OnlineElastMan: Self-Trained Proactive Elasticity Manager for Cloud-Based Storage Services

Ying Liu*, Daharewa Gureya*, Ahmad Al-Shishtawy† and Vladimir Vlassov*
*KTH Royal Institute of Technology, Sweden
Email: yinliu@kth.se, gureya.daharewa@gmail.com, vladv@kth.se
†Swedish Institute of Computer Science, Stockholm, Sweden
Email: ahmad@sics.se

***Abstract*—The pay-as-you-go pricing model and the illusion of unlimited resources in the Cloud initiate the idea to provision services elastically. Elastic provisioning of services allocates/de-allocates resources dynamically in response to the changes of the workload. It minimizes the service provisioning cost while maintaining the desired service level objectives (SLOs).**

Model-predictive control is often used in building such elasticity controllers that dynamically provision resources. However, they need to be trained, either online or offline, before making accurate scaling decisions. The training process involves tedious and significant amount of work as well as some expertise, especially when the model has many dimensions and the training granularity is fine, which is proved to be essential in order to build an accurate elasticity controller.

In this paper, we present OnlineElastMan, which is a self-trained proactive elasticity manager for cloud-based storage services. It automatically trains and evolves itself while serving the workload. Experiments using OnlineElastMan with Cassandra indicate that OnlineElastMan continuously improves its provision accuracy, i.e., minimizing provisioning cost and SLO violations, under various workload patterns.

Keywords*—*Elasticity Controller, Cloud Storage, Workload prediction, SLO, Online Training, Time series analysis.

I. Introduction

Hosting services in the Cloud are becoming more and more popular due to a set of desired properties provided by the platform, such as low application setup cost, professional platform maintenance and elastic resource provisioning. Elastically provisioned services are able to use platform resources on demand. Specifically, VMs are spawned when they are needed for handling an increasing workload and removed when the workload drops. Since users only pay for the resources that are used to serve their demand, elastic provisioning saves the cost of hosting services in the Cloud.

On the other hand, services are usually provisioned to match a certain level of Quality of Service (QoS), which is usually defined as a set of Service Level Objectives (SLOs) in Cloud context. Thus, there are two contradictory goals to be achieved, i.e., saving the provisioning cost and meeting the SLO, while services are elastically provisioned.

Elastic provisioning is usually conducted automatically by an elasticity controller, which monitors the system status and makes corresponding decisions to add or remove resources. An elasticity controller needs to be trained, either online or offline, in order to make it smart enough to make such decisions. Generally, the training process allows the controller to build up a model that correlates monitored parameters, such as CPU or incoming workload, to controlled parameters, i.e., the SLO, which could be, for example, percentile request latency. The accuracy of the model, directly affects the accuracy of the elasticity controller, which dominates service provisioning cost and commitment of the SLO.

It is non-trivial to build an accurate and efficient elasticity controller. Recent works have been focusing on improving the accuracy of elasticity controllers by building different control models with various monitored/controlled metrics [1], [2], [3], [4], [5], [6]. However, none of the works have considered the practical usefulness of an elasticity controller, which involves the following challenges. First, an elasticity controller usually needs to be tailored according to a specific application. To be concise, sometimes, it requires complicated instrumentations to the provisioned application or even not possible to obtain the metrics that are used to build the control model. Furthermore, even with all the metrics, it requires tremendous and tedious works to train the control model. A general training procedure involves the redeployment and reconfiguration of the application and collecting and analyzing data by running various workloads against various configurations of the application. Second, the hosting environment of the provisioned application may change due to some unmonitored factors, for example, platform interference or background maintenance tasks. Then, even with well-trained control models, it may not be able to adjust to these factors and leads to inaccurate control decisions. Third, it is always too late for the elasticity controller to react to a workload increase when the workload is already saturating the application. Thus, we argue that prediction of the workload is always a compulsory element to an elasticity controller.

In this work, we propose OnlineElastMan, which is a generic elasticity controller for distributed storage systems. It excels its peers with its practical aspects, which includes straightforward obtainable control metrics, automatically online trained control models and embedded generic workload prediction module. It makes OnlineElastMan an "out-of-the-box" elasticity controller, which can be deployed and adopted by different storage systems without complicated tailoring/configuring efforts. Specifically, OnlineElastMan requires only monitoring on the two most generic metrics, i.e., incoming workload and service latency, which is obtainable from most of the storage systems without complicated instrumentation. Using the monitored metrics, OnlineElastMan analyzes the workload composition in depth, which includes

978-1-5090-3536-6/16 $31.00 © 2016 IEEE
DOI 10.1109/ICCAC.2016.11

read/write request intensity and data size of the requested item, which defines the dimensions of a control model. OnlineElastMan can easily plug in more interested dimensions if needed. After fixing the dimensions, a multi-dimensional control model can be automatically built and trained online while the storage system is serving requests. After a sufficient amount of warm up on the control model, OnlineElastMan is able to issue accurate control decision based on the incoming workload. Furthermore, the control model continuously improves itself online to adjust to unknown/unmodeled events of the operating environment. Additionally, a generic workload prediction module is also integrated to facilitate the decision making of OnlineElastMan. It allows OnlineElastMan to scale the storage system well in advance to prevent SLO violations caused on workload increase and scaling overhead [5]. Specifically, the prediction module aggregates multiple prediction algorithms and chooses the most appropriate prediction algorithm based on the current workload pattern using a weight majority selection algorithm. Contributions of the paper are as follows.

- Implementation of an "out-of-the-box" generic elasticity controller framework, which is easily applicable to most of the distributed storage systems.
- Integration of an online self-trained control model to OnlineElastMan, which avoids repetitive and tedious system reconfiguring and model training.
- Proposal of a multi-dimensional control model based on workload characteristics, which proves to have better control accuracy.
- Realization of a generic workload prediction module in OnlineElastMan, which is adjustable to multiple workload patterns.
- Open-source implementation [1] of OnlineElastMan framework.

