Workload and Resource Aware Proactive Auto-Scaler for PaaS Cloud

R.S. Shariffdeen, D.T.S.P. Munasinghe, H.S. Bhathiya, U.K.J.U. Bandara, and H.M.N. Dilum Bandara
Department of Computer Science & Engineering
University of Moratuwa
Katubedda, Sri Lanka
{*ridwan.11, tharindu.11, bhathiya.11, bandaraukju.11, dilumb*}*@cse.mrt.ac.lk*

***Abstract*—Elasticity is a key feature in Cloud Computing where virtualized resources are provisioned and de-provisioned via auto-scaling. However, auto-scaling in most Platform-as-a-Service (PaaS) systems is based on reactive, threshold-driven approaches. Such systems are incapable of catering to rapidly varying workloads, unless the associated thresholds are sufficiently low. Alternatively, maintaining low thresholds leads to resource over-provisioning under relatively stable workloads. Moreover, thresholds are not a good indication of QoS compliance, which is a key performance indicator of a cloud application. Hence, it is nontrivial to determine an optimum threshold while minimizing costs and meeting QoS demands. We propose inteliScaler, a proactive and cost-aware auto-scaling solution to address these issues by combining a predictive model, cost model, and a smart killing feature. An ensemble workload prediction mechanism is introduced based on time series and machine learning techniques for making accurate predictions on drastically different workload patterns. Utility of the solution is demonstrated using both simulations and empirical evaluations using Apache Stratos PaaS (deployed on the AWS EC2), as well as RUBiS and real-world workload traces. Results show significant QoS improvements and cost reductions by inteliScaler compared to a typical reactive and threshold-based PaaS auto-scaling solution.**

Keywords: Auto-scaling; Cloud computing; PaaS; Prediction

I. Introduction

Auto-scaling is the process of dynamically allocating and deallocating resources for a particular application deployed in the cloud. This is of vital importance for clients to optimize their resource utilization. The goal of a cloud auto-scaling mechanism (i.e., auto-scaler) is to achieve higher Quality of Service (QoS) levels while minimizing the associated cost.

Cloud auto-scaling systems have to address two major challenges. First, an auto-scaler needs to be aware of the workload that the system has to deal with. Second, auto-scaler should allocate the right amount of resources to the system in a cost-effective manner while maintaining its QoS. Auto-scalers use two main approaches to address the first challenge. Depending on the selected approach, the auto-scaler would gain workload awareness in a proactive or reactive manner. In the reactive approach, the auto-scaling decision would be triggered by a predefined set of events. In contrast, proactive auto-scaling mechanisms forecast the workload ahead such that the auto-scaler can make decisions based on the anticipated workload instead of waiting for a trigger. A major challenge in cloud workload prediction is to come up with a solution that would perform well against drastically different workload patterns. For example, a model that typically performs well on workloads with seasonal trends does not perform well with frequently fluctuating loads [1]. The reactive approach is commonly used in PaaS auto-scalers [2], [3]. It is relatively simple to implement, but greatly reduces the solution space available when it comes to addressing the second problem, i.e., resource allocation. Whereas an accurate understanding about the future workload enables better resource management such as proactive spawning of new a Virtual Machine (VM) and not killing a VM until end of the billing cycle. Hence, we have opted for proactive auto-scaling over reactive auto-scaling in our solution, as proactive mechanisms supported by accurate prediction mechanism enable auto-scalers to make more detailed decisions.

Much research work has been conducted (as discussed in Section II) in relation to resource allocation problem as well [4], [5]. However, almost all the available PaaS solutions are built on rule-based scaling. In the rule-based approach, for example, spin up decisions will be taken when the average memory consumption of the cluster of VMs is over 75%. Such mechanisms mostly rely on user defined threshold parameters for configuring policies or rules to govern the scaling decisions [6]. One of the major drawbacks with this type of rule based, threshold-driven auto-scaling is that the user is expected to be a domain expert, capable of setting up appropriate threshold values for memory usage and CPU utilization for a given application. Also, mapping application metrics such as response time and throughput to system-level metrics such as CPU utilization and I/O Operation per Second (IOPS) is nontrivial.

In this paper, we propose a proactive and cost-aware auto-scaling solution to address these issues by combining a predictive model, cost model, and a smart killing feature. We utilize a workload prediction mechanism based on time-series forecasting and machine-learning techniques. Experimental results show that the proposed ensemble method outperforms individual techniques, as well as some of the popularly used ensemble models, when it comes to accuracy. Moreover, we propose a greedy heuristic-scaling algorithm to address the resource allocation problem considering both the QoS and cost factors. In the algorithm, we introduce the idea of *penalty factors* for quantifying and incorporating performance degradations to the scaling model, an idea inspired by penalties introduced in Service Level Agreements (SLAs) of popular PaaS platforms such as Google App Engine. The scaling algorithm evaluates all possibilities and selects the optimum resource configuration

considering the lease cost and penalty due to performance degradation. This scaling mechanism mitigates the problem of having to incorporate threshold values as in rule-based scaling.

In addition to introducing a scaling algorithm to calculate the required resources, we take a novel perspective in addressing the auto-scaling problem by injecting pricing model awareness to our solution. In our opinion, ignorance of the pricing model in the scaling decision is a major drawback in current PaaS auto-scalers. For example, consider a typical application deployed on Amazon Web Services (AWS). On a sudden fluctuation in the workload, a typical auto-scaler would scale-out by spawning a new VM instance, and when the workload is back to normal, the auto-scaler would scale-in by killing one of the VM instances which has already been paid for an hour. Thus, the customer has effectively lost 50 minutes of utilization of the instance, though it has been paid. The *smart killing* feature, which we adapted to our solution from [7] would apprehend the utilization of each VM instance, and scale-in instances only when their lease periods are about to expire. It could result in extra saving by mitigating the requirement to spin up another instance on a sudden fluctuation of the workload within the paid hour.

