Using Machine Learning for Black-Box Autoscaling

Muhammad Wajahat, Anshul Gandhi
Stony Brook University
{mwajahat, anshul}@cs.stonybrook.edu

Alexei Karve, Andrzej Kochut
IBM Research
{karve, akochut}@us.ibm.com

***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 minimize 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 50% compared to the optimal static policy, (ii) is within 15% 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.**

I. 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], [9], [10], [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 II.

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 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 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%). Our evaluation results show that MLscale reduces cost by around 50% when compared to the optimal static policy. Compared to the optimal dynamic policy, MLscale is within 15% 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 minimizing overheads due to provisioning changes.

978-1-5090-5117-5/16/$31.00 ©2016 IEEE

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 minimizing SLA violations without expert application knowledge or tuning.

Organization: To provide more context for this paper, we first present related work in Section II. We then present the design of MLscale in Section III, followed by our experimental setup and experimental results in Sections IV and V, respectively. We conclude the paper in Section VI.

II. 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 Botran 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. By contrast, MLscale is designed to be application-agnostic (neural network-based online modeling). 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. Bahati et al. [9] also employ RL to adapt the action space of threshold-based autoscaling. VCONF [10] leverages 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].

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. [21] employ multi-variate regression to model the impact of rebalancing data for elastic storage. Gandhi et al. [5], [22] 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.

III. MLscale

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

A. 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 [23]. A disadvantage is that they are prone to over-fitting, but this can be solved by using more training data [24]. For more details, we refer the readers to Haykin [17].

Inputs to our neural network model consist of request rate and average (across all nodes in a tier) system usage statistics such as CPU usage, number of context switches, number of interrupts, and network and disk I/O statistics; these metrics can be easily obtained online (see Section IV-D). 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]. Due to lack of space, we are not including a diagram of our network. Note that the network can be easily modified to add 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 data, takes a few seconds. 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%ile) response time is about 8.2%. It is important that the training data cover a moderate range of request rates and several scaling actions. These scaling actions could be naive, for example, based on changes in request rate. In general, the larger and richer the training data set, the lower will be the modeling error. The amount of 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 I 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 IV 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.

B. 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.

Technique	Training Time	Training Error	Test Error
LR	2.5ms	9.2%	9.3%
SVR	1294s	2.1%	6.2%
KR	8141s	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%

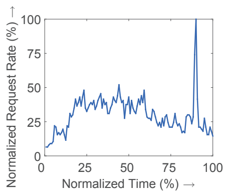
TABLE I
COMPARISON OF ML TECHNIQUES FOR PERFORMANCE MODELING.

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:

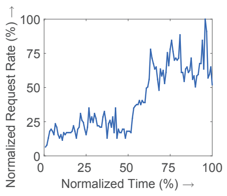
$$m' = c_0 + c_1 \cdot m \cdot w/(w+k) + c_2 \cdot m \cdot k/(w+k), \quad (1)$$

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 [2], [5], [15], and motivates the structure of the specific terms used in Eq. (1). 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.

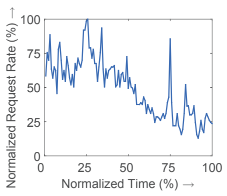
We use a subset of the training data employed for neural network modeling; specifically, we consider data points immediately before and after a scaling action. Based on training, we find that c_1 ranges from $0.8 - 1$ and c_2 ranges from $0.4 - 1.1$ for different metrics. c_0 varies much more widely 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 V-E. 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. (1), provides acceptable prediction accuracy. However, this was not the case under performance modeling (see LR in Table I).



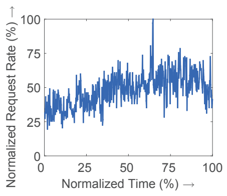
(a) T2 trace [25]



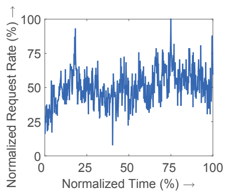
(b) T4 trace [25]



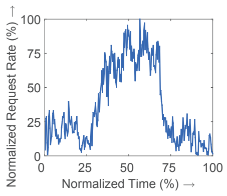
(c) T5 trace [25]



(d) Clarknet trace [26]



(e) UCB trace [26]



(f) SAP trace [27]

Fig. 1. Traces used in our experiments. Y-axis represents normalized request rate. X-axis represents normalized trace time.

