Delta Scaling: How Resources Scalability/Termination Can Be Taken Place Economically?

Yousri Kouki, Md Sabbir Hasan, Thomas Ledoux

DeltaScaling : How resources scalability/termination can be taken place Economically?

Yousri Kouki*, Md Sabbir Hasan†‡ and Thomas Ledoux‡
* *Linagora, Paris, France, Email: ykouki@linagora.com*
† *Myriads team (Insa Rennes, Inria, IRISA), France*
‡ *Ascola team (Mines Nantes, Inria, LINA), France, Email: firstname.lastname@mines-nantes.fr*

***Abstract*—Cloud Computing promises to completely revolutionize the capacity management of resources. The elasticity and the economy of scale are the intrinsic elements that differentiate it from traditional computing paradigm. A good capacity planning method is a necessary factor but not sufficient to fully exploit Cloud elasticity. This paper proposes innovative policies for resource management to achieve the optimal balance between capacity and quality of Cloud services while supporting Cloud technical and conceptual limitations. The main idea is to control finely the scalability and the termination of virtual machines in regards of several criteria such as the life-cycle of the instances (e.g. initialization time) or their cost. The approach was evaluated with a real infrastructure (Amazon EC2) and an application testbed. Experimental results illustrate the soundness of the proposed approach and the impact of scalability/termination resource policies. Using DeltaScaling, the cost saving of as much as 30% can be achieved while causing the minimum number of violations, as small as 1%.**

***Keywords*-Cloud Computing, Elasticity, Service Level Agreement (SLA), Auto-Scaling, Capacity Planning, Scalability, Termination**

I. Introduction

Cloud computing promises to completely revolutionize the way to manage resources. Thanks to elasticity, resources can be provisioned within minutes to satisfy a required level of Quality of Service (QoS) formalized by Service Level Agreements (SLAs) between different Cloud actors. Then, finding the best capacity management to maintain the consumer's satisfaction level while minimizing the service costs due to resources fees, becomes the main challenge of service providers.

A good capacity planning method is a necessary key but not sufficient enough to fully exploit Cloud elasticity. With respect to the existing work such as [2] [3], proposed solutions focused on capacity planning method accuracy but ignore Cloud technical limitations such as the non-negligible resource initiation time and Cloud conceptual constraints such as the billing model. Indeed, the non-negligible virtual machine (VM) initiation time [15] may prevent just-in-time provisioning to absorb workload peaks and thus avoid SLA violations. This can be even worse during oscillating scenarios of workload as it may have a "ping-pong effect" in the request/release of resources. In addition, the time granularity of resource reservation and the implied resource billing model (e.g., hourly, daily, etc.) may also lead service providers to either pay more than they actually consume (e.g., when it requests resource during a workload peak and releases it right after) which is known as *partial usage waste* [19], or to take into consideration the reservation duration before deciding to request more resources.

To have efficient capacity management model, the following questions have to be addressed: *how many* resources to add or terminate? and *how* resources scalability/termination can be taken place economically while respecting the Service Level Agreement (SLA)? This paper focuses more on the second question since existing work proposed solutions about the first question (i.e. capacity planning) [2] [3]. In order to address this issue, we propose a new auto-scaling architecture, called DeltaScaling, driven by policies which controls finely the scalability and the termination of VMs in regards of several criteria such as the life-cycle of the instances (e.g., initialization time) or their cost.

DeltaScaling is an extension of our previous work that can be found in [1]. The key contributions of this paper are the design of a flexible capacity management architecture, the development and validation of innovative policies: i) *scalability policies* to absorb the non-negligible resource initiation time. Usually resource initiation time refers to start-up time of a new VM to meet the current resources demand on the fly, if needed and ii) *termination policies* to harmonize capacity management with the economic model of the resource fees. Our proposed DeltaScaling is capacity planning model agnostic since the proposed policies are independent of any capacity planning method. In this paper, we illustrate both with static threshold-based rules method and Queuing Networks based method. We validate DeltaScaling through case studies and extensive experiments with on-line services hosted in Amazon EC2.

The remainder of this paper is organized as follows. Section 2 illustrates the assumptions, which are used to ease the understanding of our approach. Section 3 describes DeltaScaling in more details. The results obtained from experimental evaluation are presented and discussed in Section 4. Related research is discussed in Section 5. Finally, Section 6 concludes this paper and provides some discussion on future work.