In an attempt to implement our solution and demonstrate the utility of the proposed PaaS auto-scaler, we target Apache Stratos PaaS (however, the techniques detailed here are extensible to other PaaS systems as well). Apache Stratos [8] is an open source PaaS framework that encapsulates IaaS-level details to the level of reduced granularity and complexity of a PaaS, while offering multi-tenancy and multi-cloud deployment capabilities. Stratos supports multiple IaaS providers, including AWS, OpenStack, GCE (Google Compute Engine), and Microsoft Azure [8]. Stratos auto-scaler is based on policy-driven decisions, which performs workload prediction for a small time window (usually a few minutes) but does not utilize resource optimization approaches explicitly, thereby incurring unnecessary costs to the customer, such as over provisioning of resources and naive scale-down decisions. Apart from the typical characteristics of a PaaS auto-scaler, Stratos offers other features such as a modular architecture allowing easy modifications, and the availability of a mock IaaS as a component for testing and evaluation, which would greatly assist any research in terms of monetary and implementation cost.

We evaluated the proposed solution, namely inteliScaler, by deploying Apache Stratos on AWS Elastic Compute Cloud (EC2). We deployed the three-tier bidding benchmark RUBiS as the user application and experimented with several workload traces. Our results demonstrate that inteliScaler successfully scales the application in response to fluctuating workloads, without user intervention and without off-line profiling. More importantly, we compare our solution with existing rule-based triggers and show that inteliScaler is superior to such approaches.

Rest of the paper is organized as follows. Section II presents the related work. High-level architecture of inteliScaler is presented in Section III. Prediction model is presented in Section IV while cost-aware resource allocation is presented in Section V. Section VI and VII present simulation and experimental results, respectively. Concluding remarks are presented in Section VIII.

II. Related Work

Significant research has been conducted in the domain of cloud auto-scaling, particularly at IaaS level. However, we limit our focus to PaaS level and IaaS solutions that can be adopted to PaaS.

A pluggable auto-scaling system that adds hardware and QoS awareness while capturing the cost of a scaling decision to complement AppScale PaaS is presented in [7]. Authors seek to provide an auto-scaling solution, which learns the behavior of a web application and provides optimal scaling decisions on AWS using hot spares and spot instances. While the paper emphasizes the importance of QoS factors as well as cost model awareness, it lacks a reliable workload prediction mechanism that resource allocation mechanism can rely on. Dependable Compute Cloud (DC2) is an application agnostic, model driven, adaptive auto-scaling system proposed in [6]. DC2 employs a Kalman Filtering technique in combination with a queueing theoretic model to proactively scale resources according to the varying workload. DC2 addresses the important segment of auto-scaling, namely the removal of user input to specify scaling decisions. However, it does not capture the cost incurred by the scaling process and therefore is not a complete auto-scaling solution that is cost effective. SLA is an important factor when providing cloud resources as services. A SLA defines the contract between a service provider and a service consumer on an agreed QoS level. SLA-driven Cloud Auto-scaling (SCAling) an advanced implementation of cloud elasticity based on SLA [9]. It successfully handles the trade-off between profit and customer satisfaction level without requiring manual intervention. The main idea is to exploit the SLA requirements to propose dynamic resource provisioning.

Yang et al. [10] have used use a sliding window based Linear Regression Model (LRM) for workload prediction and showed a low prediction deviation. They have also proposed an auto-scaling mechanism to scale virtual resources at different resource levels in service clouds which combines real-time scaling and pre-scaling under three scaling techniques, namely self-healing scaling, resource-level scaling and VM-level scaling. Simultaneous optimization of resource cost, QoS and availability is a major challenge in the context of cloud auto-scaling. Roy et al. [11] have addressed this challenge with a resource allocation algorithm based on model predictive techniques, which allocates or deallocates machines to applications with the goal of optimizing their utility over a limited prediction horizon. As part of their research, they have used a second order ARMA filter for workload prediction on the World Cup 98 traces [12] and showed accurate results.

Even though extensive research has been conducted on various aspects of cloud auto-scaling [13], [14], [15], [16], we have recognized that available PaaS cloud solutions stick to threshold driven, rule-based reactive auto-scaling. This is mainly due to the lack of a solution that addresses both future workload prediction and resource allocation.

III. Overall Solution - inteliScaler

We propose an automated scaling service for the PaaS systems that would take scaling decisions based on the characteristics of the application while considering the operational cost including the penalty cost. As seen in Fig. 1 there are

two main components in our solution, each being individually responsible for one aspect of the scaling decision. *Workload Predictor* predicts the future workload by processing statistic from the Real Time Event Processor. Prediction results are then sent to the Resource Monitor. *Resource Monitor* is responsible for resource allocation and deallocation. Resouce Monitor consists of two subcomponents, namely Resource Quantifier and Cost Optimizer. *Resource Quantifier* decides the number of instances required to handle predicted or anticipated workload. *Cost Optimizer* handles qualitative aspect by deciding which instances to spin-down or which type of instance to spin-up while taking the pricing models of the underlying IaaS into consideration. Next, the prediction model and cost-aware resource allocation applicable for these components are discussed.

