Improvements of the Reactive Auto Scaling Method for Cloud Platform

Conference Paper *in* Communications in Computer and Information Science · May 2017

DOI: 10.1007/978-3-319-59767-6_33

1 author:

Dariusz Rafał Augustyn
Silesian University of Technology
34 PUBLICATIONS **179** CITATIONS

Improvements of the reactive auto scaling method for cloud platform

Dariusz Rafal Augustyn

Silesian University of Technology, Institute of Informatics,
16 Akademicka St., 44-100 Gliwice, Poland
draugustyn@polsl.pl

Abstract. Elements of cloud infrastructure like load balancers, instances of virtual server (service nodes), storage services are used in an architecture of modern cloud-enabled systems. Auto scaling is a mechanism which allows to on-line adapt efficiency of a system to current load. It is done by increasing or decreasing number of running instances. Auto scaling model uses a statistics based on a standard metrics like CPU Utilization or a custom metrics like execution time of selected business service. By horizontal scaling, the model should satisfy Quality of Service requirements (QoS). QoS requirements are determined by criteria based on statistics defined on metrics. The auto scaling model should minimize the cost (mainly measured by the number of used instances) subject to an assumed QoS requirements. There are many reactive (on current load) and predictive (future load) approaches to the model of auto scaling. In this paper we propose some extensions to the concrete reactive auto scaling model to improve sensitivity to load changes. We introduce the extension which varying threshold of CPU Utilization in scaling-out policy. We extend the model by introducing randomized method in scaling-in policy.[1]

Keywords: cloud computing, auto scaling, custom metrics, load balancing, overload and underload detection

1 Introduction

Most of modern system architectures allow to use scaling capability provided by cloud platform. The cooperating components of the information system may be run in cloud environment on separated virtual machines called instances or service nodes. Inside the cloud, a load balancer can distribute a stream of requests among many operating service nodes. Cloud platform provides mechanisms (like software tools, APIs etc.) for managing such service nodes. Especially, these

[1] This is the manuscript of:
Dariusz R. Augustyn: Improvements of the reactive auto scaling method for cloud platform. Computer Networks. Communications in Computer and Information Science. vol. 718, 2017, pp. 422–431
The original publication is available on www.springerlink.com

mechanisms allow to horizontal scaling-out and scaling-in by programmatic create/destroy a virtual server. This gives a possibility to apply some model of auto scaling [1], where the number of service nodes is adapted to a system load. Such approaches may be reactive [2, 3] (they use information about current load and system state) or predictive [4–6] (they additionally use an extrapolation of load and system state in near future). The reactive auto scaling models are rather simple, but they may be applied to a poorly predictable load.

Obviously a scaling-in increases a cost of system. To measure the cost we may define a simple objective function:

$$MeanCost = \frac{1}{Time} \int_0^{Time} Number\_of\_service\_nodes(t)dt \tag{1}$$

which evaluates a system respect to usage of service nodes during $Time$.

A decision of scaling-in or scaling-out may be taken according to assumed Quality of Service (QoS) requirements or system resource-based ones. A user may assume some high-level criterion of quality based on statistics (e.g. mean, high order quantile) of some application-level metrics like execution time of selected business service. The approach to auto scaling model which uses the application-level metrics will be denoted as CMAS (Custom Metrics Auto Scaling). A user may also define less intuitive low-level criterion based on statistics (e.g. mean) of a resource-level metrics like CPU Utilization of a service node. Such approach will by denoted as SMAS (Standard Metrics Auto Scaling). The approach proposed in [2] combines these two approaches.

The optimization problem to solve in auto scaling domain can be formulated as choosing such methods and values of their parameters to minimize the objective function subject to QoS requirements.

This paper focuses on extending the model and method of the reactive auto scaling module presented in [2]. In this paper we propose the following improvements of that method:

- the additional error-based criterion in determining overloaded state of system (sec.3),
- the method of obtaining limits for group CPU Utilization (that may cause better choosing the moment of launching scaling-out) adapted to number of currently launched virtual machine instances (sec.4),
- the more aggressive strategy of scaling-in based on a function probability of turning off a redundant service node (sec.5).