II. Background and assumptions

In this section, we present the assumptions we made on Cloud applications and elasticity models to build our auto-scaling architecture.

A. SaaS Service model

SaaS services are mostly designed with multi-tier Web applications hosted by VMs for flexibility, reusability and scalability. Usually, we distinguish between two types of scalability: i) *vertical scaling – scale up/down* – typically refers to resizing existing resources of existing VM instances (e.g., CPU or RAM); and ii) *horizontal scaling – scale out/in* – usually refers to adjusting (add/remove) the number of VM instances. In this paper, for the sake of simplicity, we consider only horizontal scaling. The main challenge of a SaaS provider is to ensure that the number of used instances increases seamlessly during demand spikes to maintain performance, and decreases automatically during demand lulls to minimize costs.

B. Service-Level Agreement model

The response time and the availability are key metrics of interest for quantifying the performance and dependability of SaaS services. One might want a service level to attain a given objective that is the Service Level Objective (SLO). A SLO has usually one of the following form: a QoS metric with a value higher/lower than a given threshold. Therefore, a SLA is a set of SLOs to meet and is negotiated between two parties, the Cloud service provider and its customer. We use the CSLA language [1] to express SLA. CSLA allows defining SLA in any language (e.g., XML, Java). CSLA features such as *confidence* and *fuzziness* have been introduced to deal with QoS uncertainty in unpredictable Cloud environment. These concepts will be explained in Section IV.

C. Capacity Planning Model

Capacity planning is the activity within auto-scaling task with determining the capacity plan: defining how many instances to add or remove to adjust automatically according to a workload pattern. DeltaScaling embeds two models: i) Threshold-based rules model and ii) Queuing Network model. The first one is user-friendly but requires a deep understanding of the workload. The second one is more efficient but computational times and memory requirements are important.

Threshold-based rules Model. Static threshold-based rules model is the most popular for capacity planning and it has been used by Cloud providers like Amazon EC2.

The main idea behind this method is that the capacity of resources might vary according to a set of rules, where each rule is based on one or more metrics (e.g., response time, availability or CPU usage). A rule may be composed of an upper threshold *upper*, a lower threshold *lower*, two time values ($time_{upper}$, $time_{lower}$), during which the metric is greater/lower than the corresponding threshold, and two calm durations: $calm_{add}$ and $calm_{terminate}$ during which no scaling decisions can be committed in order to prevent system oscillations. Based on those parameters, the rules can be defined as follows:

If $m > upper$ *for* $time_{upper}$ *then* $capacity = capacity + k_{add}$
Do nothing for $calm_{add}$
If $m < lower$ *for* $time_{lower}$ *then* $capacity = capacity - k_{terminate}$
Do nothing for $calm_{terminate}$

where m is the metric (i.e., response time, availability), capacity is the current capacity, k_{add} and $k_{terminate}$ are respectively the number of resources to add and to terminate.

Queuing Network Model. The capacity planning problem can be formulated using a queuing model or a Queuing Network Model (QNM). A multi-tier system (implementing the SaaS application) can be modeled as a multi-server queueing model where their services are considered as closed loops to reflect the synchronous communication model that underlies these services. That is a client waits for a response before sending another request. Then, we rely on Mean Value Analysis (MVA) algorithm [5] for evaluating the performance of the queuing network. The MVA algorithm predicts the response time and the availability based on the current service workload and the capacity. A utility function is defined to combine performance, availability and financial cost objectives:

$$\theta(t) = \frac{\zeta(t)}{\omega(t)} \tag{1}$$

where ω corresponds to the cost of the service at time t and ζ corresponds to the SLA function (predicate) that checks SLA objectives (response time and availability).

$$\zeta(t) = (Rt(t) \leq Rt_{max}) \wedge (Av(t) \geq Av_{max}) \tag{2}$$

where Rt, Av are respectively the current response time

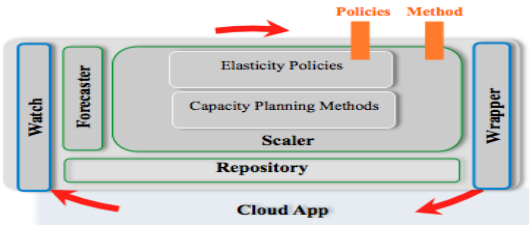


Figure 1. Top-level view of the DeltaScaling Architecture

and availability. Note that $\forall Rt, \forall Av$, $\zeta(Rt, Av) \in \{0, 1\}$.

