Sustainable Computing: Informatics and Systems 22 (2019) 287–299

Contents lists available at ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

MLscale: A machine learning based application-agnostic autoscaler

Muhammad Wajahat [a,*], Alexei Karve [b], Andrzej Kochut [b], Anshul Gandhi [a]

[a] Stony Brook University, NY, United States
[b] IBM T.J. Watson Research Center, NY, United States

ARTICLE INFO

Article history:
Received 16 March 2017
Accepted 2 October 2017
Available online 10 October 2017

Keywords:
Cloud computing
Autoscaling
Dynamic provisioning
Machine learning

ABSTRACT

Autoscaling is the practice of automatically adding or removing resources for an application deployment to meet performance targets in response to changing workload conditions. However, existing autoscaling approaches typically require expert application and system knowledge to reduce resource costs and performance target violations, thus limiting their applicability. We present MLscale, an application-agnostic, machine learning based autoscaler that is composed of: (i) a neural network based online (black-box) performance modeler, and (ii) a regression based metrics predictor to estimate post-scaling application and system metrics. Implementation results for diverse applications across several traces highlight MLscale's application-agnostic behavior and show that MLscale (i) reduces resource costs by about 41%, on average, compared to the optimal static policy, (ii) is within 14%, on average, of the cost of the optimal dynamic policy, and (iii) provides similar cost-performance tradeoffs, without requiring any tuning, when compared to carefully tuned threshold-based policies.

1. Introduction

Online services typically experience significant variations in workload demand [1,2]. Statically provisioning resources for the peak demand can minimize performance degradation during high loads, but can result in unnecessary overprovisioning, to the tune of 40–50% [3,4], leading to substantial resource costs and energy expenditure.

Motivation: A promising approach that is widely employed to mitigate resource costs without compromising on performance is *autoscaling – the ability and knowledge to add/remove the required amount of resources in response to changes in demand*. Underestimating the resource provisioning can lead to costly service level agreement (SLA) violations; overestimating the provisioning can result in unsustainable operating costs and poor resource utilization. While autoscaling can be employed both in physical clusters (to save on energy costs) and in virtual clusters (to save on rental costs), successfully autoscaling the system is challenging. Specifically, it is not obvious as to *how many nodes (machines/VMs)* should be added or removed to meet the required performance.

Objective: Typical approaches to autoscaling rely on a deep understanding of the application, the underlying infrastructure, and their dynamics, to accurately scale resources. In the absence of such information, exhaustive instrumentation and experimentation is required to carefully study the system [4,5]. However, as established by prior work, gaining such an understanding of a given system is itself a challenging task worthy of research [6]. *Given the diversity of applications running in the data center and cloud today, can we develop an application-agnostic autoscaling approach?* While predictions of future load can aid in autoscaling, they still require an understanding of how load relates to performance in order to successfully scale the system; further, accurate predictions are often not available for all applications [4,7].

Prior approaches: Black-box modeling techniques, such as those based on machine learning (ML), have emerged as a promising solution for autoscaling. Typical ML approaches to autoscaling rely on reinforcement learning, which aims to learn the best scaling action for a given system state using historical data and trial-and-error [8–11]. However, such approaches incur high overhead due to the requirement of a large state space for learning [12,13]. Other approaches employ linear regression based techniques to capture the relationship between system state and performance [1,14]. However, performance is often non-linearly related to resource usage [5,15]. We discuss related work in detail in Section 2.

Our approach: In this paper, we present *MLscale*, an application-agnostic autoscaler based on ML. In particular, MLscale first employs neural networks to develop an *online black-box model* that

* Corresponding author.
E-mail addresses: mwajahat@cs.stonybrook.edu (M. Wajahat), karve@us.ibm.com (A. Karve), akochut@us.ibm.com (A. Kochut), anshul@cs.stonybrook.edu (A. Gandhi).

https://doi.org/10.1016/j.suscom.2017.10.003
2210-5379/

relates observable monitored metrics with the performance metric, such as response time. MLscale then employs regression to estimate the values of the monitored metrics after a hypothetical scaling event, and uses these estimates to *predict performance after scaling*. These predictions enable MLscale to accurately scale the system.

We evaluate the benefits of MLscale on an OpenStack private cloud and an AWS public cloud under six different request traces using three applications: (i) IBM's DayTrader that emulates an online stock trading system, (ii) a PHP-based load-balanced web server tier, and (iii) a PHP-MySQL deployment that emulates an online database service. Despite the difference in behavior and complexities, MLscale is able to accurately model the (average and tail) response time for all three applications with a low modeling error (6–9%). Armed with our accurate modeling, MLscale effectively autoscales the application to provide low SLA violations while reducing the resource costs. Our evaluation results show that MLscale can reduce costs by around 23–50% when compared to the optimal static policy. Compared to the optimal dynamic policy, MLscale is typically within 1–38% of the optimal cost. Furthermore, compared to existing finely-tuned application-agnostic utilization threshold based systems, MLscale provides comparable cost-performance tradeoffs without requiring any manual tuning or trial-and-error approaches for setting model parameters. Finally, the resource provisioning under MLscale is significantly more stable than other approaches, thus reducing overheads and wear-and-tear due to provisioning changes.

Contributions: This paper makes the following contributions:

- We develop a neural network-based online black-box approach for application-agnostic performance modeling.
- We develop a regression-based metrics estimator to accurately predict the impact of scaling on application metrics.
- We present MLscale, a black-box autoscaler that provides near-optimal resource usage while reducing SLA violations without expert application knowledge or tuning.

Organization: The rest of the paper is organized as follows. To provide more context for this paper, we first present related work in Section 2. We then present MLscale in Section 3, including the design of the black-box neural network performance modeler and the regression-based metrics predictor. Our experimental setup is described in Section 4, followed by our experimental results in Section 5. We conclude the paper in Section 6.

2. Related work

Autoscaling has received significant attention from academia and industry, especially with the emphasis placed on sustainable computing and the emergence of cloud computing and its elastic resources. Most approaches to autoscaling are application-specific, and rely on expert knowledge of the application, or on exhaustive experimentations to derive such knowledge. Armed with this knowledge, reactive and predictive solutions are proposed for autoscaling; a survey of such existing solutions can be found in Lorido-Botrán et al. [13]. Such solutions are orthogonal to our application-agnostic MLscale approach and are not discussed here.

Recent approaches to autoscaling have leveraged ML to avoid the challenging and tedious performance modeling required for arbitrary complex applications. AGILE [1] uses online modeling to estimate the relationship between resource metrics and performance, thus making it application-agnostic. However, AGILE leverages polynomial curve fitting (using polynomials of degree up to 16) by first running controlled experiments at various resource intensities. AppRM [16] uses a regression model to approximate the non-linear relationship between resource allocation and application performance with a linear model. Our neural network based method is more generic than polynomial curve fitting or linear approximations as it can model nonlinearities in the system [17]. Gandhi et al. [18] employ machine learning to derive black-box performance models *specifically* for Hadoop workloads, and then leverage these models for autoscaling. SCADS [19] uses ML to offline build performance models for storage systems that can predict SLA violations. When the need for scaling is detected, a model-predictive controller is used to drive the scaling actions. In response to workload changes, the authors suggest periodically retraining the model. By contrast, MLscale is designed to be application-agnostic (neural network-based online modeling). Further, MLscale can automatically detect the need for retraining and online retrain its performance model (see Section 5.7). Gmach et al. [20] use fuzzy logic to derive autoscaling rules based on resource usage. While fuzzy logic is different from ML, it does provide similar black-box modeling ability. However, the initial set of rules must be provided, preferably by an expert, for fuzzy logic to work well.

A popular ML approach to autoscaling is reinforcement learning (RL). This approach learns the best action for a given system state (such as request rate or load specification) based on past experience and via trial-and-error. Dutreilh et al. [8] use RL to learn the scaling actions that minimize the sum of number of servers and violations. Similarly, Amoui et al. [21] and Padala et al. [22] employ variants of RL to learn the best actions for each state. Bahati and Bauer [9] also employ RL to adapt the action space of threshold-based autoscaling. URL [23] and VCONF [10] leverage RL to automatically configure various system parameter values, including the number of resources. One of the main drawbacks of RL is the time taken by the system to learn "good" actions. During this learning time, performance can be quite poor. To overcome this issue, Tesauro et al. [12] use queueing theory to initially manage the system, and then use RL, once enough training has been done, to control the system. While useful, RL is often not scalable given the large state space (exponential in the number of variables) that it has to maintain. In fact, RL is often integrated with neural networks to reduce its state space [11].

Elastisizer [24] uses black-box modeling to estimate the resource provisioning for a MapReduce job. Verma et al. [25] employ regression on resource and performance profiling for small data sets to derive scaling rules for larger data sets under Hadoop. These works focus on *statically* sizing the application prior to deployment. Gandhi et al. [18] employ machine learning to derive black-box performance models *specifically* for Hadoop workloads, and then leverage these models for autoscaling. By contrast, MLscale is designed to be application-agnostic (neural network-based online modeling), and is tailored for autoscaling (regression based post-scaling predictions).

ML has also been used to tune the parameters of application-specific models. Iqbal et al. [14] employ polynomial regression to model the number of servers in a tier as a function of the number of static and dynamic requests received by the RUBiS web application. Horvath et al. [15] use regression to model the relationship between response time and (only) CPU utilization, and server power and CPU utilization. These models are then used for energy-efficient application scaling. Lim et al. [26] employ multi-variate regression to model the impact of rebalancing data for elastic storage. Gandhi et al. [5,27] employ Kalman filters to estimate parameters of performance models that are not easily observable. However, the authors start with a generic queueing-theoretic model whose parameters are then derived online. By contrast, MLscale does not assume any prior application-specific knowledge or performance model.