II. Problem Statement

There is a large body of work on elasticity controllers for the Cloud [2], [3], [4], [5], [6]. Most of them focus on improving the control accuracy of the controller by introducing novel control techniques and models. However, none of them tackles the practical issues regarding the deployment and application of the controllers. We examine the usefulness of an elasticity controller while deploying it in a Cloud environment. Specifically, we investigate the configuration steps for an elasticity controller before it starts provision services. Typically, it involves the following steps to setup an elasticity controller.

1) Acquire metrics for the elasticity controller from the provisioned application or the host platform.
2) Deploy the provisioned application in order to construct a training case for the elasticity controller.
3) Configure the provisioned application according to the deployment.
4) Configure and run a specific synthesized workload against the application.
5) Collect training data from the training case and train the control model accordingly.
6) Repeat step 2 to 5 until the control model is fully trained before serving the real workload.

It is intuitively clear that the more metrics considered in a control model, the more accurate it will be. However, increasing the metric dimensions of a control model comes with a significant amount of overhead during the training phase. Specifically, training a control model with only 3 dimensions results in 27 (3^3) training cases when only 3 trials/runs are conducted for each dimension. This means that step 2 to step 5 needs to be repeated 27 times to train the control model. Obviously, it is extremely time consuming to train a control model manually, especially when the model has many dimensions, which is needed for higher control accuracy.

OnlineElastMan alleviates the training process with online training. Specifically, the model automatically trains and evolves itself while serving the workload. After a short period of warm up, the controller is able to provision the underlying application accurately. Thus, it is no longer needed to manually and repetitively reconfigure the system in order to train the model. Furthermore, in order to make OnlineElastMan as general as possible, its input metrics are easily obtainable from the application. Specifically, it directly uses the information in the incoming workload, which does not need application specific instrumentation, and service latency, which is the most accurate and direct reflection of QoS and can be easily sampled from system entry points or proxies.

On the other hand, previous works [5], [2] have demonstrated that, in order to keep the SLO commitment, a storage system needs to scale up in advance to tackle with a workload increase since scaling a storage system involves non-negligible overhead. Thus, we have made a design choice to integrate a workload prediction module for OnlineElastMan. Again, to make it as general as possible, the workload prediction module is able to produce accurate workload prediction for various workload patterns. Specifically, it has integrated several prediction algorithms that are designed to cope with different time series patterns. The most appropriate prediction algorithm is chosen online using a weight majority selection algorithm.

III. OnlineElastMan Design

In this section, we present the design of OnlineElastMan by explaining its three major components, i.e., workload prediction, online model training, and elasticity controller. Figure 1 presents the architecture of OnlineElastMan. Components operate concurrently and communicate by message passing. Briefly, workload prediction module takes input from the current workload and predicts workload for a near future (the next control period). Online Model training module updates the current model by mapping and analyzing the monitored workload and the performance of the system. Then, the elasticity controller takes the predicted workload and consults the updated performance model to issue scaling commands by calling the Cloud API to add or remove servers for the underlying storage system.

A. Monitored Parameters

Auto-scaling technique requires a monitoring component that gathers various metrics that reflect the realtime status of the targeted system at an appropriate granularity (e.g per

[1] https://github.com/gureya/OnlineElasticityManager

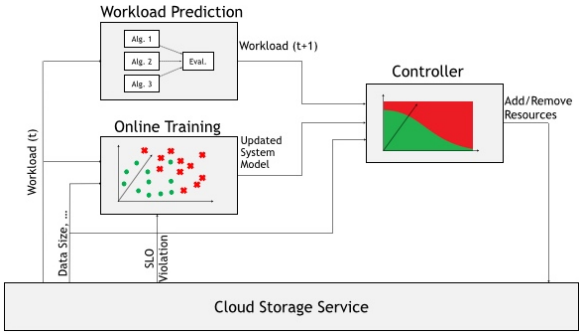


Fig. 1: OnlineElastMan Architecture

second, per minute, per hour). It is essential to review the metrics that can be obtained from the target system and the metrics that best reflect the status of the system. To ease the configuration of OnlineElastMan framework and to make it as general as possible, we consider the target storage system as a black box. OnlineElastMan adopts the most general and direct metrics that dominate the QoS of the targeted storage system. Specifically, we take the **workload**, which causes the variations in those system metrics, directly as the input. OnlineElastMan requires the workload monitoring to provide the read/write intensity and the size of the requested data in the workload in small intervals. The monitored data can be obtained by sampling the traffic passing through the entry points, e.g. proxies or load balancers, of the storage system. The **percentile latency**, which defines and directly reflects the QoS, is collected either from entry proxies or the storage system itself depending on the design and workflows of storage systems. Then, the collected percentile latency is used to adjust and improve control decisions/models. In Section IV-A1, we provide details on how we obtain these metrics in a distributed storage system, such as, Cassandra [7].

B. Multi-dimensional Online Model

One of the core components in OnlineElastMan is the multi-dimensional online SML (Statistical Machine Learning) model. It correlates the input metrics (workload characteristics) with the SLO (percentile latency). The goal of the model is to keep the target system operating with the percentile latency varying only in a small controlled range. It is intuitively clear that with more provisioned resources (VMs), the system is able to respond to requests with reduced latency. However, on the other hand, we would also like to provision as little VMs as possible to save the provisioning cost. Thus, the controlled latency range is always desired to be slightly under (just satisfying) the percentile latency requirement defined in the SLO to minimize the provisioning cost. We refer this region to be **optimal operational region (OOR)**, where a system is not very much over-provisioned but satisfying the SLO.

In order to keep the system operating in the OOR while the incoming workload is dynamic, an elasticity controller needs to react to the workload changes and allocating/de-allocating VMs to the system. Previous works [3], [4], [5], [8] designs an elasticity controller based on an offline statistical model. OnlineElastMan builds the model online and continuously improves/updates the model while serving requests. The online

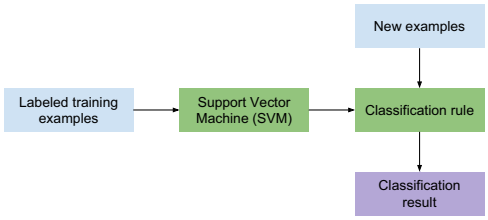


Fig. 2: Classification using SVM

training feature frees system administrators from the tedious offline model training procedure, which includes repetitive system configurations, system deployments, model updates, etc., before putting the controller online. Additionally, the continuous evolving model in OnlineElastMan enables the system to survive with factors that are not considered in the model, e.g. platform interference [9], [10].