$$\omega(t) = \sum_{n=1}^{N} Cost_{VM/T_n}(t) + \sum_{n=1}^{N} Cost_{Software/T_n}(t) \quad (3)$$

where $Cost_{VM,n}$ is the cost of VM at tier T_n and $Cost_{Software,n}$ is the software pricing and licensing costs at tier T_n.
The demanded capacity is the one that provides the highest utility (Equation 1). To compute it, we can use any search algorithm. In our implementation, we used a dichotomic search.

III. DeltaScaling Architecture

In this section, we first provide an overview of the DeltaScaling architecture before introducing the scalability/termination policies.

A. Overview

In order to cope up with a dynamic environment which is intrinsic to Cloud computing systems, it becomes imperative to rely on models that allow the system to react to the execution context so as to keep it running in an optimized way (in terms of QoS and cost). For this purpose we rely on autonomic computing [4]. Figure 1 illustrates the main components of the DeltaScaling architecture. On the highest level, sensors gather data from the managed elements (*Cloud App*), which allow the Autonomic Manager (*Scaler*) to monitor (*Watch*) and perform changes on them via effectors (*Wrapper*). The component *Repository* is in charge of maintaining some information about infrastructure, workload, SLA and elasticity policies that determine the scope within which decisions can be taken by the *Scaler*. Our solution presents a high level of reusability and extensibility in particular to the capacity planning method and elasticity (e.g., scalability, termination) policies.

B. Auto-scaling Lifecycle

The component *Watch* permits the monitoring of Cloud application via two modes: pull and push. Using the pull mode, the *Watch* gathers information about infrastructure, workload and service performance. The gathered information is raised periodically to the *Scaler* component. In addition, we developed a push mode to take into account the deferred terminations (see Section III-D). Indeed, the *Watch* is notified of started/stopped instances events and send full-time-instance consumption notification to the *Scaler*. Then, the *Scaler* analyzes gathered information whether it is necessary to (re)plan capacity. *Scaler* is based on a capacity planning method to calculate the optimal configuration that guarantees the SLA constraints while minimizing the cost of the service according to elasticity policies. The *Scaler* considers the result of capacity planning method to produce the optimal plan: a set of actions have an effect on the Cloud application in immediate and/or deferred manner. Finally, having the new optimal plan, the *Wrapper* takes this plan and executes it.

C. Scalability Policies

Cloud elasticity is the ability to automatically *scale out* resources as you need to accommodate varied workloads while maintaining QoS. To absorb the non-negligible VM initiation time [15], we propose three scalability policies: Reactive (with/without *Application Elasticity*), Proactive and Hybrid.

Reactive. A reactive policy refers to a run-time decision – based on the current demand and system state – to add resources on the fly. This policy can be easily designed and implemented when decisions are taken based on monitored performance metrics.

On the contrary, a reactive policy cannot absorb the non-negligible resource initiation time. To address this issue, we rely on *Application Elasticity* which has proven to be efficient in our previous work [1]. Application elasticity allows a SaaS application to support resource elasticity limitations by degrading *vs* upgrading services between different modes (e.g., 2D *vs* 3D display, a degree of security levels). In our proposal, we propose to degrade the functionality of the Cloud service in order to absorb the maximum requests until all resources have been activated.

Proactive. Proactive policy can be used to predict future demands in order to overcome the issues that may be raised by the non-negligible resource initiation time by adding resources in advance.

In our implementation, we used a prediction-based model (using the past history to predict future demand). The time-series analysis [16] offers a number of methods for predicting values at certain specific future times (*prediction interval*), based on a set of previously observed values (*prediction window*).

We rely on OpenForecast[1] package to predict future values. We propose a *Forecaster* component to obtain

[1]http://openforecast.sourceforge.net/

the best forecasting model for the given workload history. Finally, to take into account the non-negligible resource initiation time, we customize the prediction window according to the instance initiation time.