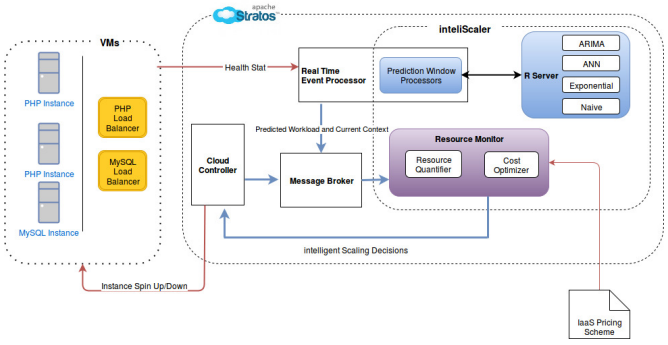

Figure 1. inteliScaler System Architecture.

IV. Prediction Model

A. Limitations of Existing Approaches

The time-variant metrics employed in measuring cloud workloads such as CPU utilization, memory consumption, and in-flight request count can effectively be treated as a set of time series, transforming the prediction problem to the more generalized notion of time series analysis. Time-series analysis of workload data is a popular research domain. Techniques like single order auto-regression, quadratic exponential smoothing and ARMA filters have been shown to produce accurate results in this regard [11], [17], [18]. In addition, machine learning techniques including sliding window based linear regression, Artificial Neural Networks (ANNs), and Hidden Markov Models (HMM) have been applied by other researchers [10], [19].

These researches often focus on specific datasets, so that the resulting models are not sufficiently capable of adapting to different workload characteristics present in multi-application environments. In addition, most models are derived using offline training, and hence do not dynamically adapt to latest workload variations. Given that specific time series prediction techniques like ARIMA and exponential moving average require certain conditions to be satisfied by the input datasets and require parameter adjustments to fit to specific datasets, an online training process is essential for obtaining an accurate fit.

While proactive auto-scaling can be beneficial for a PaaS cloud with diverse applications and drastically differing resource demands, such auto-scalers should be capable of accurately predicting future workloads over a reasonable time horizon in order for their proactive decisions to be effective. Making early scaling decisions, such that the required resources would be allocated in advance to cater for future workloads, is highly dependent on the availability of such accurate predictions. Following can be identified as key characteristics of a good cloud workload prediction model:

- Ability to predict accurately over a sufficiently large time horizon, considering factors like VM start-up and shutdown delays.
- Online training capability, as evolution and continuous learning of new workload characteristics are essential as the workload dataset grows with time.
- Bounded processing time per auto-scaling request.
- Immunity to overfitting to specific patterns, as it should be able to deal with different applications and hence a wide variety of dynamic workload patterns.

B. Proposed Approach

For adaptive workload prediction, we propose an error-based ensemble technique that tries to address situations where historical data is initially unavailable, making offline training impossible. Ensemble technique uses a mixture of prediction algorithms from time-serial analysis and machine leaning. As the dataset accumulates over time, the prediction algorithm will adapt its newer forecasts to latest characteristics of the accumulated workload data.

In existing error-based combining techniques, mean values of error metrics (e.g., absolute error, absolute percentage error, squared error, etc.) are taken into account while calculating the contributing factors for individual methods. Treating the currently available dataset as a time series $X = [x_1, x_2, .., x_t]$ we want to calculate predictions for x_{t+h}. If the final predicted value is $\hat{x}_{t+h}$ and the predicted value from the i-th model of the ensemble set-up for x_{t+h} is $\hat{x}^{(i)}_{t+h}$, the ensemble value $\hat{x}_{t+h}$ can be defined as a weighted sum of predictions from each model:

$$\hat{x}_{t+h} = \sum_{i=1}^{k} w_i \hat{x}^{(i)}_{t+h} \quad \forall k \in \{1, 2, 3, ..., n\} \tag{1}$$

where the i-th forecasting method is assigned a weight w_i. Considering that the weights should add up to unity for the sake of unbiasedness [20], we define the contribution from the i-th model to the final result as c_i, so that the above equation becomes:

$$\hat{x}_{t+h} = \frac{\sum_{i=1}^{k} c_i \hat{x}^{(i)}_{t+h}}{\sum_{i=1}^{k} c_i} \tag{2}$$

where $w_i = \frac{c_i}{\sum_{j=1}^{k} c_j}$. To determine the contribution coefficients c_i for our technique, we use past forecast errors of each model. Due to the need for the accurate predictions for the next time horizon, the contributions are calculated using inverses of the past forecasting errors.

Among the popular error quantification techniques, models whose errors are based on the last observation overlook overall accuracy and highlight the error of the last prediction, whereas average errors assign equal significance to all the last prediction errors. However, our situation requires assigning a larger

significance to errors in more recent predictions and smaller significance to earlier predictions. We address this requirement by using exponential smoothing to fit the past forecast errors, and the contribution coefficients $c_{i,t}$ are calculated using the inverses of the the fitted values (e_t). If $b_{i,t}$ is the fitted value of the past forecasting error from the i-th model at t-th time interval, $c_{i,t} = \frac{1}{b_{i,t}}$. Where e_t can be the absolute error, squared error, or absolute relative error at t-th prediction. $b_{i,t}$ can be defined as:

$$b_{i,t} = \alpha e_{(i,t)} + (1 - \alpha) b_{(i,t-1)} \tag{3}$$

where $0 \leq \alpha \leq 1$

$$b_{i,t} = \alpha e_{(i,t)} + \alpha(1-\alpha) e_{(i,t-1)} + \alpha(1-\alpha)^2 e_{(i,t-2)} + ...$$

$$c_{i,t} = \frac{1}{b_{i,t}} \tag{4}$$

For the proper use of the error-based weighting mechanism described above, it is necessary that the models being ensembled should capture different characteristics of workloads, and hence be collectively capable of covering a wide range of workload characteristics. After careful consideration, we have included an Autoregressive Integrated Moving Average (ARIMA) model which assumes a linear correlation structure among the time series values, a neural network model that can capture complex nonlinear relationships via a data-driven approach, an exponential model for preserving generality and capturing exponential patterns, and a naive forecast model which forecasts the last known data point for the next interval as a sentinel for mitigating situations where insufficiently trained models (especially the early-stage neural network) produce out-of-range forecasts. A detailed discussion on selection of these four models is presented in [21].

