundary (downwards for a DROP decision or upwards for an ADMIT decision), after an AC decision is made. Therefore, with reference to Fig. 4, we calculate the reward for making a decision while the quantized load is x_i and observing a resulting level of utilization x as:

$$R(x \mid x_i) = \begin{cases} x_k; & \text{if DROP;} \\ x_{k+1}; & \text{if ADMIT,} \end{cases} \tag{11}$$

where $k = \arg\max_j (x_j < x)$. At the boundary, $x_k = x_n$, and $x_{k+1} = x_{bnd}$. Beyond the boundary, when $x > x_{bnd}$, $R(x)$ is de ned as

$$R(x \mid x_i) = \begin{cases} x_{bnd}; & \text{if DROP} \\ \frac{1}{2}(x_{bnd} \quad 1); & \text{if ADMIT}: \end{cases} \tag{12}$$

As $x_{bnd} < 1$, Eq. (12) states that the reward for an ADMIT decision beyond the boundary is negative. This represents a penalty for violating the allowable CPU utilization limit.

For each x_i, the AC agent learns the optimal ADMIT/DROP policy by using the weighted fair exploration mechanism detailed in Section III-F: initially, the agent drops the tasks with probability 0.5, and subsequently it drops or accepts them

according to the action that corresponds to the highest Q-value (as computed with (4)) with probability . The training continues until each state (i.e., each quantized load region) is eventually marked as either ADMIT or DROP. This is when the AC policy converges. To do so, the final ADMIT/DROP marking of a region is determined after a minimum number of visits. In our case, goes to 0 after 1000 learning steps per load interval, so that if the accumulated Q-value for ADMIT is higher than the one for DROP, the load region will be marked as ADMIT, and DROP otherwise. Therefore, by training the AC agent with stochastic load variations, we can identify x_{lim} as the highest quantized value x_q for which the AC agent admits tasks.

Given the structure of the chosen reward function, x_{lim} prudently aims at maximizing the utilization of resources at a VM without violating response time requirements. Therefore, the scaling agent can use x_{lim} to make optimal scaling decisions, as shown in the next subsection IV-E.

Training the AC agent in practice—Having determined x_{bnd} to be 60%, the geometric quantizer is fully defined and the AC training phase can start. In practice, we only need to train one VM, because we have assumed homogeneity across VMs. Thus, we send tasks towards one VM and start updating the Q-values for ADMIT an DROP actions in each state. To explore all quantized levels of CPU utilization, we generate workload with high variability in inter-arrival times. When we visit a state for the number of times prescribed in (5), this state converges, and we lock the policy for this state into either an ADMIT or DROP decision, according to which one has the highest accumulated Q-value (4).

E. Scaling agent

We design and implement a Q-Learning scaling agent whose objective is to achieve as low a blocking rate as possible with as few resources as possible, according to the task admission policy AC agents previously learned.

Action and state spaces for horizontal scaling—The scaling agent adds VMs (scale-out) or removes VMs (scale-in) as appropriate, given the recent history of utilization of active VMs. Therefore the action space for the scaling agent is given by the range of VMs that can be added or removed. The state space consists of three values (i) the current number of VMs, (ii) the quantized values of the average CPU utilization for the set of active VMs in the *previous* epoch, and (iii) the quantized values of the average CPU utilization for the set of active VMs in the *current* epoch.

We show the state and action spaces for our horizontal scaling agent in Fig. 5. We represent each permissible action as a "card" indicating the number of VMs that must be added or removed when taking the action associated with the card. Moreover, each card consists of a grid (see Fig. 6) whose rows and columns are indexed with load levels. These levels represent the immediate past load and the current quantized load, respectively, thus expressing the short-term memory hidden in the state. The cells contain the cumulative reward obtained by a given state-action pair. Here, we quantize VM loads uniformly, so as to obtain a more granular view of system-wide resource utilization than a geometric quantizer would achieve. We choose uniform steps of 2% in the region between 0 and 20% of utilization, and steps of 5% in the region between 20% and x_{lim}. The finer sampling between 0 and 20% utilization yields better control when the workload is low: in these cases, only a marginal change is observed when a VM is added or removed. By way of contrast, if utilization is already high, adding or removing a VM causes significant utilization changes. Thus, quantizing utilization more coarsely already enables the detection of such changes, while reducing the state space. Finally, the region above x_{lim} conglomerates into a single, large level. In fact, at this region of utilization, a coarse scale-out decision is most likely, and does not require a fine resolution in the state space representation.

Fig. 6. State space detail for a "card" in the action space. Each cell's index pair is given by the quantized level of average system-wide resource utilization in successive epochs.

Scaling rewards—The scaling reward function (R_{sqlr}) consists of two components: (i) R_{blk}, computed by comparing the blocking probability P observed after a scaling to the maximum allowable blocking rate P_{blk}, and (ii) $R_{res} \leq 0$, which expresses the cost of resources, and depends on the number K of active VMs after the scaling decision). Specifically:

$$
\begin{aligned}
R_{sqlr} &= R_{blk} + R_{res}; \\
R_{blk} &= \begin{cases} R_{min}; & \text{if } P \leq P_{blk}; \\ (P_{blk} \quad P); & \text{if } P > P_{blk}; \end{cases} \\
R_{res} &= (1 \quad K);
\end{aligned}
\tag{13}
$$

where R_{min} is a small positive reward that the agent accrues as an incentive for keeping the system within the allowable service outage limits. The training parameters and act as modifiers, so that blocking probability violations receive a different penalty than the use of unneeded extra resources. This makes the scaling agent flexible. In fact, different CSPs may give different weight to SLOs violation penalties and cost savings achieved by reducing resource usage.