Hybrid. In order to be benefited from the advantages of both policies and thus be able to finely adjust the capacity, a hybrid approach (i.e., combination of reactive and proactive policies) would be necessary. For instance, proactive policy can be applied for a long-term demand prediction; in case of prediction error, the capacity can be reactively adjusted for the short-term.

D. Termination Policies

Cloud computing is typically based on full-hour billing model[2][3][4]. For example, with Amazon EC2, pricing is per instance-hour consumed for each instance, thus each partial instance-hour consumed will be charged as a full hour! In order to harmonize capacity management with the economic model, we introduce deferred policies. Then, we distinguish two types of policies: immediate and deferred.

Immediate. When a *scale in* condition is met (i.e. fewer resources are required), the termination is immediate. The *Scaler* component can configure a policy to terminate the oldest, the newest, the closest to next instance hour or a specific instance for the immediate termination.

Deferred. We propose a termination policy driven by the billing model called "Use as you Pay" (UaP) [6]. UaP policy is constructed to use the full-instance period before terminating any instance. For example, if a SaaS provider uses a instance for 2.5 hours, they will still have the flexibility to use remainder minutes of the hour to handle other workloads. In this way, SaaS providers pay exactly according to their usage. In contrast to the UaP policy, the Amazon EC2 billing policy can be described as: if SaaS providers terminate an instance and then start the same instance again, they are charged for another hour upon starting it, even if it is still within the same 60 minutes period. Furthermore, UaP policy allows to avoid system oscillations particularly for On and Off workload [7]. According to the auto-scaling lifecycle, UaP policy notifies stopped instances events to the *Watch* component (push mode). We can consider UaP policy as a proactive strategy: an instance waiting for the end of an instance-hour to terminate can be reused in the presence of a new increasing workload.

[2]http://aws.amazon.com/fr/ec2/pricing/
[3]https://azure.microsoft.com/en-us/pricing/overview/
[4]https://cloud.google.com/pricing/

IV. Experiments

This section presents results obtained from some experiments performed with the proposed approach. This work exploits the execution of a real use case scenario running on a real physical infrastructure.

A. Experimental Testbed

In this section, we describe the testbed used in the experiments.

SaaS application. In recent years, cryptography services are on high demand since banks, e-commerce sites and normal users need data protection for online transactions against identity theft and cyber crimes. Therefore, we prefer to design a simple cryptography service in a SaaS fashion that is deployed on Amazon infrastructure (EC2 instances). Our cryptography service is architecturally organized in 3-tier architecture[5], distributed and replicated in the Cloud: i) a load balancer tier that is responsible to distribute incoming application traffic across multiple application servers, ii) an application server tier that implements business logic functionality, and iii) a back-end database.

In this paper, we consider the SaaS provider offers *Application Elasticity* (see Section III-C): that proposes two cryptography services offering the same functionality but not the same quality. S_1 may operate on only one mode (normal), whereas S_2 may operate on two (normal and degraded). In the normal mode, the service uses 3DES algorithm, while in the degraded one, the service uses DES algorithm.

Table I
SLA between SaaS provider and its consumers

	metric	oper.	value	fuzz.	% of fuzz.	conf.	penalty
S1/S2	Rt	≤	0.1s	0.05 s	11.11%	90%	0.003$/req
	Av	≥	99%	2%			0.002$/req
S2	Mu	≤	30%	5%			0.0015$/req

Table II
Most relevant combinations

implementation	planning	scalability	termination	app. elastic.
QN-UaP	QN	Reactive Proactive	UaP	Yes
QN-Newest	QN	Reactive	Newest	Yes
AmazonUaP (TBR-UaP)	TBR	Reactive	UaP	No
AmazonLike (TBR-Newest)	TBR	Reactive	Newest	No

SLA. A SLA is established between the SaaS provider and each one of its clients stating that the provider has to guarantee a minimum level of service with respect to usage mode (normal/degraded)

[5]http://support.rightscale.com/03-Tutorials/02-AWS/02-Website_Edition

and other quality of service criteria such as response time or availability. Upon contract violations, penalties should be paid back to clients as compensation.

The SLA is expressed using the language CSLA [1] as it is shown in Table I. It is composed of three Service Level Objectives (SLOs): a performance SLO (response time Rt), a dependability SLO (availability Av) and a mode usage SLO (Mu). All metrics are measured within an observation interval of 1 min and evaluated on a time window of 10 min. We distinguish three states of a time interval: ideal, degraded and inadequate. Ideal means that the objective threshold is respected. The QoS degradation consists of utilizing a error margin (*fuzziness value*). Beyond this margin, it is an inadequate state.

