Journal of Computing and Security

October 2015, Volume 2, Number 4 (pp. 271–279)

http://www.jcomsec.org

A Workload-Adaptable Method for Migrated Software to the Cloud

Zahra Reisi [a], Omid Bushehrian [a,*], Farahnaz Rezaeian Zadeh [b]

[a] *Department of Computer Engineering and IT, Shiraz University of Technology, Shiraz, Iran.*

[b] *Department of Electronics, Computer and Information Technology Engineering, Foolad Institute of Technology, Fooladshahr, Isfahan 8491663763, Iran.*

ARTICLE INFO.

Article history:
Received: 26 June 2015
Revised: 09 April 2016
Accepted: 21 July 2016
Published Online: 26 August 2016

Keywords:
Migration to Cloud, Auto-Scaling, Learning Automata, Transient Workload.

ABSTRACT

Due to the elastic nature of the cloud environments, migration of the legacy software systems to the cloud has become a very attractive solution for service providers. To provide Software-as-a-Service (SaaS), an application provider has to migrate his software to the cloud infrastructure. The most challenging issue in the migration process is minimizing the cloud infrastructure (VM's) costs while preserving the quality requirements of the service consumers. In this paper a self-adaptive method for migrated applications to the cloud is proposed in which an intelligent auto-scaling component continuously monitors the incoming application workloads and performs vertical or horizontal scaling. Since reacting to the transient workload changes always results in useless sequences of "acquire-release" actions of cloud resources and imposes unwanted overhead costs on the service provider, the auto-scaling component recognizes the transient workloads using a Learning Automata and only reacts to the stable ones. The OpenStack platform is used for evaluating the applicability of proposed method in real cloud environments. The experimental results demonstrate the ability of the proposed method in recognizing the transient workloads and consequently reducing the overall costs of the service provider.

1 Introduction

The SLAs (Service Level Agreements) between service provider and service consumers are the most important artifact upon which the required resource capacities of the migrated application to the cloud are planned. However the fluctuating workload of recent cloud applications requires an elastic mechanism to adapt the amount of the acquired resources to the current workload [1, 2]. It is essential for the cloud provider to minimize the cost of rented resources while preserving the QoS requirements specified in the SLAs during the application execution. To achieve this, an important task is to find the optimal amount of resources for different kind of workloads before migrating the application to the cloud which is performed during the capacity planning activity. Usually the service provider scaling is driven by a set of scaling rules which specify the correct time for scaling up or down [3, 4].

With APIs provided by commercial cloud providers such as Amazon, the developer can define and configure the auto-scaling rules. In order to scale up/down the Amazon VM's, an object called *Amazon Cloud Watch Alarm* is applied. This object is a monitor agent responsible for checking a VM QoS status over a period of time and triggering the scaling rules when the

* Corresponding author.
Email addresses: z.reisi@sutech.ac.ir (Z. Reisi), Bushehrian@sutech.ac.ir (O. Bushehrian), fr.rezaeeian@gmail.com (F. Rezaeian Zadeh)
ISSN: 2322-4460

QoS falls below a predefined threshold. The scaling rules are defined for changing (add/delete) the number of VM instances (horizontal scaling) or changing the size of current VMs (vertical scaling).

However an important open issue regarding the auto-scaling mechanisms is the adaptation of scaling rules based on runtime conditions: The scaling rule set may involve some probabilistic rules whose triggering probabilities change over time based on the stability or instability of the application workloads. For instance, consider scaling rule saying that "if the workload changes to L_i then add one VM to current VM group". However, if the workload L_i is not stable and immediately vanishes, changing the current VM set and software deployment is useless. Generally, reacting to the transient workload changes always results in useless sequences of "acquire-release" actions of cloud resources which impose unwanted overhead costs to the service provider due to the configuration changes. Therefore, it is essential to foresee the future conditions of the system workload by learning from previous events.

In this paper, a new self-adaptive method for migrated applications to the cloud is proposed. The main objective of our strategy has been predicting the transient workloads and reacting intelligently to them. We believe that by this strategy, the total resource costs of the service provider is reduced significantly as the reconfiguration process happens only when the change is stable. We have implemented the proposed model on the OpenStack [1] environment.

The paper is organized as follows: Section 2 includes a review of the literature on auto-scaling in the cloud. Our strategy is introduced in Section 3 and the experiment results are demonstrated in Section 4. The conclusion and the future work are described in Section 5.