2 The Auto-scaled Distributed System Designed for AWS Cloud Infrastructure

In the considered model [2] a quality of service requirement is a constraint based on statistics for execution times of a selected business critical service. A user may explicitly set $T_{q\ acc}$ – a value of a threshold (a max value) for T_q - a high order quantile of execution times (the q^{th} quantile). If $T_q > T_{q\ acc}$

than we assume that a system is overloaded. A user may also set MV_{acc} – a value of threshold (a min value) for MV – a mean value of execution times. If $MV < MV_{acc}$ than we assume that a system is not enough loaded.

In our work we consider a simple cloud-aware information system (Fig. 1) that consists of:

- load balancer which exposes service outside a cloud and internally distributes requests among service nodes,
- multiple (n) service nodes, that internally expose SOAP/WebServices to the load balancer; the so-called auto scaling group consists of such service nodes and load balancer,
- DaaS node (PostgreSQL Relational Database Service) which persists data.

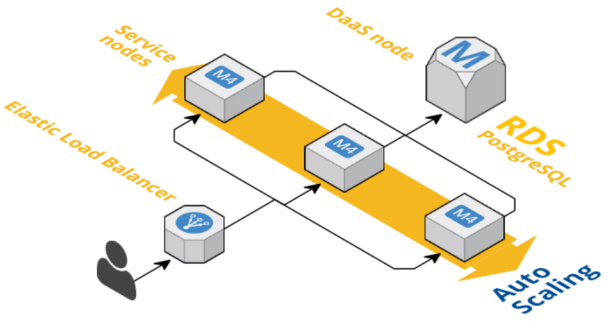

Fig. 1. The cloud-based architecture of the system: Elastic Load Balancer, and ($n = 3$) Elastic Compute Instances (service nodes), and Amazon RDS service (DaaS node).

The proposed in [2] software module that controls virtual machines (i.e. creates/destroys an instance of service node) is responsible for scaling-out/-in.

A scaling-out procedure uses an application-level custom metric – T_q and a AWS built-in resource-level metric – CPU Utilization. The module tries to use the estimator of T_q. If estimator of T_q exceeds $T_{q\ acc}$ the scaling-out should be performed. When estimator of T_q is not available (too less observations so we cannot positively verify at the assumed level that an estimator T_q belongs to the assumed confidence interval) the module uses $GroupCPUUtil$ – a group CPU utilization (a mean of CPU utilizations of service nodes). If $GroupCPUUtil$ exceeds a $MaxCPUUtil$ that means that system is overloaded (but not by the selected business critical service) and the scaling-out should be performed, too.

A scaling-in procedure uses a application-level custom metric – MV and again a built-in resource-level built-in metric – CPU utilization. When estimators of MV is not available (too few observations what causes that it is not statistical confident) the module uses $GroupCPUUtil$. If $MinCPUUtil$ exceeds $GroupCPUUtil$ it means that the system is not loaded enough and the scaling-in should be performed.

The algorithm based on a custom metrics (T_q or MV) was called CMAS (Custom Model of Auto Scaling). The supplementary algorithm based on a built-in metrics ($GroupCPUUtil$) was called SMAS (Standard Model of Auto Scaling). Both CMAS and SMAS checks the conditions for T_q, MV, $GroupCPUUlil$ in some regular moments of time (determined by interval T_i). They launch scaling only if the condition is satisfied at least m times during last M tries (commonly $m > M/2$).

3 Analysis of System Efficiency

To describe an efficiency characteristic of a system, we consider to load it by a sequence of requests of selected business service. We assume the exponential distribution of intervals between subsequent requests with a mean value of intervals equals $1/\lambda$. The results of loading a system with $n = 1, 2, 3$ service nodes may look like those shown in Fig.2.

Fig.2 presents how the mean value of execution time – $MV^{(n)}$, the q^{th} quantile of execution times – $T_q^{(n)}$, the % of error requests per unit of time – $Err^{(n)}$ for $n = 1, 2, 3$ depend on increasing system load – λ.

In most cases, the error requests appear because nodes may be overloaded. This may happen either for service node or DaaS node. In fact, we may directly scale out in the system by multiply service nodes but we have no direct influence on scaling DaaS node. Thus we may expect that for overloaded system with many service nodes most of errors requests results from overloading of single DaaS node.

Quality of Service requirements define a not overloaded system where both criteria $T_q^{(n)} \leq T_{q\ acc}$ and $Err^{(n)} \leq Err_{acc}^{(n)}$ are satisfied. By increasing system load we may obtain the highest values of λ – $\lambda_{max}^{(n)}$ (blue color in Fig.2) where $T_q^{(n)} \approx T_{q\ acc}$ and $Err^{(n)} \leq Err_{acc}^{(n)}$ for $n = 1$ (Fig.2a, 2b), $n = 2$ (Fig.2c), $n = 3$ (Fig.2d).