Specifically, the online model is built with the monitored parameters mentioned in Section III-A. It classifies whether a VM is able to operate in the OOR under the current workload, which breaks down to the intensity of read and write requests and the requested data size. Ideally, a storage node hosted in a VM can be either operating under commitment to the SLO or with violation to the SLO. Therefore, with a given workload and VM flavor, the classifier model is a line that separates the plane into two regions, in which the SLO is either met or violated as shown in Figure 3. Different models need to be built for different VM flavors and different storage systems hosted. While building the model, there are several configurable parameters that affect the accuracy of the model.

Granularity of the model: Since the collected data for the model can be decimal, it is impossible to analyze the data with infinite combinations. We group the collected data with a pre-defined granularity, which makes a two-dimensional plane to be separated to small squares or a three-dimensional plane to be separated to small cubes. These squares and cubes are the groups where data are accumulated and analyzed. The granularity of data groups can be configured depending on the memory limits and the precision requirements of the model.

Historical data buffer: For data collected and mapped to each group, we maintain a historical record for the most recent n reads and writes.

Confidence level: The historical data in each group is analyzed to define whether the workload that corresponds to the data collected in this group violates the SLO or not. For example, 95% confidence level implies that 95% of all the Read/write percentile latency sampled satisfy the SLO.

Update frequency: The model updates itself periodically with a fixed configurable rate. A higher update frequency allows the model to swiftly adapt to execution environment changes while a lower update frequency makes the model more stable and tolerate transient execution environment changes.

1) SVM Binary Classifier: : SVMs have become popular classification techniques in a wide range of application domains [11]. They provide good performance even in cases of high-dimensional data and a small set of training data. Figure 2 shows the flow of a classification task using SVM.

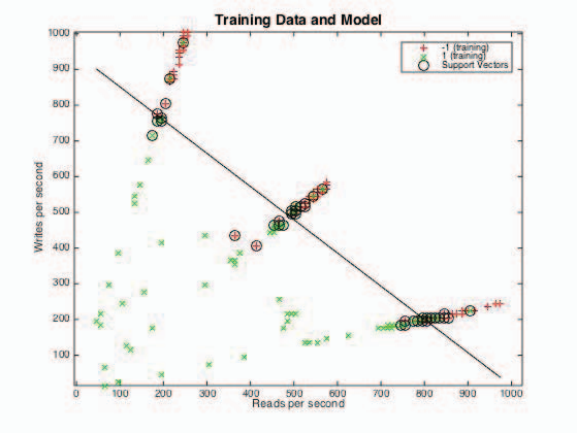


Fig. 3: 2 dimensional SVM performance model

We describe techniques to build the model of OnlineElastMan with SVM. Each instance of the training set contains a class label and several features or observed variables. The goal of SVM is to produce a model based on the training set. More concretely, given a training set of instance-label pairs $(x_i, y_i), i = 1, ..., l$ where $x_i \in R^n$ and $y_i \in \{1, -1\}^l$, the SVM classification solves the following optimization problem:

$$min_{w,b} \qquad \| w \|^2 + C \sum_i \xi_i \tag{1}$$

subject to:

$$\begin{aligned} y^{(i)}(w^T x^i + b) \geq 1 - \xi_i, \quad & i = 1, 2, \ldots, m \\ \xi_i \geq 0, \quad & i = 1, 2, \ldots, m \end{aligned} \tag{2}$$

After solving, the SVM classifier predicts 1 if $w^T x + b \geq 0$ and -1 otherwise. The decision boundary is defined by the following line:

$$w^T x + b = 0 \tag{3}$$

Generally, the predicted class can be calculated using the linear discriminant function:

$$f(x) = wx + b \tag{4}$$

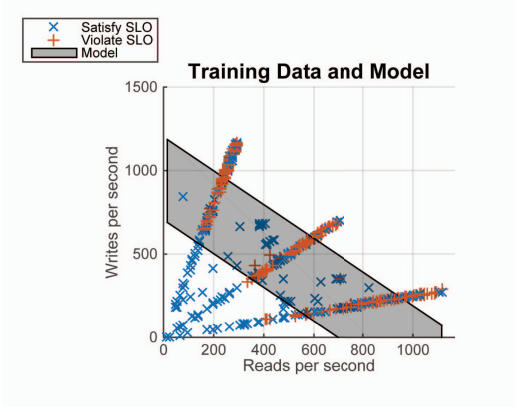
x refers to a training pattern, **w** as the weight vector and **b** as the bias term. **wx** refers to the dot product, which calculates the sum of the products of vector components $w_i x_i$. For example, in case of training set with three features (e.g. x, y, z), the discriminant function is simply:

$$f(x) = w_1 x + w_2 y + w_3 z + b \tag{5}$$

SVM provides the estimates for w_1, w_2, w_3 and b after training.

Given Equation 3, the SML model is a line (Figure 3) when only monitoring read/write request intensity in the workload or a plane (Figure 4a) when another dimension, i.e., data size, is modeled. Figure 4b is a 2 dimensional projection of Figure 4a, which shows that different data sizes cause different separations of the 2 dimensional model space. It indicates that data size plays an essential role to build an accurate control model for storage systems. The line/plane separation in the model represents the maximum workload that a VM can serve under the specified SLO (percentile latency).

(a)



(b)

Fig. 4: (a) 3 dimensional SVM performance model taking into account request data size (b) top angle to view (a), where the model plane is projected to a 2 dimensional surface and the shaded area is caused by the varying data sizes

Online model Training: Using the SVM model training technique, the performance model is updated periodically according to the **update frequency** using the data in the **historical data buffer** processed with the **confidence level**.

We believe that every VM can have significant performance difference even when they are spawned with the same favor. This can be caused by the interference from the host platform [9], [10] or background tasks, such as data migration [5]. Thus, individual SML model is built for each VM participating in the system. They automatically evolve and update continuously while the system is serving workload. Periodically, the updated SML models for each VM are sent to the elasticity controller module to make scaling decisions.

