FOX: Cost-Awareness for Autonomic Resource Management in Public Clouds

Veronika Lesch
University of Würzburg
Würzburg, Germany
veronika.lesch@uni-wuerzburg.de

André Bauer
University of Würzburg
Würzburg, Germany
andre.bauer@uni-wuerzburg.de

Nikolas Herbst
University of Würzburg
Würzburg, Germany
nikolas.herbst@uni-wuerzburg.de

Samuel Kounev
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

ABSTRACT

Nowadays, to keep track with the fast changing requirements of internet applications, auto-scaling is an essential mechanism for adapting the number of provisioned resources to the resource demand. In the context of public clouds, there exist different natures of cost-models for charging resources. However, the accounted resource units and charged resource units may differ significantly due to the applied cost model. This can lead to a significant increase of charged costs when using an auto-scaler as it tries to match the demand of the application as close as possible. In the literature, several auto-scalers exist that support cost-aware scaling decisions but they introduce inherent drawbacks.

In this work, this lack of existing cost-aware mechanisms is addressed by introducing a mediator between an application and the auto-scaler. This cost-aware mechanism is called FOX. It leverages knowledge of the charging model of the public cloud and reviews the scaling decisions found by the auto-scaler to reduce the charged costs to a minimum. More precisely, FOX delays or omits releases of resources to avoid additional charging costs if the resource is required in the future. Hereby, FOX is not restricted to use one specific auto-scaler but offers interfaces to use any auto-scaler.

For an evalation under controlled conditions, FOX scales a multi-tier application deployed in a private cloud that is stressed with two real world workloads: BibSonomy and IBM CICS. As FOX provides an interface for auto-scalers, we evaluate the cost-aware mechanism with three state of the art auto-scalers: React, Adapt, and Reg. The experiments show that FOX is able to reduce the charged costs by 34% at maximum for the Amazon EC2 charging model. According to the cost model, FOX provisions more resources than required. This results in a decreased SLO violation rate from 28% to 2% at maximum. The accounted instance time increases at max. by 30%.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184415

CCS CONCEPTS

• **General and reference → Cross-computing tools and techniques**; • **Networks → Cloud computing**; • **Computer systems organization → Self-organizing autonomic computing**; • **Software and its engineering → Virtual machines**;

KEYWORDS

Cloud Computing, Public Cloud, Auto-Scaling, Cost-Awareness, Charging Model

ACM Reference Format:
Veronika Lesch, André Bauer, Nikolas Herbst, and Samuel Kounev. 2018. FOX: Cost-Awareness for Autonomic Resource Management in Public Clouds. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184415

1 INTRODUCTION

In order to face the dynamic behavior respectively requirements of internet applications, cloud computing emerged as computing model that allows fast access to resources and has a high level of scalability. Due to these benefits, the usage of auto-scalers arose in cloud computing. The developed mechanisms try on the one hand to adapt the supplied resources as close as possible to the demanded resources; on the other hand they try to consider the predefined service level objectives. When using auto-scalers in public clouds, the desired effect of adapting the number of resources can lead to high costs as the accounted costs and the charged costs can deviate depending on the cloud. For example, if a virtual machine (VM) is charged hourly, the hour has to be paid although the accounted time is less than one hour. In order to minimize the charged costs, a cost-aware mechanism is required. While taking the future demand into account, the cost-aware mechanism modifies the auto-scaling decisions.

In this work, a cost-aware mechanism, called FOX, is proposed. FOX serves as mediator between an application deployed in a public cloud and an auto-scaler. The working principle of FOX bases on the MAPE-K control loop [16] and has additional knowledge of the cost-models. The main idea is to proactively plan the resource allocation and release. In order to reduce the charged cost, FOX modifies the found scaling decisions of the auto-scaler. If the resource is already charged and will be required in the future, FOX does not

stop the VM to avoid additional charging intervals. The design of FOX provides an interface for auto-scalers, i.e., FOX is able to add cost-aware functionality to any auto-scaler using this interface. Additionally, there is also an interface for a forecasting mechanism that allows to add different existing forecasters.

We evaluate FOX using a multi-tier application deployed under controlled conditions in a private cloud. The application is stressed using two real-world workloads: BibSonomy and IBM CICS. To show that FOX can add cost-awareness to multiple auto-scalers, three auto-scaling mechanisms are selected and evaluated: React [8], Adapt [2], and Reg [13].

The results of the experiments with the Amazon EC2 cost model show that FOX is able to decrease the charged costs, the cost you have to pay, for all auto-scalers by 34% at maximum. In addition to the cost reduction, the accounted instance time is increased which results in 26% less SLO violation at maximum. The elasticity metrics consider how well the supply curve fits the demand. FOX actively decides not to stop resources if they will be required, thus SLO improvements are achieved in trade for a slightly worse auto-scaling performance considering the elasticity metrics.

In order to investigate the impact of cost-aware mechanism for auto-scaling, we pose ourselves the following research questions: RQ1 *What kind of cost models exist and what are the popular ones?*, RQ2 *How can we modify the scaling decisions so that the charged costs are reduced?*, and finally, RQ3 *How well does FOX perform in the context of auto-scaling?*

The contributions of this paper align with the three addressed research questions and structure the paper as follows: in Section 2, we survey existing cost models, i.e., we answer RQ1. In Section 3, we address RQ2 by introducing the approach of FOX. Afterwards, the used tools are introduced in Section 4. Section 5 discusses the results of the experiments and addresses RQ3. In Section 6, we summarize related work before concluding the paper.

2 PUBLIC CLOUD COST MODELS

Multiple public infrastructure cloud provider exist each offering their own charging model. However, a classification into three groups can be found: hourly charging, two phase charging and minute-by-minute charging.

Hourly-based Charging. The first group, hourly charging, charges for every started hour regardless of stopped instances before the full hour is over. By this rough granularity, a huge charging overhead can occur, e.g., Amazon EC2[1], ORACLE Cloud[2], IBM Bluemix[3], Digital Ocean[4], and OHV[5] charge on an hourly basis.

Two-Phase Charging. The Google Cloud Platform[6] is a representative providing the two phase charging model. The first phase consists of a fixed interval of ten minutes, which has to be paid regardless of a shorter runtime. Afterwards, the model switches to the second phase, where a minute-by-minute charging is applied. This cost model introduces overheads for the first ten minutes of a virtual machine but afterwards they do not introduce large overheads.

Minute-Based Charging. The third group of public cloud providers charge the used resources minute-by-minute. So, all instance times are rounded to the next minute. This introduces a small overhead that is negligible when looking at the minute price. Example public cloud providers that charge on this basis are the Open Telekom Cloud [7], Microsoft Azure[8], and 1&1[9].

In this work, FOX considers the cost models of the first two categories: hourly charging and two phase charging. For the hourly charging, we expect that the costs can be lowered by a significant amount when using FOX, as there is a large overhead when rounding runtimes to the next full hour. For the second group, the two phase cost models, we expect that the cost savings are not as significant as they are for the hourly charging, as the overhead by rounding to the next full minute is very small.