Statistical techniques, such as regression [28], neural networks [29], and pattern matching [4], have also been employed to predict the request rate or load based on historical trends. However, as

mentioned in Section 1, application knowledge is still needed to translate predictions to resource requirements.

Machine learning has also been recently used to determine *where* an application VM should be placed to minimize resource contention [30,31]. By contrast, our focus here is on *when* and *how many* application VMs should be provisioned.

3. MLscale

We now present our approach, MLscale. The two important components of MLscale are: (i) an online neural network-based performance modeler (Section 3.1), and (ii) a regression-based post-scaling metrics predictor (Section 3.2). Together, the two components allow MLscale to accurately and efficiently autoscale the system (Section 3.3).

3.1. Black-box neural network performance modeling

Our goal here is to develop an *online* performance model that can accurately estimate performance, mean or tail response time in our case, as a function of observable metrics, such as request rate and resource utilizations. Importantly, our model should *not* rely on any expert application knowledge.

We employ ML for our performance modeling objective. In particular, we leverage neural networks (more specifically, a multi-layer feed-forward network) to model response time. Neural networks take a set of inputs, and then learn how to best combine them, using adaptive weights, to estimate the output that is close to the observed metric of interest (mean or tail response time). Neural networks offer a number of advantages, including requiring less formal statistical training and the ability to implicitly detect complex nonlinear relationships between dependent and independent variables [32]. A disadvantage is that they are prone to over-fitting, but this can be solved by using more training data [33]. For more details, we refer the readers to Haykin [17].

Inputs to our neural network model consist of application and average (across all nodes in a tier) system usage statistics. Specifically, our inputs are:

1. RR: Number of requests received per second,
2. CPU: CPU usage averaged across cores,
3. Inter: Number of interrupts generated per second,
4. CTXSW: Number of context switches per second,
5. KBIn: KB of data received per second,
6. PktIn: Number of packets received per second,
7. KBOut: KB of data sent per second, and
8. PktOut: Number of packets sent per second.

These metrics can be easily obtained online (see Section 4.4) and provide sufficient coverage of the application's resource consumption. In general, we can include other useful metrics depending on the environment. For example, we show in Section 5.8 how including Cycles Per Instruction as yet another input metric can be helpful in cloud environments that experience performance interference due to resource contention among colocated VMs. We use average statistics for our load-balanced homogeneous nodes, as opposed to per-node statistics, to reduce our state space. The neural network leverages these inputs and outputs estimates of response time; this gives us an accurate model for performance using training data collected online. We use a single hidden layer in our network since the Universal Approximation Theorem is well known for feed-forward networks with sigmoid functions [17].

An illustration of our neural network is shown in Fig. 1. Here, m_i represents the eight inputs enumerated above, and RT in the output layer refers to Response Time, our metric of interest. We employ

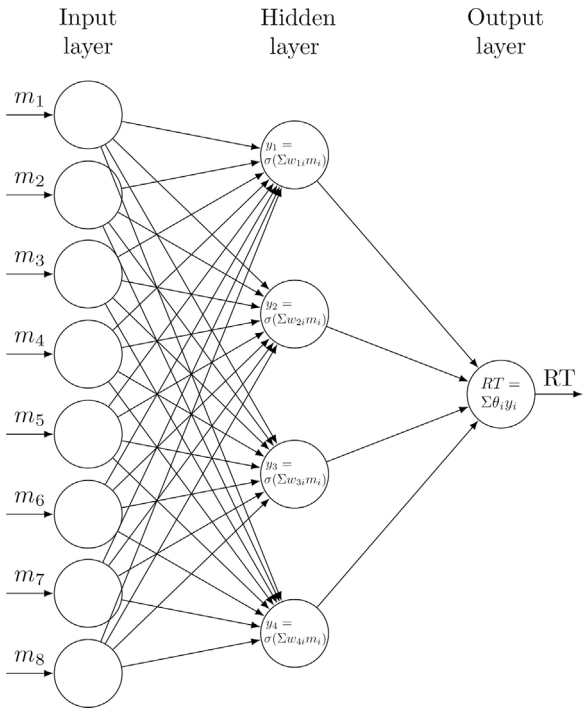


Fig. 1. Architecture of our neural network. Input layer consists of system and application metrics, the hidden layer applies the sigmoid activation function to a weighted sum of input values, and the output node applies a linear activation function to the weighted sum of values from hidden nodes.

four nodes in the hidden layer based on the heuristic of using the average of the number of input and output nodes [34]. The σ in the hidden nodes refers to the sigmoid function. Based on the network shown in Fig. 1, the output node (RT metric) is computed as follows:

$$RT = \theta_0 + \sum_{k=1}^{4} \theta_k \left(1 + \exp(-(w_{k,0} + \sum_{i=1}^{8} w_{k,i} \cdot m_i)) \right)^{-1}, \quad (1)$$

where the weights for transformation between input and hidden layers are represented by the matrix w, where $w_{k,i}$ is the weight for the kth hidden node and ith input node. Since we have 4 hidden nodes and 8 input nodes, k ranges from 1 to 4 and i ranges from 1 to 8 for the w matrix. $w_{k,0}$ is the weight of the bias node, which always takes a value of 1. θ_k are the weights for transformation between hidden and output layers. Since there is only one output unit, θ is a vector of length 4 (and not a matrix). The sigmoid activation function, ($\sigma(x) = \frac{1}{1+e^{-x}}$), is only applied in the hidden layer, while the output unit applies the linear activation function. Finally, θ_0 is the weight of the bias for the output unit.

Note that the network can be easily modified to include additional output variables such as estimates of tail response time or other metrics such as resource usage, power consumption, etc. The size of the hidden layer will have to be adjusted accordingly.

Training our neural network model, given the input training data, takes a few seconds. To obtain the training data, we run the application under a naive scaling policy, such as a response time threshold based scaling or CPU threshold based scaling [7], and vary the request rate to trigger scaling actions. We collect training data during this run consisting of average response time, request rate, current size of tier, and the average system metrics such as CPU usage, network usage, etc. This training data is then used for neural network training to learn response time as a function of system metrics and request rate. The test error for modeling the mean response time of our three applications using neural networks is about 6%; the test error for modeling the tail (95‰) response time

Table 1
Comparison of ML techniques for performance modeling.

Technique	Training time	Training error	Test error
LR	2.5 ms	9.2%	9.3%
SVR	1294 s	2.1%	6.2%
KR	8141 s	1.8%	8.2%
KNN-uniform	5 ms	5.0%	6.4%
KNN-distance	5 ms	0%	6.2%
Neural network	7.5 s	5.9%	6.0%

is about 8.2%. It is important that the training data cover a moderate range of request rates and several scaling actions. In general, the larger and richer the training data set, the lower will be the modeling error. The amount of input training data required for obtaining the above-mentioned error is a few hours worth of application run time.

We also compare neural network with other ML techniques that can be used for performance modeling. Specifically, we train models using non-parametric techniques such as Support Vector Regression (SVR), Kernel Regression (KR), and K-Nearest Neighbors (KNN), and also Linear Regression (LR), which is a parametric technique. Table 1 shows the training times, and training and test errors, across all applications, for mean response time modeling under these techniques. We use a 70–30% training and test split over the observations (see Section 4 for details on our experimental setup). While all techniques, except LR, have low modeling error, SVR and KR need substantial training time as both require cross-validation to set model parameters. KNN can be trained quickly, but has very high overhead as it keeps the entire training data in memory. Neural networks has low error, moderately low training time, and low overhead. We thus employ neural networks for MLscale's performance modeling component. Note that the other ML techniques we consider can still be useful in specific scenarios. For example, SVR and KR can be leveraged in cases where longer training time is acceptable, and KNN can be leveraged in cases where there is adequate capacity to hold the entire training data in memory.

3.2. Regression-based metrics prediction

The next logical step is to determine how the above performance model can be used to decide the resource scaling. Typically, the model is queried to determine the number of resources needed to achieve a response time below the SLA target. However, there is a subtle issue here. The model estimates response time based on *current* inputs (request rate and resource usage). To predict the response time *after* scaling, we need estimates of the post-scaling metrics. Unfortunately, the metrics are dependent on the resource scaling. For example, average CPU usage will likely drop after adding a new node, and thus we cannot use current, pre-scaling, estimates of CPU usage for predicting post-scaling response time.

To predict new estimates of input metrics after the proposed scaling action, we again use ML. In particular, we employ multiple linear regression to estimate the post-scaling metrics, m', as a function of the current metric value, m, current number of nodes employed in the target tier, w, and proposed number of additional nodes, k; note that k can take negative values to indicate scale-in. Mathematically, we have:

$$m' = c_0 + c_1 \cdot m \cdot w/(w+k) + c_2 \cdot m \cdot k/(w+k), \quad (2)$$

where c_0, c_1, and c_2 are regression coefficients that are derived via training. If we naively assume that the metrics scale perfectly with the number of nodes, that is, $m' = m \cdot w/(w+k)$, then we have $c_0 = 0$, $c_1 = 1$, and $c_2 = 0$. This "naive metric scaling" is often assumed in existing work for simple metrics such as request rate and CPU utilization [5,2,15], and motivates the structure of the specific terms used in Eq. (2). However, this naive scaling relationship is not very accurate in practice as CPU utilization and other metrics often depend on background activities as well. Further, even an idle node will have non-zero context switches and interrupts. Our regression coefficients account for such discrepancies.

We use a subset of the training data employed for neural network modeling; specifically, we consider data points immediately before and after a scaling action. Based on training, we find that c_1 ranges from 0.8 to 1 and c_2 ranges from 0.4 to 1.1 for different metrics. c_0 varies much more widely depending on the metric, ranging from near-zero for CPU utilization to the thousands for interrupts and context switches. Our test error for regression across all applications is a low 9% (across all metrics). The non-zero values for c_0 and c_2 validate the inaccuracy of naive metric scaling. We further illustrate the need for this regression-based metrics predictor via experiments in Section 5.5. Note that while we can use other ML techniques, such as neural networks, for this metrics prediction component, we find that the simple (low-overhead) linear regression technique, given by Eq. (2), provides acceptable prediction accuracy and a low training time of 1.6 ms for all 8 metrics. However, this was not the case under performance modeling (see LR in Table 1). We discuss results for other metrics prediction techniques in Section 5.6.