C. Elasticity Controller

An elasticity controller makes scaling decisions in configurable control periods/intervals to prevent system from oscillations. When making a scaling decision, the elasticity controller

collects the aggregation of the input workload of all VMs (W) and the aggregation of the capacity of all VMs (C). The capacity of a VM is the maximum workload that it can handle under the SLO, which is obtained from the multi-dimensional SML model. The elasticity controller also observes the input workload (w_x) and capacity (c_x) for each VM individually to identify fine-grained SLO violations. Specifically, the capacity of each VM is calculated by intersecting the plane of its SML model with a line from the origin that points to the current workload representation, which is a point corresponding to read and write workload intensity and the averaged data size. The capacity of the VM is the intersection point, which represents the capability to serve workload with specific read/write request intensity of a specific data size. If the current workload representation point is beyond the capacity representation point in the model, the SLO is violated.

The responsibility of an elasticity controller is to keep the provisioned system operating with commitment to the SLO. The strictest requirement is that each VM operates with the commitment to the SLO, which can be denoted by $\forall i \in N, w_i < c_i$, where N is the complete set of all participating VMs. However, this is not trivial to achieve without over-provisioning the system because of the imbalance of workload distribution. It is challenging to balance workload in storage systems with respect to each VM. This is because that storage systems are stateful, i.e., usually each VM is responsible only for a part of the total data stored. Thus, a specific request can only be served by a specific set of VMs, which host the requested data. Given that different storage systems have different data distribution as well as load balancing strategies and OnlineElastMan is designed to be a generic framework to provision storage systems elastically, we choose not to manage workload/data distribution for provisioned systems. Furthermore, managing data distribution or rebalancing among VMs is orthogonal to the design goal of OnlineElastMan. Nevertheless, OnlineElastMan provides/outputs suggestions for workload distributions to each participating VMs based on their capacity learnt from our SML models.

In order to tolerant load imbalance among VMs to some extent, OnlineElastMan introduces an optional tolerance factor α when computing scaling decisions to prevent too much over-provisioning. Specifically, a scaling up decision is issued when the SLO violation $c_x < w_x$ is observed from more than α VMs, where $\alpha \geq 0$. When $\alpha = 0$, there is no tolerance on load imbalance. The number of VMs to add is calculated individually for each VM and aggregated globally. $\frac{w_x - c_x}{c_x}$ number of VMs with the same flavor as c_x is expected to be added. Thus, when $\frac{w_x - c_x}{c_x} < 0$, it represents that a VM has more capacity than the incoming workload. We aggregate results of $\frac{w_x - c_x}{c_x}$ on each VM flavors and ceiling the aggregated results. When the result on a specific VM flavor is negative, we do nothing because it is in a scaling up procedure. When the result on a specific VM flavor is positive, we add the number of VMs of that flavor accordingly.

For scaling down, there is also a corresponding load imbalance tolerance factor β. β denotes the over-provisioning number of VMs in each VM flavor. A scaling down procedure is triggered by first satisfying that there is no VM that violates the SLO, which gives $\forall i \in N, w_i < c_i$, where N is the complete set of all participating VMs. Then, the number of VMs to de-allocate is calculated through similar process comparing to scaling up. The aggregated results of $\frac{w_x - c_x}{c_x}$ on each VM flavors are floored after subtracting β. Last, the corresponding number of VMs are de-allocated when the floored results are greater than zero.

When a scaling up/down decision is made, the elasticity controller interact with Cloud/platform API to request/release VMs. Where applicable, the elasticity controller also calls the API to rebalance data to the newly added VMs or to decommission the VMs that are about to be removed. Adding/removing VMs to a distributed storage system introduce a significant amount of data rebalance load in the background. This leads to fluctuations on sensitive performance measures, such as percentile latency. Usually, the extra data rebalancing load is not long lasting. So, this fluctuation can be filtered out in our SML model with proper setting on the **confidence level** and **update frequency** introduced in Section III-B.

D. Workload Prediction

An optional but essential component of OnlineElastMan is the workload prediction module. It is always too late to make a scaling out decision when the workload is already increased since preparing VMs involve non-negligible overhead, especially for storage systems, which require data to be migrated to the newly added VMs. Thus, there is a prediction module that facilitates OnlineElastMan to make decisions in advance.

Often, there are patterns that can be found in the workload, such as the diurnal pattern [12]. These patterns become vague when the workload is distributed to each VM. Thus, we are not predicting the incoming workload for each VM. Rather, the workload is predicted for the whole system. Then, it is proportionally calculated for each VM based on the current workload portion that is served by the VM. Finally, instead of using the current incoming workload to make a scaling decision in the previous section, we are able to use the predicted workload as the input.

However, even predicting the workload for the whole system is not trivial since there are many factors that contribute to the fluctuation of the workload [13]. Some workloads have repetitive/cyclic pattern, such as diurnal patterns or seasonal patterns while some other workloads experience exponential growth over a short period of time, which can be caused by market campaigns or special offers. Considering that there are no perfect predictors and different applications' workloads are distinct, no single prediction algorithm is general enough to be suitable for most workloads. Thus, we have studied and analyzed several prediction algorithms that are designed for different workload patterns, i.e., **the regression trees**, **first-order autoregressive**, **differenced first-order autoregressive**, **exponential smoothing**, **second-order autoregressive** and **random walk**. Then, a weighted majority algorithm (Section III-D3) is used to select the best prediction algorithm.

1) Regression Trees model: Regression trees predict responses to data and are considered as a variant of decision trees. They specify the form of the relationship between predictors and a response. We first build a tree using the time series data through a process known as recursive partitioning (Algorithm 1) and then fit the leaves values to the input predictors like Neural Networks. Particularly, to predict a

response, we follow the decisions in the tree from the root node all the way to a leaf node which contains the response.

