Unsupervised Learning of Dynamic Resource Provisioning Policies for Cloud-Hosted Multitier Web Applications

Waheed Iqbal, Mathew N. Dailey, and David Carrera

***Abstract*—Dynamic resource provisioning for Web applications allows for low operational costs while meeting service-level objectives (SLOs). However, the complexity of multitier Web applications makes it difficult to automatically provision resources for each tier without human supervision. In this paper, we introduce unsupervised machine learning methods to dynamically provision multitier Web applications, while observing user-defined performance goals. The proposed technique operates in real time and uses learning techniques to identify workload patterns from access logs, reactively identifies bottlenecks for specific workload patterns, and dynamically builds resource allocation policies for each particular workload. We demonstrate the effectiveness of the proposed approach in several experiments using synthetic workloads on the Amazon Elastic Compute Cloud (EC2) and compare it with industry-standard rule-based autoscale strategies. Our results show that the proposed techniques would enable cloud infrastructure providers or application owners to build systems that automatically manage multitier Web applications, while meeting SLOs, without any prior knowledge of the applications' resource utilization or workload patterns.**

***Index Terms*—Cloud computing, multi-tier applications, resource management, scalability, service-level agreement (SLA), system performance.**

I. Introduction

CLOUD computing is attractive to Web service owners because it empowers them to provide highly available and manageable applications at low cost. The dynamic resource provisioning capabilities of cloud infrastructures further enables Web application owners to scale their applications on the fly with low operational cost. A variety of criteria could potentially be used to measure the performance of a dynamic resource provisioning policy. From the Web application user's point of view, however, *response time* is the most important quality attribute of a Web application, yet current service-level agreements (SLAs) offered by cloud infrastructure providers do not address response time.

One of the typical architectures for cloud-hosted applications is the multitier Web application consisting of at least a presentation tier, a business logic tier, and a data management tier running as separate processes. Multitier Web applications hosted on a specific fixed infrastructure can only service a limited number of requests concurrently before some bottleneck occurs. Once a bottleneck occurs, if the arrival rate does not decrease, the application will saturate, service time will grow dramatically, and eventually, requests will fail entirely.

It is important for Web applications to service all requests reliably and minimize service time in order to be useful to their end users. Cloud providers can offer dynamic resource provisioning [1], [2] and autoscaling [3] to maintain maximum service time guarantees, while minimizing resource utilization for a given workload. However, optimal proactive resource provisioning and scaling for a specific Web application require, at the least, a technique to automatically identify bottlenecks and scale the appropriate resource tier. For simple (one-tier) Web applications, it is possible to detect bottlenecks and achieve low operational costs by first minimizing resource allocation and then dynamically scaling allocated resources as needed to handle increased loads. However, for multitier Web applications, it is more difficult to automatically identify the exact location of a bottleneck and scale the appropriate resource tier accordingly. This is because multitier applications are complex, and bottleneck patterns may be dependent on the specific pattern of workload at any given time.

In principle, it is possible to identify bottlenecks for a specific application by monitoring and profiling the low-level hardware resource utilization in each tier under a variety of workloads. However, these fine-grained techniques have to deal with extra monitoring overhead, virtualization complexity, and end-user security concerns involved in installing monitoring agents on rented virtual machines. Web application owners usually cannot deploy low-level hardware profiling agents on the physical machines, and cloud providers generally do not have insight into the applications they are hosting. Techniques based on low-level hardware profiling that do not also monitor response time metrics would further be unable to identify performance issues

Manuscript received May 21, 2014; revised September 4, 2014 and March 3, 2015; accepted April 18, 2015. Date of publication May 21, 2015; date of current version November 22, 2016.

W. Iqbal was with the Asian Institute of Technology, Pathumthani 12120, Thailand. He is currently with the Punjab University College of Information Technology, University of the Punjab, Lahore 54890, Pakistan (e-mail: waheed.iqbal@pucit.edu.pk).

M. N. Dailey is with the Department of Computer Science and Information Management, Asian Institute of Technology, Pathumthani 12120, Thailand (e-mail: mdailey@ait.ac.th).

D. Carrera is with the Department of Computer Architecture and Barcelona Supercomputing Center, Universitat Politìcnica de Catalunya, 08034 Barcelona Spain (e-mail: dcarrera@ac.upc.edu).

Digital Object Identifier 10.1109/JSYST.2015.2424998

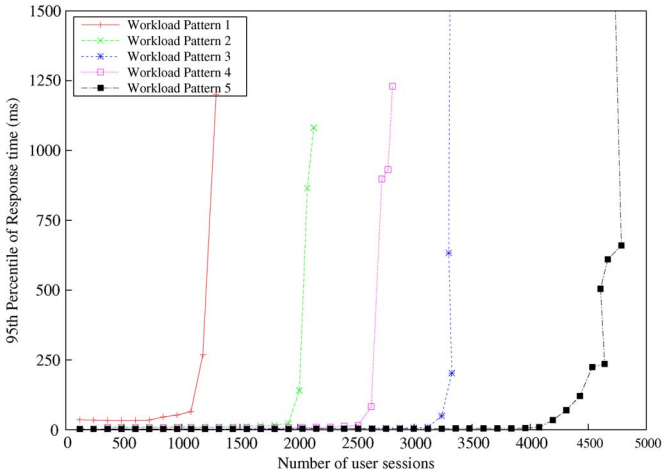

Fig. 1. Response time saturation of a benchmark Web application under different workload patterns.

created by software misconfiguration. One example is whether the number of connections from each server in the Web server tier to the database tier is appropriate.

A generic black box approach based on high-level information such as throughput and access logs without any instrumentation of the virtual machines would therefore be much more preferable. We advocate the use of coarse-grained monitoring techniques based on application access logs. This course-grained information coupled with methods for *learning* application- and workload-specific resource provisioning policies can enable cloud providers to automatically identify and resolve bottlenecks in the applications they are hosting.

Although learning bottleneck resolution policies from coarse-grained monitoring has potential benefits, one possible difficulty is that the optimal action to take when a bottleneck occurs might well depend on the nature of the workload, which can change rapidly over time. For example, the FIFA website [4] observed sudden traffic spikes during the soccer World Cup of 1998 [5]. Search engines observed sudden spikes on the death of Farrah Fawcett and Michael Jackson [6]. Online travel and booking sites exhibit different workload patterns at different times of the day and week [7].

As a simple example of the effect of workload patterns on application bottlenecks, consider the following, in which we model five arbitrary but reasonable workloads for a specific application and profile the system's behavior. Each workload contains ten different mixes of dynamic and static requests for the Rice University Bidding System (RUBiS) benchmark auction application [8]. Fig. 1 shows the service time saturation points for each of the five different workload patterns. We observe that the system performance varies from 1000 user sessions to 4500 user sessions, depending on the specific workload pattern. These results show clearly that the appropriate action to take to resolve a bottleneck might depend strongly on the current workload pattern.

In this paper, we present a method for learning appropriate application- and workload-specific resource provisioning policies. We develop a formal model for and techniques to identify the parameters of workload patterns for multitier Web applications using access logs and unsupervised machine learning. The method automatically identifies groups of uniform resource identifiers (URIs) with similar resource utilization characteristics from historical access log data. We also design and empirically evaluate a method for satisfying a service-level objective (SLO) that provides a maximum service time guarantee that works by reactively identifying bottlenecks for specific workload patterns and then learning resource allocation policies, all based on coarse-grained access log monitoring in real time. The policy learner initially uses a trial-and-error process to identify an appropriate bottleneck resolution policy in the context of a specific workload pattern and then exploits that policy to reduce violations of the SLO, while minimizing resource utilization. The approach does not require predeployment profiling or any insight about the application. We evaluate our proposed system on the Amazon public cloud and the RUBiS benchmark Web application. We also compare our proposed approach with industry-standard rule-based scale-out strategies (as explained in Section V-A) and with a baseline approach similar to that of Urgaonkar *et al.* [9], which scales up all replicable tiers whenever a bottleneck occurs.

Our results show that the proposed techniques would enable cloud infrastructure providers or application owners to build systems that automatically manage multitier Web applications, while meeting SLOs, without any prior knowledge of the applications' resource utilization or workload patterns.