2 Related Work

Workload prediction is the most applied technique for auto-scaling in cloud environments. The main objective here is to address the challenges of horizontal and vertical scaling. A framework is proposed in [5] for automatic scaling called "SmartScale" in which the horizontal or vertical scaling is applied at each stage to make a trade-off between cost changing configuration and SLA violation. SmartScale uses horizontal and vertical scaling together and in correct time by means of the decision tree technique. The scaling process is executed when the workload is changed. In [6], a controller is embedded within the auto-scaling component for capacity planning to find the optimal number of resources corresponding to the upcoming workload. In [7] a linear regression with an auto-correlation function is used for predicting the number of web requests. The main objective of this mode is to find an optimal trade-off between cost and latency when resources are allocated. The relationship between the number of VMs and the latency is defined based on an M/M/m queuing model. In [8] an efficient model is proposed for resource allocation based on the prediction techniques. Hereby using the second order Auto Regression Moving Average method (ARMA) it was shown that using a small number of VMs can effectively satisfy both QoS requirements and cost reduction. In [9] the scaling problem is formulated as an integer programming problem. The architecture for scaling has two parts including runtime and pre-runtime. The results showed an optimal trade-off between cost and SLA objectives.

In some studies on transient and stable workloads, the stableness of the workload is defined statically [10, 11]. However, it is highly dependent to workload changes. To address this shortcoming, in [12] a dynamic threshold based on meta-rules is proposed by which the threshold is adapted based on the workload changes.

An important and ignored aspect in most of the previous studies is the effect of transient workload on the cloud cost which is paid by the service provider. We addressed this shortcoming in our previous paper in [13] where an intelligent Auto-Scaling Engine (iScale) is presented which recognizes the transient workload changes using a Learning Automata and only reacts to the stable workloads.

In this paper the architecture presented in [13] is extended by augmenting a new component called "Virtualization Manager" which is responsible for interfacing with cloud platform to perform vertical or horizontal scaling.

3 Cloud Auto-Scaling Approach

The proposed auto-scaling approach applies both vertical scaling and horizontal scaling. Auto-scaling is performed based on the type of workload. There are two types of workloads: stable load and transient load. Identifying the type of workload is performed by Learning Automata. The proposed method in this paper satisfies three important requirements: reducing the cost of application deployment on the cloud, satisfying the SLA constraints, and decreasing the number of runtime software re-configuration due to new resource demands. Since these requirements are conflicting with each other, achieving the optimal trade-off is an important issue in migrated-to-cloud applications.

[1] http://www.openstack.org

Corresponding to each workload change from L_i to L_j a *durability threshold* is required to classify the new workload L_j as "stable" or "transient". If workload L_j is not changed before its corresponding durability threshold, it is recognized as a "stable" load, otherwise, it is a "transient" load. These thresholds are computed based on the cloud costs paid due to the newly required resources for supporting workload L_j and SLA violation penalty. The durability of each load change from L_i to L_j is learned by a Learning Automata LA_{ij} based on its durability history. If a load change from L_i to L_j is learned as "stable" by LA_{ij}, it means that the act of scaling out/in the current configuration to the new configuration C_j is more profitable than keeping the current configuration.

Assuming that the migrated software initially is deployed on a set of virtual machines acquired from the IaaS provider, scaling up/down the system based on changing the workload means that the allocated resources to the software and the assignment of software components to the virtual machines may change and a new configuration is installed for the software. Before starting the migration process, the required configuration G_i to support the known workload L_i has to be specified by the designer during an activity called Capacity Planning [14]. Here, based on the QoS requirements of the end-users specified in the Service Level Agreements (SLA), the designer determines the required capacity (the number of virtual machines and the size of allocated resources on each one) and the optimal assignment of software components to virtual machines. So in this case, the SLA constraints are not violated for each known workload L_i. The outcome of this activity is the Configuration Plan of the migrated software which is defined as a mapping between each workload L_i and its corresponding configuration G_i. The resources in G_i are sufficient to provide the services to the end-users when the workload is changed to L_i according to the predefined SLA [13]. Each configuration G_i is formally defined as triple (C, M, A) where C is the set of software components to be started, M is the set of VM's on which the software components are deployed and A is a function that maps each component $c_k \in C$ to a $vm_j \in M$.