Algorithm 1: Recursive Partitioning Algorithm

Data: A set of N data points, x_i, i = 1,...,n
Result: A regression tree
if *termination criterion exist* **then**
  Generate Leaf Node and allocate it a Given Value;
  Return Leaf Node;
else
  Identify Best Splitting test $s*$;
  Generate node t with $s*$;
  Left_branch(t) = $RecursivePartitioning(< x_i, y_i >: x_i = s*)$;
  Right_branch(t) = $RecursivePartitioning(< x_i, y_i >: x_i \neq s*)$;
  Return Node t;

2) ARIMA: Autoregressive moving average (ARMA) is one of the most widely used approaches to time series forecasting. ARMA model is convenient for modelling time series data which is stationary. In order to handle non-stationary time series data, ARMA model adopts a differencing component to help deal with both stationary and non-stationary data. This class of models with differencing component is referred to as the autoregressive integrated moving average (ARIMA) model. Specifically, ARIMA model is made up of autoregressive (AR) component of lagged observations, a moving average (MA) of past errors and a differencing component (I) needed to make a time series to be stationary. The MA component is impacted by past and current errors while the AR component shows the recent observations as a function of past observations [14].

In general, an ARIMA model is represented as ARIMA(p,d,q) model where **p** is the number of autoregressive terms (order of AR), **d** is the number of differences needed for stationarity, and **q** is the number of lagged forecast errors in the prediction equation (order of MA). The following equation represents a time series expressed in terms of AR(n) model:

$$Y^{'}(t) = \mu + \alpha_1 Y(t-1) + \alpha_2 Y(t-2) + \ldots + \alpha_n Y(t-n) \quad (6)$$

Equation 7 represents a time series expressed in terms of moving averages of white noise and error terms.

$$Y^{'}(t) = \mu + \beta_1 \epsilon(t-1) + \beta_2 \epsilon(t-2) + \ldots + \beta_n \epsilon(t-n) \quad (7)$$

In OnlineElastMan, apart from regression tree, we have integrated five models using ARIMA implementations, which are first-order autoregressive ($ARIMA(1,0,0)$), differenced first-order autoregressive ($ARIMA(1,1,0)$), simple exponential smoothing ($ARIMA(0,1,1)$), second-order autoregressive ($ARIMA(2,0,0)$) and random walk ($ARIMA(0,1,0)$). In our view, they represent most of the common workload patterns. For example, the first-order autoregressive model performs well when the workload is stationary and autocorrelated while, for non-stationary workload, a random walk model might be suitable. Then, the challenge is to detect and select the most appropriate prediction model during runtime.

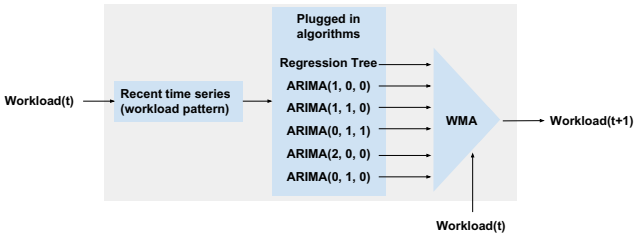


Fig. 5: Architecture of the workload prediction module

3) The Weighted Majority Algorithm: A Weighted Majority Algorithm(WMA) is implemented to select the best prediction model during runtime. It is a machine learning algorithm that is used to build a combined algorithm from a pool of algorithms [15]. The algorithm assumes that one of the known algorithms in the pool will perform well under the current workload without prior knowledge about the accuracy of the algorithms. The WMA have many variations suited for different scenarios including infinite loops, shifting targets and randomized predictions. We present our WMA implementation in Algorithm 2. Specifically, the algorithm maintains a list of weights w_1,...,w_n for each prediction algorithm. The prediction result from the most weighted algorithm, based on a weighted majority vote, is selected and returned.

Algorithm 2: The Weighted Majority Algorithm

1. Initialize the weights $w_1, ..., w_n$ of all the prediction algorithms to a positive weight (1).
2. Return the prediction result of the prediction algorithm with the highest weight.
3. Compare the predicted value with the real value, penalize the prediction algorithms, which missed the prediction more than a predefined tolerance interval n, by multiplying their weights with a fixed penalize factor m ($0 \leq m < 1$).
4. Wait until next prediction interval and go to 2.

The prediction module of OnlineElastMan is shown in Figure 5. Additional prediction algorithms can be plugged into the prediction module to handle more workload patterns.

E. Putting Everything Together

OnlineElastMan operates according to the flowchart shown in Figure 6. The incoming workload is fed to two modules, i.e., the prediction module and the online training module. The prediction module utilizes the current workload characteristics to predict the workload in the next control period using the algorithm described in Section III-D. The online training module records the current workload composition and samples the service latency under current workload. Then, the module trains/updates the performance model with the **update frequency**. The actuation is calculated based on the predicted workload for the next control period using the updated performance model according to the algorithm explained in Section III-C. Finally, the actuation is carried out on the Cloud platform that hosts the storage service.

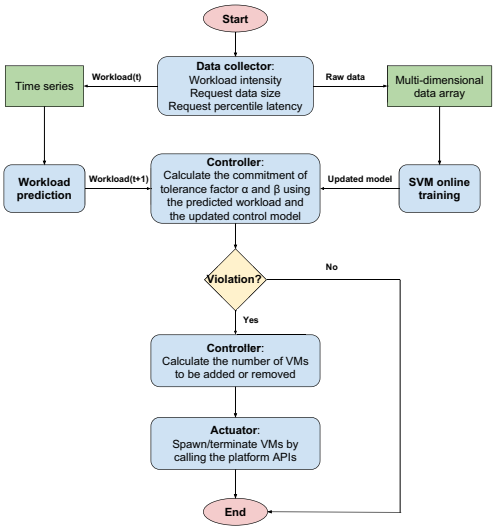


Fig. 6: Control flow of OnlineElastMan

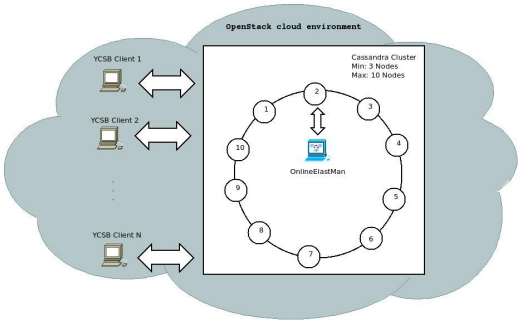


Fig. 7: Different number of YCSB clients are used to generate workload with different intensity. OnlineElastMan resizes the Cassandra cluster according to the workload.