Initialization— As stated earlier, each card in the bubble shown in Fig. 5 consists of a grid, whose cell indices correspond to the average level of utilization of the active VMs over the previous epoch and the current epoch (cf. Fig. 6). We initialize the diagonal elements for the cases where the

number of VMs remain unchanged (card "0" in Fig. 5) as non-zero values. This helps drive initial decisions, e.g., to penalize scale-in and promote scale-out if the average utilization of the current number of VMs is too high. We set these diagonal values based on the following rationale: if the average utilization is below x_{lim}, i.e., the safe limit learned by the AC agent, no blocking is expected (zero probability). Conversely, if the utilization exceeds x_{bnd}, blocking is almost sure to happen. Instead, intermediate utilization values $x_{lim} < x < x_{bnd}$ yield a blocking probability that increases with x. Formally, we set the diagonal elements to equate the blocking probability $P_0(x)$, defined as follows:

$$P_0(x) = \begin{cases} 0; & x < x_{lim} \\ 1; & x > x_{bnd} \\ \frac{1}{2}\left[1 + \mathrm{erf}\left(\phi(x)\frac{e}{\sqrt{2}}\right)\right]; & \text{otherwise} \end{cases} \quad (14)$$

where

$$\phi(x) = \frac{x - x_{lim}}{x_{bnd} - x_{lim}}; \quad (15)$$

and $\mathbf{x} = [x_1^{(t)}; x_2^{(t)}; \ldots; x_n^{(t)}; 1]$. The above definition for the case $x_{lim} < x < x_{bnd}$ yields a smooth transition between blocking probabilities 0 and 1, as shown in Fig. 7. We recall the AC agent adaptively learns x_{lim} during its own training phase, so we can assume that the scaler knows the safe value of x_{lim} for any admissible number of active VMs. The diagonal elements computed above serve as the reference action values, $Q(S^{(t+1)}; a)$, for the updates in (4) after horizontal scaling.

The scaling procedure—With the above described actions, states, rewards and initialization, the scaling process is unrolled in Algorithm 1. Fig. 8 details a scale-out action. To describe the latter, we consider starting after a previous action having taken place at instant $t-1$. The first cell index is the quantized level of the average utilization in the interval $[t-2; t-1)$. At instant t (bottom bubble), our scaler obtains the quantized level of the average utilization in the interval $[t-1; t)$. This serves as the second cell index to be considered in selecting the action. The current number of active VMs, K, is also evaluated.

With this triplet of values, the current state is established, and we are ready to choose a scaling action (i.e., scale-in, scale-out, or keep the current number of VMs) based on (5) and (6). To do so, recall that every card corresponds to a scaling action, e.g., add 1 VM, remove 2 VMs, etc. We check the convenience of every action by reading the Q-value of the cell indexed by the quantized average utilization values at the current epoch (t), and at the previous one ($t-1$). Then, we choose a scaling action based on (5) and (6), by leveraging the above Q-value entries in every card of the action space within the bubble defined by K VMs.

For later reference we term the cell of the chosen action card as R-Cell (marked red in Fig. 8). After waiting shortly for the VMs to start up or shut down, and for the effect of the change to become manifest, we reach instant $t+1$. We can now compute the immediate reward as described in (13); this accounts for the blocking probability observed between time instants t and $t+1$, and for the number of active VMs at instant $t+1$. We also take into account the accumulated reward stored in card "0" at the diagonal cell indexed by the quantized average utilization value over the interval $[t, t+1)$ (this cell is colored green in Fig. 8). The above two values are used to update the Q-value in R-Cell as prescribed in (4).

How to train the scaling agent—After having trained the AC, we need to create a set of tables of the Q-values for all scaling actions. The number of tables depends on the highest number of VMs that can be provisioned, as well as on the number of VMs that can be added or removed within a single scaling decision. Then, with an instance of the (already trained) AC running at each active VM and a global scaling agent running, we expose the system to varying offered load profiles, and

Algorithm 1: Scaling Agent Algorithm

```
Result: Scaling action, updated table of Q-values
RunCount ← 0
n ← 0
while True do
    N_t ← GETACTIVEVMS()
    U_current ← 0
    foreach vm ∈ VMs do
        U_current ← U_current + GETUTILS(interval; vm)
    end
    x_t = GETQUANTIZEDUTIL(U_current / N_t)
    if RunCount >= 2 then
        Q_{t+1} = READQTABLE(N_t; x_t; x_{t-1})
        n = GETSTATEVISITS(N_t; x_t; x_{t-1})
        Q_t = READQTABLE(N_{t-1}; x_{t-1}; x_{t-2})
        R = COMPUTER(N_t; GETBLOCKING(interval))
        R' = R + γ Q_{t+1}
        δ = R' − Q_t
        // update Q-value
        Q(S^(t); A^(t)) ← (n/(n−1)) Q_t + (1/n)δ
        // update state visits
        N(S^(t); A^(t)) ← n + 1
    end
    if n < N_RefVisits then
        ε ← 1 − (n/N_RefVisits)
    else
        ε ← ε_min
    end
    ArrWFE = 0_{1000ε × 1000}
    ArrGRD = 1_{1000(1−ε) × 1000(1−ε)}
    ArrALL = CONCATENATE(ArrWFE; ArrGRD)
    if ArrALL [RANDOMINT()] == 0 then
        Scale with weighted fair exploration
    else
        Scale according to
          max(READQTABLE(N_t; x_t; x_{t-1}))
    end
    N_{t-1} ← N_t
    x_{t-2} ← x_{t-1}
    x_{t-1} ← x_t
    RunCount ← RunCount + 1
end
```

Fig. 7. Modified error function to estimate the blocking probability component of the initial Q values of card "0" (Fig. 5) diagonals.