3.3. Putting it all together – MLscale

MLscale works by leveraging the neural network-based online performance modeler and regression-based metrics predictor. Once trained, the MLscale policy works by periodically monitoring request rate and resource usage metrics (obtained via collectl [38]), and using these monitored metrics as input to the trained performance model to estimate response time. We use a monitoring interval of ten seconds, over which the metrics are averaged. While ten second intervals worked well for most of the traces we employ, the interval length can be made longer or shorter based on how bursty the workload is expected to be.

We invoke autoscaling if both the observed *and* estimated response time exceed the SLA target. Note that we also rely on our performance model's estimate of response time to avoid responding to noisy measurements of application response time. In practice, we invoke scaling conservatively before the SLA target is violated; we scale-out when response time is within 10% of the SLA target. This is similar to the approach used by existing autoscaling techniques [1,5]. We initiate scale-in when the response time drops below 60% of the SLA target. These thresholds can be adapted as needed depending on the user's cost-performance tradeoff preferences. Our experimental results in Section 5 show that MLscale provides a balanced cost-performance tradeoff between resource usage and SLA violations.

To execute autoscaling, we first leverage the metrics predictor to predict post-scaling metrics for a proposed scaling action, and use these post-scaling metrics as input to the performance model to predict the response time after the proposed scaling. This allows us to determine the minimum scaling (k in Eq. (2)) required to maintain response times below the SLA target by evaluating scaling options around the current provisioning (w in Eq. (2)). Note that k can be greater than 1, thus allowing for arbitrary provisioning changes. We show in Section 5 that MLscale often adds/removes multiple nodes simultaneously in response to large variations in load.

Finally, we note that the underlying MLscale models can be easily retrained to adapt to changes in the workload. This is useful for workloads that are updated in real time (e.g., newer versions of an existing web page), or workloads that have different phases of

resource consumption. We discuss and evaluate model retraining in Section 5.7.

4. Experimental setup

MLscale can be employed in physical clusters or virtual clusters to save on operating costs and improve resource utilization. Our evaluation focuses on virtual clusters. We use an OpenStack-based private cloud (hosted on Dell C6100 servers with 12 cores and 48 GB each, connected via 1Gb links) and an AWS-based public cloud (m4.large and c4.large instances [39]) for our experiments. We create VMs on these clouds to host our applications. Unless otherwise specified, we report results and details for the OpenStack setup.

4.1. Applications

PHP-based web application. We set up a 10-VM (4 GB RAM, 2vCPU) tier of Apache-PHP web servers that each host a computationally-intensive microbenchmark. These VMs are behind an Apache load balancer [40], hosted on a different VM, that distributes incoming requests among the VMs in a round robin manner, and also allows enabling/disabling the web servers. Note that, because of autoscaling, the number of active web VMs will vary dynamically. We use httperf [41], on another VM, as our load generator with exponential inter-arrival times.

DayTrader. DayTrader [42] is an open source benchmark application emulating an online stock trading system. DayTrader is an end-to-end Java Enterprise Edition (J2EE) web application composed of several Java classes, Java Servlets, Web Services and Enterprise Java Beans, making it an ideal benchmark for evaluating the scalability and performance of a J2EE application server like IBM WebSphere Application Server (WAS). The TradeDatabase is hosted on DB2.

Our DayTrader deployment consists of 7 VMs. Four of these VMs (4 GB RAM, 2vCPU each) are application tier VMs running IBM WAS and have the DayTrader application deployed on them. The backend database VM (18 GB RAM, 4vCPU) runs DB2 using a ramdisk [43] for improved performance. Another VM serves as a front-end load-balancer running IBM HTTPServer (similar to Apache). The last VM acts as the client and simulates requests to the DayTrader application through iwlengine. The client supports several request classes, such as home, register, buy, sell, etc. Several of the request classes execute on both the WAS tier and the DB, and have dependencies between them.

PHP-MySQL application. We set up 3 database VMs (4 GB RAM, 4vCPU each), hosting exact replicas (for read-only) of 5 million records of stack exchange posts data [44]. An httperf VM (8 GB RAM, 4vCPU) generates requests that are directed to the PHP application tier VMs (4 GB RAM, 2vCPU each) via an Apache load balancer (8 GB RAM, 4vCPU). The application tier VMs generate a query for each received request; the query consists of read requests for 10 randomly selected records. Queries are sent to an HAproxy [45] (TCP) load balancer VM (8 GB RAM, 4vCPU), which selects one of 3 replicated database VMs to serve the 10 associated read requests. Load balancers use the (default) round robin policy.

4.2. Traces

We use various request traces from NLANR [35], ITA [36], and enterprise applications [37] for driving our application load. The specific traces we use for evaluation are shown in Fig. 2. We only show normalized request rate and trace length values as these are modified per the system capabilities.

4.3. Metrics

We focus on cost-performance tradeoffs. To this end, we use the percentage of response time violations over the monitoring intervals, **V**, as our performance metric. The response time SLA for the PHP web application is set to 120 ms, that for DayTrader is set to 10 ms, and that for PHP-MySQL is set to 60 ms. For DayTrader, we only focus on the "quotes" request class which makes up the majority of generated requests and accesses the WAS and DB nodes.

For cost, we consider the time-averaged number of nodes (VMs) employed over the entire experiment length, **N**, as our metric. N can be used as a proxy for the dollar cost of renting resources, or as a proxy for power usage in physical clusters. We also consider the number of scaling actions taken during the experiment, **C**, as a proxy for the wear and tear costs associated with scaling.

4.4. MLscale implementation

MLscale is implemented as a simple controller in python using a few hundred lines of code, and is hosted on the application's load balancer VM. MLscale leverages the ffnet library for implementing neural networks. The scaling action is executed by issuing directives to OpenStack or AWS and the application (specifically, the load balancer for each application) to add/remove the VMs. For monitoring, we use a 10 s interval to provide responsive autoscaling while avoiding hasty decision making. All VMs monitor resource statistics using the collectl [38] utility. Load balancers monitor the request rate and response time. MLscale collects all statistics periodically.

5. Evaluation results

We now present our evaluation results. We first discuss our evaluation methodology in Section 5.1. We then present our results for the PHP-based web application (Section 5.2), DayTrader (Section 5.3), and the PHP-MySQL application (Section 5.4). We end with additional results (Sections 5.5–5.8) that highlight MLscale's unique advantages, including the ability to retrain and the ability to autoscale in the presence of VM interference.

5.1. Experimental methodology

To evaluate MLscale, we consider the percentage of violations over the entire trace, V, and the time-averaged number of VMs employed, N. Unless otherwise specified, violations refer to mean response time SLA violations. For each trace, we compare MLscale with the following scaling policies:

Opt-Static. This is an unrealistic policy that knows the *exact* request rate ahead of time and statically provisions for the peak request rate over the entire trace, thus resulting in 0 violations. As expected, Opt-Static has high resource usage.

Opt-Dynamic. This is the ideal dynamic autoscaling policy that also knows the request rate ahead of time and dynamically provisions the system to incur 0 violations. We do not implement Opt-Dynamic, but instead model its resource usage by benchmarking the application to derive the peak throughput of a single VM, and use this information to estimate provisioning at each point in time, allowing for fractional resources, to obtain a lower bound. Similar approaches have been used in prior work [7] to emulate the optimal policy for comparison.

CPUscale. This policy works by setting upper and lower thresholds for scaling based only on monitored (10 s intervals) CPU utilization. Specifically, if the average monitored CPU utilization of the VMs, in the last 3 monitoring intervals, exceeds the upper threshold, a scale-out is initiated. Likewise, when the average falls below the lower threshold, a scale-in is initiated. CPUscale is representative of existing rule-based policies offered by cloud services

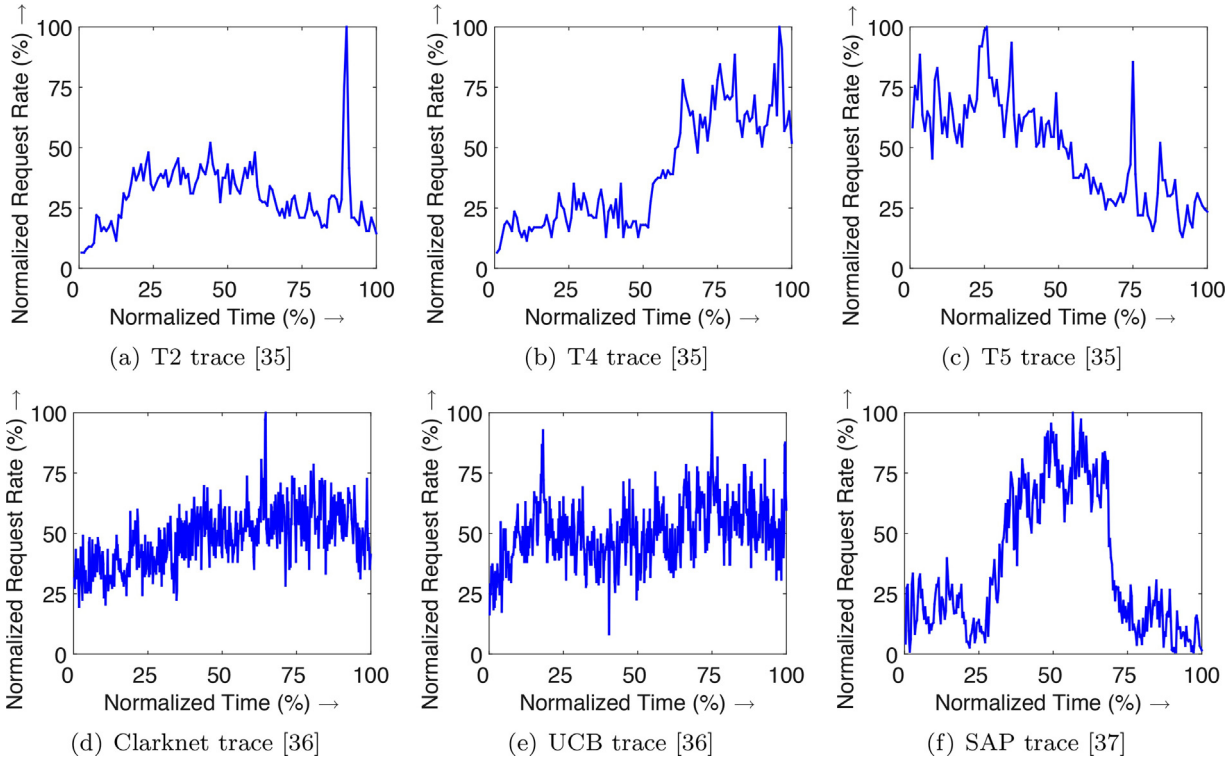


Fig. 2. Traces used in our experiments. *Y*-axis represents normalized request rate. *X*-axis represents normalized trace time.