In the use case, the SaaS provider should guarantee an average response time less than or equal to 0.1*s*, with confidence, fuzziness and percentage of fuzziness are 90%, 0.05*s* and 11.11%, respectively. It means that in the ten observation intervals of the evaluation window, we accept 1 inadequate interval (Rt > 0.15*s*), 1 degraded interval (0.1*s* < Rt < 0.15*s*) and 8 ideal intervals (Rt < 0.1*s*). Regarding the availability, it is defined as the number of accepted requests divided by the total number of requests in interval of 1 min. The SLA states that at least 99% of availability should be guaranteed, with 2% of fuzziness, meaning that we accept 1 inadequate interval (Av < 97%), 1 degraded interval (97% < Av < 99%) and 8 ideal intervals (Av > 99%). Concerning the functionality of degradation, for S1, only normal mode is accepted (i.e., 3DES), whereas for S2, up to 30% of the observation periods may be accepted on degraded mode (i.e., DES).

A violation of the response time SLO (resp. the availability and mode usage SLOs) implies a penalty equal to 0.003$/*request* (resp. 0.002$/*request* and 0.0015$/*request*).

Evaluation Scenarios. We evaluate DeltaScaling and its elasticity policies with representative workload patterns in the Cloud environment such as Unpredictable Bursting [7]. We use Apache JMeter as load injection system.

The initial architecture of our experiments for each workload pattern is composed of one instance per tier. The following experiments are conducted with EC2 small on-demand instances. The duration of the experiments is fixed at two hours. We evaluate reactive and proactive scalability policies combined with two termination policies : i) immediate (Newest) and ii) deferred (UaP). We also use the Application Elasticity to absorb the requests until resources have been activated which suggests, we degraded the functionality of service to accept more user requests until a new VM is activated, thus non-negligible resource initiation time is handled. Finally, since our approach is capacity planning model agnostic, we experiment with two capacity planning methods: Queuing Network (QN) and threshold-based rules (TBR). Table II summarizes the most relevant combinations.

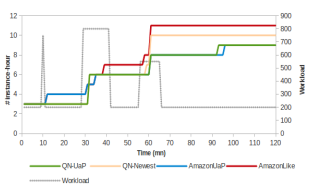
B. Results and Discussion

We present the results obtained from an extensive set of experiments. Results show how scalability/termination policies can be used so as to cope with some technical and conceptual limitations of Cloud elasticity. The impact of these policies is shown in Figure 2 according to different workload patterns. We observe their pertinence considering Customer Satisfaction Level (CSL) and SaaS provider profit. CSL refers to the percentage of SLOs compliance (evaluation windows without penalty / total of evaluation windows) . The SaaS provider's profit is calculated from the difference between income (service price) and expense (resources cost and penalties).

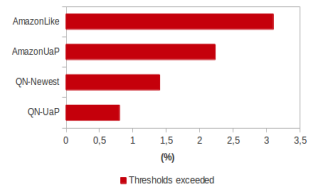
Unpredictable Bursting workload. Figure 2 depicts the results for the experiments with the Unpredictable Bursting workload considering the SLA given in Table I. As shown in the graphs in Figure 2(a), a variable number of clients was used in this experiment (initially 200, then 750, 800 and finally 200 again). The Figures 2(a), 2(b), and 2(c) show respectively the resource cost, the percentage of exceeded thresholds and the ratio of provider profit *vs* customer satisfaction level (CSL) for each combination from Table II. In Figure 2(a), *QN-UaP* and *QN-newest* have cushioned the addition of resources during peak load at t=10 min using the *Application Elasticity*. Further, the UaP termination policy performs best to reduce *#instance – hour* to 9 only *vs* 11 for Newest policy as shown in Figure 2(a). At t=29 min, thanks to Queuing Network (QN) model, the SaaS service needs nearly 3 minutes only to meet SLA requirements (performance, availability and mode) while it needs at least 6 minutes using AmazonLike policy.