Fig. 8. SQLR's horizontal scaling mechanism. We compare the Q-values of the grey-shaded cells in order to determine the best action according to (6). Here, we choose a scale-out of +1 VM. After the scaling action a, the Q-value in the red cell receives the update as specified in (4). One component of the update is the Q-value contained in the green cell of card "0" in bubble "$\{K+1\}$".

act according to Algorithm 1. As the scaling agent adds or removes VMs from the host, we monitor the blocking rates experienced and the number of running VMs, and generate rewards to update the table of Q-values related to the scaling agent's decision.

We check the relevant table every 120 seconds, which constitutes one epoch, and immediately call for a scaling decision whose action is selected according to (5) and (6). When the number of visits of a state reaches the prescribed count level M, then $\epsilon = \epsilon_{min}$ and the policy for that state has converged.

Fig. 9. Testbed setup. (1) Dell T640 server: Hosts KVM hypervisor, VMs, Admission controllers and Scaling Agent. (2) Client PCs: Generate requests towards the server according to demand profile. (3) Gbps switch: Creates LAN between Clients and Server.

V. EXPERIMENTS

A. Testbed

In order to evaluate the effectiveness of our scheme, we run experiments on a testbed that mirrors the operations of a CSP. We set up the testbed as shown in Fig. 9. The architecture of our Dell T640 server consists of two processor sockets with non-uniform memory allocation (NUMA), 10 hyper-threaded CPU cores per socket for a total of 40 logical cores with a variable clock rate. The server memory is 128 GB.

The server runs Ubuntu 18.04 LTS as its operating system and acts as a host for VMs. The client PCs run on Ubuntu 16.04.3 LTS. We use the KVM hypervisor, and manage the VMs using libvirt [34]. Each instance of a VM is configured with 4 virtual CPUs and 4 GB of memory. The client PCs and the server are connected via a Cisco switch to form a Gigabit/s local area network. The PCs function as ASPs running bash scripts that generate requests to the server with varying rates as depicted in Figs. 10 and 11.

As our cloud application, we choose the algorithm used for proof-of-work computation in bitcoin mining [35]. It is a suitable stand-in for resource-hungry, computationally challenging tasks that are commonly deferred to the cloud such as encryption [36] and transcoding [37], [38]. Each iteration of this computation involves incrementing a counter variable (nonce), hashing it together with a given hash code and merkle root, and then hashing the outcome again. The hashing mechanism is the 256-bit Secure Hash Algorithm (SHA-256). We will use the word *job* to refer to one proof-of-work iteration from hereon. In order to mimic the varying degrees of complexity of typical cloud applications, we consider a different number of iterations for each request. Specifically, a request can generate any number of iterations in the discrete set $\{300k, 400k, \ldots, 1200k\}$.

The server launches VMs to handle incoming requests according to one of the following schemes: static provisioning, extended Kalman filtering based prediction [19], RLPAS [17]

and our proposed scaling scheme. All agents, including the admission control and load balancer, are implemented in Python and run within the host operating system. For reproducibility, we fully share SQLR's code[3].

As mentioned in Section II, the scheme in [19] leverages a queuing system model enhanced with an EKF. It makes near time predictions of response times based on measurements of arrival rates and system utilization. Using a queue model refined by a tuned EKF with the maximum allowable response time (from an SLA) as input, the scheme then calculates the number of nodes needed and scales appropriately to approach this number.

We make some slight modifications to the EKF algorithm to make it more robust. We increase the interval between the predict and update phases from 10s to 90s. This provides sufficient time for starting up a VM and letting it handle tasks. Additionally, instead of the instantaneous measured system utilization and response times, we provide their average over the predict and update intervals of the filter as input to the EKF. This prevents the scaler from over/under estimating input parameters, and thus yields a fairer comparison to our scheme. Further, we dispense with the network delay in the system model as the response times are taken directly on the server. We consider a single-tiered application, and one class of requests. This also has the effect of simplifying the process and measurement noise covariance matrices to have size 2×2 (as only two parameters are taken into account in each case), thereby enhancing the tuning of the EKF.

We also compare our scaling system to the state-of-the-art RLPAS proposed in [17]. We only consider the response time parameter in our implementation and not throughput, since our stand-in cloud application is compute-intensive. Owing to the use of the load balancer, which distributes the offered load evenly, we set the ratio of utilized to provisioned VMs to 1.

For our scheme, we limit the number of VMs that can be added or removed within a single scaling action to 2. This truncates the action space, reducing the number of visits required for a state to achieve a stable policy $M_o = 50$ (cf. Table III). Therefore, it also limits the number of learning steps needed to attain a stable policy.

As part of the training for our scheme, we combine several workload profiles with different averages, resulting in the composite shown in Fig. 10.

Fig. 10. Pre-training workload profile. The red line is the moving average of the number of requests per minute, computed over windows of 30 samples.

For the test workload, we again use a combination of several profiles with different averages to obtain the composite shown in Fig. 11. To achieve this, we configure requests to be sent with inter-arrival times ω drawn uniformly at random from the set of values $\{0, 1, \ldots, \omega_{max}\}$ seconds, for each hour slot. For example for the busy-hour slot $\omega_{max} = 5$ s and the for low workload period $\omega_{max} = 9$ s. This results in high entropy (given the uniform distribution of inter-arrival times) but still allows us to procure similarities between workloads, and evaluate contextual knowledge re-use. Our choice of inter-arrival statistics leads to patterns encountered in real workloads, with rapid variations over short intervals, but with veritable trends over longer observation windows. It also includes sudden bursts and drops, such as those observed at the start of hours 8, 10, 14 and 18.