Relation to Spot Markets. In addition to these groups where instances can be provisioned and released on demand, Amazon Web Services offer a Spot Market[10]. Here, the prices of instances vary dependent on supply and demand. The customer can specify a maximum price he wants to pay for an instance. If the price for an instance drops below this maximum, the instance is provisioned for this customer. The instance is released if the price rises above the maximum price the customer defined or if the customer stops the instance by himself. The cost-aware mechanism of FOX also supports a deployment with spot instances, as the logic how scaling decisions are modified to save costs is not affected. The deployment with spot instances introduces the risk that instances may not be provisioned on demand or are terminated by the platform if the actual price is higher than the bid. Thus, optimal bid placing could be the responsibility of another independent component and is not considered as a feature of FOX. To ensure reproducible results, the scenario using spot instances is omitted in the evaluation of FOX.

In this section, we address the research question RQ1: *What kind of cost models exist and what are the popular ones?* Among public cloud environments available, multiple cost models exist that can be classified into three groups: hourly charging, two phase charging and minute-by-minute charging. Example cloud providers for these cost models are Amazon EC2, Google Cloud Platform and Open Telekom Cloud. In this paper, we consider only the first two groups as the third group does not introduce large overheads that can be optimized by a cost-aware mechanism.

3 APPROACH

The main idea of FOX is that it operates as mediator between the auto-scaling mechanism and the application for adapting the associated scaling decisions based on a predefined cost-model. To this end, FOX contains a knowledge base, a forecast component and the interface for the auto-scaler. The knowledge base holds all found

[1] https://aws.amazon.com/de/ec2/pricing/on-demand/
[2] https://cloud.oracle.com/infrastructure/pricing
[3] https://www.ibm.com/cloud-computing/bluemix/de/pricing?lnk=hm
[4] https://www.digitalocean.com/pricing/#faq
[5] https://www.ovh.de/g677.informationen_zur_dedicated_cloud_abrechnung
[6] https://cloud.google.com/compute/pricing#machinetype
[7] https://cloud.telekom.de/fileadmin/CMS/Information/Kundenflyer/Open-Telekom-Cloud_Pricing-Models.pdf
[8] https://azure.microsoft.com/pricing/details/virtual-machines/linux
[9] https://hosting.1und1.de/cloud-computing
[10] https://aws.amazon.com/ec2/spot/pricing/

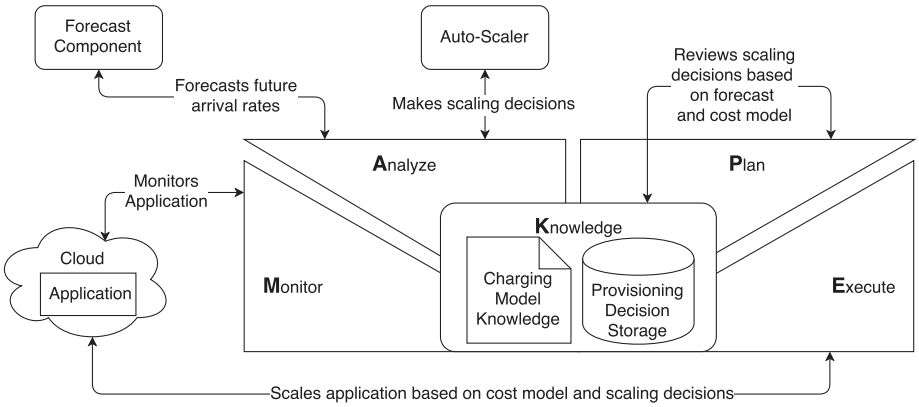

Figure 1: MAPE-K model of the Cost-Component.

scaling decisions from the auto-scaler, the predicted future arrival rates, and the knowledge about the cost model of the cloud platform provider. An overview of existing cost models are explained in Section 2. In our experiments, a simplistic forecaster returns the arrival rates of the last observerd day as forecast for the current day. As auto-scalers React, Adapt, and Reg are used (see Section 4). The auto-scaler and the forecast component can be replaced with other mechanisms. An example forecasting tool developed for auto-scaling contexts in cloud computing is the hybrid, decomposition-based approach called Telescope [28].

3.1 MAPE-K Adaptation

The approach of FOX is based on the MAPE-K control loop [16] as depicted in Figure 1. In the first phase called *Monitor*, FOX monitors the application and gathers information such as arrival rates and saves them into the knowledge base. As most of the existing auto-scalers in the literature are designed for homogeneous requests, i.e. single class case, FOX also holds this assumption. The monitoring interval is set to two minutes. Then, during the *Analyze* phase FOX fetches forecast values for the next 30 minutes from the forecast component. Based on these forecasts, the auto-scaler makes scaling decisions for all tiers for the next 15 intervals and saves them also in the knowledge base. While the application can consist of different resources, the resource types in each tier are assumed to be homogeneous. In the *Plan* phase, FOX reviews the scaling decisions based on the decisions found for the future forecasts and changes them according to the cost model. That is, for instance, that some scaling down decisions are delayed or cancelled according to the charging interval. Finally, in the *Execute* phase, FOX scales the application based on the adapted scaling decisions. The *Analyze*, *Plan* and *Execute* phases are described in more detail below.

Analyze: In the *Analyze* phase, FOX sends the observed arrival rate history to the forecaster component and receives the forecast values for the next 30 minutes, i.e., 15 forecast values. This is done every 15 minutes so that an overlap in forecasts exists. This overlap is required since FOX evaluates future events to adapt the scaling decisions. For each forecast value and each tier, the auto-scaler is called for making scaling decisions. The auto-scaler receives the forecast value via the interface, the amount of running VMs and the request rate that a single VM can handle at the specific tier. The amount of running VMs for the first forecast value is the amount of current running VMs. For the following forecast values, the planned amount from previous decisions are used. Based on this information, scaling decisions for each forecast value are made per tier and added to the knowledge base. From the second forecaster call on, the overlap of the decisions appears. As the new decisions have more recent information, the old decisions are omitted and replaced by the new ones.

Plan: For our experiments, FOX takes two common cost models into account: First, the Amazon EC2 model with an hourly charging, and secondly, the Google Cloud Platform model where the first ten minutes are charged fix and then the charging switches to a minutely basis, see Section 2 for a more detailed explanation of the cost models. The idea of FOX is to modify the current scaling decisions based on planned decisions for the future. Hereby, a down-scaling should be avoided when a VM will be required again in the near future. In case an up-scaling should be processed, the decision will not be modified. The decision logic how FOX changes the decisions is depicted in Figure 2 and summarized in Algorithm 1. First, FOX checks whether the current decision triggers a down-scaling (Algorithm 1, L. 1). If this holds (r.t. two lower cases in Fig. 2), all future decisions are fetched that are planned during the next charging interval (Algorithm 1, L. 2), i.e., one hour for Amazon EC2 and ten minutes for Google Cloud Platform. Then, FOX iterates over all future decisions (Algorithm 1, L. 3) and checks whether the amount of the future decision is higher than the amount of the current decision (Algorithm 1, L. 4), i.e., whether a down-scaling should be processed even if the VM will be required in the future. If this holds, the amount of the current decision is changed to the number of running VMs or the amount of the future decision, depending on which one is smaller (Algorithm 1, L. 5). In case the amount of the future decision is smaller than the amount of the current decision, the current decision is not modified. Finally, the revised decision is returned.

Execute: The *Execute* phase is responsible for scaling the application according to the found scaling decisions that are reviewed

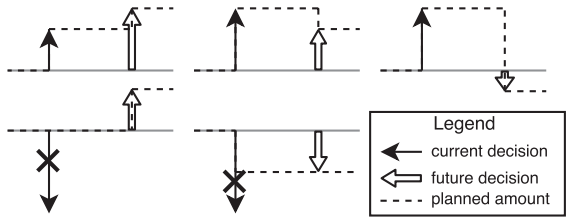

Figure 2: Decision logic for comparison to future decisions.