The exact contributions of this paper are:

i) **Workload pattern identification**: we propose a formal model for workload patterns for multitier Web applications and an automated method for identifying the parameters of workload patterns online using access logs and unsupervised machine learning.
ii) **Resource provisioning policy learning**: we propose and evaluate an automated method for learning appropriate application- and workload-specific resource provisioning policies using reinforcement learning.

There are a few limitations to this work. First, although our policy learning method is equally applicable to any multitier architecture, we have only instrumented and empirically evaluated the method on one particular two-tier Web application installed on the Amazon Elastic Compute Cloud (EC2). We also assume that a sufficient bandwidth exists and that enough time is given up front to collect access logs from the application, in order to identify URIs with similar resource utilization characteristics.

The rest of this paper is organized as follows: Related work is discussed in Section II. The proposed workload model is discussed in Section III, and the policy learning method is discussed in Section V. Experimental details are provided in Section VI. A detailed experimental evaluation is provided in Section VI. Finally, conclusions are drawn in Section VII.

II. Related Work

There have been several efforts toward adaptive allocation of cloud resources to satisfy performance criteria. For example, Bodik *et al.* [2] present a statistical machine learning approach

to predict the system performance for a single-tier application and minimize the amount of resources required to maintain the performance of an application hosted on a cloud. Liu and Wee [10] monitor the CPU and bandwidth usage of virtual machines hosted on an Amazon EC2, identify the resource requirements of applications, and dynamically switch between different virtual machine configurations to satisfy the changing workloads. However, none of these solutions address the issues of multitier Web applications or database scalability, a crucial factor in dynamic management of multitier workloads.

There have also been several efforts to use machine learning to manage application resources dynamically. For example, Bu *et al.* [11] use a reinforcement learning approach to identify the best server configuration settings (e.g., maximum number of clients and maximum number of threads) to maximize the performance of their system. Tesauro *et al.* [12] present a reinforcement learning approach to automatic resource allocation for single-tier Web applications using offline initial policy learning. Bodik *et al.* [13] present an approach to learn a performance model using local regression (a nonlinear regression technique) for Web applications hosted on clouds. The learned model is then used to provision resources necessary to satisfy SLA requirements. However, none of these approaches consider the effects of different workload patterns.

Thus far, only a few researchers have begun to address the problem of resource provisioning for multitier applications. Urgaonkar *et al.* [14] present an analytical model using queuing networks to capture the behavior of each tier. Rao and Xu [15] present an online method for capacity identification [16] of multitier Web applications hosted on physical machines using hardware performance counters. Singh *et al.* [17] present a technique to model dynamic workloads for multitier Web applications, using k-means clustering of service times, based on logs collected at each tier. The method uses queuing theory to model the system's reaction to the workload and to identify the number of instances required for an Amazon EC2 to perform well under a given workload. Our method identifies workload patterns using clustering, but we do not separately monitor each tier of the Web application. Instead, we treat the whole multitier application as a black box. Our proposed approach is able to learn resource allocation policies in real time.

Dejung *et al.* [18] present a dynamic resource provisioning approach for multitier Web applications hosted on clouds, aiming to ensure homogeneous performance from every instance in each tier despite deployment in a heterogeneous environment. However, the authors do not consider different workload patterns.

Most of the work in dynamic resource provisioning only identifies changes in the application's workload volume, using this information to provision more resources to maintain application performance [2], [18], [19]. However, a few researchers have incorporated workload pattern modeling into their application's resource provisioning policies. Sharma *et al.* [20] present a machine learning-based method to automatically characterize Web application resources by measuring CPU usage, number of requests, and network utilization. Bodik *et al.* [21] present a workload model for sudden increases in volume and demand for objects in stateful systems.

Some of the recent work in dynamic scaling of Web applications uses rule-based and machine learning approaches. For example, Fernandez *et al.* [22] present a system to dynamically provision resources for a Web application hosted on a heterogeneous cloud. The main focus is to provide customers with different SLA level options (gold, silver, and bronze), with higher quality of service available at higher cost. The proposed system autoscales the Web application using rule-based techniques incorporating CPU utilization, arrival rate, throughput, and response time. However, the authors only evaluate the system on a single-tier Web application. Seracini *et al.* [25] present a resource provisioning system for Web applications based on queuing theory. The system profiles nonfunctional qualities of the Web application (response time, arrival rate, and throughput) as well as low-level infrastructural resources (CPU utilization, cache, and main memory) in deciding how to autoscale the application. However, the proposed system only scales out the application tier, i.e., other tiers are simply overprovisioned to prevent them from becoming bottlenecks. Gandhi *et al.* [24] use a combination of Kalman filtering and a queueing model to determine how to dynamically scale cloud-hosted applications using average response time, the request arrival rate, and CPU utilization of the virtual machines hosting the application. However, the authors do not address the issue of scaling multiple tiers of the application. Yazdanov and Fetzer *et al.* [25] present a reinforcement learning approach to vertically scale the CPU and memory of virtual machines hosting an application.

The work in this area most relevant to ours [26] proposes a cost-effective resource allocation approach to adaptively manage cloud resources to satisfy response time and availability guarantees. The authors profile the response time of each tier of the application to identify bottlenecks, but they do not consider workload patterns. Our approach is more coarse-grained; it only profiles the application's load balancing proxy traces in the context of specific workload patterns to learn bottleneck resolution policies in real time.

To our knowledge, there is no existing system for dynamic resource allocation using coarse-grained monitoring able to learn optimal resource allocation policies for multitier Web applications in real time. We take the first step in this direction with a method to identify workload patterns and learn optimal resource allocation policies for a given workload.

The research reported upon in this paper extends the work reported on in a previous conference paper [27], in which industry-standard rule-based resource provisioning methods (as explained in Section V-A) are used to explore cost–performance tradeoffs with Amazon EC2 compute resources. This paper proposes and empirically evaluates a new unsupervised machine learning method for dynamic provisioning of multitier Web applications under the constraints imposed by user-defined performance goals; the industry-standard rule-based resource provisioning methods are used as a baseline for comparison.

III. Workload Pattern Model

It is possible to design or learn a resource provisioning policy based on raw workload levels (request arrival rates)

alone, but we advocate for the need to incorporate not only the raw workload rate but also the workload pattern in a resource provisioning policy. From the sample data and discussion in Section I, clearly, identifying the specific types of resources used by an application at any given time would enable the specification of a more precise resource provisioning policy than would otherwise be possible. Here, we introduce a model for workload patterns, whose parameters can be automatically identified at runtime via observations of the system's performance under different conditions over time, and in Section IV, we show how to learn a resource provisioning policy that, besides handling the current raw workload level, also tailors provisioning actions to the current workload pattern. The model consists of two components:

i) **URI space partitioning**: a partitioning of the application's URI space into requests with similar resource utilization characteristics;
ii) **workload pattern**: a probabilistic model of request arrivals over the URI space.

We describe both of the components in the following subsections.

A. URI Space Partitioning

We assume that the URI space for a particular Web application can be partitioned into a set of k discrete *clusters* $\{c_1, \ldots, c_k\}$ with similar resource utilization characteristics. Within each cluster, we assume that the amount of any resource required (CPU time, network bandwidth, disk access, and so on) to service any particular request is random but follows an identical distribution for every distinct URI path in the URI space.

In the limit, in which k equals the number of distinct URI paths in the application's URI space, this is certainly a reasonable assumption. However, in practice, to make model identification tractable, we further assume that it is a reasonable approximation to fix k to a small number and thus map many URI paths to the same cluster.

The cost of violations of this assumption, i.e., having large clusters that contain URIs with dramatically different resource requirements, is that resources may be over- or underprovisioned when the actual workload consists of different patterns of URIs mapped to the same cluster but having different actual resource requirements. Increasing the number of clusters can reduce this cost, but if the number of clusters is too large, the neural network tasked with learning the effects of scaling operations over different workload patterns (as described in Section IV) would require a prohibitively large number of training examples before generalizing well, and this would in turn make the learning algorithm's exploration phase prohibitively long.