3.1 Architecture

The proposed self-adaptable architecture is shown in Figure 1. As explained in the previous section, a *configuration* is a set of virtual machines which are allocated to the application components. The current configuration is selected by the *Scaling Manager* component based on the Configuration Plan. *Load Watch* monitors the incoming traffic of virtual machines to detect changes in the workload so that the Scaling Manager component decides if the current configuration needs to be adapted to the new workload. After identifying a change in the workload, the new workload is classified into one of the predefined workload classes by measuring the time intervals between subsequent requests. A *Learning Automaton (LA)* decides when to scale up/down the current configuration by learning from the previous history of workloads. The idea is to change the current configuration only if the new workload is not transient. LA learns the type of workload (transient/stable) gradually from the past events. When the decision about the next configuration is made by the Scaling Manager, the Virtualization Manager component prepares the resizing commands according to the standards of the cloud infrastructure platform and sends them to change the current configuration.

3.2 Virtualization Manager

Scaling commands are prepared by the Virtualization Manager. Virtualization Manager receives the new configuration and uses VM resizing to scale up (or down) the current configuration.

Virtualization Manager consists of the following sub-modules:

(1) Rectifier: This module monitors the current workload and current configuration periodically and triggers a configuration change request if needed. This configuration change request is essential when the last action chosen by the LA in Scaling Manager was wrong and needs to receive penalty. This situation happens when the LA action is "not change the current configuration" however after the predefined threshold is passed it is observed that the load is not transient and the LA action was false.

(2) Adapter: This module hides the variations of the cloud infrastructure API from the other modules. It translates the auto-scaling requests (vertical/horizontal) to the specific language of the IaaS platform. Hence changing the cloud infrastructure API has no effect on other modules in the proposed architecture.

3.3 Cost model and Parameters

Suppose that the input workload is L_c and the current configuration of the application is C_i. If the workload changes to another class such as L_j, it may be reacted by the auto-scaling engine either by scaling out the configuration C_i and installing the new configuration C_j or ignoring the load change and continuing with the current configuration. If the first action is taken, a vertical or horizontal scaling may happen. The cost of horizontal or vertical scaling is computed by Equa-

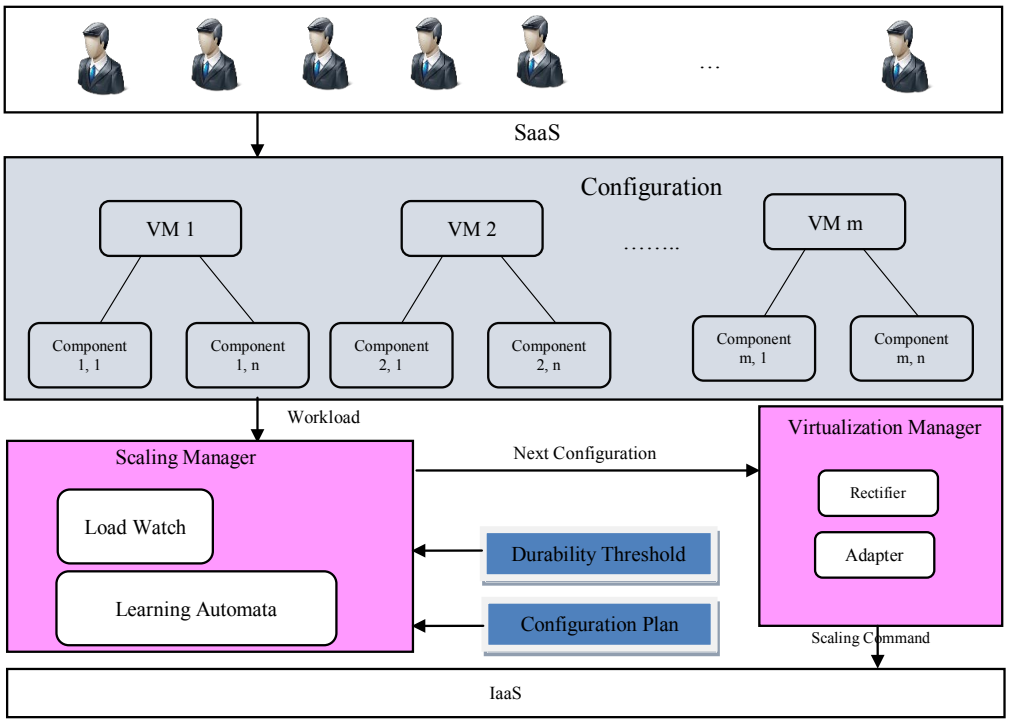

Figure 1. The Components of the Auto-Scaling Architecture

tions (1) and 2 respectively as follows [13]:

$$Cost_H(t,i,j) = \int_0^{t_{c_{ij}}} P_{change}(t) + Cost_{C_j} \times (t - t_{c_{ij}}) + Cost_{License}(i,j) \quad (1)$$

$$Cost_v(t,i,j) = Cost_{c_i} \times t \quad (2)$$

Where $t_{c_{ij}}$ denotes the reconfiguration time from C_i to C_j, $P_{change}(t)$ is the SLA violation function related to the horizontal configuration, $Cost_{C_j}$ is the cost of new configuration, and $Cost_{License}(i,j)$ is the cost of new software licenses in configuration C_j. These two equations compute the cost that has to be paid to the cloud provider up to time t plus the penalty that has to be paid to the end-users due to violating SLA during the configuration change (only when the scaling is horizontal) plus the cost of the new software license (only when the scaling is horizontal). If the action is ignoring the workload change, the current configuration is kept and the total cost incurred up to time t is calculated as follows:

$$IgnoreCost(t,i,j) = \int_0^t P_{c_i l_j}(t) + Cost_{C_i} \times t \quad (3)$$

The penalty function due to the violation of SLA at time point t is denoted by $P_{c_i l_j}(t)$, the cost of the current configuration is shown by $Cost_{C_i}$. This formula computes the cost of penalty which is paid to the end-users due to the violation of SLA plus the cost of keeping configuration C_i up to time t.

The threshold value α_{ij} is the time point at which the cost of changing the configuration calculated by Equation(1) or (2), and the cost of keeping the current configuration calculated by Equation (3) are equal. During the workload change, as the SLA violation is increased gradually, the cost of keeping the current configuration is always less than the cost of reconfiguration up to time α_{ij} . This reasoning is obvious due to the fact that function $P_{c_i l_j}(t)$ is assumed as a strictly increasing function.

3.4 Learning Method

A Learning Automaton is a finite state machine that aims to apply the best action on the environment through a learning process. The best action is the one that maximizes the probability of receiving rewards from the environment. LA chooses an action repeatedly based on the action probabilities and then updates the action probabilities considering the environment responses.

A Learning Automaton can be presented formally as the tuple: $x, \phi, \alpha, P, A, G)$, where x is the input set,$\alpha = \{\alpha_1, \cdots, \alpha_r\}$ is a set of automaton actions, $F : \phi \times \beta \mapsto \phi$ is the production function which determines the new automaton state based on the current state and the inputs, $P(n) = \{P1(n), \cdots, Pr(n)\}$ denotes the action probabilities at stage n, $G : \phi \mapsto \alpha$ is a function that maps the current state into the current output, $\phi = \{\phi_1, \phi_2, \cdots, \phi_k\}$ is the set of internal states and A is the learning algorithm [15].

In the proposed method, to learn the durability of each individual load change, corresponding to each load change from L_i to L_j a learning automaton LA_{ij} with two actions "ignore", "scale" is defined. "Ignore" means stay on the current configuration and "scale" means change to the new configuration. Once a load change happens, the learning algorithm compares the observed durability of the last workload with its predefined threshold; if it was greater, the "scale" action has to be rewarded, otherwise, the "ignore" action is rewarded.

Suppose that m is the number of configurations and n is the number of workload classes. For each current workload L_i and next workload L_j $(i \neq j)$ the stability of L_j has to be learned by a LA_{ij} so we need $n \times (n-1)$ LAs for all combinations of workloads. However, to determine that the change L_iL_j is stable or not, the observed stability of L_j has to be compared with the value of durability threshold $\alpha_{c,j}$, which is dependent on the current configuration C_c as explained in previous subsection. For example, if $\alpha_{c,j} = 100$ and L_j lasts for 120 time units, L_j is assumed to be stable. However, if the change to L_j happens when the current configuration is C_t and $\alpha_{t,j} = 150$, L_j is assumed to be transient. As a result, for each current configuration C_c the interpretation of stability is different.