In Fig.2 (and Fig.3), the green color is used for marking acceptable operating points, the blue for boundary ones, and the red or brown for unacceptable ones.

We want to notice that the single criterion $T_q^{(n)} \leq T_{q\ acc}$ is not enough to determine a not overloaded system. When DaaS node becomes overloaded some time-out barrier may be crossed in communication between a service node and DaaS node. The architecture of the system should be adapted to such situations. Modern systems (see e.g. Repository of Electronic Medical Documentation – RepoEDM [2]) are based on a micro services architecture and supported by functionality which minimizes propagating of failure cascade and accelerates the backward information about time-outs (Hystrix[2]). So-called self-healing[3] mechanism (based on Hystrix/Eureka) reports that the service as unavailable so that subsequent requests do not run into the same timeouts. This results in very fast

[2] GitHub - Netflix/Hystrix (2016) `https://github.com/Netflix/Hystrix`

[3] Hystrix and Eureka: the essentials of self-healing microservices (2016) `https://www.dynatrace.com/blog/top-2-features-self-healing-microservices`

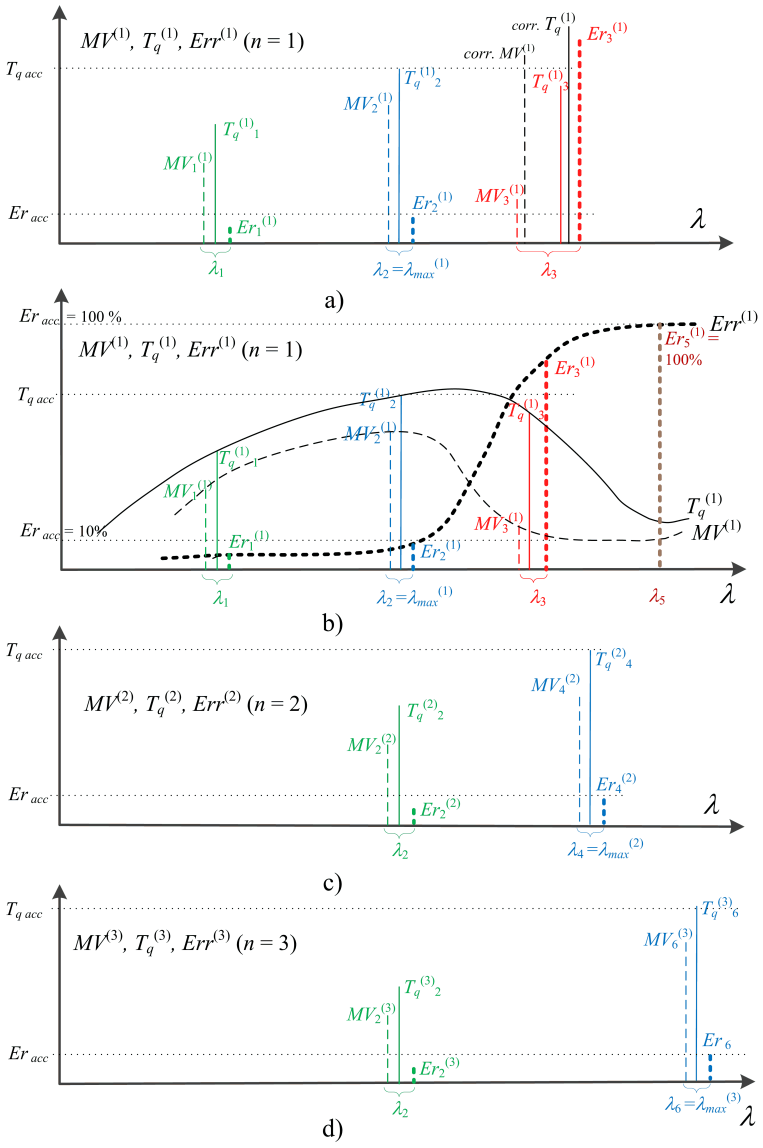

Fig. 2. Dependency between a load intensity λ and:
– the mean value of execution times – $MV^{(n)}$ (dashed line),
– the q^{th} quantile of execution times – $T_q^{(n)}$ (solid line),
– the % of error requests per unit of time – $Err^{(n)}$ (fat dashed line);
a) some operating points for a one-service-node system ($n = 1$),
b) outlines of hypothetical courses for $MV^{(1)}, T_q^{(1)}, Err^{(1)} (n = 1)$,
c) some operating points for a two-service-nodes system ($n = 2$),
d) some operating points for a three-service-nodes system ($n = 3$).