To identify the URI space partitioning model from observations, we require that it is possible to collect a sufficiently large set of historical access logs. For most applications, these historical logs should be collected for a fairly long period of time such as days or weeks. We collect the logs and preprocess them to extract the URI path, document size, and service time for each request. We use a Gaussian mixture model clustering algorithm [28] to group the requests into clusters based on document size and service time. We then construct a map from each URI path to the corresponding cluster ID. In cases where URI paths are mapped to multiple clusters, we use majority voting.

B. Workload Patterns

Our workload pattern model is probabilistic. In each independent trial of the random experiment, we wait for the arrival of a single hypertext transfer protocol (HTTP) request from a remote client and observe the cluster $c \in \{c_1, \ldots, c_k\}$ that the request URI path falls into. Let C be the random variable describing which of the k clusters a request falls into. A probability distribution $P(C)$ over the clusters defines a *workload distribution* for the Web application.

In our model, a *workload pattern* is simply a specific workload distribution P over random variable C. We assume that over short periods of time, the workload distribution is stationary; hence, we write the workload pattern at a specific time as the vector

$$P(C) = (P(c_1), P(c_2), \ldots, P(c_k)).$$

(For convenience, we abbreviate the event $C = c_i$ as simply c_i.)

Given the mapping from URI paths to cluster IDs, identifying the workload pattern for a specific interval of time is a simple matter of observing the frequency of arrival of requests for URI paths in each cluster.

IV. Policy Learning Method

Here, we develop an online unsupervised method for learning a policy for adaptive resource allocation to a multitier Web application based on the trial-and-error approach of reinforcement learning. Reinforcement learners are agents attempting to maximize their long-term reward by taking appropriate actions in an unknown environment. Our learning agent uses a simplistic method to find the policy (a mapping from application and workload state to resource allocation action) maximizing an objective function that encourages satisfying a response time SLO with minimal resources. We first define our learning agent and then give the detailed policy learning algorithm.

A. Model

The **system state** $s_t = (U, P(C), \lambda, p)$ at time t contains the configuration of the Web application's tiers U, the current workload pattern $P(C)$, the current arrival rate λ, and the current 95th percentile of the service time p. For an n-tier application, we define a configuration by the vector $U = (u_1, \ldots, u_n)$, where element u_i indicates the number of machines allocated to tier i.

The **action** a that the agent can select at any point in time is a particular scale-up strategy. For our benchmark two-tier Web application, the possible scale-up strategies are to scale up the Web tier (a^w), to scale up the database tier (a^d), to scale up both tiers (a^b), or do nothing ($a^\emptyset$). The set of possible actions that the agent can perform is thus $A = \{a^w, a^d, a^b, a^\emptyset\}$.

Input: Environment functions E and E', state space S, initial state s_0, actions A, service time threshold τ
Output: Estimated state-action value function $Q : S \times A \mapsto \mathbb{R}$.

```
t ← 1
s_1 ← E(s_0, a^∅)
while true do
    Log s_t for learning model
    Extract p (95^th percentile service time) from s_t
    if p > τ (SLA violation detected) then
        For all a ∈ A\a^∅, q(a) ← 0
        for each a ∈ A\a^∅ do
            t ← t + 1
            s_t ← E(s_{t−1}, a)
            Log s_t for learning model
            q(a) ← r(s_t)
            s_t ← E'(s_t, a)
        end
        a_t ← argmax_a q(a)
    else
        a_t ← a^∅
    end
    t ← t + 1
    s_t ← E(s_{t−1}, a_t)
end
```

Algorithm 1: Policy learning exploration (on-line learning) approach.

Input: Environment functions E and E', state space S, initial state s_0, actions A, service time threshold τ, estimated value function $Q : S \times A \mapsto \mathbb{R}$.

```
t ← 1
s_1 ← E(s_0, a^∅)
while true do
    Extract p (95^th percentile service time) from s_t
    if p > τ (SLA violation detected) then
        a_t ← argmax_a Q(s_t, a)
    else
        a_t ← a^∅
    end
    t ← t + 1
    s_t ← E(s_{t−1}, a_t)
end
```

Algorithm 2: Policy learning exploitation (decision making) approach.

A **policy** π is a mapping from system states s_t to corresponding actions a. We use the *value function* approach, in which the agent predicts the value (future reward) of each possible action a and selects the action maximizing the predicted reward: $\pi(s_t) = \arg\max_a Q(s_t, a)$. Our value function $Q(\cdot, \cdot)$ incorporates a neural network regression model trained on observations recorded during an *exploration phase*.

Critically, since the input state s_t to π includes not only the raw arrival rate λ but also the workload pattern $P(C)$, the value function $Q(s_t, a)$ can assign different value levels to state–action pairs, in which the arrival rates are the same but the workload patterns are different. Without this flexibility, the learner would be forced to assign the highest value Q for a particular arrival rate λ to actions that would either unnecessarily overcommit resources for workload patterns that are relatively light in terms of resource consumption or would allow a bottleneck to appear or persist when the workload pattern is more resource-hungry.

The **reward function** r encourages the learning agent, when it is successful, and discourages it, when it is unsuccessful, at maintaining the response time SLO with minimal allocation of resources. We use the immediate reward function $r(s_t)$, where $s_t = (U, P(C), \lambda, p)$ is a system state as follows:

$$r(s_t) = \frac{1}{p + \sum_{i=1}^{n} (u_i \times \alpha_i)} \tag{1}$$

α_i specifies the relative weight of the resource minimization objectives for each tier. In our experiments, we use $\alpha_w = 0.25$ and $\alpha_d = 0.5$, for the Web server tier and database tier, respectively. These settings prioritize Web tier scale-out decisions over database tier scale-out decisions when both actions achieve the same response time. We incorporate this heuristic because scaling the database tier introduces load balancing overhead at the Web tier and data synchronization overhead within the database tier.

We model the **environment** that the agent interacts with as a stochastic function $E(s_t, a)$ mapping current state s_t and action (scaling strategy) a to a new state. The agent must wait for a user-defined interval to give enough time for the system to realize the effects of the action. We also allow the agent to instantaneously retract a previously executed action in the current environment with a function $E'(s_t, a)$. E' simply returns s_t with the tier configuration scaled down by the number of machines added by action a, without affecting the other elements of s_t. Access to function E' allows the agent to explore different actions with respect to a specific configuration under possibly fluctuating workloads.

B. Policy Learning Algorithm

We use a simplified greedy version of the Q-learning approach [30] to build, through online observation, an estimate of the value $Q(s_t, a)$ of each action in each state. Algorithms 1 and 2 give pseudocode for our exploration and exploitation (online learning and decision making) approaches. The learning agent begins with no knowledge and monitors, over each interval of time, for SLA violations. During the exploration phase, when the agent detects a violation, it attempts all possible actions using a simple exhaustive exploration algorithm, and then, it selects the action that provided the highest reward. The agent also logs each state s_t encountered during exploration, in order to build a neural network regression model predicting the service time for a given tier configuration and workload. In the exploitation phase, when an SLA violation is detected, the value $Q(s_t, a)$ of each action a is calculated by first using the regression model to predict the service time p that would entail if a was applied to the configuration in s_t and then calculating

the corresponding reward. The action maximizing $Q(s_t, a)$ is chosen.

Clearly, our simplistic exploration approach requires the ability to quickly perform an action, measure application performance under the new configuration, and then retract the action. To accomplish this, we implemented a strategy similar to that proposed in [13], which maintains a pool of previously booted virtual machines ready to be quickly added to a specific tier of the application.

V. Experimental Design

Here, we provide details of rule-based strategies used to compare with our proposed technique, our application setup, workload generation methods, testbed cloud infrastructure, experimental details, and evaluation criteria.

A. Industry-Standard Rule-Based Scale-Out Strategies

A multitier Web application hosted on a cloud can satisfy specific response time requirements by performing horizontal scaling (scale-out) using a variety of policies, including rule-based methods and schedule-based methods. A rule base defines a set of rules for triggering scale-out operations; for example, if a tier's virtual machine CPU utilization reaches 85% or its memory utilization reaches 90%, we may want to add an additional virtual machine to the tier. Schedule-based approaches, on the other hand, adjust the number of virtual machines allocated to an application based on a specific schedule, e.g., based on particular hours of the day or days of the week. As a baseline for comparison with the policy learning method developed in this paper, we experiment with two fairly simple industry-standard rule-based scale-out strategies, namely, CPU reactive and response reactive.