All this will influence the number of SLA violations. The SLOs evaluation is performed on windows of 10 minutes. Each minute interval can be classified as i) degraded if one or more objectives (response time, availability or usage mode) use the associated fuzziness or ii) inadequate if one or more objectives (SLOs) exceeds the threshold +/- fuzziness. The SLA is calibrated in order to accept 1 inadequate interval and 1 degraded interval every 10 minutes, otherwise a penalty will be applied.

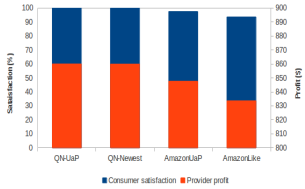
In Figure 2(c), using AmazonLike, SaaS provider profit is the least important with a customer satisfaction level of 93.6%. Indeed, the termination policy "Newest" causes the instability of the system. Such instability occurs more exceeded thresholds (see



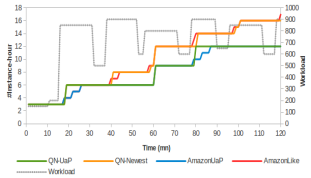
(a) Resource cost for Unpredictable Bursting



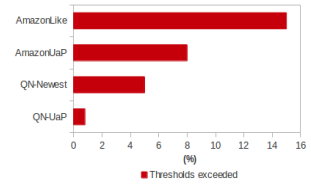
(b) Thresholds exceeded for Unpredictable Bursting



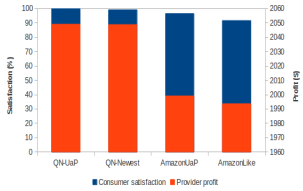
(c) Provider profit *vs* CSL for Unpredictable Bursting



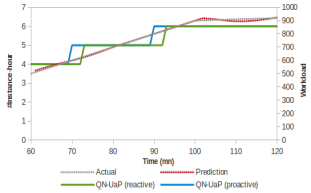
(d) Resource cost for On and Off



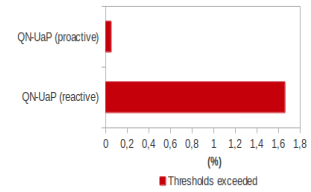
(e) Thresholds exceeded for On and Off



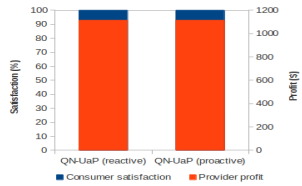
(f) Provider profit *vs* CSL for On and Off



(g) Resource cost for Predictable



(h) Thresholds exceeded for Predictable



(i) Provider profit *vs* CSL for Predictable

Figure 2. Results with different workloads

Figure 2(b)). In addition, the immediate termination causes more instance-hour to pay. The SaaS provider profit is almost the same using QN-UaP or QN-Newest and with customer satisfaction of 100%. Nevertheless, QN-UaP presents the best QoS perceived by customers: there are fewer requests with a functionality degradation (3 min *vs* 5 min in a time window of 120 min as show in Table III).

On and Off workload. Figure 2 presents the second workload pattern (*On and Off*) that we apply to the Cloud service. The workload fluctuates between 500 and 900 clients. The results are presented in the same manner as for the first workload pattern. In the case of *On and Off* workload, the maximum saving cost and the minimum SLA violations can be reached using our policies. It can be observed in Figure 2(d) that during the variation of demands, UaP policy minimizes cost to 12 $instance-hour$ only *vs* 17 $instance-hour$ for Newest policy. Figure 2(e) shows that QN model generates less number of SLA violations than TBR model (AmazonLike). In fact, UaP policy allows avoiding system oscillations and implicitly SLA violations.

In Figure 2(f), as for the unpredictable workload, the SaaS provider profit using AmazonLike is the least important with customer satisfaction of 91.67%. We note that the UaP policy allows minimizing the use of degraded mode: 2 min in a time window of 120 min while degraded mode is used 12 min with Newest policy (see Table III). The SaaS provider profit is the best using QN-UaP with customer satisfaction of 100%.