Fig. 11. Test workload profile. The red line is the moving average of the number of requests per minute, computed over windows of 30 samples.

B. Implementation on large scale

Although our experiments took place on a small testbed, our scheme still lends itself well to large-scale deployments, e.g., in data centers. The latter can be achieved via the modular approach presented in [38]. A conventional layer-4 load balancer such as [39] can be used to route tasks to physical servers as shown in Fig. 12. Given the cost benefits of operating homogeneous hardware in large scale settings [40], most servers in a data center will have the same specifications. This means that, in most cases, the scaling policies learned for one server can be re-used with no need for retraining.

VI. Results

In this section, we show the effectiveness of the AC and scaling policies. We then examine the results with respect to two SLOs: service availability (as measured via blocking rates) and response times.

A. Admission control policy convergence

First, we briefly discuss our AC agent. We recall that this component learns the appropriate utilization limit x_{lim}, that ensures bounded response times.

Fig. 13 shows how the learning algorithm for the AC trades off exploration and exploitation using our weighted

[3]https://github.com/Constantine-Ayimba/SQLR

Fig. 12. Schematic of a modular large-scale deployment. (From [38].)

fair exploration scheme, cf. Eq. (6). The evolution of the accumulated reward for a subset of three state-action pairs is shown in Fig. 13a. Red, blue and teal-colored lines denote the Q-value evolution for low, intermediate, and high utilization levels, respectively. Dashed lines refer to Q-values for drop decisions, wheres solid lines refer to admit decisions. In the initial learning phases, the difference between the values is not as distinct, and the admission control agent makes a DROP or ADMIT decision with about the same probability. After 1000 learning steps, the agent has understood which decisions yield better rewards (or at least lower penalties). For example, in high utilization regimes (teal lines), DROP decisions (dashed line) have a much higher Q-value, and are thus much more likely than ADMIT decisions (solid line). Conversely, at low utilization (red lines), ADMIT decisions are much more likely. At intermediate utilization, the difference between the Q-values of ADMIT and DROP decisions is not as stark, but still associated with a DROP decision.

The above results suggest that utilization levels up to 0.45 result in ADMIT decisions, whereas levels exceeding 0.45 start making DROP decisions more convenient. In other words, the scaling agent learns the limiting value of utilization, x_{lim}, to be 45%. Therefore, once the LB has chosen a VM that should serve an incoming task, the AC agent drops the task if the VM's utilization is higher than this learned value of x_{lim}, and accepts the requests otherwise.

B. Scaling agent's policy convergence and complexity

In order to characterize the overall state of convergence of the scaling agent, we consider the probability of randomness in action selection, . Recall that, for each state, we decrease from 1 down to 0 linearly with the number of visits to that state. Therefore, we use $1 - \bar{}$ to express the convergence level, where $\bar{}$ is the average value of computed across all states.

As shown in Fig. 14a, in the initial stages, e.g., after one full cycle of the test workload in Fig. 11 (green curve), is high in every state, such that its distribution has a mean $= 0.97$, corresponding to approximately 3% convergence.

(a) Reward accumulated with experience.

(b) Frequency of DROP and ADMIT decisions.

Fig. 13. Admission Control training. Red curves: low utilization level between 30% and 45%. Blue curves: intermediate utilization levels between 45% and 53%. Cyan curves: high utilization levels of 60% and above. Dashed lines convey the Q-values of DROP decisions, solid lines of ADMIT decision.

The scaling agent progressively gains greater experience about the workload profile and the corresponding system states. Thanks to this, the agent develops the appropriate scaling policy for each state, and acts less randomly. This yields diminishing values of in the related states and its distribution shifts leftwards and upwards, such that $= 0.065$ at the 30th cycle (brown curve) in Fig. 14a, which corresponds to approximately 93.5% convergence.

Over subsequent runs, the weighted fair exploration mechanism drives the scaling agent to visit the most pertinent states more often, as they procure better rewards. These states correlate more strongly to the underlying workload profile. At advanced levels of convergence, with low values of , the scaling agent chooses actions promising higher rewards, resulting in more visits to familiar states with fully converged policies. This is shown in Fig. 14b, where those states for which the policy has converged (yellow bars) are visited most likely as expected. However, weighted fair exploration still ensures a few visits to less familiar states (blue and green bars), and guarantees that the agent will be able to learn different policies, should it observe different workload patterns in the future.

For a reinforcement learning agent, the ultimate aim is take actions that maximize the accumulated reward; or minimize the penalties (negative rewards). Fig. 15 shows the sum of all the Q-values corresponding to any action, computed at different snapshots. For an increasing number of learning

(a) Convergence inferred from the distribution of the randomness factor. The rate is faster in the initial cycles as the weighted fair exploration drives the agent to the most promising states more often.

(b) State visitation probability with convergence at approx 93.5% ($=$ 0:065). The scaling agent acts more greedily, exploiting actions in the converged states.

Fig. 14. Scaling agent convergence behavior.

Fig. 15. Cumulative Q-Values, i.e., the sum of Q-values for all states, taken at different snapshots in the course of the experiment.

Fig. 16. Markov chain of possible actions from selected states. The numbers in curly brackets within each bubble, $\{i\}$, point to the number of VMs. Right-pointing arrows from a state indicate scale-out, whereas left-pointing arrows indicate scale-in. The re-entrant arrows above each bubble indicate no scaling.

steps, the scaling agent approaches convergence, and accrues progressively lower penalties. As depicted in Fig. 15, policy convergence occurs rapidly within the first 5000 epochs. This is because the parts of the workload with similar patterns influence the scaling agent to visit some states more often. The rate then slows as the scaling agent should observe less frequent workload patterns repeatedly in order to decide on the best policy. After about 30k epochs, most pertinent states have fully converged. The negative values are expected because of the way we structured the reward function in (13): this function issues penalties commensurate to the number of VMs provisioned in excess of the first one.