C. Proposed Prediction Algorithm

The proposed workload prediction algorithm can be defined as follows:

1) Consider the time series history window at time t, $X = [x_1, x_2, ..., x_t]$.
2) Calculate forecast value from the i-th time series forecasting method over the time horizon h, $\hat{x}_{t+h}^{(i)} \forall i \in \{1, 2, 3, ..., k\}$, where k is the number of forecasting methods used.
3) Fit a history window for the last t actual data points using the i-th method.
4) Use an exponential smoothing model to fit the errors resulting from Step 3, and use them to calculate the contribution factor for the i-th model at t, $c_{(i,t)}$.
5) Calculate the point forecast for time $(t + h)$ using $\hat{x}_{t+h} = \frac{\sum_{i=1}^{k} c_{(i,t)} \hat{x}_{t+h}^{(i)}}{\sum_{i=1}^{k} c_{(i,t)}}$.
6) At time $(t + 1)$, actual value for time $(t + 1)$ will be available. Add this value to the history window $X = X \cup \{x_{t+1}\}$ and repeat from Step 2.

While this algorithm does not specify a particular PaaS workload or performance metric, it can be used with any time-variant metrics such as CPU utilization, memory consumption, and request in flight.

Table I. COMPARISON OF WORKLOAD PREDICTION ACCURACY ON CLOUD DATASETS.

Model	Google Cluster		Memory		CPU	
	RMSE	MAPE	RMSE	MAPE	RMSE	MAPE
ARIMA	12.963	0.051	7.238	0.136	2.976	0.036
Exponential	12.886	0.041	7.005	0.160	3.150	0.048
Neural net.	12.530	0.036	8.169	0.135	**2.792**	0.031
Stratos	19.757	0.116	9.928	0.172	5.692	0.024
ARMA-based model	12.549	0.069	7.185	0.180	3.477	**0.023**
Mean Ensemble	12.099	0.051	7.036	0.130	2.900	0.029
Median Ensemble	12.059	0.055	7.010	0.141	2.944	0.028
Proposed Ensemble	**11.934**	**0.027**	**6.972**	**0.129**	2.873	0.027

D. Evaluation

We implemented the proposed prediction algorithm in R and tested it against a several publicly available datasets, as well as several real-world cloud workloads [22] collected from server applications. Our solution was compared with each of the constituent models in the ensemble solution, two other popular ensemble techniques (mean and median ensemble), the existing prediction technique in Apache Stratos, and the prediction model described in [11]. Table I shows the comparison of prediction approaches against CPU and memory usage traces obtained from a dedicated standalone server from a private cloud, as well as and a summarized portion of the Google cluster dataset [23], which includes a series of task execution details against their starting and ending times. Boldfaced and underlined values in the table denote instances of the best and worst performances observed for each dataset across all methods, respectively. For the evaluation we used T = 15 minutes, based on the fact that AWS offers hourly billing, adding a 5-minute lookahead time (for deciding whether the VM would be useful during the first few minutes of its next billing cycle as well, as the initialization of a new VM to a working state took about 5 minutes in our Stratos set-up) to the existing 10-minute grace period that we were using for smart killing operations. From the results it is clear that our proposed technique provides the best prediction results for several datasets while never performing worse than the individual prediction methods. See [21] for a detailed discussion on results.

V. COST-AWARE RESOURCE ALLOCATION

Most of the public PaaS solutions like Google App Engine provide SLAs based on the QoS provided. According to these SLAs, a penalty will be charged from the provider to the PaaS user in cases where the provider was unable to provide the agreed level of service. In most cases this penalty is defined as a percentage of the monthly cost of users. We propose a greedy heuristic scaling algorithm inspired by already existing PaaS SLAs. However, unlike most PaaS providers, we do not consider service uptime as the measure of QoS, instead we consider any performance degradation with respect to the metric considered (e.g., memory consumption, CPU utilization, and requests in flight) as a violation.

A. Scaling Algorithm

Considering both the resource cost and the penalty for performance degradation, we define the total cost in the next T minutes as follows:

$$C_t(n) = C_{ins}.n + C_{ins}.n.f(\frac{v_i}{T}.100) \tag{5}$$

$$n_{opt} = argmin_{n \in N \ \wedge \ n \in [min,max]} C_t(n) \quad (6)$$

Here f is a predefined function which calculates penalizing factor based on the percentage of performance degradation. T is the total time for prediction, $C_t(n)$ is the total cost for time T, and C_{ins} is the cost for an VM instance, n is the number of instances, and v_i is the violation time.

Hypothetical example in Fig. 2 demonstrate how the proposed algorithm works. Using the above definition, we calculate the total cost for different numbers of instances n iterating through minimum to maximum VM count. Increasing the number of instances increases resource cost, but decreases penalty factor (and therefore the violation cost). Considering values from the minimum number of VMs (shown by the bottommost line in Fig. 2) to the maximum number of VMs allowed (shown by the topmost line), we decide on an optimum number that minimizes the total cost. Such an exhaustive search is made possible by the bounded and relatively small range of VM count in most set-ups.