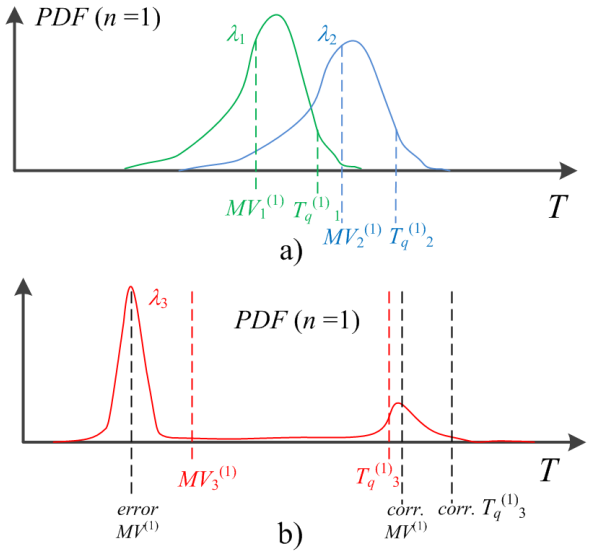

Fig. 3. Probability density function (PDF) of execution times T for a one-service-node system: a) for a not overloaded system (green line for λ_1 intensity) and a boundary overloaded one (blue line for λ_2), b) for an overloaded system (red line for λ_3).

responses from error requests targeted to the overloaded DaaS node. This is illustrated in Fig.3b where an empirical probability density function is bimodal. The execution times near the first local maximum (values close to *error* $MV^{(1)}$) correspond to error requests. The execution times near the second local maximum (values close to *corr.* $MV^{(1)}$) correspond to correctly processed requests. Although system is overloaded (Fig.3b) and most of requests are processed incorrectly with time-outs (the mass near *error* $MV^{(1)}$ is greater than the mass near *corr.* $MV^{(1)}$), the mean value and the q^{th} quantile are less relative to the ones from the not overloaded system (Fig.3a).

The satisfied condition $T_{q\,3}^{(1)} < T_{q\,2}^{(1)} = T_{q\,acc}$ in the $(3)^{th}$ operating point (red in Fig.2a) may lead to incorrect conclusion that the overloaded system from Fig.3b might be accepted as not overloaded. But it does not satisfy the Err-based criterion so finally it will be rejected according to QoS requirements.

4 Improvement of Scaling-out in SMAS

SMAS model presented in [2] was based on an assumption that $MaxCPUUtil$ obtained for one-service-node system is enough accurate for a multi-service-nodes system. This assumption is only approximately valid because we can observe that CMAS and SMAS create instances in time differently even for the same load profile. Adapting $MaxCPUUtil$ values to n – the number of running service nodes – allows SMAS to behave almost the same like CMAS, i.e. we may observe

situations when either SMAS or CMAS increases number of service nodes almost at the same moments of time.

We already noticed that a load of a single DaaS node may not be distributed like a load directed to many service nodes. During load increasing, CPU utilization of DaaS node will increase too and DaaS node becomes slower and the portion time of processing of a single request in DaaS node will increase too. Because we want to hold the same $T_{q\ acc}$ with increasing load the portion of time of processing in a service node should be decreasing thus a service node should be faster and its CPU Utilization has to be lower.

We may experimentally find values of $MaxCPUUtil$ dependent on n. Values of $MaxCPUUtil(n)$ determine when the auto scaling module should switch the system from having n service nodes to $n+1$ ones. Those values may be obtained as means of CPU Utilizations of n service nodes in boundary operational points, i.e. for a load specified by $\lambda_{max}^{(1)}$ (when $n = 1$), ..., $\lambda_{max}^{(3)}$ (when $n = 3$), ...(Fig.4). Such hypothetical shape of a decreased dependency suggests that switching from n to $n+1 (n > 1)$ will be earlier (i.e. for smaller values of $GroupCPUUtil$) than it happens in the method from [2] where we had only a one and high value – $MaxCPUUtil(1)$ for all n.