Predictable workload. The last experiment shows the result of the scalability of Proactive policy (see Figure 2) as we to compare both the reactive and the proactive policies. Figure 2(g) presents the workload that we apply to the service. The experimentation starts with a moderate load corresponding to 100 clients. Workload rises with a trend between t=20 min and t=100 min. The duration of the experiment is equal to 120 min. A history of at least 60 observations is necessary for the prediction, which is why we use the first 60 min as prediction window. The results presented are related to the second hour of the experiment (61 min - 120 min). Figure 2(g) illustrates the result of prediction using our Forecaster in the interval 61 min - 120 min. The predicted values were accurate with an average error of 0.35% (see Figure 2(g)). The prediction interval was fixed to 2 min (i.e., the average value of instances initiation time or VM start up time). The proactive scalability uses the result of prediction to anticipate the increase in capacity allowing to avoid the

overspending thresholds. While the reactive scalability adds resources following a overrun and after having tried the use of mode degradation for 1 minute.

In Figure 2(i), the SaaS provider profit is almost the same using reactive or proactive policy with customer satisfaction of 100%. However, the reactive policy produces overruns thresholds. In addition, with this policy, the functionality degradation is used 4 min in a time window of 60 min (see Table III).

Table III
CSLA and Application Elasticity

workload	implementation	total duration (min)	degraded mode (min)
Unpredictable Bursting	*QN-Newest* (reactive)	120	5
	QN-UaP (reactive)	120	3
On and Off	*QN-Newest* (reactive)	120	12
	QN-UaP (reactive)	120	2
Predictable	*QN-UaP* (reactive)	60	4
	QN-UaP (proactive)	60	0

Discussion. To sum up, the results show that the proposed scalability/termination policies are very effective to improve Cloud elasticity as well as provider's profit. The use case scenario suggests how elasticity of application can be taken into consideration to overcome the resource initiation time and scalability issues. However, in real world, using DES algorithm instead of 3DES might be a bit skeptical, but it was worth validating our approach where CPU usage intensiveness is different between the two algorithms. Since 3DES needs thrice the computing power as DES, any application which inherits elasticity capability in terms of increasing/decreasing computing power (CPU intensivity), can be benefited by our approach. Among the most relevant combinations, the different benchmarks show that the association between the capacity planning based on Queuing Network and the "UaP" policy, with the Application Elasticity feature (i.e., QN-UaP), is always the best combination without depending on the any kind of workload characteristics. It is evident from the results that, QN model is better than TBR model in terms of profit and customer's satisfaction level, but different termination policy in QN model can affect the total performance of the system divergently. The idea behind introducing QN-Newest policy is to validate the aforementioned statement since QN-Newest policy exceeds more thresholds in run-time, thus might create instability to the system. Therefore, effective termination policy of cloud resources is more important for capacity management. We found it necessary to illustrate that, using QN model can increase the system performance (i.e. response time, availability etc.) but associating UaP policy with QN model can out perform other capacity planning method in terms of provider's profit while causing minimum number of SLA violation.

V. Related work

Capacity management is the process tasked with adjusting the capacity of a system, to meet changing demands, at the right time and at optimal cost. With respect to the time resources should be added/released, capacity management can be conceived in three ways: i) reactive [8], ii) proactive [12] or iii) hybrid [13]. Several works has been done to provide solution for capacity management in Cloud environments. However, only a few works provide a combination of reaction (reactive, proactive or hybrid) [20]. Our solution is more complete and includes three types of above mentioned reactions.

Capacity planning is the sub-activity within Capacity management task with determining the optimal capacity of resources. It may be achieved in a completely ad-hoc manner by dynamically adjusting the allocation of resources (e.g., by adding or releasing resources) according to a set of predefined rules. Examples of such kind of approach are implemented in leading industrial solutions such as Amazon Auto Scaling and Autoscaling Application Block (Microsoft Azure) or research works [8] [9].

Capacity planning may combine techniques from different fields in order to be effective. For example, in order to effectively model systems performance, Queuing Theory [14] can be applied. Alternatively, Reinforcement Learning [11] and Game Theory [10] might also be used respectively to give a more effective performance profiles or to deal with economy equilibrium and thus improve estimations on resource requirements. Those techniques can be combined along with operational research techniques like Constraint and Linear Programming so as to find solutions that respect the constraints (e.g., imposed by SLAs) or even solution that optimizes a certain criterion such as the Quality of Service or cost.