As a result, for each current configuration C_c a separate LA_{ij} is needed. Since there are m different configurations C_c and n different workloads, the total number of $m \times n \times (n-1)$ LAs is needed in the proposed approach.

4 Experiments

OpenStack is one of the popular open source software packages in cloud computing domain. This software controls a set of computing, storage and network resources and provides a rich REST API to manage them [16]. The auto-scaling activity is done by Heat [2] component which is compatible with AWS Cloud and contains a set of templates which are configured within the OpenStack.

[2] https://wiki.openstack.org/wiki/Heat

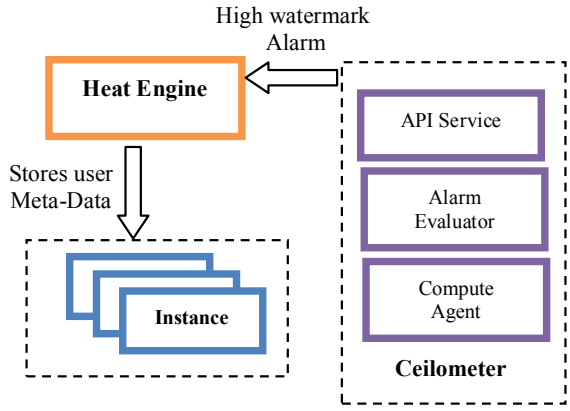

Figure 2. Heat Engine scales out stack [20]

Heat-API component sends the API request to Heat-Engine in order to be processed. The user puts all the information related to the application in a text file (compute instances, storage, floating IPs). Then, during the application execution, the required resources in the text file are provisioned by Heat whenever the scaling is requested. The environment of OpenStack is monitored by Ceilometer and collects the required information about the resources and system performance. Finally, a combination of Heat and Ceilometer activities will complete the scaling process [17–19].

The scaling process in Heat is depicted in Figure 2. The API service controls lifecycle alarms, the Compute agent gathers the statistical information and Alarm evaluator brings the alarm definition using the API Service. After delivering the alarm by Ceilometer, the Heat engine should start the scaling operations.

4.1 Experimental setup

The proposed approach is evaluated in single node OpenStack. The installation is performed with Heat and Devstack. The aim of experimental evaluation is to assess the accuracy of the proposed cost-aware scaling method in cloud services. The experiments are carried out on the OpenStack simulator, which is a framework for modeling and simulating the cloud computing infrastructures and services. The pricing scheme is based on the Amazon EC2 [3] pricing model. Four types of VMs are provided in OpenStack: small, medium, large and xlarge, and they have different virtual resource costs, as shown in Table 1.

Pre-defined classes are listed in Table 2. For each workload L_i a configuration G_i is defined. The workload used in this study is generated using the Rain tool [21]. Rain tool is a Markov-chain based workload generation toolkit which can easily use defined or empirical probability distributions to mimic different classes