such as AWS [46] and OpenStack [47]. In practice, we implement CPUscale and empirically find the best upper/lower thresholds for each trace.

5.2. Results for PHP-based web application

Fig. 3 shows the performance of our PHP web application under MLscale for all the six traces. The solid red line (*y*-axis on the right) shows the observed request rate and the blue dotted line (*y*-axis on the left) shows the observed response time. The green horizontal line denotes the mean response time target (120 ms). Note that the *y*-axis for response time does not start from 0; we set this intentionally so that it is easy to differentiate between the request rate and response time plots, which would otherwise overlap significantly.

In general, MLscale accurately reacts to load variations and autoscales accordingly, resulting in low violations. In particular, if we focus on trace T2 and T5, we see that MLscale leads to *very few violations* for both traces. This is because of the timely scaling actions, denoted by the black up and down triangles, representing scale-out and scale-in, respectively, in the figures; multiple coincident triangles represent addition/removal of *multiple VMs simultaneously*. Observe that the scaling actions are typically correlated with a sharp change in request rate. However, this is not always the case. For example, the variations in request rate for T2 between the 1 h and 2 h marks, and those for T5 between the 0.5 h and 1 h marks, do not necessitate scaling and lead to only a couple of violations. Also observe the *relatively stable provisioning* under MLscale which does not needlessly result in scaling actions. For example, the variations in load between the 2.5 h and 4 h marks for T5 result in only 1 provisioning change and no violations.

In some bursty cases, as in the Clarknet case, MLscale responds aggressively by scaling often, and adding or removing multiple VMs simultaneously to account for the steep load variations. As we show later, despite this aggressive scaling behavior, MLscale performs well in comparison to other policies for the bursty Clarknet trace.

Table 2
Comparison of policies for all traces under PHP application.

	MLscale		ΔN Opt-Static	ΔN Opt-Dynamic	Best CPUscale	
	V	*N*			*V*	*N*
T2	0.9%	2.0	59.8%	14.4%	1.8%	2.2
T4	2.1%	2.4	51.6%	14.9%	2.0%	2.8
T5	1.0%	2.7	46.0%	13.3%	1.8%	3.1
Clarknet	5.9%	5.0	49.7%	20.5%	3.4%	5.2
UCB	4.3%	5.8	42.3%	15.1%	3.5%	6.2
SAP	4.9%	4.7	53.1%	12.6%	3.6%	5.7

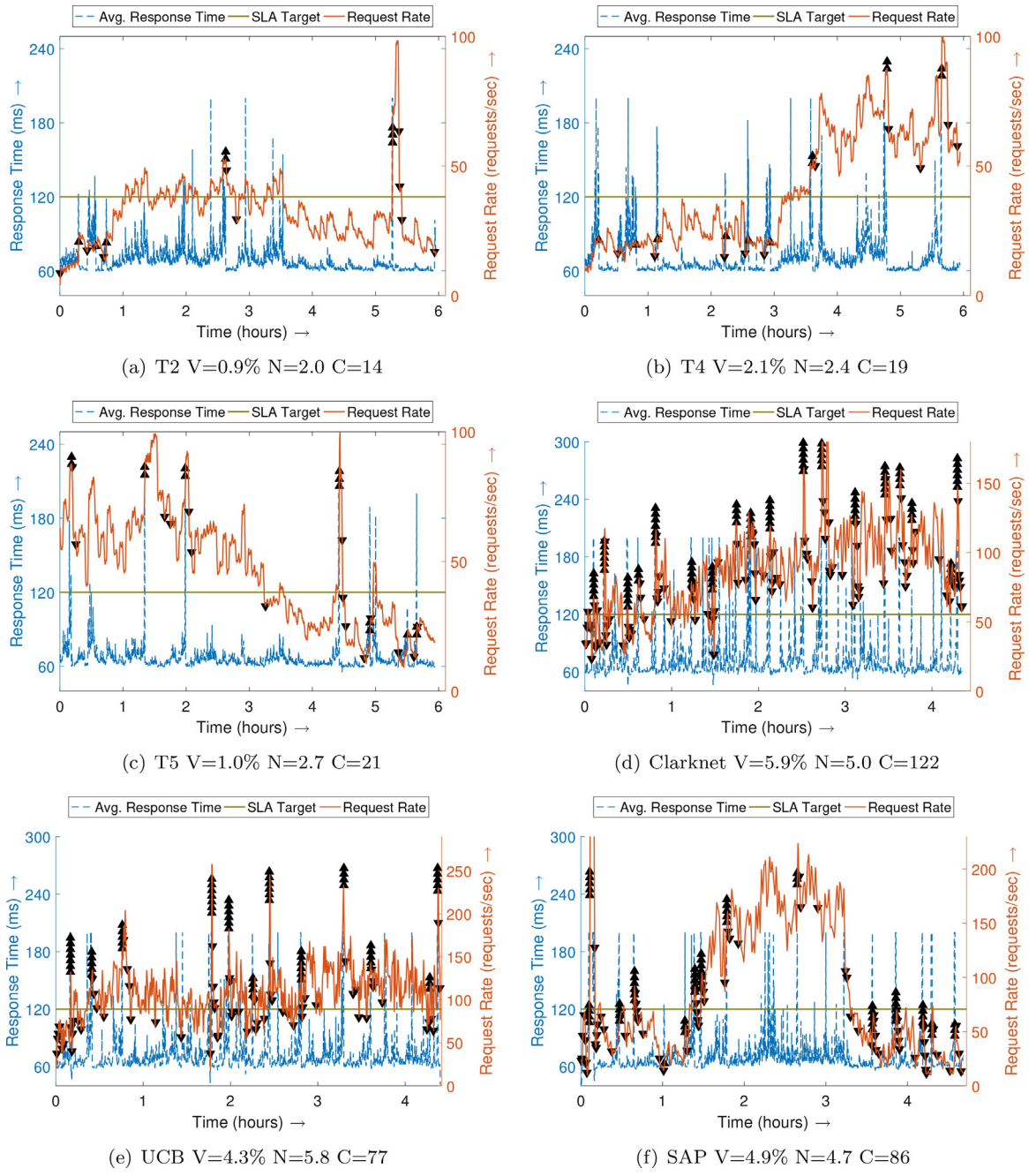
In practice, if it is known a priori that the request rate is bursty, MLscale could be tuned to react less aggressively by, for example, increasing the monitoring and decision interval length.

Full results for all traces and all policies are provided in Table 2. We see that MLscale typically incurs *less than 5%* violations. In the table, we omit *V* values for Opt-Static and Opt-Dynamic as these are 0. Instead, we report percentage reduction in *N* of MLscale over Opt-Static, ΔN Opt-Static, and percentage reduction in *N* afforded by Opt-Dynamic over MLscale, ΔN Opt-Dynamic. We use a cluster size of 5 VMs for the first three traces and 10 VMs for the last three traces. Of course, the resulting value of *N* will depend on the scaling policy and the trace; the cluster size represents the upper bound of *N*.

MLscale vs. Opt policies: Compared to Opt-Static, MLscale lowers resource costs by *at least 40%*. The reduction ranges from 42% under UCB to almost 60% under T2. This is expected as MLscale is dynamic in nature.

Compared to Opt-Dynamic, MLscale is typically within 15% of the resource usage of Opt-Dynamic, except for Clarknet. Recall that Opt-Dynamic knows the exact request rate ahead of time, which is often infeasible.