With respect to these works, most focus only on method accuracy and ignore Cloud technical and conceptual limitations. Some initiatives contributed to solve technical limitations, such as [17] and [18] while conceptual limitations (e.g., economic model) is not addressed. [17] proposes an approach based on vertical scaling to absorb duplicating time for data-base tier. However, the vertical scaling is not provided by all IaaS providers and even when offered it can be only cold scaling. So, the initialization time is not fully addressed. [18] presents a predictive model to absorb the initialization time. The accuracy of prediction method depends on the input window size and the prediction interval whereas the authors do not detail these values.

The originality of our contribution is to focus on the life-cycle of the instances (e.g., initialization, termination time) and on the impact of the pay-as-you-go pricing in elasticity management. In addition, all the related work concentrate only at the infrastructure level by optimizing resource scaling, whereas in this paper we take into account both levels: application – via *Application Elasticity* – and infrastructure. For example, this extra elasticity capability can be used as an alternative solution for short workload peaks instead of either paying a full usage cycle.

VI. Conclusion and future work

In this paper, we propose a new auto-scaling architecture, called DeltaScaling, driven by elasticity policies which is flexible enough to work with several capacity planning methods such as Threshold-based rules or Queuing Network models in a economic way. Experimenting DeltaScaling in a real infrastructure (Amazon EC2) illustrates that, 30% cost saving can be achieved while causing the minimum number of violations, as small as 1%. In the near future, we propose to extend our solution to help any Cloud provider to select the best capacity planning method from a catalog according to the service context (service topology, workload characteristics, SLA). This service is based on the capacity planning methods evaluation criteria (e.g., system stability, method scalability, solution optimality, implementation simplicity) applied to the Cloud providers context to propose the right method to choose.

References

[1] Y. Kouki, F. Alvares, S. Dupont and T. Ledoux. A Language Support for Cloud Elasticity Management. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), May 2014.

[2] S. Dutta, S. Gera, A. Verma, and B. Viswanathan. SmartScale: Automatic Application Scaling in Enterprise Clouds. IEEE 5th International Conference on Cloud Computing (CLOUD), June 2012.

[3] Z. Shen, S. Subbiah, X. Gu, J. Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. ACM Symposium on Cloud Computing (SOCC), October 2011.

[4] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. Computer, vol.36, no.1, pp.41-50, Jan 2003.

[5] M. Reiser, S. S. Lavenberg. Mean-Value Analysis of Closed Multichain Queuing Networks. J. ACM vol.27, no. 2, April 1980.

[6] Y. Kouki and T. Ledoux. RightCapacity: SLA-driven Cross-Layer Cloud Elasticity Management. International Journal of Next-Generation Computing (IJNGC), vol. 4, no. 3, November 2013.

[7] Cloud workload patterns. watdenkt.veenhof.nu/2010/ 07 /13/ workload-patterns-for-cloud-computing/, 2012.

[8] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. Lightweight Resource Scaling for Cloud Applications. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), May 2012.

[9] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi. Integrated and autonomic cloud resource scaling. IEEE Network Operations and Management Symposium (NOMS), April 2012.

[10] D. Ardagna, B. Panicucci and M. Passacantando. A Game Theoretic Formulation of the Service Provisioning Problem in Cloud Systems. International conference on World Wide Web (WWW), 2011.

[11] E. Barrett, E. Howley, and J. Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurrency and Computation: Practice and Experience, vol. 25, no. 12, August 2013.

[12] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. IEEE International Conference on Cloud Computing (CLOUD), July 2011.

[13] 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, June 2011.

[14] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. Network Operations and Management Symposium (NOMS), April 2012.

[15] M. Mao and M. Humphrey. A Performance Study on the VM Startup Time in the Cloud. IEEE International Conference on Cloud Computing (CLOUD), June 2012.

[16] G. E. P. Box and G. M. Jenkins. Time Series Analysis : Forecasting and Control. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 1994

[17] L. Wang, J. Xu, M. Zhao, Y. Tu, and J. A. B. Fortes. Fuzzy Modeling Based Resource Management for Virtualized Database Systems. IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), July 2011.

[18] 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, January 2012.

[19] H. Jin, X. Wang, S. Wu, S. Di, and X. Shi. Towards Optimized Fine-Grained Pricing of IaaS Cloud Platform. IEEE Transactions on Cloud Computing, 2014.

[20] L. Moore, K. Bean, and T. Ellahi. A Coordinated Reactive and Predictive Approach to Cloud Elasticity. International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING), May 2013.