We believe that the CPU reactive, response reactive, and baseline methods used in the paper provide broad coverage of the most common rule-based methods one might select. Currently, we do not see any other intelligent autoscaling methods being offered commercially by cloud providers to application owners. This section describes CPU reactive, response reactive, and baseline strategies in further detail.

1) CPU Reactive: Most Web applications that generate dynamic content require a significant amount of CPU capacity, and such applications often saturate their CPUs when attempting to serve an unexpectedly heavy workload. In preliminary experiments with our sample benchmark Web application on EC2, we established that CPU always saturates before the Web application's response times go beyond a specific threshold. This means that CPU is the main resource limitation under heavy workloads; therefore, in our CPU reactive strategy, we configure the system to trigger scale-out operations whenever average CPU utilization crosses a specific threshold. For our two-tier Web application, which comprises a Web tier and a database tier, the policy is as follows:

i) Trigger a scale-out operation whenever the average CPU utilization over all virtual machines allocated to a specific tier goes beyond a specific threshold (α_{cpu}).
ii) Scale-out operations are performed by adding one virtual machine to the tier(s) triggering the operation.

The CPU Reactive approach may scale out none, one, or both tiers depending on the CPU utilization of the tier-specific virtual machines over a given monitoring interval.

2) Response Reactive: The response reactive strategy relies on response time monitoring rather than CPU utilization monitoring to trigger scale-out operations. This is sensible because, as previously discussed, a maximum response time requirement would be an important SLO in the SLA for any operations team setting out to manage an application's performance.

In the response reactive approach, we trigger scale-out operations whenever the application's response time exceeds a specific threshold. For our two-tier Web application, our response reactive technique is as follows:

i) Trigger a scale-out operation whenever the 95th percentile of the response time goes beyond a specific threshold (τ_{rt}).
ii) Scale-out operations are performed by adding one virtual machine to each tier(s), for which the average CPU utilization is higher than a specific threshold (α_{cpu}).

Like the CPU Reactive strategy, the response reactive strategy may similarly scale out none, one, or both tiers. Although the triggering event is different, the scale-out operation itself is identical in the two cases.

3) Baseline: We use a baseline approach similar to that of Urgaonkar *et al.* [9], which scales up all replicable tiers whenever a bottleneck occurs.

B. Benchmark Web Application

RUBiS [8] is an open-source benchmark Web application for auctions. It provides typical core functionality of an auction site, such as browsing, selling, and bidding for items, and it provides three user roles: visitor, buyer, and seller. Visitors are not required to register; they are allowed to browse for items that are available for auction. We use the PHP implementation of RUBiS as a sample Web application for our experimental evaluation.

C. Testbed Cloud

Amazon owns multiple geographically dispersed data centers around the world known as *regions*. Each region is divided into multiple locations known as *availability zones*. Users are able to place their EC2 instances in any region and any availability zone. We performed all experiments in the `us-west-2` region and the `us-west-2c` availability zone of the Amazon public cloud. Fig. 2 shows the testbed cloud infrastructure used during the experiments. We use an EC2 medium instance (c1.medium) as the head node. The head node is configured to fulfill the following responsibilities:

i) Generate the workload (user sessions).
ii) Act as a proxy server for the benchmark Web application.

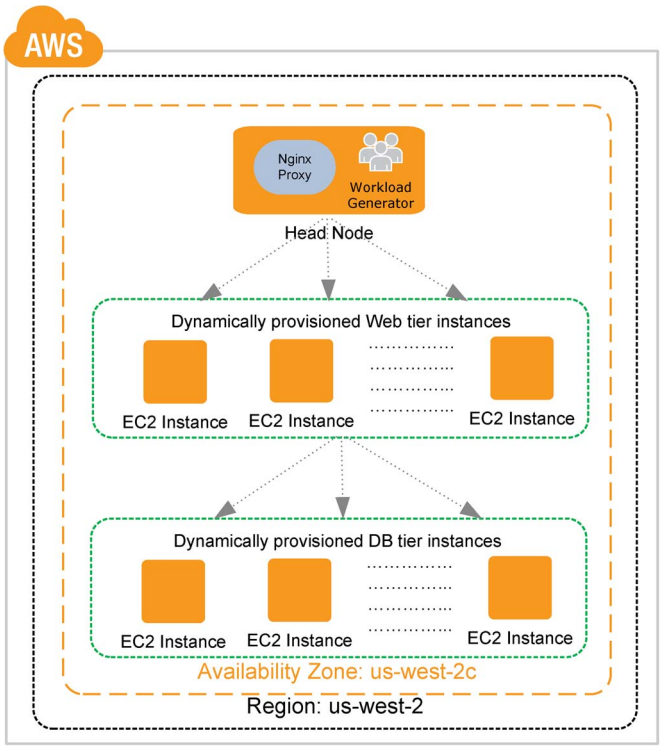

Fig. 2. Testbed cloud infrastructure used for the experiments.

iii) Load balance incoming requests among the servers in the provisioned Web server tier.
iv) Dynamically provision resources for the application.

We use a pool of dynamically provisioned EC2 micro instances for the Web and database tiers. The pool always contains at least one virtual machine allocated to the Web tier and one virtual machine allocated to the database tier for the benchmark application. We set the maximum number of dynamically provisioned instances to 14 for all experiments. We never observed any of the head node's resources (CPU, memory, input/output, or network bandwidth) saturate during the experiments.

We use Nginx as the load balancer for the Web tier because it offers detailed logging and allows reloading of its configuration file without terminating existing client sessions. Since RUBiS does not currently support load balancing over a database tier, we modified it to use round-robin balancing over a set of database servers listed in a database connection settings file, and we developed a server-side component to update the database connection settings file after a scaling operation has modified the configuration of the database tier. As the focus of our experiments is on scaling, not on database consistency, during our experiments, the workload generator only submits read requests, and each database server is set up with a replica of the same database. In a real-world deployment, we assume that a mechanism such as xkoto [31] would exist to ensure consistent reads after updates to a master database.

D. Synthetic Workload Generation

We use `httperf` [32] to generate synthetic workload for our experiments. We generate workloads for specific durations with a required number of user sessions per second. A user session emulates a visitor that browses items up for auction in specific categories and geographical regions and also bids on items up for auction. We generate traffic in a step-up fashion, starting from a specific number of user sessions per second, and increase the number of user sessions every 60 s.

In the experiments reported on in this paper, the workload pattern $P(C)$ is fixed during each experiment, but we use different workload patterns in experiment 1 (a relatively light workload pattern) and experiment 2 (a relatively heavy workload pattern). Details follow in the next section.

VI. Experimental Results

To evaluate the proposed policy learning method, we performed an experimental evaluation, in which we first learned a URI partitioning model for the RUBiS benchmark Web application and then performed two experiments using different workload patterns. Both experiments executed in five phases, in turn named **CPU Reactive**, **Response Reactive**, **Exploration**, **Exploitation**, and **Baseline**, using the same workload pattern.

i) In the **CPU Reactive** phase, we used the previously described CPU reactive strategy configured with a very high average response time threshold value (α_{cpu}) of 99%, allowing the tier to nearly saturate before scaling.
ii) In the **Response Reactive** phase, we used the previously described response reactive strategy configured with an average response time threshold value (τ_{rt}) to 500 ms, i.e., an acceptable maximum response time requirement for most Web applications. This strategy scales one or both resource tiers only when τ_{rt} is exceeded.
iii) In the **Exploration** phase, we initialize an empty value function and let the system learn in real time using the proposed policy learning algorithm.
iv) In the **Exploitation** phase, the agent uses the policy learned during exploration and resolves bottlenecks automatically using dynamic resource provisioning.
v) In the **Baseline** phase, we use the simple baseline strategy, in which both tiers are reactively scaled up whenever a SLA violation occurs.