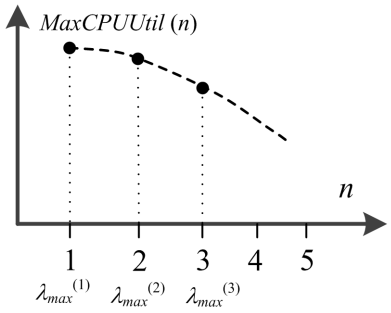

Fig. 4. $MaxCPUUtil$ threshold for SMAS adapted to n – the number of operating service nodes.

5 Improvement of Scaling-in in CMAS

According to a goal of minimizing the objective function $MeanCost$ (Eq.1) some improvement of the scaling-in procedure was proposed. Let us remind rather conservative behavior in [2] – a scaling-in is needed when m times the criterion $MV < MV_{acc}$ is satisfied during M tries. In more detail, we obtain $\widehat{MV}$ (the estimator of MV) and verify that $\widehat{MV}$ is less than MV_{acc} at an assumed confidence level p. Such $\widehat{MV}$ we call confident. In [2], we only detect a fact of criterion satisfying but we do not use a difference value – $MV_{acc} - \widehat{MV}$.

To make the above-mentioned strategy of scaling-in more effective we propose to scaling-in when we satisfy the criterion J times where $1 < J \leq m$ but we will take into account only confident values of $\widehat{MV}$, too.

Although we will introduce some nondeterministic factor we want to hold a compatibility with the current strategy that satisfying the criterion m times launches scaling-in always i.e. with probability equals 1.

We do not want to fire scaling-in upon only a one try.

Let us denote as follows:

- $\widehat{MV}$ – confident estimator for $j \leq J$,
- $s = \sum_{j=1}^{J} \widehat{MV}_j$,
- $s_0 = JMV_{acc}$.

We introduce function of probability (p-function) of launching scaling-in as follows:

$$p(s,J) = \begin{cases} 0 \text{ for } s > s_0 \\ 0 \text{ for } J = 1 \\ 1 \text{ for } 0 \leq s \leq s_0 \wedge J = m \\ \frac{1-\frac{1}{m-1}(J-1)}{0-\frac{s_0}{m-1}(J-1)}(s-0)+1 \text{ for } 0 \leq s \leq \frac{s_0}{m-1}(J-1) \wedge J \in \{2,\ldots,m-1\} \\ \frac{\frac{1}{m-1}(J-1)-0}{\frac{s_0}{m-1}(J-1)-s_0}(s-s_0)+0 \text{ for } \frac{s_0}{m-1}(J-1) < s \leq s_0 \wedge J \in \{2,\ldots,m-1\} \end{cases} \tag{2}$$

which is easier understandable using Fig.5.

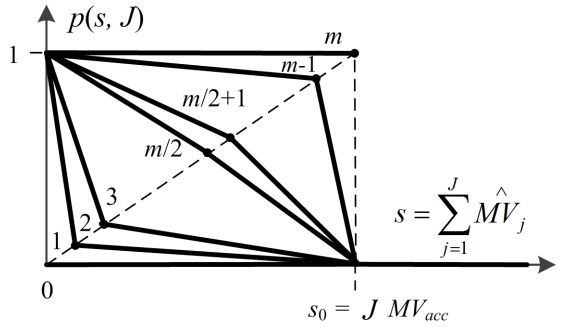

Fig. 5. Function of probability of launching scaling-in ($J = 1,\ldots,m$).

In new scaling-in method denoted by CMAS* we decided to scale-in with probability obtained from p-function (Eq. 2).

6 Some Experimental Results

In experiments we used real RepoEDM system described in [2] and run it in Amazon Web Services Cloud. The results illustrate SMAS/CMAS behavior after the improvements introduced in sec.4 and 5.

According to the idea from sec.4 for some assumed $T_{q\ acc} \approx 4000$ms we obtain $MaxCPUUtil(1) \approx 82\%$, $MaxCPUUtil(2) \approx 71\%$, $MaxCPUUtil(3) \approx 63\%$. The differences among those values proved to be statistical significant at assumed confidence level ($p = 0.9$).

To evaluate the improvement of SMAS scaling-in we test both SMAS and CMAS with linearly increased λ in time during 20min and with no load during 20min. Test was repeated 10 times. The improved model is denoted by SMAS*.

