Unsupervised Learning of Dynamic Resource Provisioning Policies for Cloud-hosted Multi-tier 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. However, the complexity of multi-tier 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 multi-tier 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 EC2 cloud 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 multi-tier Web applications while meeting service level objectives, without any prior knowledge of the applications' resource utilization or workload patterns.

Index Terms—Cloud computing, Resource management, System performance, Multi-tier applications, Service-level agreement, Scalability.

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.

W. Iqbal is currently with the Punjab University College of Information Technology, University of the Punjab, Lahore, Pakistan. The work reported upon in this paper was performed while he was at the Asian Institute of Technology, Thailand.
E-mail: waheed.iqbal@pucit.edu.pk

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

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

One of the typical architectures for cloud-hosted applications is the multi-tier Web application consisting of at least a presentation tier, a business logic tier, and a data management tier running as separate processes. Multi-tier 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 auto-scaling [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 then dynamically scaling allocated resources as needed to handle increased loads. However, for multi-tier 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 multi-tier 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 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 coarse-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 Web site [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 10 different mixes of dynamic and static requests for the RUBiS benchmark auction application [8]. Figure 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 multi-tier Web applications using access logs and unsupervised machine learning. The method automatically identifies groups of 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 then exploits that policy to reduce violations of the SLO while minimizing resource utilization. The approach does not require pre-deployment 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 (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 multi-tier Web applications while meeting service level objectives, without any prior knowledge of the applications resource utilization or workload patterns.

The exact contribution of this paper are:

- **Workload pattern identification**: we propose a formal model for workload patterns for multi-tier Web applications and an automated method for identifying the parameters of workload patterns online using access logs and unsupervised machine learning.
- **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 multi-tier architecture, we have only instrumented and empirically evaluated the method on one particular two-tier Web application installed on the Amazon EC2 cloud. We also assume that sufficient bandwidth exists 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 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 et al. [10] monitor the CPU and bandwidth usage of virtual machines hosted on an Amazon EC2 cloud, 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 multi-tier Web applications or database scalability, a crucial factor in dynamic management of multi-tier workloads.

There have also been several efforts to use machine learning to manage application resources dynamically. For example,
Rao 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. Gerald 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) [14] for Web applications hosted on clouds. The learned model is then used to provide 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 multi-tier applications. Uragaonkar et al. [15] present an analytical model using queuing networks to capture the behavior of each tier. Jia et al. [16] present an on-line method for capacity identification [17] of multi-tier Web applications hosted on physical machines using hardware performance counters. Rahul et al. [18] present a technique to model dynamic workloads for multi-tier 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 cloud 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 multi-tier application as a black box. Our proposed approach is able to learn resource allocation policies in real time.

Jiang et al. [19] present a dynamic resource provisioning approach for multi-tier 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], [19], [20]. However, a few researchers have incorporated workload pattern modeling into their application’s resource provisioning policies. Sharma et al. [21] 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. [22] 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, Hector et al. [23] 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, 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. Filippo et al. [24] present a resource provisioning system for Web applications based on queuing theory. The system profiles non-functional 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 — other tiers are simply over-provisioned to prevent them from becoming bottlenecks. Anshul et al. [25] use a combination of Kalman filtering and a queuing 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. Lenar et al. [26] 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 [27] 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 multi-tier 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 [28], in which industry standard rule-based resource provisioning methods (explained in Section V-A) are used to explore cost-performance trade-offs with Amazon EC2 compute resources. This paper proposes and empirically evaluates a new unsupervised machine learning method for dynamic provisioning of multi-tier 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. In this section, 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:

- **URI space partitioning**: a partitioning of the application’s URI space into requests with similar resource utilization characteristics.