A. URI Partition Model Learning

We generated synthetic workloads comprising different combinations of URIs corresponding to dynamic and static contents for our sample benchmark Web application. We collected 19 200 log entries for the purpose of clustering URIs on document size and response time features. We used the Weka implementation of the EM algorithm for Gaussian mixture models [28] to cluster the preprocessed log entries. Weka's EM implementation automatically identifies the number of clusters (k) by maximizing the log-likelihood of future data. We used the default parameter values for the learning algorithm and identified five different clusters, as shown in Fig. 3. We obtained the majority cluster ID for each URI path in the log file and retained this mapping for the training stage.

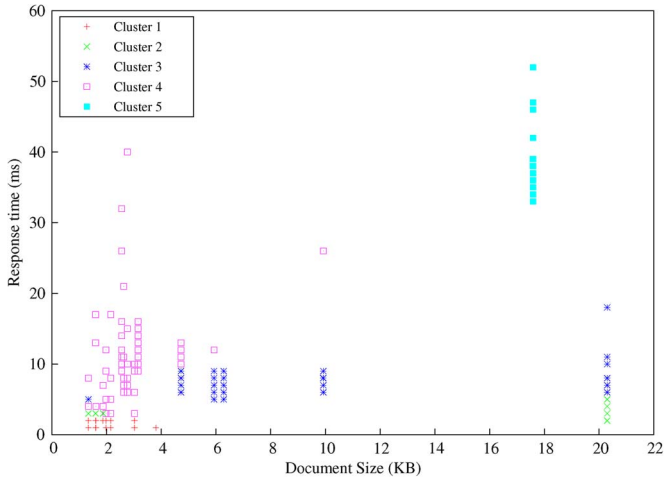

Fig. 3. Clustering of RUBiS URIs based on document size and response time.

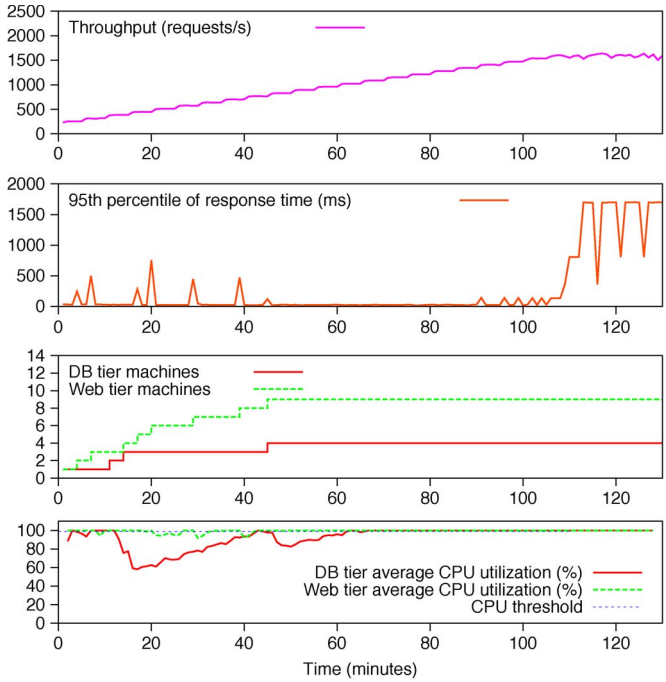

Fig. 4. Experiment 1 results with CPU reactive scale-out strategy. The graphs show throughput (requests served/second), 95th percentile of response time, dynamic addition of instances in each tier, and average CPU utilization over the Web and database tier instances.

B. *Experiment 1 (Light Workload Pattern)*

In this experiment, we modeled each user session by 32 user requests following the workload distribution $P(C) = (0.34, 0.38, 0.12, 0.12, 0.03)$, according to the learned URI partitioning model. We call this workload a "light" workload pattern because only 3% of the URIs map to Cluster 5. As clearly shown in Fig. 3, the URIs in Cluster 5 are the most heavy users of resources and take a long time to process.

The synthetic workload for this experiment began with eight user sessions per second and was incremented every 5 min. For each of the five phases, we repeated the same workload generation process.

1) CPU Reactive: Fig. 4 shows the results of experiment 1 with the CPU reactive scale-out strategy. Whenever the system

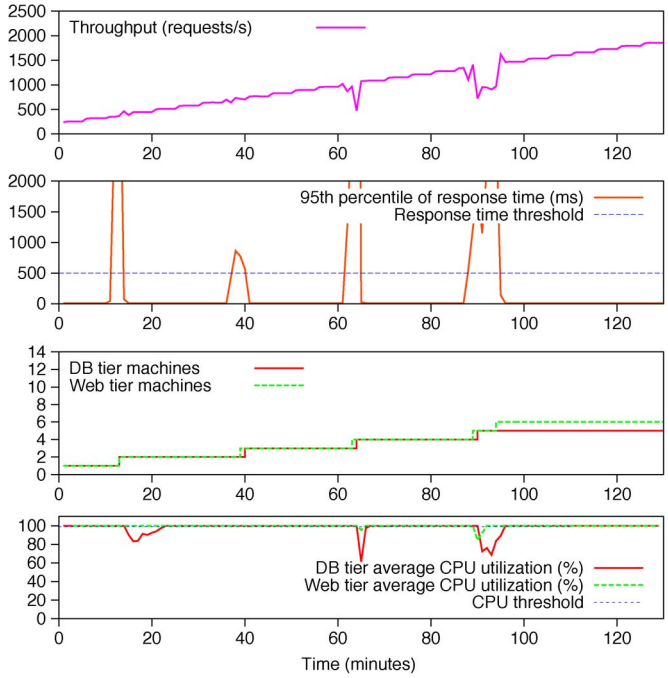

Fig. 5. Experiment 1 results with response reactive scale-out strategy. The system is capable of reacting on performance violations and restoring application performance.

detects a violation of the CPU utilization threshold, it uses the CPU reactive scale-out strategy to identify the tier(s) to scale out; then, it dynamically adds micro instances (virtual machines) to the identified tier(s). The system quickly reaches the maximum allocation limit (14 instances) during the experiment. By the 43rd minute, all 14 instances were consistently utilizing 100% of their CPUs, but we do not observe any significant growth in the 95th percentile of the response time until the 107th minute. At that point, we observe a sudden increase in the 95th percentile of the response time. The downward spikes after minute 107 are due to a special feature of EC2 micro instances: switching of the CPU from *background level* to *max level*. Max-level allocation is allowed for short periods of time to accommodate short spikes in CPU requirements. The system throughput grows linearly in response to the growing workload until the system's capacity to service requests fully utilized, after which the throughput remains constant. Clearly, the system's capacity is much higher than the workload at the 43rd minute. Despite the fact that no further resources are allocated to the system from then onwards, it is nevertheless able to handle the increasing workload up to the 107th minute. This indicates that the CPU reactive strategy is overprovisioning the system and wasting resources.

2) Response Reactive: Fig. 5 shows results with the response reactive scale-out strategy. On each occurrence of response time requirement violations, the system determines which tier(s) to scale out; then, it dynamically adds micro instances (virtual machines) to the identified tier(s). We observe that the system is capable of reacting to performance violations quickly and that response times return to reasonable levels after action is taken. We also observe that system throughput degrades temporarily whenever the response time saturates.

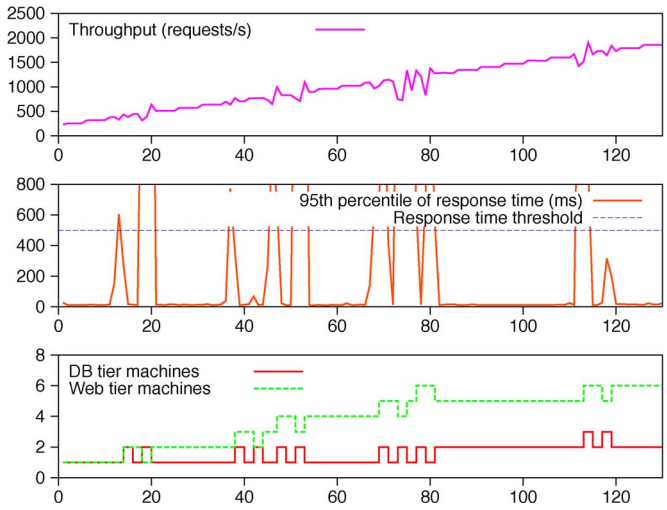

Fig. 6. Experiment 1 results with exploration scale-out strategy. The agent learns a value function for the maximum value resource allocation policy by observing SLA violations then determining appropriate bottleneck resolution actions.