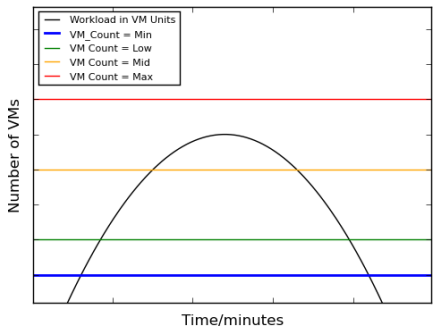

Figure 2. Calculating optimum number of VM instances n_{opt}.

B. Pricing Model Awareness

Our proposed solution also considers the pricing model of the underlying IaaS layer while taking auto-scaling decisions. We adapted the smart killing feature proposed in [7] after evaluating the concept. Cloud providers like AWS bill customers on a per-hour basis, which means a user will be charged for one hour, even if an instance is used only for a few minutes. Smart killing suggests that an instance should not be spin downed, if it has not completed a full billing cycle. Considering practical issues such as the time required to gracefully shut down an instance, we spin down an instance only if it used for 50 to 57 minutes in its billing hour. However, smart killing is only useful for IaaS models with relatively long billing cycles.

C. Evaluation of Resource Allocation

1) Behaviour of Different Auto-scaling Approaches: We evaluated proposed approach using different workloads. Fig. 3, 4, and 5 show resource allocation of proposed algorithm based on Load Average Statistics from RUBIS workload on AWS deployment.

Two graphs in Fig. 3 shows the variation of VM instance count with and without smart killing when the prediction mechanism of (original) Stratos is used, along with an 80% threshold value to calculate the required number of instances. Graphs in Fig. 4 and 5 show possible combinations of reactive

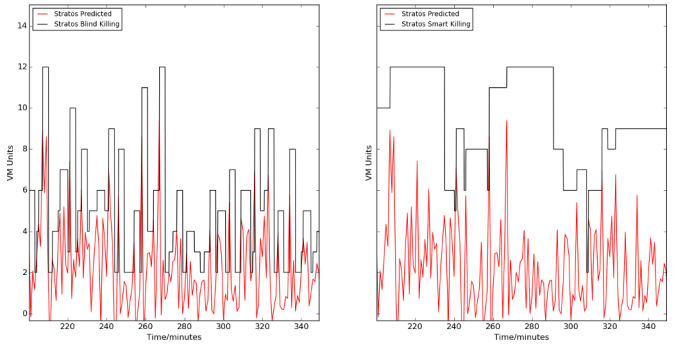

Figure 3. Simulation results using existing Apache Stratos prediction mechanism and threshold base scaling.

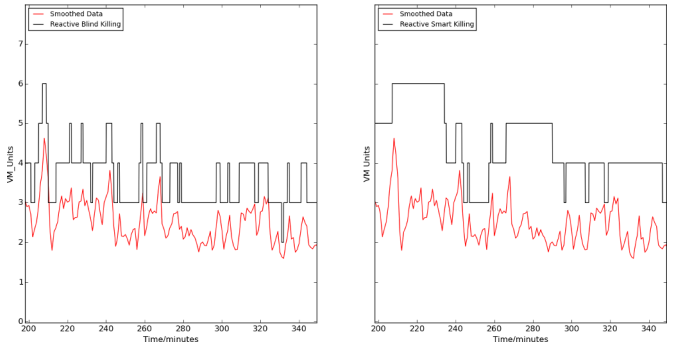

Figure 4. Simulation results using reactive auto-scaling with threshold base scaling.

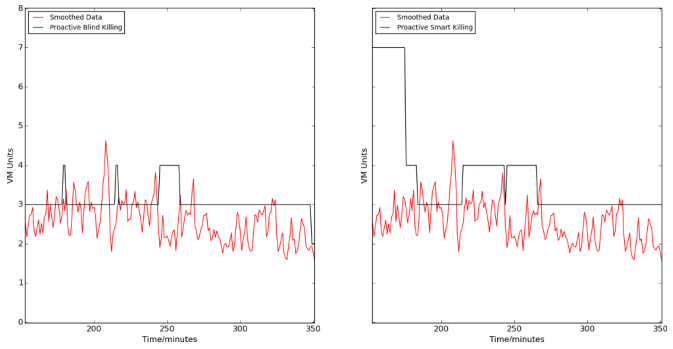

Figure 5. Simulation results using proactive auto-scaling with proposed scaling algorithm.

and proactive auto-scaling approaches with and without smart killing. An 80% threshold level is used in reactive solutions as well. In the proactive approach, the proposed heuristic is used with the following penalty function:

$$f(x) = \begin{cases} 0 & \text{if } 0 < x \leq 0.05; \\ 0.1 & \text{if } 0.05 < x \leq 1; \\ 0.2 & \text{if } 1 < x \leq 5; \\ 2^{\frac{x}{20}} & \text{if } 5 < x \leq 100; . \end{cases}$$

Fig. 6 shows the variation of cost over time for different auto-scaling approaches on AWS. It can be noted that blind killing combined with Apache Stratos incur the highest cost

Table II. BEHAVIOUR OF PROACTIVE SOLUTION WITH DIFFERENT VM INSTANCE TYPES.