C. Putting it all together - MLscale

MLscale works by leveraging the neural network-based online performance modeler and regression-based metrics predictor. In particular, we periodically monitor request rate and resource usage metrics (obtained via collectl [28]), and use these monitored metrics as input to the performance model to estimate response time. If both the observed and estimated response time exceed the SLA target, we invoke autoscaling. 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 V 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. (1)) required to maintain response times below the SLA target by evaluating scaling options around the current provisioning (w in Eq. (1)). Note that k can be greater than 1, thus allowing for arbitrary provisioning changes. We show in Section V that MLscale often adds/removes multiple nodes simultaneously in response to large variations in load.

IV. 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 48GB each, connected via 1Gb links) and an AWS-based public cloud (m4.large and c4.large instances [29]) 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.

A. Applications

PHP-based web application. We set up a 10-VM (4GB RAM, 2vCPU) tier of Apache-PHP web servers that each host a computationally-intensive microbenchmark. These VMs are behind an Apache load balancer [30], 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 [31], on another VM, as our load generator with exponential inter-arrival times.

DayTrader. DayTrader [32] 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 (4GB RAM, 2vCPU each) are application tier VMs running IBM WAS and have the DayTrader application deployed on them. The backend database VM (18GB RAM, 4vCPU) runs DB2 using a ramdisk [33] 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 (4GB RAM, 4 vCPU each), hosting exact replicas (for read-only) of 5 million records of stack exchange posts data [34]. An httperf VM (8GB RAM, 4 vCPU) generates requests that are directed to the PHP application tier VMs (4GB RAM, 2vCPU each) via an Apache load balancer (8GB RAM, 4 vCPU). 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 [35] (TCP) load balancer VM (8GB RAM, 4 vCPU), which selects one of 3 replicated database VMs to serve the 10 associated read requests. Load balancers use the (default) round robin policy.

B. Traces

We use various request traces from NLANR [25], ITA [26], and enterprise applications [27] for driving our application load. The specific traces we use for evaluation are shown in Figure 1. We only show normalized request rate and trace length values as these are modified per the system capabilities.

C. 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 120ms, that for DayTrader is set to 10ms, and that for PHP-MySQL is set to 60ms. 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.

D. 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 10s interval to provide responsive autoscaling while avoiding hasty decision making. All VMs monitor resource statistics using the collectl [28] utility. Load balancers monitor the request rate and response time. MLscale collects all statistics periodically.

V. Evaluation Results

We now present our evaluation results. We first discuss our evaluation methodology in Section V-A. We then present our results for the PHP-based web application (Section V-B), DayTrader (Section V-C), and the PHP-MySQL application (Section V-D). We end with additional results (Sections V-E) that highlight MLscale's unique advantages.

A. 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 (10s 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 in 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 such as AWS [36] and OpenStack [37]. In practice, we implement CPUscale and empirically find the best upper/lower thresholds for each trace.

B. Results for PHP-based Web Application

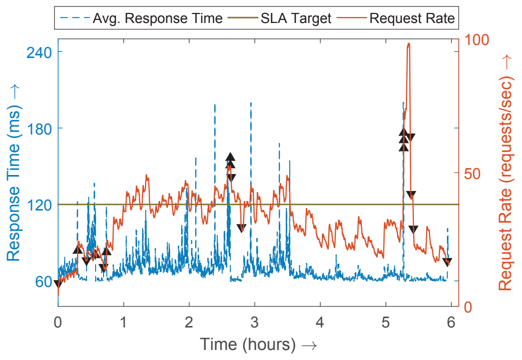
Figure 2 shows the performance of our PHP web application under MLscale for T2 and T5 traces. The solid red line (y-axis on the right) shows the observed request rate and the blue dotted line (y-axis on the left) shows the observed response time. The green horizontal line denotes the mean response time target (120ms).