Algorithm 1: Revising auto-scaler decisions

Input: Decision current, runningVms run, chargInterval ci
Result: revisedDecision
/* if running VMs > amount of cur. decision, revise (acc. to Fig. 2) */
1 **if** *run > current.amount* **then**
2 futures = getFutureDecisionsInInterval(ci);
3 **foreach** *next in futures* **do**
4 **if** *next.amount > current.amount* **then**
5 current.amount = min(run, next.amount);
6 **return** *current*

by the cost component. In this phase, the cost component is important, again, as it can decide which VMs should be stopped in case of down-scaling to minimize financial loss. The procedure of this phase works as follows. First, the decisions for the current time are given to the execution component. In case of an up-scaling decision, the component provisions the required VMs. In case of a down-scaling decision, the execution component requests the VMs that introduce minimum financial loss if stopped from the cost component. To determine the VMs that should be stopped, the cost component takes the charging model into account. For the Amazon EC2 charging model, the runtime of all VMs are gathered. Then, the VMs that are closest to the next charging interval, i.e. one hour, are selected for down-scaling. For the Google Cloud Engine, the VMs are sorted descending by their overall runtime so that the VM which ran longest is at the beginning of the list. Then, the down-scaling amount of VMs is selected from the beginning of the list. So, the VMs with longest overall runtime are selected.

3.2 Discussion

In this section, we address the research question RQ2: *How can we modify the scaling decisions so that the charged costs are reduced?* FOX is designed according to the MAPE-K control loop. Here, the *Plan* phase consists the cost-aware mechanism where the knowledge of the cost model is used to review the existing scaling decisions from the auto-scaler to minimize costs. This mechanism is two-fold: First, the mechanism reviews all down-scaling decisions whether they are meaningful. That is, if a future decision defines that the instances that should be stopped will be required in a few minutes, the down-scaling is not executed or reduced executed. Second, in case of a reviewed down-scaling decision that should be executed, the VMs are stopped that reduce the lowest financial loss, i.e., the instances that are closest to the next charging interval are stopped in case of Amazon EC2. For the Google Cloud Platform cost model the VMs with the longest runtime are stopped.

4 TOOLS

The following section introduces three categories of tools that are used in this work: forecaster, auto-scaler and elasticity benchmarking framework.

4.1 Forecaster

The forecast component of FOX is able to access multiple different forecaster. In this work a simple forecaster is used where the values of the last day are returned as forecast values for the next day. So, the forecast value for the next interval is the observed value 24 hours earlier. Though, different forecasting approaches can be used. Telescope, e.g., is a hybrid forecasting tool that is designed to perform multi-step-ahead forecasts for univariate time series, while maintaining a short runtime [28]. Besides Telescope, tBATS [9] or ARIMA [1] can be used as forecaster as well.

4.2 Auto-Scalers

For the evaluation of FOX, we selected a subset of the most cited and public available auto-scalers proposed in the survey of T. Lorido-Botran et al. [17].

React: In 2009, Chieu et al. [8] present a reactive scaling algorithm for horizontal scaling.

React provisions VM resources based on a threshold or scaling indicator of the web application. The indicators consist of the number of concurrent users, the number of active connections, the number of requests per second, and the average response time per request. React gathers these indicators for each VM and calculates the moving average. Afterwards, the current web application VMs with active sessions above or below the given threshold are determined. Then, if all VMs have active sessions above the threshold, new web application instances are provisioned. If there are VMs with active sessions below the threshold and with at least one VM that has no active session, idle instances are removed.

Adapt: In 2012, A. Ali-Eldin et al. [2] propose a proactive auto-scaler that supports horizontal scaling. It contains a model of each service of the cloud based on a closed loop control system. Adapt models the infrastructure using queueing theory as G/G/n stable queue with variable number of servers n. Using this model the authors build two adaptive controllers that are parameter independent. Any performance metric can be used as controlled parameter. Adapt estimates the future service capacity using a gain parameter that determines the estimated change in the workload in the future. The two controllers are built by using two different gain parameters: the periodical rate of change of the system load and the ratio between the change in the load and the average system service rate over time.

Reg: In 2011, W. Iqbal et al. [13], introduce their proactive auto-scaler that uses response times to find scaling decisions to remove bottlenecks. A reactive model checks if the capacity is less than the load, and makes a scale-up decision. For down-scaling, a proactive mechanism decides when and how much to deprovision. Therefore,

a regression model is used to predict the number of VMs required at each time. This model is updated every time a new observation is added. The reactive mechanism feeds these observations to the proactive mechanism at every observation interval. Then, the model is recalculated using the complete history of the workload. If the current load is lower than the capacity, the model determines the required amount of VMs that can fulfill this load.

4.3 Elasticity Benchmarking Framework

In order to evaluate the two approaches, we use the BUNGEE Cloud Elasticity Benchmark controller [11]. The working principle is depicted in Figure 3. On the left side, the system under test (SUT) is depicted. It contains the IaaS cloud that hosts the multi-tier application and the scaling controller. On the right side, the experiment controller (BUNGEE) with its four phases is illustrated. First, in the System Analysis, the controller constructs a discrete mapping function for the SUT that determines the associated minimum amount of resources required to meet the SLOs (Service Level Objectives) for each load intensity. Then, the second phase, called *Benchmark Calibration*, uses the mapping from step one to generate identical changes in the curve of the demanded resource units on every platform under comparison. Based on this mapping and a predefined workload profile, the *Measurement* phase stresses the SUT while BUNGEE monitors the supplied VMs. Finally, in the *Elasticity Evaluation* phase, the elasticity and user-oriented metrics based on the collected monitoring data are calculated.

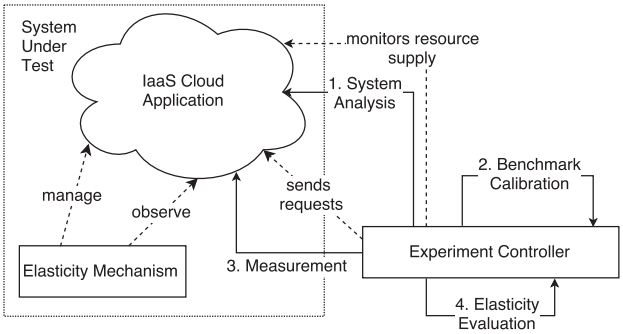

Figure 3: Elasticity Benchmarking Framework.

5 EVALUATION AND EXPERIMENT DESCRIPTION

The evaluation is split into multiple parts. First, we explain the elasticity metrics we use for evaluating the performance of the auto-scaling mechanism. Second, we introduce the cost metrics. We use these metrics to evaluate FOX and its cost saving potential. Third, we describe the experiment environment. Fourth, we explain the plots made for evaluating the scaling behavior of the auto-scaling mechanism without and with FOX in the methodology section. Fifth, we present the evaluation that contains detailed results for the experiments using React on the BibSonomy trace. Due to space limitations, we summarize the experiments for Adapt and Reg using BibSonomy in a table and omit the plots. Afterwards, we present the results of the experiment using React on the IBM workload using a detailed metric evaluation. Finally, we summarize the assumptions and limitations of the experiments and discuss whether the results can be generalized.

5.1 Elasticity Metrics

We use system-oriented elasticity metrics endorsed by the Research Group of the Standard Performance Evaluation Corporation (SPEC) [12] for quantifying the performance of FOX in context of auto-scaling. In particular, we use the provisioning accuracy and the wrong provision time share.