IV. Evaluation

We evaluate OnlineElastMan from two aspects. First, we show the accuracy of the prediction module, which consists of six prediction algorithms. It directly influences the provision accuracy of OnlineElastMan since it is an essential input parameter for the performance model. Then, we present the evaluation results of OnlineElastMan when it dynamically provisions a Cassandra cluster with the application of the online multi-dimensional performance model.

Our evaluation is conducted in a private Cloud, which runs OpenStack software stack. Our experiments are conducted on VMs with two virtual cores (2.40GHz), 4GB RAM and 40GB disk size. They are spawned to host storage services or benchmark clients. OnlineElastMan is configured separately on one of the VMs. The overview of the evaluation setup is presented in Figure 7.

A. Evaluation Environment

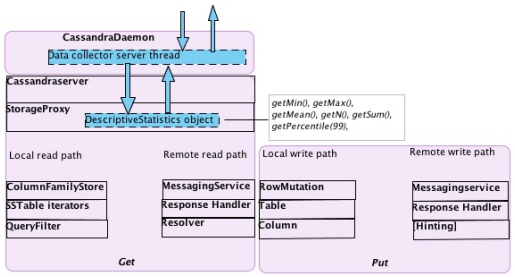


Fig. 8: Cassandra instrumentation to collect request latencies

1) Underlying Storage System: Cassandra (version 2.0.9) is deployed as the underlying storage system and provisioned by OnlineElastMan. Cassandra is chosen because of its popularity to be used as a scalable backend storage by many companies, e.g. Facebook. Breifly, Cassandra is a distributed replicated database, which is organized with distributed hash tables. Since a Cassandra cluster is organized in a peer to peer fashion, it achieves linear scalability. Minimum instrumentation is introduced to Cassandra's read and write path as shown in Figure 8. The instrumented library samples and stores service latency of requests in its repository. OnlineElastMan's data collector component periodically, every 5 minutes in our experiments, pulls collected access latencies from the repository on each Cassandra node. The collected request samples from each Cassandra node are used by the prediction module and the online training module of OnlineElastMan as shown in Figure 6 The Cassandra rebalance API is called to redistribute data when adding/removing Cassandra nodes.

2) Workload Benchmark: We adopt YCSB (Yahoo! Cloud System Benchmark) (version 0.1.4) to generate workload for our Cassandra cluster. We choose YCSB because of its flexibility to synthesize various workload patterns, including the varying read/write request intensity and the size of the data propagated. Specifically, we configure YCSB clients with the parameters shown in Table I. In order to generate stable workload to Cassandra, a fixed request rate (1200 req/s) is set to each YCSB client hosted on a separate VM. We vary the total amount of workload generated by adding or removing VMs that host YCSB clients.

Number of Threads	16
Request Distribution	uniform
Record Count	100000
Read Proportion	varied (0.0 - 1.0)
Update Proportion	varied (0.0 - 1.0)
Data size	varied (1 - 20) KB
Replication Factor	3
Consistency Level	level ONE

TABLE I: YCSB configuration

3) Multi-dimensional Performance Model: Our performance model is trained automatically when the input workload varies. OnlineElastMan takes input from the monitored parameters as specified in Section III-A. Specifically, the workload features, including read/write request intensity and request data size, and the corresponding service latency, obtained from Cassandra instrumentation, are associated to train the model. Details on model training is presented in Section III-B.

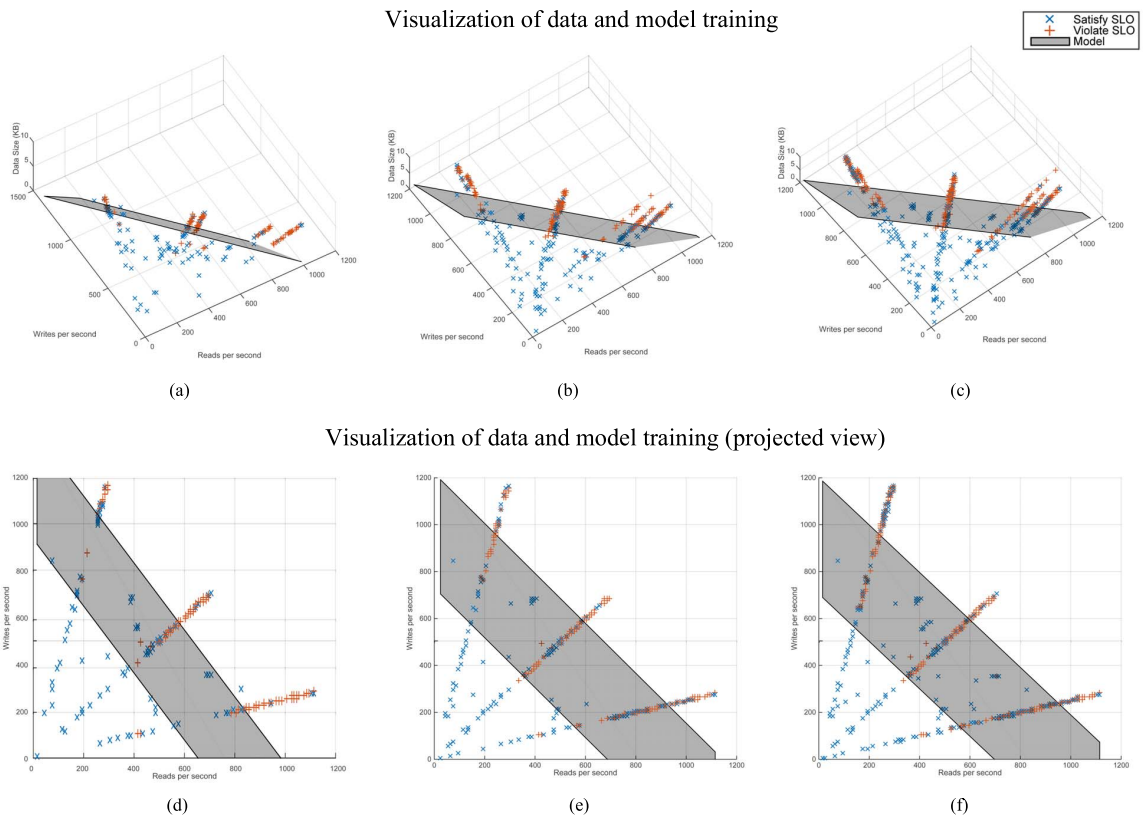


Fig. 9: The training illustration of the 3 dimensional performance model. (a), (b), and (c) are ordered by the length of training period. (d), (e), and (f) are the visualization of (a), (b) and (c) with data size dimension projected on the other 2 dimensions

In practice, the model starts empty and needs to get trained online automatically for some time. This is because that the model is application and platform specific. Thus, it needs a warm up training phase. According to our experiment experience, it takes approximately 20 to 30 minutes to train a performance model from scratch. After warm up, the model can be used to facilitate the decision making process of the elasticity controller while serving the workload.