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 1hr and 2hr marks, and those for T5 between the 0.5hr and 1hr 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 2.5hr and 4hr marks for T5 result in only 1 provisioning change and no violations.

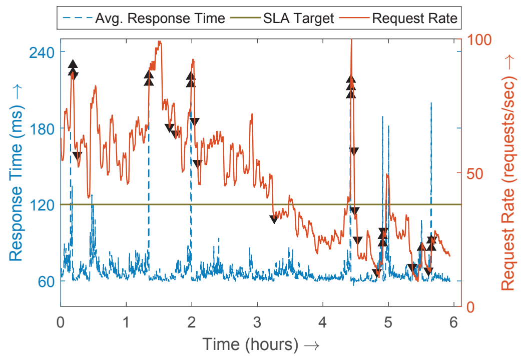
Compared to Opt-Static, MLscale lowers the resource cost by about 60% for T2 and about 46% for T5. This is expected as MLscale is dynamic in nature. Compared to Opt-Dynamic, MLscale is within 15% of the ideal resource cost. Recall that Opt-Dynamic knows the exact request rate ahead of time. Compared to the best CPUscale policy, MLscale provides superior cost-performance tradeoffs. For T2, the best CPUscale 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.

Full results for all traces and all policies are provided in Table II. 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. We see that MLscale typically incurs *less than 5%* violations and lowers resource costs by *at least 40%* when compared to Opt-Static. Further, in most cases, MLscale's resource usage is *within 15%* of the usage of Opt-Dynamic.

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



(a) T2 trace under MLscale. V=0.9%, N=2.0.



(b) T5 trace under MLscale. V=1.0%, N=2.7.

Fig. 2. Performance of (a) T2 and (b) T5 under MLscale for the PHP web application. The up and down triangles indicate scale-out and scale-in, respectively.

	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

TABLE II
COMPARISON OF POLICIES FOR ALL TRACES UNDER PHP APPLICATION.

traces, the results are comparable. 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 [1], [5], [8]. 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.

Provisioning stability: It is important to note the stability of the VM provisioning under MLscale. For the T2 and T5 traces shown in Figure 2, 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 Figure 3 for the T2 trace. The provisioning under Opt-Dynamic is even more unstable – 151 and 263 scaling actions for T2 and T5, respectively. This instability stems from the need to react to

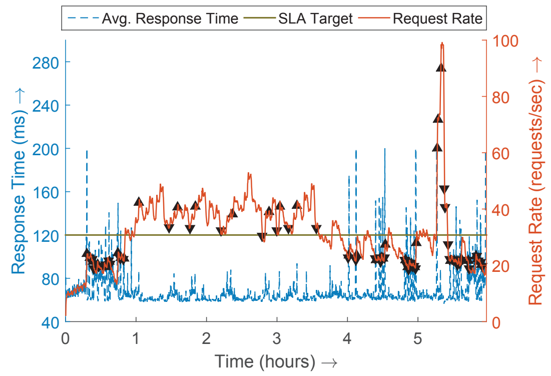


Fig. 3. Performance of T2 under the best CPUscale policy for the PHP web application. V=1.8%, N=2.2. The numerous scaling actions triggered by CPUscale illustrate its unstable resource provisioning.

the current system state without understanding the post-scaling implications. For example, the scaling actions in Figure 3 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 minimize any overhead, such as the boot up time or wear-and-tear associated with adding/removing nodes [7].

95%ile response time targets: MLscale can be easily extended to model tail response times, such as 95%ile response times. We evaluate MLscale for the Clarknet and UCB traces under a 95%ile response time target of 120ms. We use the AWS public cloud setup for these experiments. 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.