MLscale vs. CPUscale: The results in Table 2 for T2 and T5 traces show that MLscale is superior to the best CPUscale in terms of both performance violations and resource cost. For T2, the best CPUscale



(a) T2 V=0.9% N=2.0 C=14

(b) T4 V=2.1% N=2.4 C=19

(c) T5 V=1.0% N=2.7 C=21

(d) Clarknet V=5.9% N=5.0 C=122

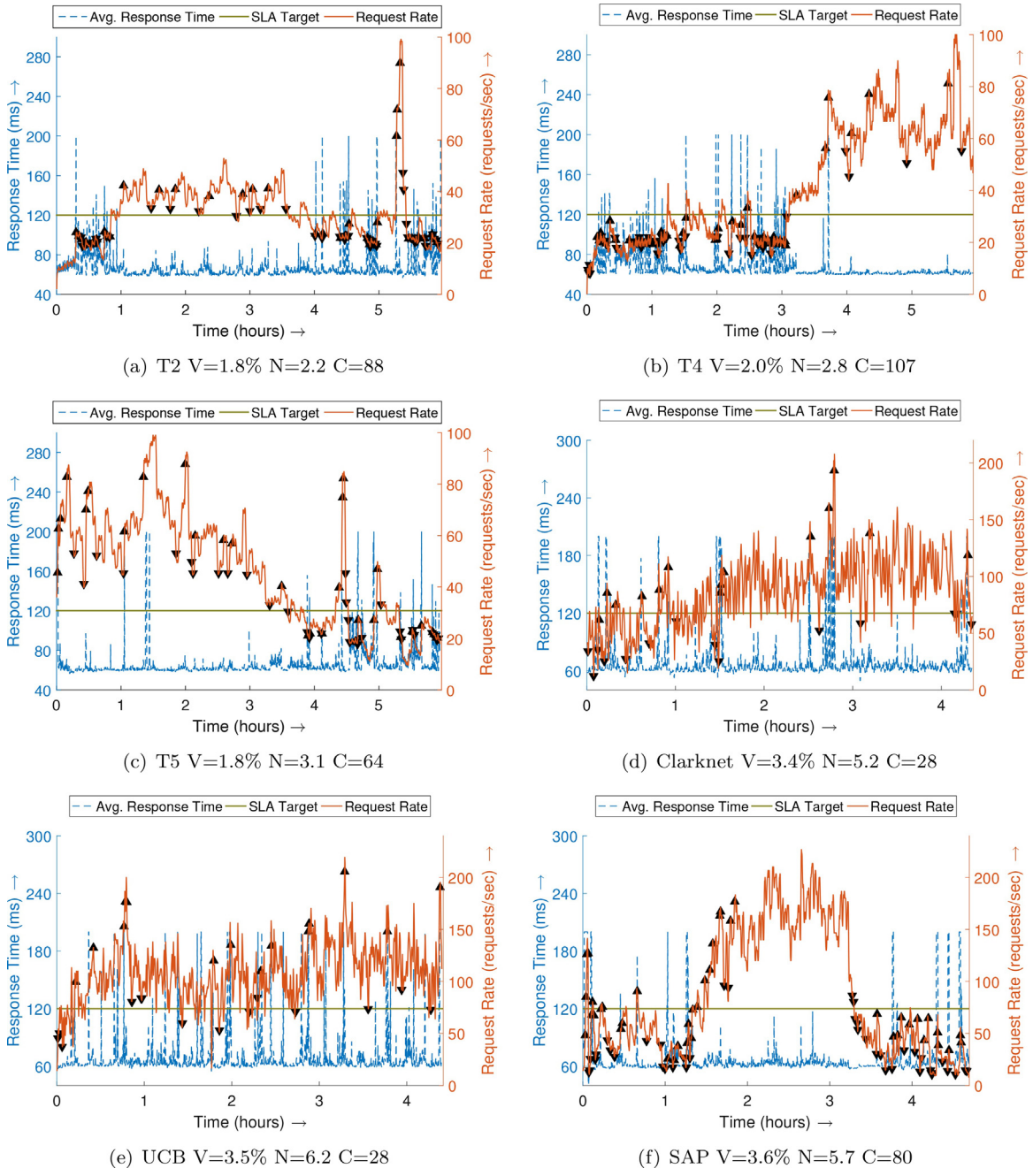
(e) UCB V=4.3% N=5.8 C=77

(f) SAP V=4.9% N=4.7 C=86

Fig. 3. All six traces under MLscale policy for PHP application. (For interpretation of the references to color in this figure citation, the reader is referred to the web version of this article.)

policy (with upper threshold of 55% and lower threshold of 35%) results in 1.8% violations and 2.2 resource cost, both marginally higher than MLscale; results are similar under T5. For T4, MLscale incurs almost similar violations but requires 14% fewer resources. For Clarknet, UCB, and SAP, the comparison is not straight forward. While the best CPUscale policy results in fewer violations, it does require more resources. The behavior of the best CPUscale policy for each trace is illustrated in Fig. 4.

However, we emphasize that while the *best* CPUscale policy provides qualitatively similar results to MLscale, it requires *exhaustive trial-and-error experiments* on *each* trace to choose the best upper/lower thresholds. For example, the best CPUscale policy provides lower violations at the expense of slightly higher resource costs compared to MLscale for Clarknet and UCB traces. However, converging on the best CPUscale policy required experimentation with 7 different pairs of upper/lower thresholds, for each of Clarknet and UCB. Further, the wide variance in performance for different threshold choices highlights the need for this experimentation – for Clarknet, N varied from 5.2 (for thresholds of 35–70%) to 8.9 (for thresholds of 35–55%, which were best for T2); similar variations were observed for V. Worse, the best choice is often *trace-dependent*, making it difficult to "learn" the best thresholds. Similar conclusions about the limitations of threshold-based policies were also made by prior work [5,8,1]. In summary, MLscale provides similar tradeoffs as the best CPUscale policy *without* requiring exhaustive experimentation on the exact workload *and* trace. While MLscale requires some training, this is a one-time effort that does not have to be repeated for each trace.



(a) T2 V=1.8% N=2.2 C=88

(b) T4 V=2.0% N=2.8 C=107

(c) T5 V=1.8% N=3.1 C=64

(d) Clarknet V=3.4% N=5.2 C=28

(e) UCB V=3.5% N=6.2 C=28

(f) SAP V=3.6% N=5.7 C=80

Fig. 4. All six traces under respective best CPUscale policy for PHP application. (For interpretation of the references to color in this figure citation, the reader is referred to the web version of this article.)

Table 3
Comparison of number of scaling actions for all traces under different policies for PHP application.

	MLscale C	Opt-Static C	Opt-Dynamic C	Best CPUscale C
T2	14	0	150	88
T4	19	0	179	108
T5	21	0	255	64
Clarknet	122	0	523	28
UCB	77	0	529	28
SAP	86	0	360	80

Wear and tear comparison: Table 3 shows total number of scaling actions taken during the trace execution for all six traces under the PHP application for different policies. The columns for Opt-Static and Opt-Dynamic are theoretically calculated assuming the same request rate as the actual experiment for MLscale. Opt-Static does not change the provisioning and therefore the C value is 0 for all cases. For Opt-Dynamic, the provisioning changes according to the observed request rate.

We see that MLscale requires far fewer scaling actions compared to Opt-Dynamic; the reduction in scaling action ranges from 76.1% (under SAP) to 91.8% (under T5). Compared to the best CPUscale policy, MLscale still provides significant reduction in scaling actions for the T2, T4, and T5 traces, with an average reduction of 78%. However, MLscale incurs more scaling actions for Clarknet, UCB, and SAP. Note that we are comparing with the best CPUscale policy, which is obtained via extensive trial-and-error.

Table 4
Comparison of MLscale vs. Best CPUscale policy using percentiles instead of averages of response time.

	MLscale		Best CPUscale	
	V	*N*	*V*	*N*
Clarknet	8.79%	3.0	9.9%	3.0
UCB	5.14%	3.5	7.0%	3.5

Provisioning stability: It is important to note the stability of the VM provisioning under MLscale. For the T2 and T5 traces shown in Fig. 3, MLscale performs 14 and 22 scaling actions, respectively (some actions result in multiple VMs being added/removed). By contrast, the best CPUscale policy performs 88 and 64 scaling actions. This unstable provisioning behavior of the CPUscale policy is illustrated in Fig. 4 for the traces. The provisioning under Opt-Dynamic is even more unstable – 150 and 255 scaling actions for T2 and T5, respectively. This instability stems from the need to react to the current system state without understanding the post-scaling implications. For example, the scaling actions in Fig. 4 typically occur in pairs of scale-out followed quickly by a scale-in. This suggests an oscillatory behavior where the CPU utilization upper threshold is violated, resulting in a scale-out, which then violates the lower threshold, resulting in a scale-in. By contrast, MLscale's metrics predictor can estimate the system state after the proposed scaling action, and can thus reduce some of this instability. A stable provisioning is preferred to reduce any overhead, such as the boot up time or wear-and-tear associated with adding/removing nodes [7].

95‰ response time targets: MLscale can be easily extended to model tail response times, such as 95‰ response times. We evaluate MLscale for the Clarknet and UCB traces under a 95‰ response time target of 120 ms. We use the AWS public cloud setup for these experiments. Table 4 shows our experimental results for Clarknet and UCB traces when MLscale uses percentiles instead of average response time to drive the scaling decisions. MLscale provides 11.2% lower violations than the best CPUscale (thresholds of 10–60%) for Clarknet while consuming the same amount of resources. For UCB, MLscale provides 26.7% lower violations than the best CPUscale (thresholds of 15–60% this time), at the expense of 1.7% more resources. Note that these experiments are different from the ones presented before as the underlying setup is different and the scaling trigger is now response time percentiles. The training data for percentile based scaling is also different from the one used for average response time based scaling.

5.3. Results for DayTrader

DayTrader is a considerably more complex application than our PHP web application. It is multi-tiered, and has several request classes. In this evaluation, we only focus on autoscaling the application tier as scaling the stateful database tier which is subject to reads and writes is a much harder problem [48] that is beyond the scope of this paper. We do, however, autoscale a read-only database tier in Section 5.4.

Fig. 5 shows the performance of DayTrader under MLscale for all six traces. We see that the percentage of SLA violations, *V*, is very low under MLscale. Also note the correlation between scaling actions and change in request rate. While this might suggest that request rate-based autoscaling strategies should work well for such traces, prior work [7] has shown that this is not the case. This is because the number of VMs in the tier affects performance due to communication overheads, thus lowering a VM's efficiency. Further, DayTrader employs a closed-loop load generator with a fixed number of clients; under this load generation model, request rate actually *decreases* as performance degrades, as each client request takes longer to complete, and the next request for each client is only generated once the previous once completes. Thus, request rate-based autoscaling would incorrectly scale-in VMs when performance degrades. MLscale does not rely only on request rate or CPU utilization, and can thus avoid incorrectly reacting to closed-loop request generation. Note that the provisioning under MLscale is also stable for DayTrader.