Complexity—We first consider complexity in terms of the number of learning steps required to attain convergence. Recall the representation of utilization in the agent's state space as depicted in the "cards" of Figs. 6 and 8. Each cell of a grid requires M updates (the number of visits until $_{min}$) for convergence. Call S the utilization component of the agent's state space and define $:= |S|$. Define also the total number of actions from all states as $:= \sum_{i \in n} |A(i)|$ where n is the maximum number of VMs available to an ASP, and $A(i)$ is the total number of actions available to the scaling agent when i VMs are active. In the worst case, the maximum number of steps required to reach convergence is at most $O(M\ )$ steps. Given that $< M <\ ^2$ by design, our complexity is between $O(\ ^2)$ and $O(\ ^3)$ which aligns with the classical analysis presented in [41].

In our implementation, for the worst-case complexity we have that the total number of VMs is $n = 10$, and the number of permissible actions $A(i)$ depends both on the number of active VMs i and on n. We clarify this through the Markov chain representation in Fig. 16, which shows all possible transitions between different numbers of VMs, denoted as the value $\{i\}$ inside each bubble, $1 \leqslant i \leqslant n$. For example, when 1 VMs is active, the agent can decide to keep 1 VM or rather scale out to 2 or 3 VMs. Instead, when 5 VMs are active, the agent can remove or add up to $N_{\ } = N_{\ } = 2$ VMs, or keep the current 5 VMs. Therefore, the admissible actions are 3 (for states having 1 and 10 VMs), 4 (for states having 2 and 9 VMs) and 5 for the rest. Therefore $= 2\ 3 + 2\ 4 + 6\ 5 = 44$. The utilization component of the state space comprises 10 quantized levels (from 0% to 20%), 5 levels (from 20% to 45%) and 1 level (beyond 45%). Therefore, $= 16\ 16 = 256$. Because we set $M = 10|A(i)|$, $1 \leqslant i \leqslant n$, the complexity of our implementation is $O(10\ 44\ 256) = O(112640)$ steps. We ameliorate this worst-case complexity by implementing weighted fair exploration (cf. III-F) and initializing the Q-values of the diagonal elements (cf. Section IV-E).

Furthermore, the operational complexity of the scaling agent presented in Algorithm 1 is $O(1)$ for all operations except for the load summation loop, which is $O(i_t)$, where i_t is the number of active VMs at epoch t.

C. Scaling profiles

We now move to discussing scaling profiles in response to the test workload of Fig. 11. We do so for all schemes considered in this work, namely SQLR, RLPAS [17], static provisioning, and the EKF-based scheme [19]. The latter produces the profile depicted in Fig. 17. The scaling behavior

Fig. 17. VM scaling for the EKF-based horizontal scaling scheme proposed in [19]. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area.

Fig. 18. VM Scaling for RLPAS: the Q-Learning horizontal scaling scheme proposed in [17]. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area.

in this scheme is quite stiff, because the EKF tends to filter out bursty workload, that would require greater agility instead.

The RLPAS scaling profile is depicted in Fig. 18. RLPAS is quite agile compared to the EKF-based scaler. However, it is susceptible to premature scaling decisions. In fact, it relies on the instantaneous workload arrival rate, which is highly stochastic in the test workload profile. Moreover, both the EKF-based scaler and RLPAS require some knowledge of the underlying application, such as its ideal response time. From empirical observations on our test application, we set such ideal response time at 5 s per job for both schemes, as this amount of time is amply sufficient to serve the greatest majority of the jobs.

The scaling profile obtained from our proposed scheme, in reference to the test workload, is shown in Fig. 19. The behavior of the scaler steadily improves with increased exposure to the test workload. As more states converge, the scaling behavior becomes more predictable, as seen by moving from Fig. 19a to Fig. 19b and Fig. 19c. The number of VMs provisioned settles around a suitable number that achieves the best tradeoff between resource cost and penalties as driven through the training parameters and .

Moreover, in Fig. 19b, we see that the first intervals to exhibit convergence (hence greater stability in the scaling behavior) are those with higher similarity to the training workload of Fig. 10. For instance the intervals of hours 6 to 8 and 16 to 18, with an average of 40 requests per minute (cf. Fig. 11), closely resemble those of hours 1 to 4 and 20 to 23 of the training workload (cf. Fig. 10). This shows that SQLR can re-use contextual knowledge learned from one workload on any subsequent one with similar characteristics.

(a) After 10 cycles, at 59% convergence ($= 0.41$). With $= 1$, $= 0.01$. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area.

(b) After 20 cycles, at 89% convergence ($= 0.11$). With $= 1$, $= 0.01$. profile. We represent resource savings with respect to static over-provisioning with 10 VMs via the gray-shaded area.

(c) After 30 cycles, at 94.5% convergence ($= 0.065$). The lighter shade of gray area represents resource savings made compared to static over-provisioning with 10 VMs for Case 2. The darker shade of gray represents extra resource savings made in Case 1 compared to Case 2.

Fig. 19. SQLR scaling behavior evolving with experience. For this training phase, we set $P_{blk} = 0.001$.