Figure 9 depicts the model built and used in our evaluation. It consists of three input parameters/dimensions, i.e., read/write request intensity and the data size. The controlled parameter is the 99^{th} percentile read latency, which is set to be $35ms$ in our case. As shown in the figure, with more training data, the model (the shaded surface) evolves itself to a more accurate state. Practically, the performance model is dynamic and trains/evolves while serving the workload. So, it can automatically evolves to a more accurate model that reflects the changes of the operating environment and the provisioned storage system. To be specific, the model adapts to unknown/unmodeled factors, such as unknown application interference or platform maintenance, gradually using more updated training data. A more accurate model leads to better provision accuracy when the elasticity controller consults it.

In our experiments, we found out that the rate at which the model evolves affects the accuracy of the decisions made by the controller. The **confidence level** and **update frequency** (as introduced in Section III-B) dictates how fast the model evolves. Ideally we should have enough confidence about the status (violate SLO/satisfy SLO) of a data point before its status changes. Setting the confidence level low and the update frequency high may result into the model oscillating (unstable model) while the opposite settings of these two parameters may delay the evolution of the model. In our experiments, we set the confidence level as 0.5, i.e., if 50% of all read/write latency queue samples satisfy the SLO then the corresponding data point satisfies SLO and vice versa. The update frequency is set to 5 minute. For applications that have distinct phases of operations, to prevent frequent retraining, one can maintain a set of models and dynamically selects the best model for the current input pattern [16].

B. Evaluation on Workload Prediction

We evaluate the prediction accuracy of the workload prediction module using a synthetic workload generated by YCSB. We have synthesized workload with different shapes of workload increase and decrease regarding the total request intensity with a fixed read/write ratio. Figure 10 presents the actual workload generated and the workload predicted by our

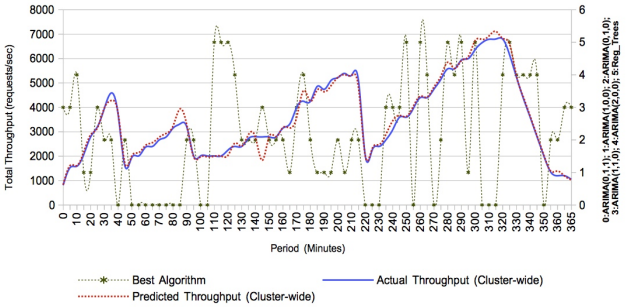


Fig. 10: Actual workload V.S. predicted workload

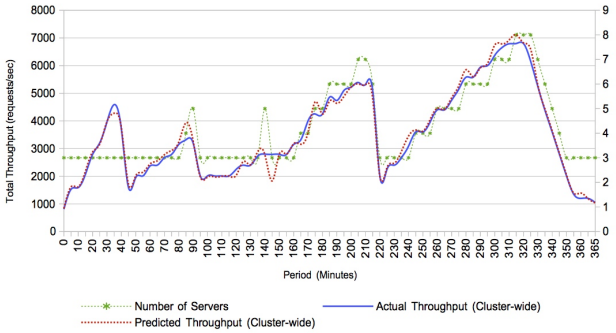


Fig. 11: VMs allocated according to the predicted workload and the updated control model

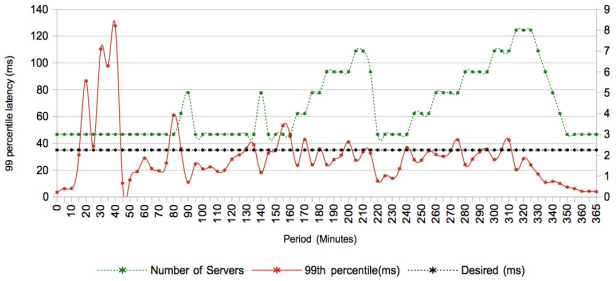


Fig. 12: The aggregated 99^{th} percentile latency from all Cassandra VMs with the allocation of VMs indicated by OnlineElastMan under the dynamic workload

prediction module. In addition, the choice of the dominant prediction algorithm proposed by the weight majority algorithm is also shown in the figure. As a result, our prediction module is able to achieve as low as 4.60% on the Mean Absolute Percentage Error for such a dynamic workload pattern.

C. Evaluation of OnlineElastMan over Cassandra

We set the goal of OnlineElastMan to keep the 99^{th} percentile of read latency to be $35ms$ as stated in the SLO. The evaluation is conducted with control period set to be 5 minutes. Even the workload of YCSB is configured to be uniform in our case, we still observe a non-trivial difference on the amount of workload served from different Cassandra storage VMs. To make a tradeoff between the uneven workload served on each VM and preventing over-provisioning, we set the tolerance factor $\alpha = 1$ and $\beta = 0.5$.