- **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-provisioned or under-provisioned 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 (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 [29] 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. URI space partitioning evaluation

To evaluate the effectiveness of the proposed URI space partitioning method, we obtained one-month’s worth of access logs from a real Web application named Lecture Buddy [30] hosted on Amazon EC2 using a “micro” instance. We used the Weka implementation of the Expectation Maximization (EM) algorithm for Gaussian mixture models [29] to cluster the preprocessed log entries on document size and response time features. Weka’s EM implementation automatically identifies the number of clusters (\( k \)) by maximizing the log-likelihood of future data. Figure 2 shows the resulting clusters of URIs based on document size and response time features.

After majority voting (to ensure that each URI maps to only one cluster), we observed only three distinct clusters (Cluster 2, Cluster 3, and Cluster 5). Cluster 2 contains only static resources, Cluster 3 contains an equal number of large static resources (javascript libraries), and dynamic resources requiring heavy database interaction, and Cluster 5 contains all dynamic resources requiring minor database interaction.

Table I shows the URI cluster mapping identified using the proposed technique along with the number of requests for each URI available in the dataset. This simple evaluation shows that the proposed technique is able to cluster Web application resources appropriately based on resource requirements.

### C. Workload patterns

Our workload pattern model is probabilistic. In each independent trial of the random experiment, we wait for the arrival of a single 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, so 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

In this section, 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) maximizing an objective function that encourages satisfying a response time SLO with minimal resources. We first define our learning agent 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 $\lambda$, and the current $95^{th}$ percentile of the service time $p$. For an $n$-tier application, we define a configuration by the vector $U = (u_1, \cdots, u_n)$, where element $u_i$ indicates the number of machines allocated to tier $i$.

The action $a$ 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 ($a^0$). The set of possible actions the agent can perform is thus $A = \{a^w, a^d, a^b, a^0\}$.

A policy $\pi$ 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 $\pi$ includes not only the raw arrival rate $\lambda$ 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 $\lambda$ 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:

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

(1)

$\alpha_i$ 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 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 $E'(s_t, a_t)$. $E'$ simply returns $s_t$ with the tier configuration scaled down by the number of machines added by action $a_t$, 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 [31] 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 (on-line 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 a 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

\[
Q(s_t, a) = \text{regression model}(s_t, a)
\]

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

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

$t \leftarrow 1$
$s_1 \leftarrow E(s_0, a^0)$

while true do
    Log $s_t$ for learning model
    Extract $p$ (95th percentile service time) from $s_t$
    if $p > \tau$ (SLA violation detected) then
        For all $a \in A \setminus a^0$, $q(a) \leftarrow 0$
        for each $a \in A \setminus a^0$ do
            $t \leftarrow t + 1$
            $s_t \leftarrow E(s_{t-1}, a)$
            Log $s_t$ for learning model
            $q(a) \leftarrow r(s_t)$
            $s_t \leftarrow E'(s_t, a)$
        end
        $a_t \leftarrow \text{argmax}_a q(a)$
    else
        $a_t \leftarrow a^0$
    end
    $t \leftarrow t + 1$
    $s_t \leftarrow E(s_{t-1}, a_t)$
end

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

\[
Q(s_t, a) = \text{regression model}(s_t, a)
\]

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 $E'(s_t, a_t)$. $E'$ simply returns $s_t$ with the tier configuration scaled down by the number of machines added by action $a_t$, 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.

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

\[
m(t) \leftarrow \text{argmax}_a q(s_t, a)
\]

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 $E'(s_t, a_t)$. $E'$ simply returns $s_t$ with the tier configuration scaled down by the number of machines added by action $a_t$, 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 [31] 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 (on-line 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 a 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

\[
Q(s_t, a) = \text{regression model}(s_t, a)
\]

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 $E'(s_t, a_t)$. $E'$ simply returns $s_t$ with the tier configuration scaled down by the number of machines added by action $a_t$, 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.

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 $\tau$
Output: Estimated state-action value function $Q : S \times A \rightarrow \mathbb{R}$.