Assigning different values to the training parameters and results in different scaling responses. As shown in Fig. 19c, a low value of relative to (Case 1) results in cost-focused scaling policies that emphasize resource cost more than service unavailability due to blocking. This is the same configuration as in Figs. 19a and 19b. When , as in Case 2, more service-focused policies are learned, giving greater importance

Fig. 20. Blocking rates over two-minute intervals. Two SQLR configurations are shown: Case 1 (= 1; = 0.01) and Case 2 (= 10; = 0.001). For clarity, a moving average filter is applied with a window size of 30 samples.

Fig. 21. Blocking rate distribution. Two SQLR configurations are shown: Case 1 (= 1; = 0.01) and Case 2 (= 10; = 0.001).

to service availability than to resource cost.

D. Blocking rates

The exploration mechanism of the Q-Learning algorithm at the core of SQLR means that it may sometimes make sub-optimal decisions in less known states, resulting in under-provisioning (such as at hour 18 for case 1, and at hour 5 for case 2 in Fig. 19c). This results in relatively high blocking rates, as shown in Fig. 20. Our guided fair exploration mechanism ameliorates the effects of such under-provisioning, ensuring that their duration is short.

Since the EKF-based scaler relies on workload measurements to predict response times and scale accordingly, it is particularly susceptible to under-estimating resource requirements when demand is low. This is evident at off-peak intervals in Fig. 20 where, between hours 0 and 7 and between hours 17 and 24, the EKF-based scaler allocates 1 VM on average, resulting in considerable blocking, much higher than the other schemes. The RLPAS scaler adjusts resources too abruptly, resulting in unpredictable blocking at both peak and off-peak hours. This is due to its dependence on direct workload measurements, which are highly stochastic.

Static provisioning results in significant under or over-provisioning, as exhibited by the black (2 VMs) and blue (10 VMs) curves, respectively. Both situations are clearly untenable: on the one hand, the CSP risks serious penalties for service unavailability; on the other hand, the CSP incurs significant yet unnecessary operational expenditure to maintain superfluous resources, even though over-provisioning results in zero blocking.

The distribution of blocking rates is shown in Fig. 21. The greater the number of VMs provisioned, the lower the blocking rate, as an incoming request will likely find a sufficiently under-utilized VM. Given the preceding insight, the EKF-based scaler that provisions the lowest number of VMs, exhibits poor blocking rate performance. RLPAS also performs poorly because of its premature scaling behavior, which occasionally leads to VM removals too soon when workload transients occur. Consider instead Case 2 of our scaling scheme. This configuration penalizes blocking more heavily than provisioning extra VMs by setting in Eq. (13). Hence, it performs nearly as well as over-provisioning with 10 VMs. If lower service availability is acceptable, our scheme can prioritize resource cost over blocking, but still achieve reasonably low blocking rates (Case 1).

E. Service times

The distribution of service times is shown in Fig. 22. We obtain the service time per job by dividing the service time of each request by the corresponding number of proof-of-work iterations it generates. These response times include the administrative overhead that the hypervisor incurs to switch between the host and guest while managing VMs. It also includes context switching between user mode and kernel mode of the corresponding operating systems.

This overhead increases with the number of VMs being administered as well as with how often they switch context. Dynamic scaling, which entails starting up and shutting down VMs, exacerbates the latter. The combination of these factors affects SQLR's case 2 greatly, as it incurs higher penalties for blocking than resource usage. This is because the policies of SQLR's case 2 implicitly employ more VMs. Therefore, SQLR's case 2 closely follows the static over-provisioned case with 10 VMs, that incurs high administrative overhead throughout.

However, the over-provisioned scenario still provides the ideal case with the lowest-variance (highly predictable) service times. Both configurations of our scheme closely approach this ideal case with about 96% of the requests being served within 5 s per job, compared to 95.5% for the over-provisioned case, 86.5% for the EKF case and 87.7% for RLPAS.

Moreover, for SQLR's Case 2 (= 10, = 0.001), the improvement in the proportion of responses within the cutoff service time of 5 s is only marginal, compared to the more cost-focused Case 1 (= 1, = 0.01). This is despite the extra amount of resources deployed in Case 2, and is primarily due to the additional administrative overheads incurred.

In order to compare the scaling schemes without the biasing effect of the administrative overhead, we carry out a process akin to noise filtering in communication systems. We do this by first obtaining the average service times over two-minute intervals, and then applying a moving average filter having a

Fig. 22. Service time distribution per job. Two SQLR configurations are shown: Case 1 ($= 1$; $= 0.01$) and Case 2 ($= 10$; $= 0.001$). The service time for each request is divided by the corresponding number of iterations it generates to obtain the time per job.

Fig. 23. Moving averages of service times (taken over a window of 30 samples to smooth out switching overheads). Two SQLR configurations are shown: Case 1 ($= 1$; $= 0.01$) and Case 2 ($= 10$; $= 0.001$).

window of 30 samples. Since context switching happens in the order of clock cycles, these two operations over intervals that are orders of magnitude longer than a clock cycle spread the cost of switching overhead over time and smooth the curves.

When we apply the operations stated above to the service time per job, we obtain the results depicted in Fig. 23. Both SQLR configurations closely follow the over-provisioned policy with the ideal response times. At low workload (hours 0-7 and 17-24), the administrative overhead to maintain a large number of VMs outweighs the gain of better service times resulting from the use of extra resources. Over these intervals, our scheme performs slightly better than the 10-VM case by provisioning fewer VMs. Conversely, the EKF-based scaler still under-performs: the single VM it provisions over these intervals is not sufficient to meet the demand within the cut-off service time. The RLPAS scaler, owing to its abrupt scaling behavior, exhibits response times that oscillate about those of the more stable EKF scaler.