As shown in Figure 11, we start the experiment with 3 Cassandra VMs. From 0 to 40 minute, the multi-dimensional performance model is trained and warmed up. The elasticity controller starts to function from 40 minute. From 40 to 90 minute, the workload increases gradually. It is observable that from 40 to 70 minute, the system is over-provisioned, as the percentile latency is far below the SLO boundary as shown in Figure 12. This is because that the elasticity controller is set to operate with a minimum number of 3 VMs, which corresponds to the replication factor of Cassandra. With the increasing of workload, the elasticity controller gradually adds two VMs from 80 minute. Then, the workload experienced a sharp decrease from 90 minute, but the controller maintains a minimum of 3 Cassandra VMs. We continue to evaluate the performance of OnlineElastMan with another two rounds of workload increase and decrease with different scales (shown from 150 to 220 minute and from 220 to 360 minute). The evaluation indicates that OnlineElastMan is able to keep the 99^{th} percentile latency commitment most of the time. On the other hand, we observe a small amount of SLO violations under the provisioning of OnlineElastMan. It is because of the tolerance factor α and β, which allows us to tolerate some imbalance of workload distribution to Cassandra nodes.

V. Related work

A. Practical Approaches

Most of the elasticity controllers available in public Cloud services and used nowadays in production systems are policy based and rely on simple if-then threshold based triggers. Examples of such systems include Amazon Auto Scaling [17], Rightscale [18], and Google Compute Engine Autoscaling [19]. The wide adoption of this approach is mainly due to its simplicity in practice as it doesn't require pre-training or expertise to get it up and running. Policy based approaches are suitable for small-scale systems in which adding/removing a VM when a threshold is reached (e.g., CPU utilization) is sufficient to maintain the desired SLO. For larger systems, it might be non-trivial for users to set the thresholds and the correct number of VMs to add/remove.

B. Research Approaches

Most of the elasticity controllers, which go beyond a simple threshold based triggers, require a model of the target system in order to be able to reason about the status of the system and decide on control actions needed to improve the system. The system model is typically trained offline using historical data and the controller is tuned manually using expert knowledge of the expected workload patterns and service behavior.

Work in this area focuses on developing advanced models and novel approaches for elasticy control such as, ElastMan [4], SCADS Director [3], scaling HDFS [2], ProRenata [5], and Hubbub-scale [6]. Although achieving very good results, most of these controllers ignore the practical

aspects of the solution which slowed down the adoption of such controllers in production systems. For example, SCADS Director [3] is tailord for a specific storage service with prerequisits that are not common in storage systems (fine grained monitoring and migration of storage buckets). ElastMan [4], uses two controllers in order to efficiently handle diurnal and spiky workloads but it requires offline manual training of both controllers. Lim et al. [2] on scaling Hadoop Distributed File System (HDFS) adopts CPU utilization, which highly correlates request latency, for scaling but it relies on the data migration API integrated in HDFS. ProRenaTa [5] minimizes the SLO violation during scaling by combining both proactive and reactive control approaches but it requires a specific prediction algorithm and the control model needs to be trained offline. Hubbub-Scale [6] and Augment Scaling [20] argue that platform interference can mislead an elasticity controller during its decision making, however, the interference measurement needs the access of many low level metrics, e.g. cache counters, of the platform.

OnlineElastMan, on the other hand, focuses on the research of the practical aspects of an elasticity controller. It relies only on the most generic and obtainable metrics from the system and alleviates the burden of applying an elasticity controller in production. Specifically, the auto-training feature of OnlineElastMan makes its deployment, model training and configuration effortless. Furthermore, an generic and extendable prediction model is integrated to provide workload prediction for various workload patterns.

VI. Conclusions and future works

In this paper, we have designed, implemented and open-sourced [2] OnlineElastMan, which is an "out-of-the-box" elasticity controller for distributed storage systems. It includes a self-training multi-dimensional performance model to alleviate model training efforts and provide better provision accuracy, a self-tuning prediction module to adjust the prediction to various workload patterns, and an elasticity controller to calculate and carry out the scaling decisions by analyzing the inputs from the performance model and the prediction module. The evaluation results of OnlineElastMan on Cassandra show that OnlineElastMan is able to provision a Cassandra cluster efficiently and effectively with respect to the percentile latency SLO in the showcase experiment.

For future work, the OnlineElastMan framework can be extended in two directions. First, it would be useful to extend the control model of OnlineElastMan with comprehensive metrics, e.g., CPU utilization, network statistics, disk I/Os, etc. Second, OnlineElastMan is essentially stateless. States are only preserved and used in the prediction and model training modules, which can be generated/trained during runtime. Thus, it is not difficult to decentralize OnlineElastMan for better scalability and fault tolerance.

Acknowledgment

This work was supported by the Erasmus Mundus Joint Doctorate in Distributed Computing funded by the EACEA of the European Commission under FPA 2012-0030 and the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research under the contract RIT10-0043. The authors would also like to thank the reviewers for their constructive comments and suggestions to improve the quality of the paper.

[2]https://github.com/gureya/OnlineElasticityManager

References

[1] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[2] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[3] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[4] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: Autonomic elasticity manager for cloud-based key-value stores. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 115–116, New York, NY, USA, 2013. ACM.

[5] Ying Liu, N. Rameshan, E. Monte, V. Vlassov, and L. Navarro. Prorenata: Proactive and reactive tuning to scale a distributed storage system. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 453–464, May 2015.

[6] Leandro Navarro Vladimir Vlassov Navaneeth Rameshan, Ying Liu. Hubbub-scale: Towards reliable elastic scaling under multi-tenancy. Accepted for publication on 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016.

[7] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[8] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16, Oct 2010.

[9] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: accelerating resource allocation in virtualized environments. *ACM SIGARCH Computer Architecture News*, 40(1):423–436, 2012.

[10] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. Technical report, 2013.

[11] Steve R. Gunn. Support vector machines for classification and regression. Technical report, University of Southampton, 1998.

[12] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *Network, IEEE*, 14(3):30–37, May 2000.

[13] R. Gusella. Characterizing the variability of arrival processes with indexes of dispersion. *Selected Areas in Communications, IEEE Journal on*, 9(2):203–211, Feb 1991.

[14] George Edward Pelham Box and Gwilym Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[15] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 108(2):212–261, February 1994.

[16] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[17] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[18] Right Scale. http://www.rightscale.com/.

[19] Google Compute Engine. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling.

[20] Leandro Navarro Vladimir Vlassov Navaneeth Rameshan, Ying Liu. Augmenting elasticity controllers for improved accuracy. Accepted for publication on 13rd IEEE International Conference on Autonomic Computing (ICAC), 2016.