Let us introduce the following coefficient:

$$Mean\Big(\frac{\hat{T}_{q\ \mathrm{CMAS}} - \hat{T}_{q\ \mathrm{SMAS}}}{\hat{T}_{q\ \mathrm{CMAS}}} 100\%\Big). \tag{3}$$

Experimentally, we obtained its value about 11% (for original SMAS with one $MaxCPUUtil$) and about 6% (after the improvement, for SMAS* with the series of $MaxCPUUtils(n)$). This result shows that the system under SMAS* becomes more similar to CMAS thanks to the scaling-out improvement.

To evaluate improvement of CMAS i.e. compare pure CMAS and CMAS with p-function of probability of lunching scaling-in (denoted by CMAS*) – we used a test profile defined by a sequence: constant λ during 20min, and linear decreased to zero during 20min, and 10min no load. $m = 4$ and $M = 7$ were used in the experiment. Test was repeated 10 times.

For the following coefficient:

$$Mean\Big(\frac{MeanCost_{\mathrm{CMAS}} - MeanCost_{\mathrm{CMAS^*}}}{MeanCost_{\mathrm{CMAS}}} 100\%\Big) \tag{4}$$

we obtained a value equals about 6% what shows a slight improvement in cost.

7 Conclusions

We rather expect poor effectiveness of load prediction during a process of mass migration of systems to cloud. Such process is complicated and it will depend on many technical factors, financial ones, or organizational ones. For such temporary situations we rather recommend a reactive model of auto scaling.

Although the idea of equivalency between custom-metrics-based QoS requirement (in CMAS) and resource-metrics-based QoS requirement (in SMAS) is not complicated but we did not meet such approach in known reactive models. We think that CMAS is well aligned to user expectations. But CMAS may not work sometimes (because of lack of metrics data) so it must be supported by adjusted SMAS.

In our work we provided a control module which implements proposed cooperative models of auto scaling (CMAS/SMAS). We also give a user a method and a software tool for finding the parameters of SMAS that are equivalent to given parameters of CMAS. The method and tool allow tuning parameter values for SMAS (adjusted to CMAS). These values may be later used in the proposed auto scaling control module.

Advantage of CMAS/SMAS approach results in its intuitiveness and simplicity comparing it to other more complex reactive models like [3] for example.

The paper presents some improvements of the reactive auto scaling model proposed in [2].

In the paper we justify the need for error metric (% of incorrectly processed requests per unit of time). This allows to minimize an impact of error requests on main QoS statistics (the q^{th} quantile) based on execution times of all requests.

The first contribution is an extension of the scaling-out model that allows to early react on an increased load by using thresholds for group CPU Utilization that depend on the number of currently operating service nodes. Early turning on an additional node may cause better QoS (early try of overloading avoidance).

The second contribution is an extension of early scaling-in model where a function of probability of launching scaling-in (decreasing number of service nodes) was introduced, giving some nondeterministic solution. Early turning off a service node may cause a lower cost (early turned-off nodes does not load a budget).

The future work will concentrate on detail experimental verification the proposed extensions according to different load profiles.

We plan to verify a usefulness of introduction non-linear elements like hysteresis and dead zones into scaling-in/-out algorithms that operate on metrics.

References

1. Qu, C., Calheiros, R.N., Buyya, R.: Auto-scaling web applications in clouds: A taxonomy and survey. CoRR **abs/1609.09224** (2016)
2. Augustyn, D.R., Warchal, L.: Metrics–based auto scaling module for amazon web services cloud platform. In: Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation: 13th International Conference, BDAS 2017, Ustroń, Poland, May 30 – June 2, 2017, Proceedings, Springer International Publishing (2017) 42–52
3. Dias De Assuncao, M., Cardonha, C., Netto, M., Cunha, R.: Impact of User Patience on Auto-Scaling Resource Capacity for Cloud Services. Future Generation Computer Systems (2015) 1–10
4. Jiang, J., Lu, J., Zhang, G., Long, G.: Optimal cloud resource auto-scaling for web applications. In: 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16, 2013. (2013) 58–65
5. Roy, N., Dubey, A., Gokhale, A.: Efficient autoscaling in the cloud using predictive models for workload forecasting. In: Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing. CLOUD '11, Washington, DC, USA, IEEE Computer Society (2011) 500–507

6. Calheiros, R.N., Masoumi, E., Ranjan, R., Buyya, R.: Workload prediction using ARIMA model and its impact on cloud applications' qos. IEEE Trans. Cloud Computing **3**(4) (2015) 449–458