This marked difference in response times, owing to differences in the scaling mechanisms, is clearly depicted in Figs. 24 to 27, where we compare the *soft blocking* performance (the proportion of admitted requests whose service times extend beyond our cut-off of 5 s per job). In these heatmaps, we consider only regions with statistical significance (30 or more responses). Moreover, the white region indicates resource allocation choices that remain unexplored for the considered input load. For the severity heatmaps of Figs. 24b, 25b, 26b, and 27b, the white region indicates those allocations leading to service times within the limit of 5 s. The offered load values on the y-axis indicate the upper bound with the value immediately below indicating the lower bound, e.g., "< 20" indicates the interval [10,20) requests per minute. Therefore, in panels (a) of Figs. 24–27, the best behavior is shown as yellow hues, as opposed to unwanted behavior (blue hues). Moreover, a larger number of yellow-colored cells denotes greater scaling agility through appropriate system states. Conversely, panels (b) of Figs. 24–27 convey best scaling behavior both through yellow hues and through the presence of a greater number of white cells.

As depicted in Fig. 24, the EKF scaler employed by [19] is prone to overruns even under light loads, given that it is very conservative in allocating extra VMs. As a result, this increases the strain on the few active ones. Fig. 25 shows that RLPAS, instead, is prone to more widespread overruns across all loads. This is because it prematurely scales in, even at high loads, after transients in workload measurements. In our scheme, whose responses are shown in Figs. 26 and 27, a significantly smaller proportion of service times exceed the cut-off time (particularly at moderate to high offered loads). In fact, our scheme is more sensitive to abrupt workload changes, and assigns resources in a more agile fashion compared to the EKF-based scheme. The effect of the EKF is to quench scaling decisions, especially in the presence of short-lived workload bursts.

Further, comparing the two configurations of our scheme shown in Figs. 26 and 27, the provisioning policies of Case 2 result in fewer instances of soft blocking than Case 1. This is because Case 2 provisions more VMs on average, which increases the likelihood of operating them at lower CPU loads. This allows more task admissions and lower service times. High severity (particularly in Case 2) comes from exploratory actions at high demand, whereby our scaler momentarily scales in. However, by evaluating the sub-optimality of these actions, our weighted fair guided exploration quickly scales out, as is evident around hours 10 and 12 in Fig. 19c.

F. CPU utilization

The average CPU utilization over the duration of the experiments is shown in Fig. 28. As before, we apply a moving average filter with a window of 10 samples to each curve. SQLR Case 1, SQLR Case 2, and the over-provisioned case with 10 VMs result in average utilization levels below 20%. The EKF scaler and the under-provisioned case result in utilization levels above 25% during the peak period. The RLPAS scaler results in a few excursions into utilization levels above 25%. Note that no curve passes 45% utilization in Fig. 28, as the AC learned not to admit tasks beyond this limit regardless of the employed scaling scheme.

(a) (b)

Fig. 24. Soft Blocking Probability for the EKF scaler. (a) Frequency of blocking: the number of responses exceeding 5 s as a fraction of the total number of responses. (b) Severity of blocking: the mean deviation from 5 s of the responses exceeding 5 s. 86.5% of the responses are within 5 s per job.

(a) (b)

Fig. 25. Soft Blocking Probability for the RLPAS Scaler. (a) Frequency of blocking: the number of responses exceeding 5 s as a fraction of the total number of responses. (b) Severity of blocking: the mean deviation from 5 s of the responses exceeding 5 s. 87.7% of the responses are within 5 s per job.

(a) (b)

Fig. 26. Soft Blocking Probability for SQLR's Case 1 (= 1, = 0.01). (a) Frequency of blocking: the number of responses exceeding 5 s as a fraction of the total number of responses. (b) Severity of blocking: the mean deviation from 5 s of the responses exceeding 5 s. 95.6% of the responses are within 5 s per job. The instances of high severity are mainly due to exploratory scale-in actions in states where x_{min} .

(a) (b)

Fig. 27. Soft Blocking Probability for SQLR's Case 2 (= 10, = 0.001). (a) Frequency of blocking: the number of responses exceeding 5 s as a fraction of the total number of responses. (b) Severity of blocking: the mean deviation from 5 s of the responses exceeding 5 s. 96.2% of the responses are within 5 s per job. The instances of high severity are mainly due to exploratory scale-in actions in states where x_{min} .

TABLE IV
SUMMARY OF RESULTS

Scheme	**Resources saved**	**Requests served with $<5\%$ blocking**	**Requests served with $<5\,\mu s$ per job**
Static 10 VMs	0.00%	100%	95.47%
Static 2 VMs	80.00%	64.86%	84.89%
SQLR Case 1	55.13%	91.50%	95.60%
SQLR Case 2	20.54%	99.17%	96.17%
EKF [19]	80.15%	65.83%	86.47%
RLPAS [17]	67.30%	71.53%	87.69%

The utilization trends closely follow the service times shown in Fig. 23, emphasizing the high correlation between these metrics. This confirms the suitability of CPU utilization as a metric to define the state of both the AC and the scaling RL.[4]

G. Summary of results

We summarize a comparison of the performance of the scaling schemes in Table IV. Here, we take the static over-provisioned case (with 10 VMs) as a reference benchmark with 0% blocking. We measure resources in terms of VM-hours. The tradeoff between resource use and service availability is apparent: schemes that procure lower blocking rates use up more resources to do so. Both configurations of our scheme achieve service times comparable to the over-provisioned benchmark with 10 VMs.

VII. CONCLUSIONS

We have presented an agile horizontal scaling system, SQLR, that learns the most appropriate horizontal scaling decision to make under highly dynamic workloads, and without any fore-knowledge of the underlying system configuration. We show that our modified Q-learning scheme enables our system to learn multiple policies and re-use any applicable knowledge to new workload profiles exhibiting previously encountered characteristics.