Table 5
Comparison of MLscale and Opt policies for all traces under the DayTrader application.

	MLscale			ΔN Opt-Static	ΔN Opt-Dynamic
	V	*N*	*C*		
T2	0.5%	1.9	6	53.0%	1.1%
T4	1.5%	1.9	3	51.8%	0.0%
T5	1.3%	3.1	8	22.5%	0.7%
Clarknet	0.9%	2.4	6	40%	0.0%
UCB	3.0%	1.5	6	63.5%	−0.7%
SAP	0.8%	2.1	9	48.0%	1.4%

Table 6
Comparison of MLscale and Opt policies for all traces under the PHP-MySQL application.

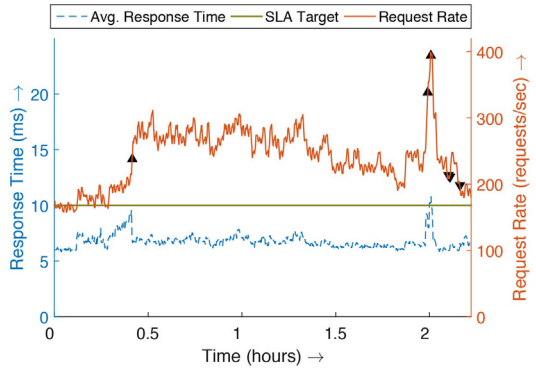
	MLscale			ΔN Opt-Static	ΔN Opt-Dynamic
	V	*N*	*C*		
T2	1.27%	1.95	10	35.0%	34.5%
T5	1.69%	2.27	17	24.3%	33.5%
UCB	2.14%	1.8	12	10.0%	45.2%

Table 5 shows the results of MLscale for all traces, and also compares its cost with the optimal approaches. We see that violations are very low for DayTrader under MLscale, often *less than 2%*. In terms of cost, MLscale again *lowers resource cost by about 50%* when compared to Opt-Static. Interestingly, for DayTrader, MLscale is typically *within 1%* of the cost of Opt-Dynamic. In fact, MLscale consumes fewer resources than Opt-Dynamic for the UCB trace, but at the expense of 3% violations. This highlights MLscale's provisioning efficacy and, combined with the low violations, shows that MLscale's autoscaling is *near-optimal* for DayTrader.

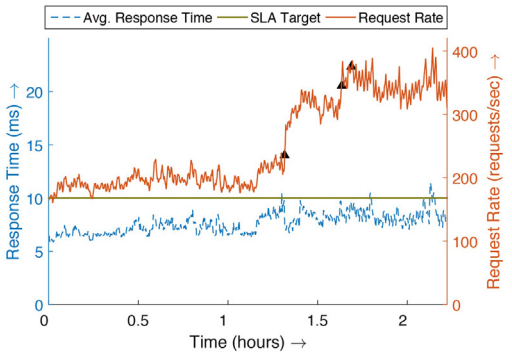
5.4. Results for PHP-MySQL application

We now present results for the PHP-MySQL application where we autoscale the database tier. Scaling the database tier is considerably more complex because of data consistency [48]; we thus consider a read-only database hosted on a VM, and add/remove exact replicas of the database VM to scale throughput according to workload demand. For this application, we experiment with the T2, T5, and UCB traces.

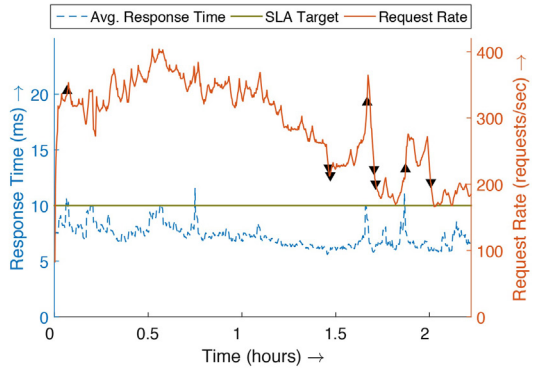
Fig. 6 shows the performance of our PHP-MySQL application under MLscale for the T2, T5 and UCB traces. When focusing on the T5 trace, we see that MLscale results in only a few violations (1.7%) despite the significant demand variations. In the first hour of the trace, MLscale autoscales 15 times, and does result in violations; this is likely because of the very erratic load variations as shown in the request rate plot (red solid line) in Fig. 6(b). However, in the second hour, MLscale autoscales only 2 times, though there are still considerable load variations. Note that there are very few violations in the second hour. This shows that MLscale correctly chooses to not autoscale the system too often in the second hour. This is because of the metrics predictor component of MLscale that can predict post-scaling response time, and can thus avoid unnecessary provisioning changes. We see a similar behavior for the UCB trace. Across the three traces, we see that MLscale incurs very few violations.



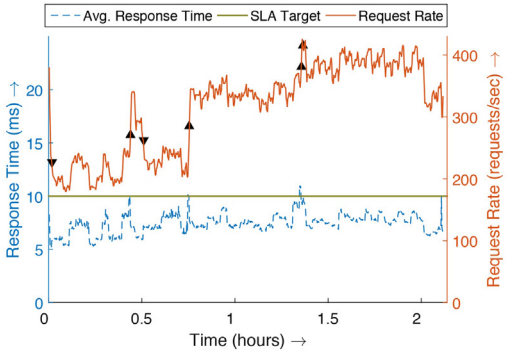
(a) T2 V=0.5% N=1.9 C=6



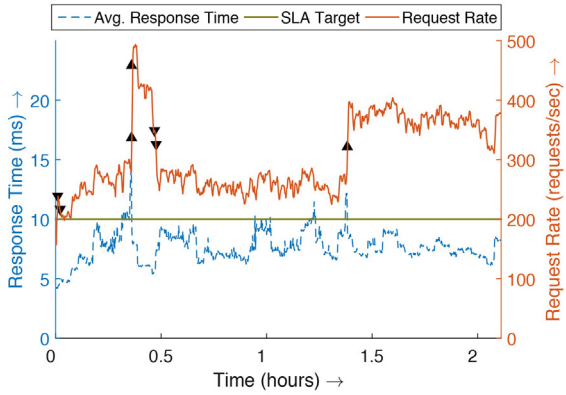
(b) T4 V=1.5% N=1.9 C=3



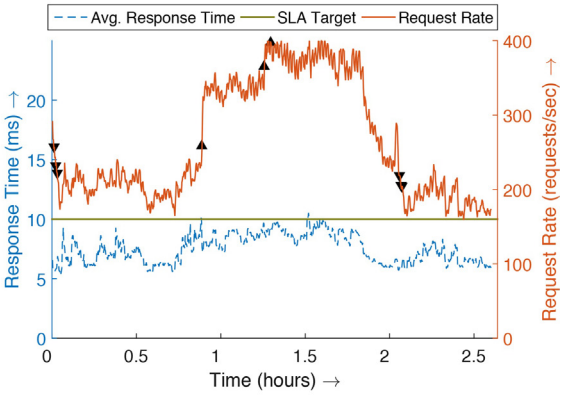
(c) T5 V=1.3% N=3.1 C=8



(d) Clarknet V=0.9% N=2.4 C=6

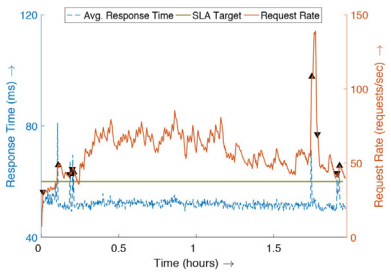


(e) UCB V=3.0% N=1.5 C=6

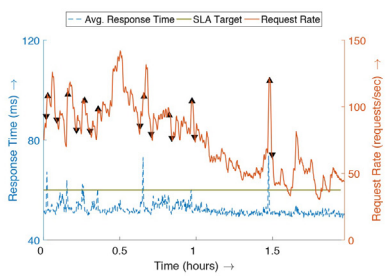


(f) SAP V=0.8% N=2.1 C=9

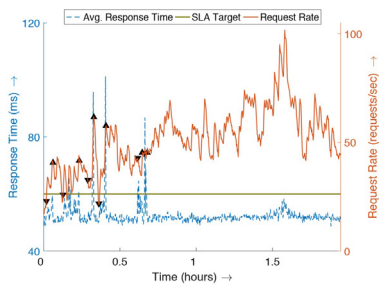
Fig. 5. All six traces under MLscale policy for DayTrader application. (For interpretation of the references to color in this figure citation, the reader is referred to the web version of this article.)



(a) T2 V=1.27% N=1.95 C=10



(b) T5 V=1.69% N=2.27 C=17



(c) UCB V=2.14% N=1.8 C=12

Fig. 6. T2, T5 and 2 h-UCB under MLscale policy for PHP-MySQL application. (For interpretation of the references to color in this figure citation, the reader is referred to the web version of this article.)

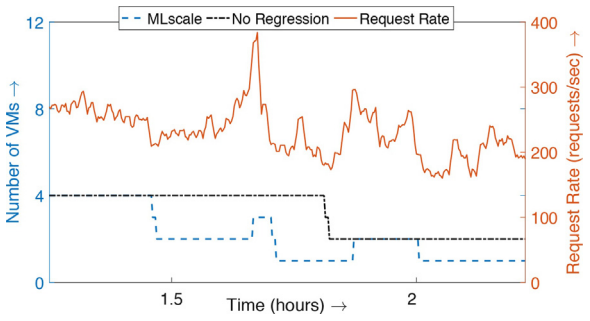


Fig. 7. Performance for T5 trace with and without the regression-based metrics predictor under DayTrader. Resource cost increases by 16% and violations increase by 23% without the regression-based metrics predictor.