For the following equations, we define:

- T as the experiment duration and time $t \in [0, T]$
- s_t as the resource supply at time t
- d_t as the demanded resource units at time t
- n as the number of tiers

The demanded resource units d_t are the minimal amount of VMs required to meet the SLOs under the load intensity at time t. Δt denotes the time interval between the last and the current change either in demand d or supply s. The curve of demanded resource units d over time T is derived by BUNGEE, see Section 4. The resource supply s_t is the monitored number of running VMs at time t.

Provisioning accuracy θ_U and θ_O: The provisioning accuracy describes the relative amount of resources that are under-provisioned, respectively, over-provisioned during the measurement interval. In other words, the *under-provisioning accuracy* θ_U is the amount of missing resources normalized by the current demanded resource units that are required to meet the SLOs normalized by the experiment time. Similarly, the *over-provisioning accuracy* θ_O is the amount of resources that the auto-scaler supplies in excess. The range of this metric is the interval $[0, \infty)$, where 0 is the best value and indicates that the supply curve lays on the demand curve during the entire measurement interval.

$$\theta_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(d_t - s_t, 0)}{d_t} \Delta t$$

$$\theta_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(s_t - d_t, 0)}{d_t} \Delta t$$

Wrong provisioning time share τ_U and τ_O: The wrong provisioning time share captures the time in which the system is in an under-provisioned, respectively over-provisioned, state during the experiment interval, i.e., the *under-provisioning time share* τ_U is the time relative to the measurement duration, in which the system is under-provisioned. Similarly, the *over-provisioning time share* τ_O is the time relative to the measurement duration in which the system is over-provisioned. The range of this metric is the interval $[0, 100]$. The best value 0 is achieved, when the system has during the measurement no over- or under-provisioning.

$$\tau_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(d_t - s_t), 0) \Delta t$$

$$\tau_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(s_t - d_t), 0) \Delta t$$

Multi-tier Auto-scaling Deviation σ_n: In order to evaluate the performance of FOX according the introduced system oriented elasticity metrics, we propose to calculate the deviation of the scaling behavior across each tier compared to the theoretically optimal auto-scaler. The theoretically optimal auto-scaler is assumed to know the future load. Therefore, it knows when and how much the demanded resources change. For calculating the auto-scaling deviation, the aforementioned metrics provisioning accuracy (θ_U, θ_O) and wrong provisioning time share (τ_U, and τ_O) are considered. As these metrics calculate the deviation of the supplied resources to the demanded, the vector of the theoretical optimal auto-scaler is assumed to be the zero vector. For the determination of the deviation, we use the Minkowski distance with these vectors. If FOX is compared to the theoretically optimal auto-scaler, the L_p-norm can be used as the Minkowski distance between a vector and the zero vector is equal to the norm. We set p to the value 4 as we have four dimensions. Thus, we define the *multi-tier auto-scaling deviation* σ_n as follows, where n is the number of tiers:

$$\sigma_n[\%] = \left(\sum_{i=1}^{n} \left(\theta_{U,i}^4 + \theta_{O,i}^4 + \tau_{U,i}^4 + \tau_{O,i}^4 \right)^{\frac{n}{4}} \right)^{\frac{1}{n}}$$

SLO violation rate ϕ: In addition to the system oriented metrics provisioning accuracy and wrong provisioning time share, the Service Level Objective (SLO) violation rate is taken into account. This metric shows how many requests the application has handled within the specified SLOs. Therefore, the requests violating the SLO are divided by the amount of sent requests during the experiment. In this work, the SLOs are specified using the response time: 95% of all requests have to be handled within two seconds.

5.2 Cost Metrics

As FOX is a cost-aware mechanism, cost metrics are also taken into account for the evaluation. To this end, we consider the instance time, however, we have to distinguish between two different instance times: *accounted instance time* and *charged instance time*. The accounted instance time is the total runtime of all VMs of all tiers. The charged instance time is the runtime, the public cloud provider charges. Figure 4 shows both instance times for the Amazon EC2 pricing model. The red blocks represent the charged instance time and the green blocks the accounted instance time. Resource instance 1 has on the left an accounted instance time of 1.25 and is charged for two hours as all started hours are charged full no matter if the resource is stopped earlier. On the right the accounted instance time matches the charged instance time of one hour. The second resource instance is started three times and runs only for a few minutes each time. However, it is charged for three full hours, even if the previous charging interval is still running. The third resource instance runs for a bit more than two hours but is charged for three hours. So, all started hours are rounded to a full hour charged instance time. In addition, each start of the same instance is considered to be a completely new instance without recognition of previous and still running charging intervals.

Cost saving rate Π: For a quantification of the cost savings FOX can provide, we introduce the cost saving rate metric Π. This metric compares the instance times of the auto-scaler (cost_{AS}) to the

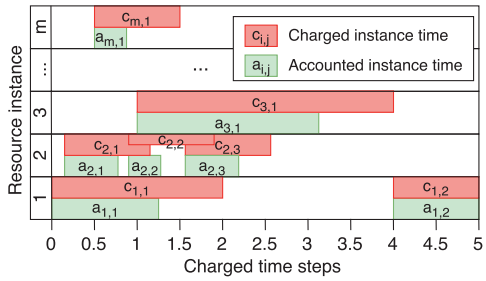

Figure 4: This example shows which instance times are accounted and which instance times are charged.

instance times of a naive approach ($\text{cost}_{\text{Naive}}$). The naive approach is assumed to provision all available resources at the start of the experiment and does not have any auto-scaling mechanisms, i.e., all available resources are running throughout the experiment. Both types of instance times are considered and a cost saving rate for accounted instance times (Π_a), respectively, charged instance time Π_c is calculated. The range of this metric is in the interval $(-1, \infty)$. If the value is negative, costs are saved. The lower the value is in the negative range the more costs are saved. If the value is greater or equals zero, the mechanism spends more or equal cost than the naive approach.

$$\Pi_x[\%] = 100 \cdot \left(\frac{\text{cost}_{\text{AS}}}{\text{cost}_{\text{Naive}}} - 1 \right)$$

5.3 Experimental Description

For the experimental evaluation, we designed a multi-tier application. It consist of three tiers with a standard workflow: The presentation tier (pt) receives requests and sends them to the business tier. An instance of the presentation tier has a processing rate of 17 requests per second. The business tier (bt) processes the forwarded requests but has a predefined number of serving units. This introduces a limitation of the number of parallel executions per instance to ten requests per second. Afterwards, the results are sent to the database tier (dt), that persists the results. The number of parallel database accesses per instance is limited to 25 per second. Finally, the results are sent back to the presentation tier that sends the response to the client. The tiers of the application are configured individually. There are different amounts of VM instances that can be provisioned per tier. At the presentation tier, 15 VMs can be provisioned. The business tier can be scaled to 25 VMs and the database tier can have 10 VMs. This configuration is made due to hardware limitations of the servers of our private cloud environment. Based on the request rates that can be served per tier and VM, 17 (pt), 10 (bt) and 25 (dt), the maximum arrival rate the application can handle with all VMs provisioned is 250 per second.