SQLR progressively optimizes its policy by tuning the tradeoff between resource cost and service availability. These constraints come from the CSP after proper determination from their business processes. Such high-level objectives make SQLR easily configurable and adaptable to any cloud application, as no domain-specific knowledge is required. We compare our proposed scheme to different state-of-the-art scaling systems, and show that our scheme achieves better performance, similar to that of an over-provisioned system.

As with most machine learning-based schemes, our scheme is subject to a training overhead. However, because of its capacity for contextual knowledge re-use, it can be trained offline with representative workloads. Also, given our weighted fair exploration mechanism, any subsequent residual learning can be done in production workloads with a much reduced risk of poor decisions in the process. We show that even prior to full convergence, our scheme performs practically as well as the unconstrained resource benchmark (static over-provisioning).

[4]Conversely, system memory allocation is not an expressive metric: in all of our experiments, all scaling agents allocate about 360 MBytes of memory, with negligible oscillations around this value.

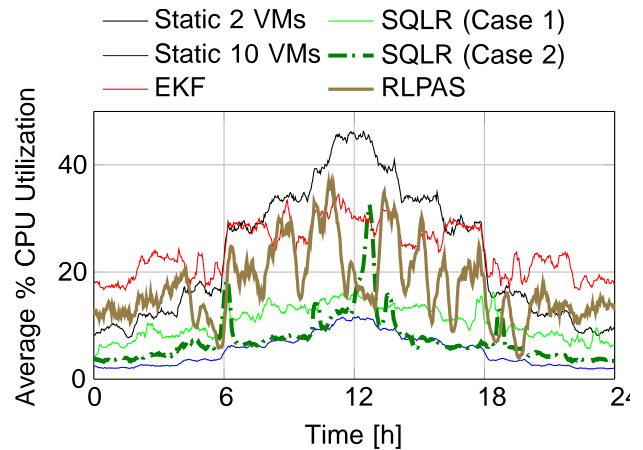

Fig. 28. Moving averages of CPU utilization (taken over a window of 10 samples to smooth out switching overheads). Two SQLR configurations are shown: Case 1 (= 1; = 0:01) and Case 2 (= 10; = 0:001).

REFERENCES

[1] K. Costello and M. Rimol, "Gartner forecasts worldwide public cloud revenue to grow 17% in 2020," Nov. 2019, accessed: 2020-12-03. [Online]. Available: https://tinyurl.com/GartnerCloudRev2020

[2] J. Anselmi, D. Ardagna, J. C. S. Lui, A. Wierman, Y. Xu, and Z. Yang, "The economics of the cloud," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 2, no. 4, pp. 18:1–18:23, Aug. 2017.

[3] C. Wang, B. Urgaonkar, G. Kesidis, A. Gupta, L. Y. Chen, and R. Birke, "Effective capacity modulation as an explicit control knob for public cloud profitability," *ACM Trans. Auton. Adapt. Syst.*, vol. 13, no. 1, pp. 2:1–2:25, May 2018.

[4] P. Cong, L. Li, J. Zhou, K. Cao, T. Wei, M. Chen, and S. Hu, "Developing User Perceived Value Based Pricing Models for Cloud Markets," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2742–2756, Dec. 2018.

[5] T. Chen, R. Bahsoon, and X. Yao, "A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 61:1–61:40, Jun. 2018.

[6] X.-L. Huang, X. Ma, and F. Hu, "Editorial: Machine Learning and Intelligent Communications," *Mobile Networks and Applications*, vol. 23, pp. 68–70, Feb. 2018.

[7] T. Young, "Deployment challenges in multi-access edge computing (MEC)," 2020, accessed: 2020-12-21. [Online]. Available: https://www.a10networks.com/blog/deployment-challenges-in-multi-access-edge-computing-mec/

[8] Fortinet, "Multi-access edge computing (MEC) and the edge cloud," 2020, accessed: 2020-12-21. [Online]. Available: https://www.fortinet.com/fr/solutions/mobile-carrier/securing-5g-innovation/mobile-edge-computing

[9] "Rightscale," https://www.rightscale.com, accessed: 2020-12-23.

[10] "Amazon EC2," https://aws.amazon.com/ec2, accessed: 2020-12-23.

[11] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth, "Coordinating CPU and memory elasticity controllers to meet service response time constraints," in *Proc. ICCAC*, Boston, MA, Sep. 2015, pp. 69–80.

[12] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Proc. IEEE/IFIP NOMS*, Maui, HI, Apr. 2012, pp. 1327–1334.

[13] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic resource provisioning for cloud-based software," in *Proc. SEAMS*, Hyderabad, India, Jun. 2014, pp. 95–104.

[14] S. Tu, M. Waqas, S. U. Rehman, T. Mir, G. Abbas, Z. H. Abbas, Z. Halim, and I. Ahmad, "Reinforcement learning assisted impersonation attack detection in device-to-device communications," *IEEE Transactions on Vehicular Technology*, vol. 70, no. 2, pp. 1474–1479, 2021.

[15] B. He, J. Wang, Q. Qi, H. Sun, and J. Liao, "Towards intelligent provisioning of virtualized network functions in cloud of things: A deep reinforcement learning based approach," *IEEE Trans. on Cloud Comput.*, 2020, in press.

[16] A. Alsarhan, A. Itradat, A. Y. Al-Dubai, A. Y. Zomaya, and G. Min, "Adaptive resource allocation and provisioning in multi-service cloud environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 31–42, Jan. 2018.