$t \leftarrow 1$
$s_1 \leftarrow E(s_0, a^0)$

while true do
    Log $s_t$ for learning model
    Extract $p$ (95th percentile service time) from $s_t$
    if $p > \tau$ (SLA violation detected) then
        For all $a \in A \setminus a^0$, $q(a) \leftarrow 0$
        for each $a \in A \setminus a^0$ do
            $t \leftarrow t + 1$
            $s_t \leftarrow E(s_{t-1}, a)$
            Log $s_t$ for learning model
            $q(a) \leftarrow r(s_t)$
            $s_t \leftarrow E'(s_t, a)$
        end
        $a_t \leftarrow \text{argmax}_a q(a)$
    else
        $a_t \leftarrow a^0$
    end
    $t \leftarrow t + 1$
    $s_t \leftarrow E(s_{t-1}, a_t)$
end

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

```

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 $E'(s_t, a_t)$. $E'$ simply returns $s_t$ with the tier configuration scaled down by the number of machines added by action $a_t$, 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 [31] 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 (on-line 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 a 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

\[
Q(s_t, a) = \text{regression model}(s_t, a)
\]

V. Experimental Design

In this section, 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 multi-tier 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 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
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. Figure 3 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:

- Generate the workload (user sessions).
- Act as a proxy server for the benchmark Web application.
- Load balance incoming requests among the servers in the provisioned Web server tier.
- 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, I/O, 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 [32]
would exist to ensure consistent reads after updates to a master database.

D. Synthetic workload generation

We use httpperf [33] 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 seconds.

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.

- In the CPU Reactive phase, we used the previously described CPU Reactive strategy configured with a very high average response time threshold value \( \alpha_{cpu} \) of 99%, allowing the tier to nearly saturate before scaling.
- In the Response Reactive phase, we used the previously described Response Reactive strategy configured with an average response time threshold value \( r_{rt1} \) to 500 ms, an acceptable maximum response time requirement for most Web applications. This strategy scales one or both resource tiers only when \( r_{rt} \) is exceeded.
- In the Exploration phase, we initialize an empty value function and let the system learn in real time using the proposed policy learning algorithm.
- In the Exploitation phase, the agent uses the policy learned during exploration and resolves bottlenecks automatically using dynamic resource provisioning.
- 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 Expectation Maximization (EM) algorithm for Gaussian mixture models [29] 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 Figure 4. We obtained the majority cluster ID for each URI path in the log file and retained this mapping for the training stage.

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 is clearly shown in Figure 4, 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 five minutes. For each of the five phases, we repeated the same workload generation process.

1) CPU Reactive: Figure 5 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 (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 till 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 [34]. 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:** Figure 6 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.

3) **Exploration:** Figure 7 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:** Figure 8 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.

**C. Baseline**

Figure 9 shows results for the Baseline phase of Experiment 1, in which adaptive scaling of both tiers is performed every time a 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.

**D. Experiment 2 (heavy workload pattern)**

In this experiment, we model each user session by 10 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 five minutes. For each of the five phases, we repeated the same workload generation process.

1) CPU Reactive: Figure 10 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 95th percentile of response time that are mainly due to the some of the provisioned micro instances switching from max level to background level. The CPU Reactive strategy quickly overprovisions the system, that are mainly due to the some of the provisioned micro instances switcing from max level to background level. 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). The number of requests in violation of the SLA is substantially decreased compared to the Exploration phase, and the total of eight 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: Figure 12 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 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: Figure 13 shows results from the 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: Figure 14 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 over-provisioning 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 II 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, so 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 multi-tier Web applications with SLAs providing specific service time guarantees while minimizing resource utilization.

One interesting result to note is that although we would expect 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 multi-tier 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 multi-tier 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 multi-tier Web application while only provisioning the necessary resources, meeting service level agreements 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

TABLE II
EXPERIMENTAL RESULTS SUMMARY. TOTAL ALLOCATED CPU HOURS AND PERCENTAGE OF REQUESTS VIOLATING THE SLA FOR EXPERIMENTS 1 AND 2.

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