In order to stress the application with authentic workloads with time-varying behavior, we choose two real world traces: (i) BibSonomy and (ii) IBM. The *BibSonomy* represents HTTP requests to servers of the social bookmarking system BibSonomy (see the paper of Benz et al. [4]) during April 2017. The *IBM* Customer Information Control System (CICS) transactions trace captures four weeks of

recorded transactions on a z10 mainframe CICS installation. Each trace was sampled in 15 minute intervals, i.e., one day consists of 96 data points. For our experiments, we accelerate each trace by the factor of 7.5. That is, one data point for each two minutes. For having an internal repetition, we select two days for each trace.

The experiments are conducted in our private cloud infrastructure. The cloud consists of eleven homogeneous, virtualized Xen-Server hosts. Eight of them are managed by Apache CloudStack[11]. The distributed application is deployed on the CloudStack environment. The last three servers are not part of the CloudStack environment and are used for hosting (i) the load-balancer (Citrix Netscaler[12]) and the cloud management for CloudStack, (ii) the auto-scaling mechanisms and FOX, and (iii) the load driver and the experiment controller. The specification of each physical machine and worker VM can be found in Table 1.

Table 1: Specification of the Servers.

Criteria	Server	Worker VMs
Model	HP DL160 Gen9	–
Operating System	Xen-Server	Ubuntu 16.06
CPU	8 cores	1 vcore
Memory	32 GB	2 GB

5.4 Methodology

All shown figures in the following have the same structure: a demand versus supply graph for each tier at the top and a request evaluation at the bottom. All graphs have the experiment duration of about 385 minutes at the x-axis. The y-axis shows the number of VMs for the demand supply plot, and the requests per second for the request evaluation. The demand and supply graph shows the demand as black dashed line and the supply as a blue solid line. If the supply line falls below the demand line there are too less VMs provisioned. In case the supply line exceeds the demand line, too many VMs are instantiated. So, the optimal auto-scaler would result in a supply line matching the demand line during the experiment. The request evaluation graph shows the sent requests as a black dashed line, the requests processed conform to the SLO as green solid line and the requests that violate the SLOs as red dashed and dotted line. The sum of the SLO conform and SLO violation lines result in the sent request line. That is, if the green line matches the black line and the red line is zero during the experiment all requests have been served within the SLO. If the red line is not equal to zero and the green line drops below the black line, more SLO violations occured. An user-oriented auto-scaler tries to configure the application so that all requests can be served within the SLO and therefore, the green line should match the black line.

5.5 Experiment Results

As mentioned earlier, the evaluation of FOX is based on two different workload traces: BibSonomy and IBM. In addition, three different auto-scalers are used to show that FOX can improve the behavior of multiple auto-scalers. Due to space limitations, detailed evaluations are presented only for React without and with FOX for both workload traces. Additionally, for the BibSonomy workload trace plots are shown where the scaling behavior with and without FOX can be observed. Due to space limitations, the evaluation for the other auto-scalers is limited to the BibSonomy workload and the results are shown in summary in Table 3.

React on the BibSonomy workload: The scaling behaviors of React without, respectively with FOX are shown in Figure 5, respectively Figure 6. Figure 5 shows that React performs many adaptations to match the current demand. In some up-scaling cases, React starts the instances too late and under-provisioning occurs that results in increasing SLO violation rates at the bottom of the figure. However, React matches the current demand most of the time. Figure 6 shows the behavior of React with FOX using the Amazon EC2 cost model. At the top three plots, the unstable behavior of React is smoothed when using FOX. The supply curve tends to over-provision the amount of VMs. That is, the supply curve stays above the demand curve most of the time. However, some scaling actions are performed to reduce the amount of unused VMs if meaningful. As the supply curve lies above the demand curve most of the time, the SLO violations are reduced to a minimum as can be seen at the bottom of the figure. This is the behavior, we expected, as FOX performs down-scaling only if the instances that should be released will not be used in the future.

Table 2: Elasticity metrics results for React on the BibSonomy trace.

Tier	Metric	React	FOX$_A$	FOX$_G$
1	θ_U	2.65%	0.45%	0.62%
1	θ_O	33.29%	66.08%	57.91%
1	τ_U	16.48%	3.04%	3.92%
1	τ_O	68.05%	93.57%	91.01%
2	θ_U	6.10%	0.99%	1.53%
2	θ_O	20.80%	54.79%	47.25%
2	τ_U	35.34%	6.67%	8.17%
2	τ_O	52.18%	88.54%	85.45%
3	θ_U	2.22%	0.15%	0.47%
3	θ_O	27.93%	82.97%	62.66%
3	τ_U	13.45%	1.12%	2.65%
3	τ_O	57.97%	96.03%	91.71%
overall	ϕ	12%	3%	3%
overall	σ_3	89%	144%	134%

The evaluation of the elasticity metrics show the same results as observed in the figures. Table 2 shows the elasticity metrics for React without and with FOX for Amazon EC2 and Google Cloud Platform cost model on the BibSonomy workload. At the first tier, the provisioning accuracy for under-provisioning of React is about 2% while the provisioning accuracy for the experiments with FOX is about 0.5%, so the under-provisioning at the first tier is reduced by 75%. However, the over-provisioning accuracy for React is about 33% and for the experiments with FOX it is 66%, respectively 58% for Amazon EC2, respectively, Google Cloud Platform cost model. In addition, the under-provisioning time share of React without

[11] Apache CloudStack: https://cloudstack.apache.org/
[12] Citrix Netscaler: https://www.citrix.de/products/netscaler-adc/

FOX is 16% and for the experiments with FOX it is reduced to 3% and 4%. The over-provisioning time share for React is 68% and for the experiments with FOX larger than 90%. The scaling behavior at the other tiers is comparable to the one at the first tier. The SLO violation rate of React is 12% and for the experiments with FOX the violation rate is 3%. So, the SLO violation rate is reduced by 75% when using FOX. However, the auto-scaling deviation for React is 89% and for the experiments with FOX the deviation is larger than 100%. For comparison, the auto-scaling deviation for the Naive approach is 221%. So, a trade-off between SLO violation rate and auto-scaling deviation can be accessed. This is the expected behavior of FOX. It reduces under-provisioning phases by delaying or cancelling scaling down actions if future decisions state that the instances will be required. Hereby, the auto-scaler performance becomes worse in trade for a reduced SLO violation rate.

The evaluation based on the cost aspects is summarized in Table 3. For the BibSonomy workload all three auto-scalers are evaluated. First, the results of React are discussed. The cost saving rate for charged costs (Π_c) compares the charged instance time of the auto-scaler run with the naive scenario. The value of -5% shows that the charged costs are reduced by 5% when using React without FOX for the Amazon EC2 cost model. FOX is able to reduce the costs by 26% for the Amazon EC2 cost model. So, FOX saves 21% more costs than the experiment without FOX. This gain is caused by the down-scaling logic of FOX, as a down-scaling is only executed if the instances are not required in the near future. In the run without FOX, many instances are stopped due to the actual request rate but are again provisioned when the load increases. This introduces additional charging intervals to start with the Amazon EC2 cost model as described earlier. This behavior is reduced when using FOX and the charged costs are lowered. The cost saving rate for accounted instance times (Π_a) compares the accounted instance times to the naive approach where all instances run throughout the experiment. The value -45% for React with the Amazon EC2 cost model shows that the accounted instance time is reduced by 45% when using React in comparison to the naive approach. When using FOX the accounted instance time is only reduced by 28%, i.e., FOX supplies more instance time than React without FOX. This also results in a significantly reduced SLO violation rate from 12% when using React to 3% when using FOX for the Amazon EC2 cost model. However, this can only be achieved in trade for a worse auto-scaling performance in terms of elasticity metrics. In summary, for the Amazon EC2 cost model, the costs can be reduced by FOX while the accounted instance time is increased. When comparing React without FOX to React with FOX for the Google Cloud Platform cost model, it can be seen that the charged costs saving rate matches the accounted cost saving rate. So, there is no significant cost savings when using the Google Cloud Platform cost model for FOX. This can be explained by the cost model, as every minute is charged separately and there is no rounding to the next full hour as seen for the Amazon EC2 cost model.