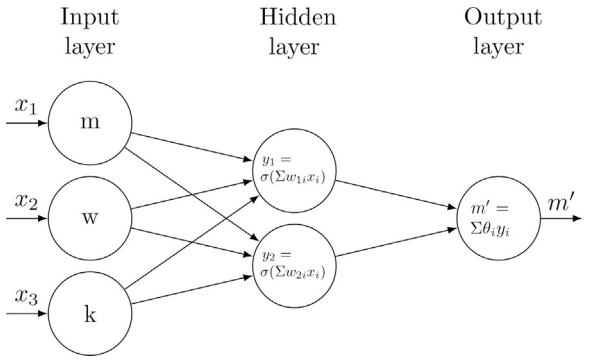


Fig. 8. Neural network for post-scale metric prediction. Bias nodes for input and hidden layer not shown here. One network is trained for each metric type.

Table 6 summarizes the results of MLscale for all traces, and also compares its cost with that of the optimal policies, for the PHP-MySQL application. Compared to Opt-Static, MLscale lowers resource costs by about 23%. Further, MLscale is within 38%, on average, of the cost of Opt-Dynamic. While there is room for improvement in this case compared to Opt-Dynamic, MLscale chooses to be conservative in terms of resource usage since the database performance degrades considerably as CPU utilization increases. In fact, for the database tier, we find that the CPU usage under MLscale is typically under 50%.

5.5. Importance of metrics predictor

The metrics predictor component of MLscale is critical for the quality of autoscaling decisions. Prior work typically implicitly assumes naive metric scaling (see Section 3.2), which says that metrics, such as per-VM CPU utilization and network activity, scale exactly with the number of VMs currently active. This incorrect assumption can negatively impact resource costs and performance. By contrast, our metrics predictor considers a more complex relationship, as given by Eq. (2), which improves cost and performance. Fig. 7 shows the performance for DayTrader under the T5 trace with and without the regression based metrics predictor; we focus on the last 40 minutes in the trace and show the observed request rate and the number of VMs employed under each technique. Note that the "no regression" refers to naive metric scaling. We see that the VM provisioning under MLscale (blue dashed line) is responsive to changes in the request rate (orange solid line). By contrast, without regression (black dash-dot line), the VM provisioning is not as responsive to changes in the workload. As a result, the resource usage, N, under no regression is 3.6 compared to 3.1 under MLscale, an increase of 16%. Similarly, the violations, V, under no regression is 1.6% compared to 1.3% under MLscale, an increase of 23%.

5.6. Post-scale metric prediction comparison

Thus far, we used a regression based model to predict the metric values after a scaling action is taken. We now discuss the other models we tried and the reason for choosing linear regression.

Apart from regression, which uses the same model as Eq. (2), we also experiment with using neural networks for metric prediction. We first consider a different neural network for each output metric and evaluate different hidden layer sizes. We then consider a neural network with multiple outputs to predict all metrics using a single network. In all cases, the input variables are the same: current metric value m, current size of tier w, and the scaling action k. The 8 different output metrics are RR, CPU, Inter, CTXSW, KBIn, PktIn, KBOut, and PktOut, as described in Section 3.1.

For the case of a different neural network for each output metric, we first evaluate the test error for metric prediction based on different sizes of the hidden layer. In general, as the size of the hidden layer increases, the computation time during training also increases. The total training time under regression for all 8 single-output networks in our case is about 1.6 ms, or about 0.2 ms per network; this is in agreement with the training time reported in Section 3.2. Compared to regression, neural network with 2 hidden units takes about 135× longer to train. Table 7 shows the comparison of test error for metric prediction under regression vs. neural network with 2 hidden units (size of hidden layer computed using the average of the number of input and output units). As can be seen, the performance of regression is comparable to the 2-unit neural network and sometimes better as in the case of RR, CPU, CTXSW and Inter. On average, regression results in a 11.4% improvement in test error over the 2-unit neural network.

Fig. 8 shows the neural network architecture that we use for individual post-scale metric prediction. This network only has 3 input nodes, which are the current metric value m, the current size of the tier w, and the scaling action k. The bias nodes are also added to the input and hidden layer which have not been shown in the figure. Based on this, the post-scale metric value prediction, m', is computed as:

$$m' = \theta_0 + \sum_{k=1}^{2} \theta_k \left(1 + \exp(-(w_0 + \sum_{i=1}^{3} w_{k,i} x_i)) \right)^{-1} \quad (3)$$

Similar to Eq. (1), w is the weight matrix for the hidden layer and θ is the weight vector for the output node. i varies from 1 to 3 for the three input nodes and k varies from 1 to 2 for the two hidden nodes. Sigmoid activation function is only applied in the hidden layer and the output node applies the linear activation function. w_0 and θ_0 are the bias weights for hidden and output layers, respectively.

We next increase the size of the hidden layer to 5 and 8 hidden units, resulting in an average test error of 8.2% and 7.9% respectively, as compared to a test error of 10.5% for regression; however, with 5 and 8 hidden units, the training time is significantly higher. In particular, the training time for 5 and 8 hidden units is 4× and 10× higher than that for 2 hidden units, respectively. When compared with linear regression, neural network with 5 and 8 hidden units takes 563× and 1287× longer to train, respectively.

We also tried a neural network with multiple output units to predict all the metrics using a single network. This network takes in all the current metric values, in addition to w and k, and outputs all the predicted metric values after the scaling action is taken. We experiment with 5, 8 and 10 hidden units, resulting in an average test error of 7.5%, 6.6% and 3.4%, respectively. Interestingly, the multi-output network can model the dependencies between the metrics themselves, and therefore provides better overall performance. However, the training time for this complex network is much higher; for the case of this multi-output network with 5,

Table 7
Comparison of test errors between regression and neural network for post-scale metric prediction.

	RR	CPU	Inter	CTXSW	KBIn	PktIn	KBOut	PktOut
Regression	9.37%	9.11%	6.85%	11.9%	10.15%	14.56%	9.84%	12.54%
Neural network	11.58%	13.84%	9.77%	12.03%	9.7%	12.99%	13.44%	11.77%

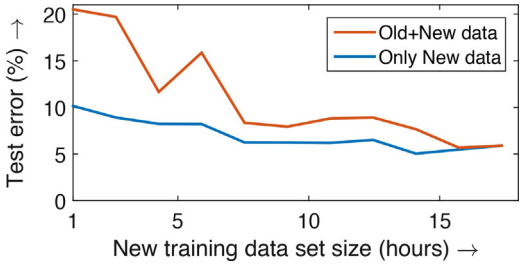


Fig. 9. Performance modeling retraining results for PHP application. (For interpretation of the references to color in this figure citation, the reader is referred to the web version of this article.)

8, and 10 hidden units, it takes about 875×, 1790×, and 2495× longer, respectively, to train the network when compared to the total training time of all 8 linear regression models.

5.7. Model retraining

MLscale can adapt to changes in the workload. Specifically, the modeling and metrics prediction components of MLscale can adapt to changes in the workload. This is useful for workloads that are updated in real time (e.g., newer versions of an existing web page), or workloads that have different phases of resource consumption.

To make MLscale adaptive, we maintain a moving window of the average modeling and prediction error. If these errors exceed a certain threshold, say 10%, then we online retrain the models. The size of the window depends on the workload variability as we do not want to retrain needlessly due to minor fluctuations in the system. For retraining, we use the collected metrics data since the need for retraining was detected (error exceeding threshold). While the retrained model error reduces with the size of the new data, we find that 1–2 h of data is sufficient to achieve less than 10% error.

Fig. 9 shows the modeling error for the retrained neural network online model; results are similar for the metrics prediction model. Here, we use the PHP web application and change the workload during runtime to make it less CPU intensive. We see that the modeling error (blue line) decreases as the new training data set size increases. Interestingly, augmenting the new data with older data worsens the model error (orange line) since the old data refers to the unchanged workload, whose dynamics are different from the new workload.

Note that, in addition to modeling error, the resulting violations (V) also go up if the model is not retrained. For the PHP web application, under SAP, Clarknet, and UCB traces, the violations go up by almost 5× if we do not retrain the model after the workload change. Further, if we only retrain the performance model (and not the regression based metrics predictor), then the violations still go up by around 3×. This simple example illustrates the need for retraining and also the need for only selecting new data for retraining.

5.8. MLscale in the presence of interference

In a cloud environment, when multiple VMs are co-located on the same physical host, performance interference can be caused due to resource contention [49,50]. In this section, we briefly explore how MLscale can be augmented to account for interference.

Detecting and measuring interference in VMs is a challenge in itself; however, the *Cycles Per Instruction* (CPI) metric is often used as a measure of CPU contention [51,52]. We can thus augment MLscale to also include CPI as another metric to our neural network model. For evaluation, we use a physical host with 12 cores in our OpenStack setup. We launch 5 VMs on this host, each requiring 4vCPUs (overcommit is enabled in our OpenStack). We use 4 of these VMs as background load and one as our application VM. We use the *stress* workload generator [53] on the background VMs to create CPU contention, and use httperf to generate load for our foreground application VM. We find that the inclusion of CPI as an input to our neural network reduces test error by about 7% compared to the default neural network model of MLscale that does not take CPI into account. This suggests the potential applicability of MLscale for autoscaling in the presence of interference.

6. Conclusion

We present MLscale, an application-agnostic autoscaler that requires minimal application knowledge and manual tuning. MLscale employs neural networks to online build the application performance model, and then leverages multiple linear regression to predict the post-scaling state of the system. This combination enables accurate autoscaling under MLscale. Experimental results on an OpenStack and AWS cloud cluster with several applications and under several request traces demonstrate the efficacy of MLscale. Comparisons with the optimal static and dynamic scaling approaches highlight the near-optimality of MLscale. The metrics predictor is an important component that helps exploit black-box modeling methodologies by predicting the impact of an action on the system state; the stable provisioning under MLscale highlights this advantage. Further, MLscale can dynamically adapt its performance model in response to workload changes through retraining.