Units	Resource Cost	Violation Cost	Total Cost	Violation Percentage
X	**3.25**	0.235	**3.485**	9.26
2X	3.617	0.240	3.857	3.738
4X	5.09	**0.192**	5.282	**0.932**

(of over 34USD in 400 minutes), while proactive solution combined with smart killing leads to the lowest cost (less than 3USD in 400 minutes). From the results in resource utilization and cost graphs, it can be observed that by introducing smart killing feature for auto-scaling improves resource utilization while reducing the cost significantly, regardless of the auto-scaling approach (reactive or proactive). It can be concluded that the proposed proactive scaling approach outperforms the reactive threshold approach, considering QoS and resource cost.

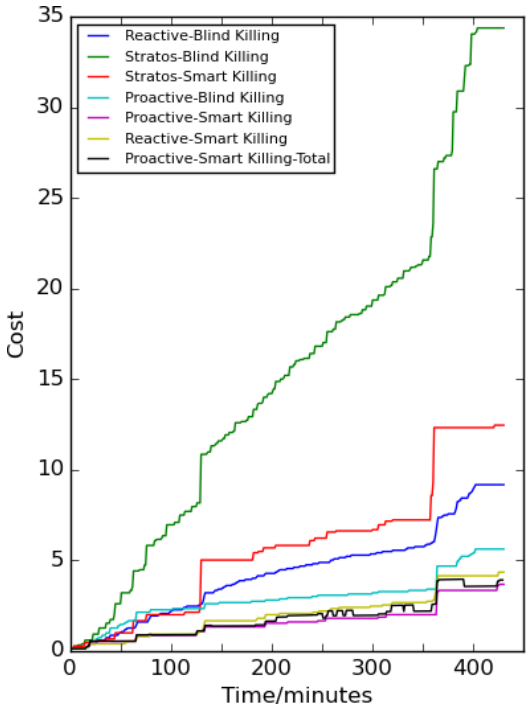

Figure 6. Variation of cost with time for different auto-scaling solutions.

2) Evaluation of Scaling Algorithm With Different Instance Sizes: We evaluated the behavior of proposed scaling algorithm with different instance types.

It can be observed from Table II that smaller the VM instance, lower the resource cost and total cost. But percentage of performance violations tends to be higher. Alterntively, larger VM instances incur higher resource cost but lower performance degradation. Considering the added complexity in handling heterogeneous platforms, we entirely focused on homogeneous clusters when implementing the solution on Apache Stratos.

VI. EXPERIMENTAL RESULTS

Here we evaluate inteliScaler's integration into Apache Stratos by deploying it on top of AWS EC2. We begin by presenting our experimental methodology and then discuss the results.

A. Experimental Setup

To evaluate the scaling process of Apache Stratos we cannot simply use either a simulation or the built-in mock IaaS feature, as they do not capture every aspect of a real system such as fault detection and handling, spawn time delay, and in-flight requests count. Hence, we deployed Apache Stratos 4.1.4 on AWS EC2 in a distributed setup to run a few workloads using the RUBiS benchmark and allow the system to scale using default policies.

Configuration details of the setup are given in Table III. At the time of evaluation, the latest stable version of Apache Stratos was 4.1.4. Apache ActiveMQ was chosen as it is the message broker recommended by Apache Stratos, although other brokers like RabbitMQ are also known to be supported. We tried almost all the options for the load balancer that Apache Stratos supported (according to their documentation), but we could only manage to set up HAProxy as the load balancer with a few modifications on our own. WSO2 CEP was the only choice for the event processor as the implementation is heavily coupled with the architecture of WSO2 CEP. We used a central database and scaled only the PHP instances, avoiding the trouble of syncing databases across multiple nodes which would have been the case, if a MySQL was also provisioned on a cluster. For the central database we had two setups, one using AWS RDS and the other with our own dedicated server. Because AWS RDS had certain limitations such as the maximum number of concurrent connections, we set up our own node by tuning its performance to handle up to 5,000 concurrent connections. Two extra nodes were set up on the same network to enumerate the workload for the RUBiS application deployed on top of Apache Stratos.

Table III. APACHE STRATOS SETUP CONFIGURATION.

Stratos Component	Implementation	EC2 Instance Type	Details
Stratos	Apache Stratos 4.1.4	t2.medium	This includes Stratos Manager and Auto-Scaler
Message Broker	Apache ActiveMQ 1.8	t2.medium	Used to communicate between each component and Cartridge Agent on each node
Load Balancer	HAProxy 1.6	t2.medium	One static load balancer to handle application requests
Complex Event Processor	WSO2 CEP 3.1	r3.large	Aggregates all health stats gathered from each node in real-time and produces average, gradient, derivative values
Database	MySQL	r3.large	Central database for all nodes to read/write
Load Generator	Rain Tool Kit	r3.large	Creates multiple connections with the Load Balancer to emulate the stipulated users and their actions

B. Evaluation

We ran several workloads, with drastically different characteristics, on the actual implementation of Apache Stratos and inteliScaler deployed in AWS EC2 setup.

1) Workload I: First we tested the same workload that was used earlier as shown in Table IV against inteliScaler. Configurations of Stratos are given in Table V.

It is clear that inteliScaler outperforms both the configurations of Stratos as shown in Fig. 7 inteliScaler has allocated less number of VM's to handle the same workload. In terms of QoS, inteliScaler has a success ratio of 91.54%, which is significantly higher than the 80% of low threshold configuration and 68% of high threshold configuration. Number of time out errors is also significantly low where in inteliScaler it is 0.8%, while low threshold configuration has 0.9% and high threshold configuration has 4.2%.

Table IV. WORKLOAD USED FOR AWS PERFORMANCE ANALYSIS.

Segment	1	2	3	4	5	6
Duration (seconds)	240	240	240	240	240	240
Users	400	800	1600	3200	800	400
Transition (seconds)	30	30	30	30	30	30

Table V. AUTO-SCALING POLICY OF STRATOS FOR DIFFERENT CONFIGURATIONS.