Adapt on the BibSonomy workload: During the evaluation of Adapt with the BibSonomy workload the results show similar behavior as seen for React. When looking at the values of Adapt without FOX for the Amazon EC2 cost model, it can be seen that the SLO violation rate is 28% and the auto-scaling deviation is 86%. The accounted costs are reduced by 57% compared to the naive approach and the charged costs are only reduced by 40%. In comparison to the run with FOX, the SLO violation shows a significant decrease to 2% but with a doubled auto-scaling deviation. When looking at the cost saving rates, the accounted costs are reduced less, so FOX manages to supply more accounted instance time. In addition, the charged costs are reduced by 43%. The increased amount of accounted instance time results in a significant decrease of SLO violation rate from 28% to 2% in trade for auto-scaling performance. The run with the Google Cloud Platform cost model shows similar behavior. The SLO violation rate is reduced significantly from 28% to 3% when comparing Adapt without and with FOX. However, the auto-scaling deviation is doubled. The accounted instance time is reduced by 57% for the run of Adapt without FOX and the charged costs are reduced by 56%. The evaluation with FOX shows that the accounted instance time and charged costs are reduced only slightly but the SLO violation rate is reduced by a significant amount.

Reg on the BibSonomy workload: The third auto-scaler we evaluated is called Reg. The experiment using Reg without FOX shows specific characteristics of Reg. At random points of the experiment, drops in the supply curve can be detected, where all VMs are stopped and immediately provisioned in the next interval. The plots of the experiment using Reg with FOX show that these drops in the supply curve are removed by FOX. The results for elasticity metrics and cost saving rates are summarized in Table 3. The run of Reg without FOX for the Amazon EC2 cost model shows a SLO violation rate of 22% and an auto-scaling deviation of 77%. The accounted instance time is reduced by 54% in comparison to the naive approach and the charged costs are reduced by 5%. When using FOX, the SLO violation rate is reduced to 4% but the auto-scaling deviation increases. The accounted saving rate is 35%, so more accounted instance time is supplied in comparison to the run of React without FOX. The charged cost saving rate is 35% that is 30% higher than without FOX. So, for the Amazon EC2 cost model, FOX is able to supply more accounted instance time while reducing the charged costs. This also results in a significant lower SLO violation rate of only 4%. The evaluation of React without and with FOX using the Google Cloud Platform cost model shows that React without FOX has a SLO violation rate of 22% while the run with FOX has a reduced SLO violation rate of only 11%. The auto-scaling deviation of the run without FOX is 77% and slightly increased for the run with FOX. The accounted cost saving rate is 53% for the run without FOX and 39% for the run with FOX. The charged cost saving rate is reduced by React by 54% and for the run with FOX by 39%. So, FOX supplies more accounted instance time that results in a significant lower SLO violation rate.

React on the IBM workload: In order to evaluate FOX with different workloads, the IBM workload is selected in addition to the BibSonomy trace. Due to space limitations, only the evaluation of React using the elasticity metrics in Table 4 and the cost saving rates summarized in Table 3 are presented and the plots are omitted. First, the elasticity metrics are discussed. The under-provisioning accuracy is reduced for the first tier from about 2% to 1% with FOX. The over-provisioning accuracy is slightly increased from 86% to 91%, respectively 89%. The under-provisioning time share at the

Table 3: Cost metrics results for the BibSonomy trace.

	BibSonomy												IBM			
	React				Adapt				Reg				React			
Metric	$React_A$	FOX_A	$React_G$	FOX_G	$Adapt_A$	FOX_A	$Adapt_G$	FOX_G	Reg_A	FOX_A	Reg_G	FOX_G	$React_A$	FOX_A	$React_G$	FOX_G
ϕ	12%	3%	12%	3%	28%	2%	28%	3%	22%	4%	22%	11%	12%	4%	12%	5%
σ_3	89%	144%	89%	134%	86%	209%	86%	187%	77%	121%	77%	100%	143%	168%	143%	156%
Π_a	-45%	-28%	-45%	-32%	-57%	-43%	-57%	-11%	-54%	-35%	-53%	-39%	-59%	-51%	-59%	-53%
Π_c	-5%	-26%	-44%	-32%	-40%	-43%	-56%	-10%	-5%	-35%	-54%	-39%	-42%	-51%	-59%	-53%

Figure 5: Scaling behavior of React without FOX on the BibSonomy trace with Amazon EC2 cost model.

first tier is reduced when using FOX from about 8% to 5%. The over-provisioning time share is increased from 86% to 90%, respectively 89%. A similar behavior for the other tiers can be derived from the metrics in the table. The SLO violation rate of React is 12% while the rate for the experiments using FOX is reduced significantly to 4%, respectively 5%. The auto-scaling deviation shows a slight increase when using FOX. So, FOX focuses on a trade-off between auto-scaler performance and SLO violation rate.

The cost saving rates shown in Table 3 show for the run with Amazon EC2 cost model that React reduces the charged instance times by 42% compared to the naive approach. When using FOX the charged costs are reduced by 51%. The accounted instance time for React without FOX is reduced by 59% in comparison to the naive approach. FOX only reduces the accounted instance times by 51%. So, FOX supplies more accounted instance time while reducing the costs in comparison to the run of React without FOX. This increased accounted instance time results in a reduction from 12% to 4% SLO violations. However, this can only be achieved in trade for a worse auto-scaling performance in terms of elasticity metrics. When comparing the charged costs for the Google Cloud Platform charging model, For the Google Cloud Platform, the results show, that FOX supplies more accounted instance time, as the saving rate is lower as in the run without FOX, while the charged costs remain stable when using FOX. This can be explained by the charging interval of one minute, as the rounding overheads to the next charging intervals are very small. Though, the SLO violation rate is reduced from 12% to 5% when using FOX. Again, the auto-scaling deviation is increased when using FOX.

5.6 Threats to Validity

In order to perform the above discussed measurements several assumptions had to be made. These assumptions may reduce the expressiveness of the results. All assumptions and their effect on the results are discussed in the following. First, the set of experiments is run in a private cloud environment under controlled conditions for reproducible performance-related results. The cost and

Figure 6: Scaling behavior of React with FOX on the BibSonomy trace with Amazon EC2 cost model.

Table 4: Elasticity metrics results for React on the IBM trace.