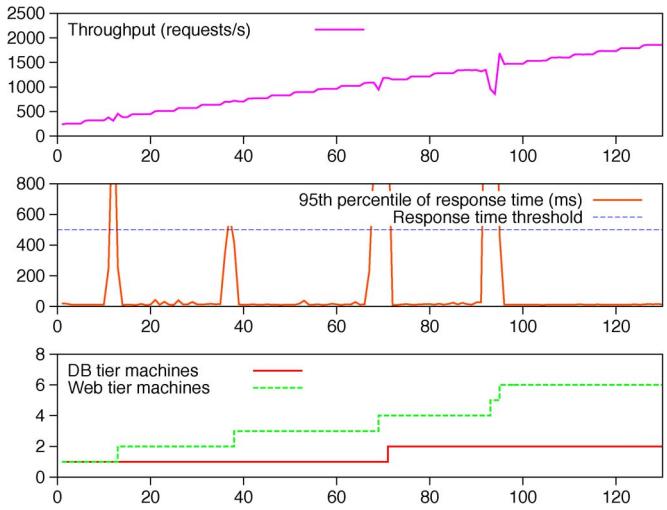

Fig. 7. Experiment 1 results with exploitation scale-out strategy. The agent exploits the policy learned during the exploration phase.

3) Exploration: Fig. 6 shows results during the exploration phase of experiment 1. The bottom graph shows the exploration behavior and adaptive addition of machines to tiers. The learning agent continuously monitors the application's performance against the service time SLO, and when it detects new violations, it explores the action space to determine the best action in the given situation and retains that best action until the next violation occurs. At the end of the exploration phase of the experiment, the agent has learned a value function that can be used with the maximum value resource allocation policy to resolve bottlenecks automatically for this workload pattern.

4) Exploitation: Fig. 7 shows results from the exploitation phase of experiment 1. The bottom graph shows the exploitation behavior and adaptive addition of machines to the tiers. The agent simply exploits the policy learned in the Exploration phase. Whenever the agent detects service time requirement violations, it uses the policy to identify the action to resolve the bottleneck by adaptively provisioning resources for the selected tier(s). The number of requests in violation of the SLA is substantially decreased compared to the Exploration phase, and the total of eight virtual machines allocated at the end of the phase is less than that allocated by the industry-standard CPU reactive and response reactive strategies.

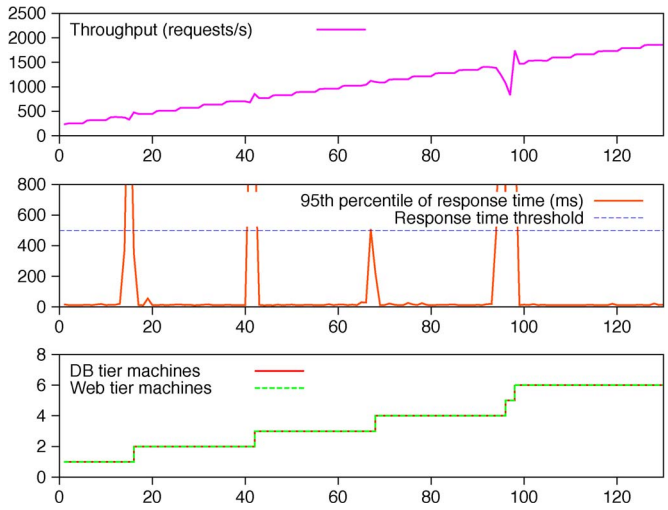

Fig. 8. Experiment 1 results with baseline scale-out strategy. The baseline scale-out agent reactively scales both tiers on every service time violation.

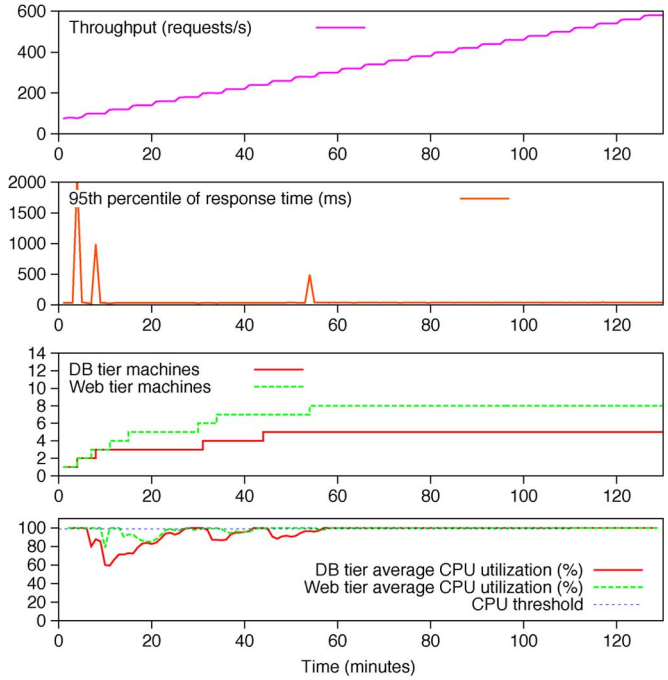

Fig. 9. Experiment 2 results with CPU reactive scale-out strategy. The system is capable of reacting on performance violations and restoring application performance.

C. Baseline

Fig. 8 shows results for the baseline phase of experiment 1, in which adaptive scaling of both tiers is performed every time an SLA violation is detected. Although fewer SLA violations are observed than during the exploitation phase, the application is clearly overprovisioned for substantial periods of time.

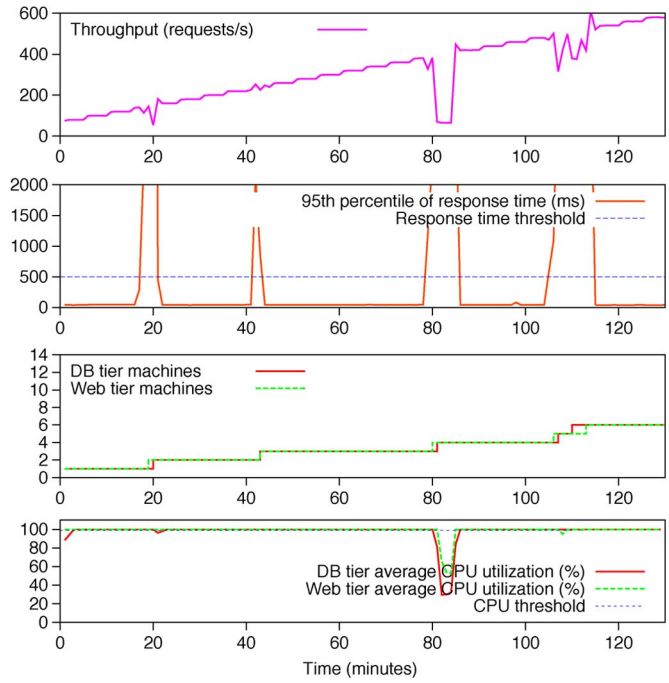

Fig. 10. Experiment 2 results with response reactive scale-out strategy.

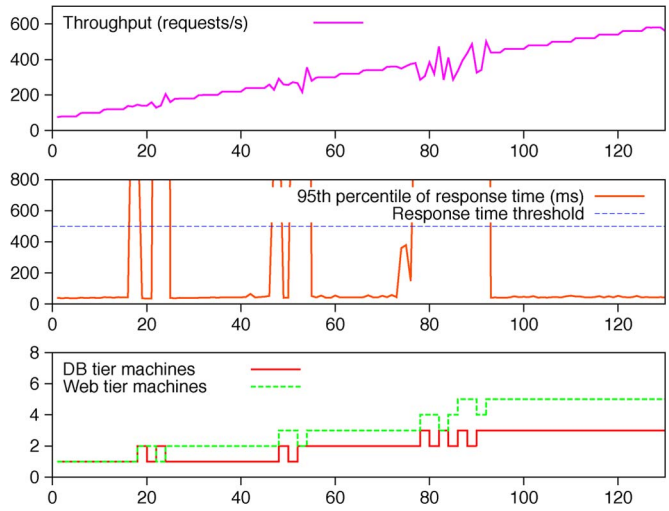

Fig. 11. Experiment 2 results with exploration scale-out strategy.