Setup	Load Average	Memory Consumption	Request in Flight
Low Threshold	70	70	100
High Threshold	95	95	150

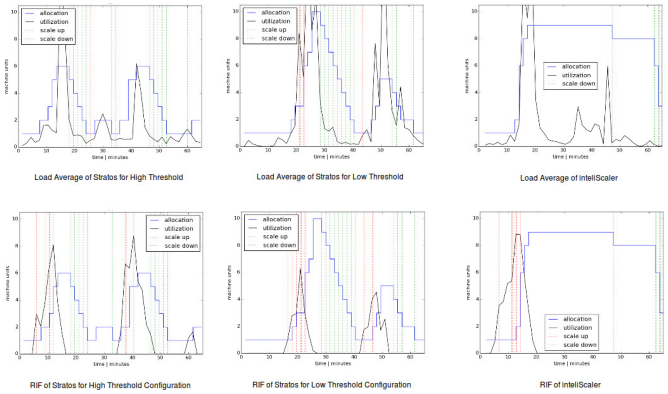

Figure 7. Performance comparison in AWS EC2 setup for Workload I.

Table VI. QOS SUMMARY FOR EVALUATION FOR WORKLOAD I ON AWS EC2

Test Case	Average Response Time	Requests Initiated	Requests Completed	Time Out Errors
low threshold	6.3839 s	254535	203067	2511
high threshold	4.4954 s	280477	191891	11863
inteliScaler	5.1288 s	286180	261979	2278

2) Workload II: We also tested both Stratos and inteliScaler with a fluctuating yet growing workload as described in Table VII.

As shown in Fig. 8, inteliScaler has the lowest resource allocation of 10 VMs whereas for the same workload Apache Stratos allocated 15 VMs, saving the cost for 5 VMs. Also the QoS of inteliScaler is better than Stratos as shown in Table VIII. inteliScaler has responded to 94.24% of the requests generated whereas Stratos has responded to only 57.41% of the total requests generated. There are significant number of time outs and drop off errors in Stratos compared to inteliScaler, which is the reason for these figures.

Table VII. WORKLOAD II FOR AWS SETUP.

Segment	1	2	3	4	5	6	7	8	9	10
Duration (s)	90	90	90	90	90	90	90	90	90	90
Users ($\times 100$)	2	1	4	2	6	4	8	6	10	8
Transition (s)	30	30	30	30	30	30	30	30	30	30

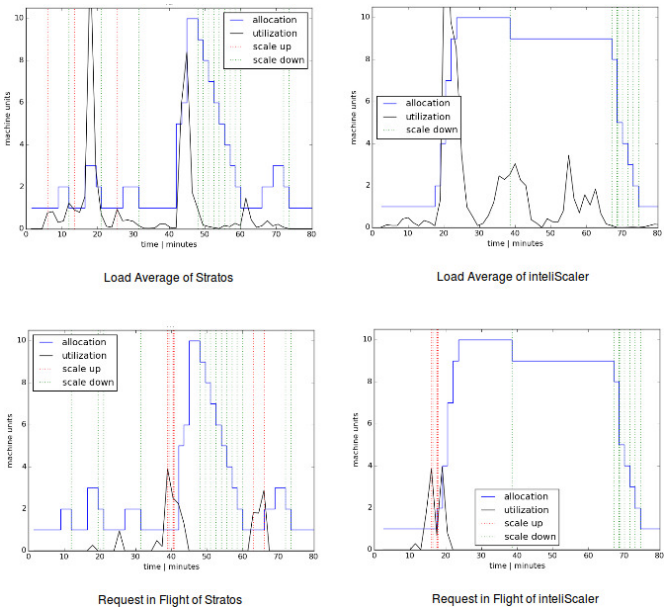

Figure 8. Performance comparison in AWS EC2 setup for Workload II.

Table VIII. QOS SUMMARY FOR EVALUATION OF WORKLOAD II ON AWS EC2.

Test Case	Average Response Time	Requests Initiated	Requests Completed	Time Out Errors
Stratos	2.7150 s	178848	102671	16257
inteliScaler	1.1207 s	214041	201719	6988

VII. FURTHER IMPROVEMENTS

The current implementation of inteliScaler is focused on a limited set of instances, since we were not able to conduct research for all the instance types available at AWS due to the cost factor. For completion of the solution, a performance model is required for mapping the workload requirement with a suitable instance type configuration available at the IaaS layer. This depends heavily on the application being deployed, but in general we will be able to accurately define the limitations of each resource type for a generic workload.

Our solution, as well as the current auto-scaling implementation on Apache Stratos assumes homogeneity of the worker nodes (members or instances). Although this results in a simpler resource management model, it prevents Stratos from enjoying many benefits and profitable aspects of heterogeneous deployment, including spawning of instances with specialized resource capacities (e.g., a memory-optimized instance, when the cluster is facing memory deficiencies rather than high load averages or requests-in-flight counts) and choosing

combinations of different instance types to fulfill a given resource demand while minimizing the total cost. However, implementing heterogeneity on the current Stratos architecture would include a substantial amount of changes to the existing code base. Moreover, our initial work on this aspect shows that the problem is difficult to model and solution space resulting from a simple model is too large to search.