Tier	Metric	React	FOX$_A$	FOX$_G$
1	θ_U	1.93%	1.11%	1.02%
1	θ_O	86.31%	90.91%	88.61%
1	τ_U	8.33%	5.59%	5.32%
1	τ_O	85.87%	90.11%	88.86%
2	θ_U	4.76%	2.07%	1.91%
2	θ_O	66.11%	112.59%	80.56%
2	τ_U	16.43%	8.29%	9.08%
2	τ_O	78.67%	88.62%	87.41%
3	θ_U	5.82%	0.97%	0.75%
3	θ_O	96.04%	105.07%	103.92%
3	τ_U	13.61%	3.97%	2.94%
3	τ_O	79.49%	93.62%	93.36%
overall	ψ	12%	4%	5%
overall	σ_3	143%	168%	156%

elasticity-related mechanisms work independent of an experienced performance variability of a public environment. Thus, the obtained results shall be meaningful for the public cloud environments. Second, the results are strongly dependent on the used auto-scaling mechanism. Therefore, three different auto-scalers that are introduced in the literature are used and compared. However, several other auto-scaler exist and the experiments could be expanded to include more auto-scalers. However, this work does not focus on optimal auto-scaling decisions. Third, the results of the cost-aware mechanism FOX depend on the quality of the forecast approach that is used. We used a simple forecasting method that returns the values of the last day as forecast. We proposed multiple alternative forecaster that could be included in the experiments to reduce variations in the results. Fourth, the experiments are conducted using one multi-tier application with homogeneous request types per tier. Other applications could behave in a different way and the results would be changed. Finally, only two workloads, BibSonomy and IBM are used to stress the application. The experiments should be expanded to use multiple other real world workload traces to show that FOX has similar results on many workloads. The results proposed in this paper cannot be generalized as there are too many assumptions and restrictions made for the experiments. However, the results have shown that FOX behaves as desired and is able to reduce the charged costs while increasing the accounted instance time and hereby reducing the SLO violation rate in the public cloud scenario.

5.7 Discussion

In this section, we address the research question RQ3: *How well does FOX perform in the context of auto-scaling?* The results of the evaluation show that FOX is able to decrease the charged costs significantly while increasing the accounted instance time for the Amazon EC2 cost model compared to the plain auto-salers. This results in reduced SLO violation rates but higher values for the metrics reflecting over-provisioning. However, this can only be achieved in trade for a higher auto-scaling deviation. For the Google Cloud Platform cost model, smaller improvements are measured. This can be explained by the nature of this cost model. All results are summarized in the following.

React on BibSonomy workload.

- React performs many scaling operations to match the demand as close as possible.
- With FOX, the supply curve is smoothed, tends to overprovision, and the oscillations are removed.
- FOX reduces the charged costs by 21% for the Amazon EC2 cost model.
- FOX provisions 17% more accounted instance time for the Amazon EC2 cost model and the SLO violation rate is reduced from 12% to 3% for the Amazon EC2 cost model.
- The auto-scaling deviation increases in trade for lowered costs for the Amazon EC2 cost model.
- The costs are not lowered for the Google Cloud Platform cost model.
- The accounted instance time is increased for the Google Cloud Platform cost model and the SLO violation rate is reduced.

Adapt and Reg on BibSonomy workload.

- When using FOX the charged costs are reduced by 3% for Adapt, respectively 30% for Reg with the Amazon EC2 cost model.
- The accounted instance times are increased by 14%, respectively 19%. This results in a decrease in the SLO violation rates of 26%, respectively 18% for the Amazon EC2 cost model.
- The results for the auto-scaling deviation for the Amazon EC2 cost model and the evaluation for the Google Cloud Platform cost model are similar to the results of React.

React on IBM workload.

- The charged costs can be reduced by 9% for the Amazon EC2 cost model, respectively 6% for the Google Cloud Platform cost model, when using FOX.
- The accounted instance time is increased when using FOX by 8%, respectively 6%. This results in a significant decrease of SLO violation rate from 12% without FOX to 4%, respectively 5% with FOX.

6 RELATED WORK

The survey of T. Lorido-Botran et al. [17] gives a broad overview of existing auto-scalers and a classification into five groups with example implementations are proposed: (i) threshold-based rules [8, 10], (ii) queueing theory [23, 26], (iii) control theory [2, 15], (iv) reinforcement learning [19, 22], and (v) time series analysis [7, 13].

In the literature, many auto-scalers exist that can be assigned to the groups mentioned above. However, only a few auto-scalers support cost-aware scaling. The cost-awareness of the existing auto-scalers can be classified into three groups: (i) general cost optimization by using heterogeneous VM image sizes, (ii) limiting costs by defining a budget or run-time constraint, and (iii) optimization of the scaling logic with knowledge of the charging models.

The first group consists of auto-scalers that find scaling decisions and select the heterogeneous VM image size combination that introduces lowest cost while still fulfilling the specified SLAs. Example auto-scalers for this group are AutoMAP [3] and the one from Sharma et al. [21]. AutoMAP calculates the required amount of resources to satisfy the SLAs and then searches for a heterogeneous configuration. This configuration should have low costs for the end user while still fulfilling the desired average response time. The auto-scaler introduced by Sharma et al. greedily searches for a configuration with low costs that has a high utilization. Therefore, first a homogeneous configuration is calculated and then, this configuration is translated into a heterogeneous solution. In the paper of Brataas et al. [5], a systematic search over vertically and horizontally scaled deployments is conducted to find cost-optimal configurations. This information could be leveraged by an auto-scaling mechanism to better support heterogeneous resources for distributed applications.

The auto-scalers of the second group have budget or runtime constraints that are specified by the user. Example auto-scalers of this group are introduced by Vaquero et al. [24], Jiang et al. [14], Xiong et al. [25], and Zhu and Agrawal [27]. The auto-scaler of Vaquero et al. requires a specified maximum runtime of all VMs. If this runtime is exceeded the application is no more scaled. The one from Jiang et al. requires a predefined budget constraint and SLA. It performs a trade off between cost and SLA satisfaction to find a minimum amount of resources while still satisfying the SLAs. Xiong et al. introduced an auto-scaler for a multi-tier application. It first determines the required amount of resources to satisfy the SLA on an overall basis and then it splits the new provisioned resources based on the budget constraints to the tiers. The auto-scaler of Zhu and Agrawal have predefined time-limit and resource budget constraints. Within these constraints, the Quality of Service (QoS) is optimized using control theory.

The auto-scalers of the third group have knowledge about the charging models of the public cloud where the application is deployed. The approach presented in this paper can be assigned to this group. Example auto-scalers for this group are the ones from Cardellini et al. [6], Naskos et al. [18], and Roy et al. [20]. The auto-scaler introduced by Cardellini et al. has knowledge about the costs per VM instance per time interval, here the charging interval, e.g. 60 minutes for the Amazon EC2 cloud, is used. With this knowledge the VMs are shut down immediately before the next charging interval starts. In case a new VM should be allocated at this time, the interval of the already running VM is renewed so that it is not stopped and runs for another charging interval. Naskos et al. introduce an auto-scaler for noSQL databases that is aware of the VM charging model and knows the runtimes of all VMs. In case of downscaling, it stops the VMs that are closest to the next charging interval. The auto-scaler introduced by Roy et al. handles a multi-dimensional cost-function. Besides the leasing costs of the VMs, it includes the distance between the estimated response time and the SLA and the reconfiguration costs. Different weights can be assigned to the three components that may result in different scaling decisions. The auto-scaler optimizes this function and finds the optimum strategy with minimum costs.

FOX belongs to the third group of the cost-aware auto-scaling classification. It combines and extends the approaches of the example auto-scalers of this category: First, it supports more complex charging models like the two-phased one as applied at the Google Cloud Platform, instead of one charging interval as presented in the paper from Cardellini et al. Second, FOX has knowledge of all running VMs and their runtimes. With this information and the

knowledge of the charging models, the VM that is nearest to the next charging interval can be selected in case of downscaling. A similar mechanism is presented in the paper of Roy et al. Third, FOX finds proactive decisions for the future. Based on these future decisions and the knowledge of the charging model, a decision logic is presented when a downscaling is meaningful and when the already running VMs should stay running. None of the mentioned auto-scalers support future decisions and reviews the actual decision using them and the knowledge of the charging model.