D. Experiment 2 (Heavy Workload Pattern)

In this experiment, we model each user session by ten user requests following the workload distribution $P(C) = (0.3, 0.2, 0.2, 0, 0.3)$, according to the learned workload model. We call this workload a "heavy" workload pattern because 30% of the URIs belong to Cluster 5, which contains the most heavy users of resources that take a long time to process.

The synthetic workload for this experiment began with eight user sessions per second and was incremented every 5 min. For each of the five phases, we repeated the same workload generation process.

1) CPU Reactive: Fig. 9 shows the results of experiment 1 with the CPU reactive scale-out strategy. Whenever the system detects a violation of the CPU utilization threshold, it uses the CPU reactive scale-out strategy to identify the tier(s) to scale out; then, it dynamically adds micro instances to the identified tier(s). During minutes 1 to 10 of the phase, we observe upward spikes in the 95th percentile of response time that are mainly due to some of the provisioned micro instances switching from max level to background level. The CPU reactive strategy quickly overprovisions the system, maintaining good response time performance for the entire phase, with linear growth in the throughput as the workload increases.

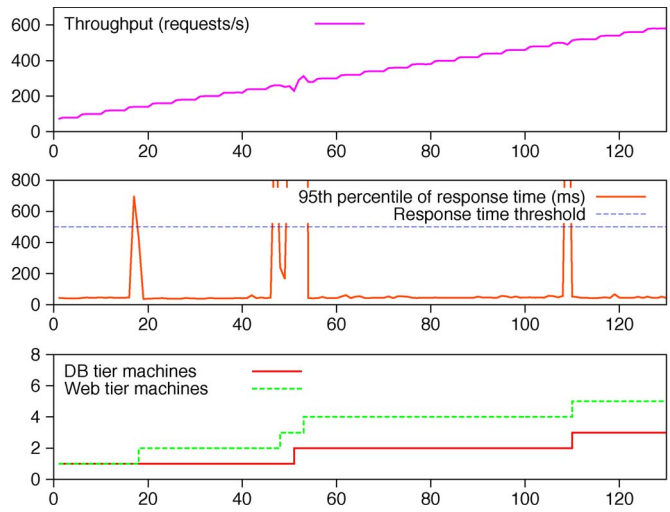

Fig. 12. Experiment 2 results with exploitation scale-out strategy.

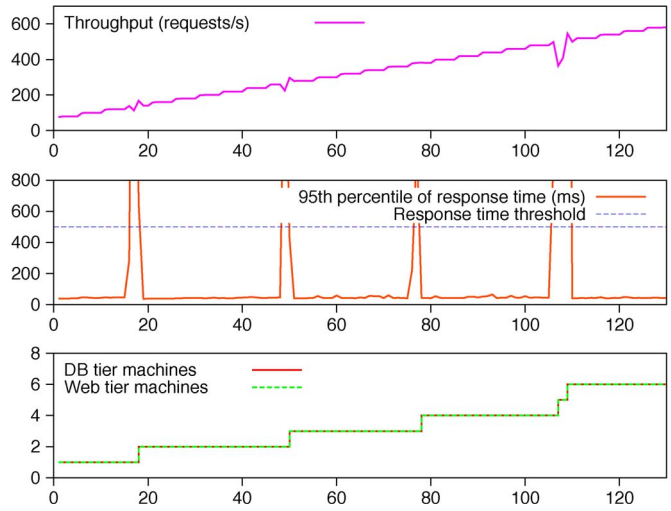

Fig. 13. Experiment 2 results with baseline scale-out strategy.

2) Response Reactive: Fig. 10 shows response reactive scale-out strategy results. On each response time requirement violation, the system identifies the tier(s) to scale out; then, it dynamically adds micro instances to the identified tier(s). We observe that the system is capable of reacting to performance violations quickly and that response times return to reasonable levels after action is taken. We also observe that system throughput degrades temporarily whenever the response time saturates.

3) Exploration: Fig. 11 shows results during the exploration phase of experiment 2. The bottom graph shows the exploration behavior and adaptive addition of machines to tiers. The learning agent continuously monitors the application's performance against the service time SLO, and when it detects

TABLE I
EXPERIMENTAL RESULTS SUMMARY: TOTAL ALLOCATED CPU HOURS AND PERCENTAGE OF REQUESTS VIOLATING THE SLA FOR THE PROPOSED POLICY LEARNING METHOD AND THE BASELINE AUTOSCALING METHOD OVER EXPERIMENTS 1 AND 2

Experiment	Method	Total allocated CPU hours	% of requests violating SLA	Total completions (requests in millions)	Cost (USD)
Experiment 1	Static Allocation	4.33	38.49	1.748	$0.087
	CPU Reactive	24.27	0.68	8.029	$0.485
	Response Reactive	15.57	0.89	8.053	$0.311
	Exploration	11.67	1.32	8.152	$0.233
	Exploitation	11.12	0.58	8.180	$0.222
	Baseline	15.50	0.47	8.173	$0.310
Experiment 2	Static Allocation	4.33	57.12	0.529	$0.087
	CPU Reactive	24.32	0.00	2.576	$0.486
	Response Reactive	14.13	2.31	2.454	$0.283
	Exploration	11.30	2.83	2.538	$0.226
	Exploitation	10.93	0.72	2.572	$0.219
	Baseline	14.10	0.71	2.565	$0.282

new violations, it explores the action space to determine the best action in the given situation and retains that best action until the next violation occurs. At the end of the exploration phase of the experiment, the agent has learned a value function that can be used with the maximum value resource allocation policy to resolve bottlenecks automatically for this workload pattern.

4) Exploitation: Fig. 12 shows results from the exploitation phase of experiment 2. The bottom graph shows the exploitation behavior and adaptive addition of machines to the tiers. As in experiment 1, the agent simply exploits the policy learned in the exploration phase. Whenever the agent detects service time requirement violations, it uses the policy to identify the action to resolve the bottleneck by adaptively provisioning resources for the selected tier(s). The number of requests in violation of the SLA is substantially decreased compared to the EXPLORATION phase, and the total of eight virtual machines is less than that allocated by the industry-standard CPU reactive and response reactive strategies.

5) Baseline: Fig. 13 shows results for the baseline phase of experiment 2. As in experiment 1, although fewer SLA violations are observed than during the exploitation phase, the application is overprovisioned for substantial periods of time.

E. Summary of Experimental Results

To analyze the tradeoff between SLA violations and overprovisioning under our policy learning method (exploration and exploitation), the industry-standard rule-based autoscaling methods (CPU reactive and response reactive), and the baseline autoscaling method, we calculate two performance metrics: the total allocated CPU hours and the percentage of requests violating the SLA. Table I shows the total allocated CPU hours and percentage of requests violating the SLA during each phase of the two experiments. In both experiments, the system allocated fewer total CPU hours under the proposed policy learning approach. However, as is clear by comparing the detailed results already presented, the percentage of requests violating the SLA is higher for the proposed technique in both experiments.

The results show clearly that CPU reactive, response reactive, and the baseline approach moderately overprovision resources but provide better performance in terms of service time. This is due to the fact that, in our experiments, the workload is always increasing; hence, proactive overprovisioning allows the system to serve requests efficiently for a longer period of time before bottlenecks occur. It is of course always possible to overprovision resources to reduce and/or eliminate SLA violations, but this obviously increases costs for users of public clouds and limits resource utilization in dedicated data centers.

Cloud providers and public cloud users thus need techniques that minimize SLA violations without overprovisioning resources. The evaluation shows that the proposed policy learning approach could help cloud providers host multitier Web applications with SLAs providing specific service time guarantees while minimizing resource utilization.