VIII. Summary

Auto-scaling is of vital importance to realize the full potential of cloud computing. However, current auto-scaling solutions available in PaaS cloud remain primitive. Almost all the PaaS providers rely on reactive approach and lacks workload awareness needed for detailed decision making and rely on rule-based decision making which expects users to set threshold parameters. To address these problems, we proposed auto-scaling solution called inteliScaler with an ensemble workload prediction mechanism based on time series and machine learning techniques, scaling algorithm that considers both cost and QoS factors when deciding the amount of resources required, and pricing model-aware decision making. We tested the proposed model in two levels. First, we evaluated the performance of individual components on simulation set-up with various workload datasets available. Second, we developed an entire solution on Apache Stratos PaaS framework and tested with a deployment on Amazon EC2. Empirical results in both levels show significant benefits for PaaS users. It is possible to improve the proposed heuristic by introducing different penalty functions based on the level of service expected by different users. For example, an application (deployed on a PaaS) that supports free vs. paid versions would require different service levels. Our proposed solution can be adapted for such scenarios by defining different penalty functions based on the type of subscription.

IX. Acknowledgement

This research is supported in part by the Amazon Web Service (AWS) educational research grant.

References

[1] H. Alipour, Y. Liu, and A. Hamou-Lhadj, "Analyzing auto-scaling issues in cloud environments," in *24th Annual Intl. Conf. on Computer Science and Software Engineering*. Riverton, NJ, USA: IBM Corp., 2014, pp. 75–89. [Online]. Available: http://dl.acm.org/citation.cfm?id=2735522.2735532

[2] "Application Scaling," https://developers.openshift.com/en/managing-scaling.html, [Online; accessed 19-July-2008].

[3] "Getting started with the Auto-Scaling service," https://www.ng.bluemix.net/docs/services/Auto-Scaling/index.html, [Online; accessed 19-July-2008].

[4] S. Khatua, A. Ghosh, and N. Mukherjee, "Optimizing the utilization of virtual resources in Cloud environment," in *Proc. IEEE Intl. Conf. on Virtual Environments Human-Computer Interfaces and Measurement Systems*, sep 2010. [Online]. Available: http://dx.doi.org/10.1109/vecims.2010.5609349

[5] 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, jun 2011. [Online]. Available: http://dx.doi.org/10.1016/j.future.2010.10.016

[6] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *Proc. 11th Intl. Conf. on Autonomic Computing*, Philadelphia, PA, June 2014.

[7] C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hegde, and A. Srivastava, "A pluggable autoscaling service for open cloud PaaS systems," in *Proc. 5th IEEE Intl. Conf. on Utility and Cloud Computing*, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1109/ucc.2012.12

[8] "WSO2 Private PaaS," http://wso2.com/cloud/private-paas, [Online; accessed 19-July-2008].

[9] Y. Kouki and T. Ledoux, "SCAling: SLA-driven cloud auto-scaling," in *Proc. 28th Annual ACM Symposium on Applied Computing*, New York, NY,gin, Mar., pp. 411–414.

[10] J. Yang, C. Liu, Y. Shang, Z. Mao, and J. Chen, "Workload predicting-based automatic scaling in service clouds," in *Proc. 6th IEEE Intl. Conf. on Cloud Computing*, June 2013. [Online]. Available: http://dx.doi.org/10.1109/cloud.2013.146

[11] N. Roy, A. Dubey, and A. Gokhale, "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting," in *Proc. 4th IEEE Intl. Conf. on Cloud Computing*. Institute of Electrical & Electronics Engineers (IEEE), jul 2011. [Online]. Available: http://dx.doi.org/10.1109/cloud.2011.42

[12] M. Arlitt and T. Jin, "1998 World Cup web site access logs," 1998. [Online]. Available: http://www.acm.org/sigcomm/ITA/

[13] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, jan 2012. [Online]. Available: http://dx.doi.org/10.1016/j.future.2011.05.027

[14] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive Elastic ReSource Scaling for cloud systems," in *Proc. Intl. Conf. on Network and Service Management*, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1109/cnsm.2010.5691343

[15] L. R. Moore, K. Bean, and T. Ellahi, "Transforming reactive auto-scaling into proactive auto-scaling," in *Proc. 3rd Intl. Workshop on Cloud Data and Platforms - CloudDP '13*, 2013. [Online]. Available: http://dx.doi.org/10.1145/2460756.2460758

[16] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, Dec. 2014.

[17] J. Kupferman, J. Silverman, P. Jara, and J. Browne, "Scaling into the cloud," Available at http://cs.ucsb.edu/~jkupferman/docs/ScalingIntoTheClouds.pdf (2015/07/20), 2009.

[18] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan, "Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers," in *Proc. IEEE Intl. Conf. on Services Computing*, July 2010. [Online]. Available: http://dx.doi.org/10.1109/scc.2010.69

[19] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *Proc. IEEE Network Operations and Management Symposium*, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1109/noms.2012.6212065

[20] R. Adhikari and R. K. Agrawal, "Combining multiple time series models through a robust weighted mechanism," in *Proc. 1st Intl. Conf. on Recent Advances in Information Technology (RAIT)*, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1109/rait.2012.6194621

[21] R. S. Shariffdeen, D. T. S. P. Munasinghe, H. S. Bhathiya, U. K. J. U. Bandara, and H. M. N. D. Bandara, "Adaptive workload prediction for proactive auto scaling in PaaS systems," in *Proc. 2nd Intl. Conf. on Cloud Computing Technologies and Applications (CloudTech '16)*, Marrakesh, Morocco, May 2016.

[22] "AutoscaleAnalyser," https://github.com/hsbhathiya/AutoscaleAnalyser/tree/master/datasets/cloud_traces, 2015.

[23] J. L. Hellerstein, "Google cluster data," Jan. 2010. [Online]. Available: http://googleresearch.blogspot.com/2010/01/google-cluster-data.html