C. 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

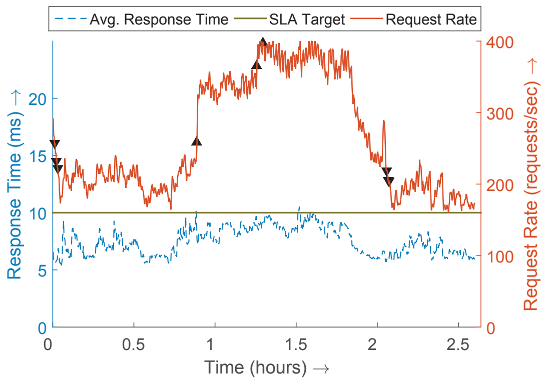


Fig. 4. SAP trace under MLscale for DayTrader application. V=0.8%, N=2.1.

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

TABLE III
COMPARISON OF MLSCALE AND OPT POLICIES FOR ALL TRACES UNDER THE DAYTRADER APPLICATION.

problem [38] that is beyond the scope of this paper. We do, however, autoscale a read-only database tier in Section V-D.

Figure 4 shows the performance of DayTrader under MLscale for the SAP trace. 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 III 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.

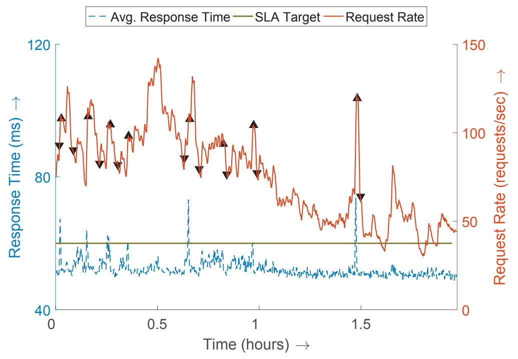


Fig. 5. T5 trace under MLscale for PHP-MySQL application. V=1.7%, N=2.3.

D. 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 [38]; 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.

Figure 5 shows the performance of our PHP-MySQL application under MLscale for 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 Figure 5. 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.

Compared to Opt-Static, MLscale lowers resource costs by about 24%. Further, MLscale is within 25% 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.

Results are similar for other traces under MLscale: $V = 0.1\%$, $N = 2.0$ for T4; $V = 1.3\%$, $N = 1.9$ for T2; and $V = 2.1\%$, $N = 1.8$ for UCB.

E. 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 III-B), 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. (1), which improves cost and performance. Figure 6 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

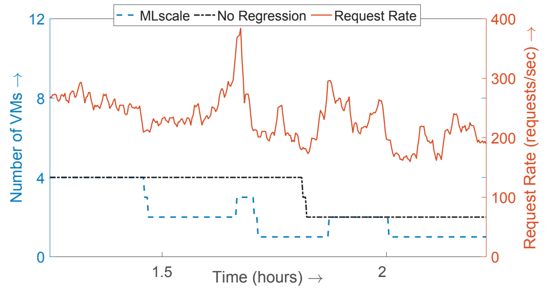


Fig. 6. Performance for T5 trace with and without the regression-based metrics predictor under DayTrader. Resource cost increases by 16% and violations increase by 23% without the regression-based metrics predictor.

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%.

VI. 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. 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.

As part of future work, we will investigate the use of MLscale to autoscale in response to performance interference in colocated VMs. We can include relevant metrics, such as cycles-per-instruction (CPI), as inputs to our neural network model to detect interference. If we oversubscribe CPU in our OpenStack setup, we find that including CPI improves modeling accuracy by around 6.7%. We will also explore modifications to MLscale to make it adaptive to long-term changes in the workload. We believe that our modeling and metrics prediction components can be retrained online in response to an increase in the modeling error, although at the expense of some retraining time.

Acknowledgment

This research was supported by an NSF CRII CSR Grant 1464151 and a 2015 IBM Faculty Award.

References

[1] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service," in *ICAC 2013*, San Jose, CA, USA, pp. 69–82.

[2] B. Urgaonkar and A. Chandra, "Dynamic Provisioning of Multi-tier Internet Applications," in *ICAC 2005*, Seattle, WA, USA, pp. 217–228.

[3] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. Kozuch, "SOFTScale: Scaling Opportunistically For Transient Scaling," in *Middleware 2012*, Montreal, Quebec, Canada, pp. 142–163.