7 CONCLUSION

In this paper, we examine the problem of increasing costs when using an auto-scaling mechanism in the public cloud environment. Our approach, called FOX, operates as a mediator between the application and the cloud to reduce the charged costs while still satisfying the specified SLOs. Therefore, FOX is based on the MAPE-K control loop: It monitors the application and analyzes future arrival rates using a forecaster. Based on these forecasts, the future configurations of the application are determined using the auto-scaling mechanism. Afterwards, the cost-aware mechanism reviews all found scaling decisions and modifies them to reduce the costs. Finally, the modified decisions are executed. FOX has a knowledge base, where observed and future arrival rates, as well as, the scaling decisions are stored. The cost-aware mechanism has knowledge about two different charging strategies: Amazon EC2, where a hourly charging is defined, and Google Cloud Platform, where the first ten minutes are charged fix and then, the charging switches to a minute-by-minute charging.

The evaluation of FOX is based on a multi-tier application deployed in the private cloud environment. This application is stressed using two real world workloads: BibSonomy and IBM CICS trace. To show that FOX can handle multiple auto-scaling mechanisms, three auto-scalers from the literature are used for evaluation: React, Adapt, and Reg. The results of all experiments show, that FOX is able to reduce the charged costs significantly, while increasing the accounted instance time for the Amazon EC2 charging model. This results in a significant decrease of SLO violation rate in trade for a slightly worse auto-scaling performance. For the Google Cloud Platform charging model, smaller (6%) cost savings are achieved. This can be explained due to the nature of the charging model, as there are no rounding overheads charged as for the Amazon EC2 charging model.

In the future, we plan to evaluate FOX with more real world workload traces and for different applications. In addition, other forecasting mechanisms like Telescope [28] will be integrated. Moreover, the experiments of all considered auto-scalers will be expanded to evaluate all of them at all workloads. For the future, it is planned to publish FOX as a tool on our website[13].

ACKNOWLEDGEMENTS

This work was funded by the German Research Foundation (DFG) under grant No. KO 3445/11-1. This research has been supported by the Research Group[14] of the Standard Performance Evaluation Corporation (SPEC).

[13] http://descartes.tools/

[14] SPEC Research: http://research.spec.org

REFERENCES

[1] R. Adhikari and R. Agrawal. 2013. An introductory study on time series modeling and forecasting. *arXiv preprint arXiv:1302.6613* (2013).

[2] A. Ali-Eldin, J. Tordsson, and E. Elmroth. [n. d.]. An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures. In *IEEE NOMS 2012*. IEEE, 204–212.

[3] M. Beltrán. 2015. Automatic provisioning of multi-tier applications in cloud computing environments. *The Journal of Supercomputing* 71, 6 (2015), 2221–2250.

[4] D. Benz and more. 2010. The social bookmark and publication management system BibSonomy. *VLDB* 19, 6 (2010), 849–875.

[5] G. Brataas, N. Herbst, S. Ivansek, and J. Polutnik. 2017. Scalability Analysis of Cloud Software Services. In *Companion Proceedings of the 14th IEEE ICAC 2017, Self Organizing Self Managing Clouds Workshop (SOSeMC 2017)*. IEEE.

[6] V. Cardellini, E. Casalicchio, F. Presti, and L. Silvestri. 2011. Sla-aware resource management for application service providers in the cloud. In *First International Symposium on Network Cloud Computing and Applications (NCCA)*. IEEE, 20–27.

[7] G. Chen and more. 2008. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services.. In *NSDI*, Vol. 8. 337–350.

[8] T. C Chieu, A. Mohindra, A. A Karve, and A. Segal. 2009. Dynamic scaling of web applications in a virtualized cloud computing environment. In *E-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*. IEEE, 281–286.

[9] A. De Livera, R. Hyndman, and R. Snyder. 2011. Forecasting time series with complex seasonal patterns using exponential smoothing. *J. Amer. Statist. Assoc.* 106, 496 (2011), 1513–1527.

[10] R. Han and more. 2012. Lightweight Resource Scaling for Cloud Applications. In *IEEE/ACM CCGrid 2012*. IEEE, 644–651.

[11] N. Herbst, S. Kounev, A. Weber, and H. Groenda. 2015. BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In *SEAMS 2015*. IEEE Press, 46–56.

[12] N. Herbst and more. 2016. Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics. *CoRR* abs/1604.03470 (2016).

[13] W. Iqbal, M. Dailey, D. Carrera, and P. Janecek. 2011. Adaptive Resource Provisioning for Read Intensive Multi-tier Applications in the Cloud. *Future Generation Computer Systems* 27, 6 (2011), 871–879.

[14] J. Jiang, J. Lu, G. Zhang, and G. Long. 2013. Optimal cloud resource auto-scaling for web applications. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2013*. IEEE, 58–65.

[15] E. Kalyvianaki, T. Charalambous, and S. Hand. 2009. Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters. In *ACM ICAC 2009*. ACM, 117–126.

[16] J. O. Kephart and D. M. Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (Jan. 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[17] T. Lorido-Botran, J. Miguel-Alonso, and J. Lozano. 2014. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing* 12, 4 (2014), 559–592.

[18] A. Naskos, A. Gounaris, and P. Katsaros. 2017. Cost-aware horizontal scaling of NoSQL databases using probabilistic model checking. *Cluster Computing* (2017), 1–15.

[19] J. Rao and more. [n. d.]. VCONF: a Reinforcement Learning Approach to Virtual Machines Auto-configuration. In *ACM ICAC 2009*. ACM, 137–146.

[20] N. Roy, A. Dubey, and A. Gokhale. 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *IEEE International Conference on Cloud Computing (CLOUD), 2011*. IEEE, 500–507.

[21] U. Sharma, P. Shenoy, and D. Towsley. 2012. Provisioning multi-tier cloud applications using statistical bounds on sojourn time. In *Proceedings of the 9th international conference on Autonomic computing*. ACM, 43–52.

[22] G. Tesauro, N. K Jong, R. Das, and M. Bennani. 2006. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *IEEE ICAC 2006*. 65–73.

[23] B. Urgaonkar and more. 2008. Agile Dynamic Provisioning of Multi-tier Internet Applications. *ACM TAAS* 3, 1 (2008), 1.

[24] L. Vaquero, D. Morán, F. Galán, and J. Alcaraz-Calero. 2012. Towards runtime reconfiguration of application control policies in the cloud. *Journal of Network and Systems Management* 20, 4 (2012), 489–512.

[25] P. Xiong and more. 2011. Economical and robust provisioning of n-tier cloud workloads: A multi-level control approach. In *31st International Conference on Distributed Computing Systems (ICDCS), 2011*. IEEE, 571–580.

[26] Q. Zhang, L. Cherkasova, and E. Smirni. 2007. A Regression-based Analytic Model for Dynamic Resource Provisioning of Multi-tier Applications. In *IEEE ICAC 2007*. IEEE, 27–27.

[27] Q. Zhu and G. Agrawal. 2010. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 304–307.

[28] M. Züfle and more. 2017. Telescope: A Hybrid Forecast Method for Univariate Time Series. In *Proceedings of the International work-conference on Time Series (ITISE 2017)*.