One interesting result to note is that, although we would expect the exploration of bottleneck resolution policies to be expensive, in fact, the exploration phases of the two experiments were less costly than those of any other successful strategy besides exploitation. This indicates that occasional exploration to fine tune bottleneck resolution policies would be quite feasible in production, if the cost in terms of requests violating the SLA can be tolerated.

VII. CONCLUSION

Designing new algorithms to minimize cost and maximize performance in cloud computing is an active research topic. The scale of cloud-hosted services is enormous, encouraging individuals to develop highly scalable applications at minimal cost that attract large user bases. Despite the great potential for developing innovative applications, choosing an appropriate resource allocation is always a difficult task. Autoscaling a multitier Web application hosted on the cloud requires a great deal of domain knowledge and knowledge of the application's performance on the specific infrastructure, making it difficult. Industry-standard strategies such as the CPU reactive and response reactive strategies explored in this paper are useful, but as we have shown, they have a tendency to overprovision resources for a given workload level. In this paper, we have presented and evaluated a new method for unsupervised online autoscaling policy learning for multitier Web applications under different workload patterns. We have evaluated the proposed policy learning algorithm using two different workloads and compared it with industry-standard rule-based (CPU reactive and response reactive) strategies as well as a baseline method.

The proposed approach is novel in that it does not require any prior knowledge of the application's resource utilization and minimizes the overhead needed to monitor, detect, and resolve bottlenecks. Our experimental evaluation shows the strength of the approach in resolving bottlenecks in a multitier Web application while only provisioning the necessary resources, meeting SLAs at minimal cost.

We are currently investigating the use of long-term exploration as necessary to revise existing policies under dynamically changing workload distributions, introducing policy learning for scale-down actions, and planning more sophisticated experiments with real-time varying workloads.

REFERENCES

[1] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Gener. Comput. Syst.*, vol. 27, no. 6, pp. 871–879, Jun. 2011.

[2] P. Bodik *et al.*, "Statistical machine learning makes automatic control practical for Internet datacenters," in *Proc. HotCloud*, 2009, pp. 1–5.

[3] "Amazon Web Services Auto Scaling," Amazon Inc., Seattle, WA, USA, 2009. [Online]. Available: http://aws.amazon.com/autoscaling/

[4] "The Official FIFA Web Site," F. I. de Football Assoc., Zurich, Switzerland, 2011. [Online]. Available: http://www.fifa.com/

[5] A. Martin and J. Tai, "A workload characterization of the 1998 World Cup web site," *IEEE Netw.*, vol. 14, no. 3, pp. 30–37, May/Jun. 2000.

[6] M. McGee, "Michael Jackson's Death: An Inside Look at How Google, Yahoo, and Bing Handled an Extraordinary Day in Search," 2009. [Online]. Available: http://searchengineland.com/michael-jackson-extraordinary-day-in-search-21641

[7] P. Nicolas, C. David, G. Ricard, T. Jordi, and A. Eduard, "Characterization of workload and resource consumption for an online travel and booking site," in *Proc. IEEE IISWC*, Atlanta, GA, USA, 2010, pp. 1–10.

[8] "RUBiS: An auction site prototype," OW2 Consortium, Paris, France, 1999. [Online]. Available: http://rubis.ow2.org/

[9] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier Internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1–39, Mar. 2008.

[10] H. Liu and S. Wee, "Web server farm in the cloud: Performance evaluation and dynamic architecture," in *Proc. 1st Int. Conf. CloudCom*, Berlin, Germany, 2009, pp. 369–380.

[11] X. Bu, J. Rao, and C.-Z. Xu, "A reinforcement learning approach to online web systems auto-configuration," in *Proc. 29th IEEE ICDCS*, 2009, pp. 2–11.

[12] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2006, pp. 65–73.

[13] P. Bodik *et al.*, "Automatic exploration of datacenter performance regimes," in *Proc. 1st Workshop ACDC*, New York, NY, USA, 2009, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1555271.1555273

[14] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier Internet services and its applications," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst. SIGMETRICS*, 2005, vol. 33, pp. 291–302.

[15] J. Rao and C.-Z. Xu, "Online capacity identification of multitier websites using hardware performance counters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 3, pp. 426–438, Mar. 2010.

[16] J. Allspaw, *The Art of Capacity Planning*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2008.

[17] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, "Autonomic mix-aware provisioning for non-stationary data center workloads," in *Proc. 7th IEEE ICAC*, Washington, DC, USA, 2010, pp. 21–30.

[18] J. Dejun, G. Pierre, and C.-H. Chi, "Resource provisioning of Web applications in heterogeneous clouds," in *Proc. 2nd USENIX Conf. Web Appl. Develop.*, 2011, p. 5.

[19] D. Villela, P. Pradhan, and D. Rubenstein, "Provisioning servers in the application tier for e-commerce systems," *ACM Trans. Internet Technol.*, vol. 7, no. 1, p. 7, Feb. 2007. [Online]. Available: http://doi.acm.org/10.1145/1189740.1189747

[20] A. Sharma *et al.*, "Automatic request categorization in Internet services," *SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 2, pp. 16–25, Sep. 2008.

[21] P. Bodik, O. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proc. 1st ACM SoCC*, New York, NY, USA, 2010, pp. 241–252.

[22] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *Proc. IEEE Int. Conf. Cloud Eng.*, Boston, MA, USA, 2014, pp. 195–204.

[23] F. Seracini *et al.*, "A comprehensive resource management solution for web-based systems," in *Proc. 11th ICAC*, Philadelphia, PA, USA, 2014, pp. 233–239.

[24] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *Proc. 11th ICAC*, Philadelphia, PA, USA, 2014, pp. 57–64.

[25] L. Yazdanov and C. Fetzer, "Lightweight automatic resource scaling for multi-tier web applications," in *Proc. 7th IEEE Int. Conf. Cloud Comput.*, 2014, pp. 466–473.

[26] B. Nicolas, P. Thanasis G., and A. Karl, "Automatic SLA-driven provisioning for cloud applications," in *Proc. Int. Symp. Cluster, Cloud Grid Comput.*, Newport Beach, CA, USA, 2011, pp. 434–443.

[27] W. Iqbal, M. Dailey, and D. Carrera, "Low cost quality aware multi-tier application hosting on the Amazon cloud," in *Proc. Int. Conf. FiCloud*, 2014, pp. 202–209.

[28] C. M. Bishop, *Pattern Recognition and Machine Learning*. Berlin, Germany: Springer-Verlag, 2007.

[29] W. Iqbal, "Lecture Buddy: Real Time Teaching Evaluation and Feedback System," 2011, [Online]. Available: http://www.lecturebuddy.com

[30] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, pp. 279–292, 1992.

[31] xkoto, "GRIDSCALE," 2009. [Online]. Available: http://www.xkoto.com/products/

[32] D. Mosberger and T. Jin, "httperf: A tool for measuring Web server performance," in *Proc. 1st Workshop Internet Server Perform.*, 1998, pp. 59–67.

Waheed Iqbal received the Ph.D. degree in computer science from the Asian Institute of Technology, Bangkok, Thailand, in 2012. Since 2014, he has been an Assistant Professor with the Punjab University College of Information Technology, University of the Punjab, Lahore, Pakistan. His research interests lie in cloud computing, distributed systems, machine learning, and big data.

Mathew N. Dailey received the Ph.D. degree in computer science and cognitive science from the University of California, San Diego, CA, USA, in 2002. In 2006, he joined the Department of Computer Science and Information Management, Asian Institute of Technology, Bangkok, Thailand, where he is currently an Associate Professor. His research interests lie in machine learning, machine vision, robotics, and cloud computing.

David Carrera received the Ph.D. degree from the Polytechnic University of Catalonia (UPC), Barcelona, Spain, in 2008. He is currently an Associate Professor with the Department of Computer Architecture, UPC and an Associate Researcher with the Barcelona Supercomputing Center, Barcelona. His research interests are focused on the integrated performance management of virtualized data centers that host applications with differentiated quality-of-service objectives.