[4] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," in *CNSM 2010*, pp. 9–16.

[5] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, Model-driven Autoscaling for Cloud Applications," in *ICAC 2014*, Philadelphia, PA, USA, pp. 57–64.

[6] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. Parekh, "Online response time optimization of apache web server," in *IWQoS 2003*, Berlin, Germany, pp. 461–478.

[7] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. Kozuch, "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers," *Transactions on Computer Systems*, vol. 30, 2012.

[8] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow," in *ICAS 2011*, Venice, Italy, pp. 67–74.

[9] R. Bahati and M. Bauer, "Towards adaptive policy-based management," in *NOMS 2010*, Osaka, Japan, pp. 511–518.

[10] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration," in *ICAC 2009*, pp. 137–146.

[11] K. O. Stanley and R. Miikkulainen, "Efficient Reinforcement Learning Through Evolving Neural Network Topologies," in *GECCO 2002*, San Francisco, CA, USA, pp. 569–577.

[12] G. Tesauro, N. Jong, R. Das, and M. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," in *ICAC 2006*, Dublin, Ireland, pp. 65–73.

[13] T. Lorido-Botrán, J. Miguel-Alonso, and 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, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871–879, 2011.

[15] T. Horvath and K. Skadron, "Multi-mode Energy Management for Multi-tier Server Clusters," in *PACT 2008*, Toronto, ON, Canada, pp. 270–279.

[16] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. Smirni, "Application-driven dynamic vertical scaling of virtual machines in resource pools," in *NOMS 2014*, Krakow, Poland, pp. 1–9.

[17] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[18] A. Gandhi, P. Dube, A. Kochut, L. Zhang, and S. Thota, "Autoscaling for Hadoop Clusters," in *IC2E 2016*, Berlin, Germany.

[19] B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "The SCADS director: scaling a distributed storage system under stringent performance requirements," in *FAST 2011*, San Jose, CA, USA, pp. 163–176.

[20] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper, "Adaptive quality of service management for enterprise services," *ACM Transactions on the Web*, vol. 2, pp. 1–46, 2008.

[21] H. C. Lim, S. Babu, and J. S. Chase, "Automated Control for Elastic Storage," in *ICAC 2010*, Washington, DC, USA, pp. 1–10.

[22] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Modeling the Impact of Workload on Cloud Resource Scaling," in *SBAC-PAD 2014*, Paris, France.

[23] D. Specht, "A general regression neural network," *IEEE Transactions on Neural Networks*, vol. 2, no. 6, pp. 568–576, 1991.

[24] P. Cunningham, J. Carney, and S. Jacob, "Stability problems with artificial neural networks and the ensemble solution," *Artificial Intelligence in Medicine*, vol. 20, no. 3, pp. 217–225, 2000.

[25] "NLANR Anonymized access logs." ftp://ftp.ircache.net/Traces.

[26] ITA, "http://ita.ee.lbl.gov/index.html," http://ita.ee.lbl.gov/index.html.

[27] SAP, "SAP application trace from anonymous source." 2011.

[28] "Collectl," http://collectl.sourceforge.net.

[29] "Amazon EC2 Instances," http://aws.amazon.com/ec2/instance-types.

[30] "Apache HTTP SERVER PROJECT," https://httpd.apache.org.

[31] "httperf," http://www.labs.hpe.com/research/linux/httperf.

[32] "DayTrader," http://geronimo.apache.org/GMOxDOC20/daytrader.html.

[33] M. Nielsen, "How to use a RAMdisk," *Linux Gazette*, vol. 44, 1999.

[34] "Stack Exchange Data Dump," https://archive.org/details/stackexchange.

[35] "HAproxy: TCP/HTTP Load Balancer," http://www.haproxy.org.

[36] Amazon, "Amazon Auto Scaling," http://aws.amazon.com/autoscaling.

[37] Openstack.org, "OpenStack Heat," https://wiki.openstack.org/wiki/Heat.

[38] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: practical power-proportionality for data center storage," in *EuroSys 2011*, Salzburg, Austria, pp. 169–182.