Our end-goal with this research is to develop an application-agnostic autoscaler for use in cloud computing. We envision such an autoscaler to be employed by the cloud service provider to offer superior autoscaling solutions for cloud users. Current rule-based autoscaling solutions employed by cloud providers such as Amazon [46] and Rightscale [54] require the user to set up the scaling rules. With MLscale, the user can be relieved of this burden.

Acknowledgment

This research was supported by NSF grants 1464151, 1622832, and 1617046.

References

[1] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, J. Wilkes, AGILE: elastic distributed resource scaling for infrastructure-as-a-service, in: ICAC 2013, San Jose, CA, USA, 2013, pp. 69–82.
[2] B. Urgaonkar, A. Chandra, Dynamic provisioning of multi-tier Internet applications, in: ICAC 2005, Seattle, WA, USA, 2005, pp. 217–228.
[3] A. Gandhi, T. Zhu, M. Harchol-Balter, M. Kozuch, SOFTScale: scaling opportunistically for transient scaling, in: Middleware 2012, Montreal, Quebec, Canada, 2012, pp. 142–163.
[4] Z. Gong, X. Gu, J. Wilkes, PRESS: PRedictive Elastic ReSource Scaling for cloud systems, CNSM 2010 (2010) 9–16.
[5] A. Gandhi, P. Dube, A. Karve, A. Kochut, L. Zhang, Adaptive, model-driven autoscaling for cloud applications, in: ICAC 2014, Philadelphia, PA, USA, 2014, pp. 57–64.

[6] X. Liu, L. Sha, Y. Diao, S. Froehlich, J.L. Hellerstein, S. Parekh, Online response time optimization of apache web server, in: IWQoS 2003, Berlin, Germany, 2003, pp. 461–478.
[7] A. Gandhi, M. Harchol-Balter, R. Raghunathan, M. Kozuch, AutoScale: dynamic, robust capacity management for multi-tier data centers, Trans. Comput. Syst. 30 (2012).
[8] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, I. Truck, Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow, in: ICAS 2011, Venice, Italy, 2011, pp. 67–74.
[9] R. Bahati, M. Bauer, Towards adaptive policy-based management, in: NOMS 2010, Osaka, Japan, 2010, pp. 511–518.
[10] J. Rao, X. Bu, C.-Z. Xu, L. Wang, G. Yin, VCONF: a reinforcement learning approach to virtual machines auto-configuration, ICAC 2009 (2009) 137–146.
[11] K.O. Stanley, R. Miikkulainen, Efficient reinforcement learning through evolving neural network topologies, in: GECCO 2002, San Francisco, CA, USA, 2002, pp. 569–577.
[12] G. Tesauro, N. Jong, R. Das, M. Bennani, A hybrid reinforcement learning approach to autonomic resource allocation, in: ICAC 2006, Dublin, Ireland, 2006, pp. 65–73.
[13] T. Lorido-Botrán, J. Miguel-Alonso, J.A. Lozano, Auto-Scaling Techniques for Elastic Applications in Cloud Environments, University of the Basque Country, Tech. Rep. EHU-KAT-IK-09-12, 2012.
[14] W. Iqbal, M.N. Dailey, D. Carrera, P. Janecek, Adaptive resource provisioning for read intensive multi-tier applications in the cloud, Future Gen. Comput. Syst. 27 (6) (2011) 871–879.
[15] T. Horvath, K. Skadron, Multi-mode energy management for multi-tier server clusters, in: PACT 2008, Toronto, ON, Canada, 2008, pp. 270–279.
[16] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, E. Smirni, Application-driven dynamic vertical scaling of virtual machines in resource pools, in: NOMS 2014, Krakow, Poland, 2014, pp. 1–9.
[17] S. Haykin, Neural Networks: A Comprehensive Foundation, 3rd ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.
[18] A. Gandhi, P. Dube, A. Kochut, L. Zhang, S. Thota, Autoscaling for hadoop clusters, in: IC2E 2016, Berlin, Germany, 2016.
[19] B. Trushkowsky, P. Bodik, A. Fox, M.J. Franklin, M.I. Jordan, D.A. Patterson, The SCADS director: scaling a distributed storage system under stringent performance requirements, in: FAST 2011, San Jose, CA, USA, 2011, pp. 163–176.
[20] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, A. Kemper, Adaptive quality of service management for enterprise services, ACM Trans. Web 2 (2008) 1–46.
[21] M. Amoui, M. Salehie, S. Mirarab, L. Tahvildari, Adaptive action selection in autonomic software using reinforcement learning, in: Proceedings of the 4th International Conference on Autonomic and Autonomous Systems, Gosier, Guadeloupe, 2008, pp. 175–181.
[22] P. Padala, K.-Y. Hou, K.G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, Automated control of multiple virtualized resources, in: Proceedings of the 4th ACM European Conference on Computer Systems, Nuremberg, Germany, 2009, pp. 13–26.
[23] C.-Z. Xu, J. Rao, X. Bu, URL: a unified reinforcement learning approach for autonomic cloud management, J. Parallel Distrib. Comput. 72 (2) (2012) 95–105.
[24] H. Herodotou, F. Dong, S. Babu, No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics, in: SOCC 2011, Cascais, Portugal, 2011, 18:1–18:14.
[25] A. Verma, L. Cherkasova, R.H. Campbell, Resource provisioning framework for MapReduce jobs with performance goals, in: Middleware 2011, Lisbon, Portugal, 2011, pp. 160–179.
[26] H.C. Lim, S. Babu, J.S. Chase, Automated control for elastic storage, in: ICAC 2010, Washington, DC, USA, 2010, pp. 1–10.
[27] A. Gandhi, P. Dube, A. Karve, A. Kochut, L. Zhang, Modeling the impact of workload on cloud resource scaling, in: SBAC-PAD 2014, Paris, France, 2014.
[28] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, D. Patterson, Statistical machine learning makes automatic control practical for internet datacenters, in: HotCloud 2009, San Diego, CA, USA, 2009.
[29] S. Islam, J. Keung, K. Lee, A. Liu, Empirical prediction models for adaptive resource provisioning in the cloud, Future Gen. Comput. Syst. 28 (1) (2012) 155–162.
[30] R. Chiang, H. Huang, TRACON: interference-aware scheduling for data-intensive applications in virtualized environments, in: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Ser. SC'11, Seattle, WA, USA, 2011, pp. 1–12.
[31] C. Delimitrou, C. Kozyrakis, Quasar: resource-efficient and QoS-aware cluster management, in: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, Ser. ASPLOS'14, Salt Lake City, UT, USA, 2014, pp. 127–144.
[32] D. Specht, A general regression neural network, IEEE Trans. Neural Netw. 2 (6) (1991) 568–576.
[33] P. Cunningham, J. Carney, S. Jacob, Stability problems with artificial neural networks and the ensemble solution, Artif. Intell. Med. 20 (3) (2000) 217–225.
[34] J. Heaton, Introduction to Neural Networks for Java, 2nd ed., Heaton Research Inc., 2008.
[35] NLANR Anonymized Access Logs. //ftp.ircache.net/Traces.
[36] ITA. http://ita.ee.lbl.gov/index.html, http://ita.ee.lbl.gov/index.html.
[37] SAP, SAP Application Trace from Anonymous Source, 2011.
[38] Collect. http://collectl.sourceforge.net.
[39] Amazon EC2 Instances. http://aws.amazon.com/ec2/instance-types.
[40] Apache HTTP SERVER PROJECT. https://httpd.apache.org.
[41] httperf. http://www.labs.hpe.com/research/linux/httperf.
[42] DayTrader. http://geronimo.apache.org/GMOxDOC20/daytrader.html.
[43] M. Nielsen, How to use a RAMdisk, Linux Gazette 44 (1999).
[44] Stack Exchange Data Dump. https://archive.org/details/stackexchange.
[45] HAproxy: TCP/HTTP Load Balancer. http://www.haproxy.org.
[46] Amazon, Amazon Auto Scaling. http://aws.amazon.com/autoscaling.
[47] Openstack.org, OpenStack Heat. https://wiki.openstack.org/wiki/Heat.
[48] E. Thereska, A. Donnelly, D. Narayanan, Sierra: practical power-proportionality for data center storage, in: EuroSys 2011, Salzburg, Austria, 2011, pp. 169–182.
[49] D. Novaković, N. Vasić, S. Novaković, D. Kostić, R. Bianchini, DeepDive: transparently identifying and managing performance interference in virtualized environments, Proceedings of the 2013 USENIX Conference on Annual Technical Conference, Ser. USENIX ATC'13 (2013) 219–230.
[50] A. Javadi, S. Mehra, B. Vangoor, A. Gandhi, UIE: user-centric interference estimation for cloud applications, in: Proceedings of the 2016 IEEE Internation Conference on Cloud Engineering (Work-in-Progress Track), Ser. IC2E'16, Berlin, Germany, 2016.
[51] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, J. Wilkes, CPI^2: CPU performance isolation for shared compute clusters, in: Proceedings of the 8th European Conference on Computer Systems, Ser. EuroSys'13, Prague, Czech Republic, 2013, pp. 379–391.
[52] A. Maji, S. Mitra, S. Bagchi, ICE: an integrated configuration engine for interference mitigation in cloud services, in: Proceedings of the 2015 IEEE International Conference on Autonomic Computing, Ser. ICAC'15, Grenoble, France, 2015, pp. 91–100.
[53] A. Waterland, Stress. http://people.seas.harvard.edu/apw/stress.
[54] RightScale, Inc., Auto-Scaling Arrays. http://www.rightscale.com/products/automation-engine.php.