[3] http://aws.amazon.com/ec2

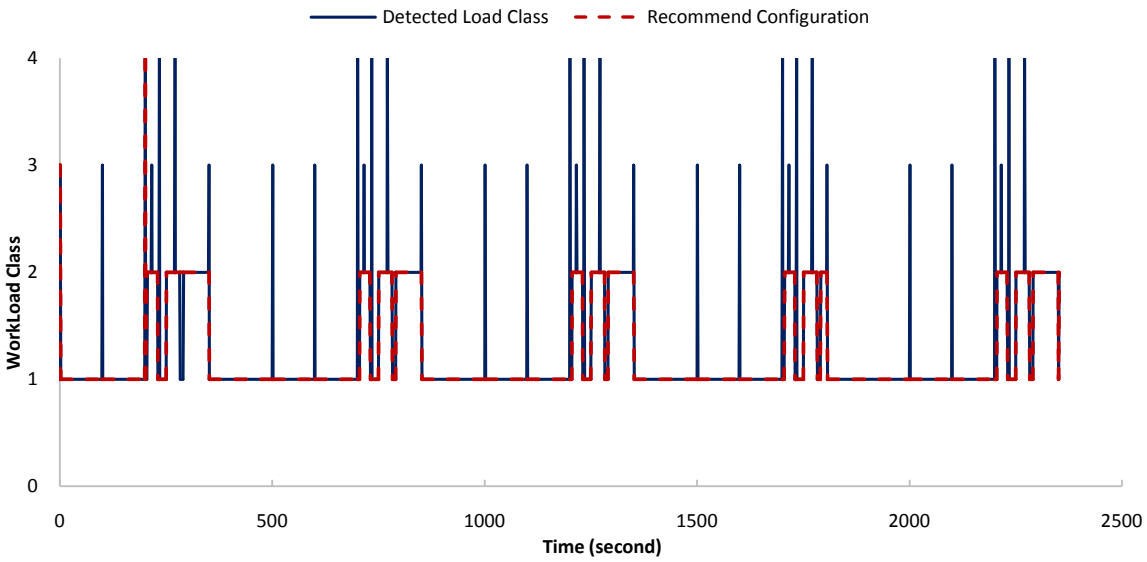

Figure 3. Detected Transient Workload (Mixed Workload)

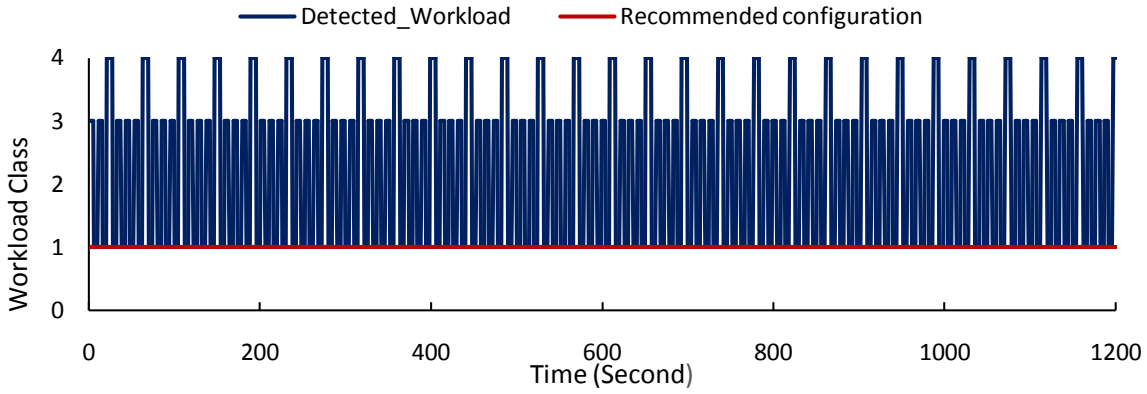

Figure 4. Detected Transient Workload (Transient Workload)

Table 1. The Patch Dimensions of the Antenna

Configuration	Memory	CPUs	Disk	Price (per hour)
Small	2 Gb	1	20Gb	$0.026
Medium	4 Gb	2	40Gb	$0.116
Large	8 Gb	4	80Gb	$0.232
xLarge	16 Gb	8	160Gb	$0.420

Table 2. Different Classes of Workload

Workload Class	Mean time between requests(ms)
L_1	100
L_2	150
L_3	200
L_4	300

of load variations. We implemented the Simple Scaling Method (SSM) and the proposed scaling method (ASM). In the SSM re-configuration is performed after each workload change regardless of its durability.

The first one will give a baseline for comparison purposes. Two types of composite workloads are used for experiments: Mixed and Transient. The former is a combination of stable and transient loads and the latter only consists of transient workloads.

4.2 Experimental Results

Experiment 1: Workload Type Recognition This experiment was designed to evaluate the accuracy of the proposed method in recognizing the transient loads. Two types of workloads were generated by means of Rain. As shown in Figures 3 and 4, ASM could successfully recognize the transient load changes. The learning process explained in previous section penalizes the "stable" action of its corresponding LA whenever a transient behavior is observed. As shown

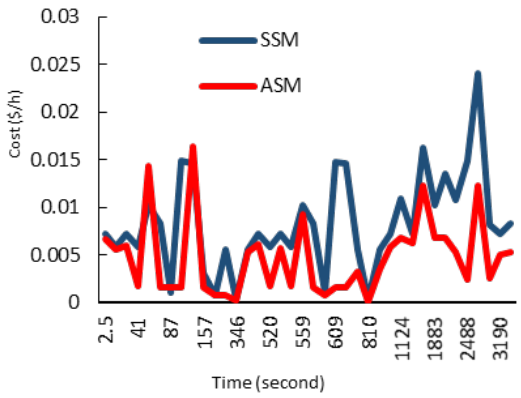

Figure 5. Total Cloud Cost

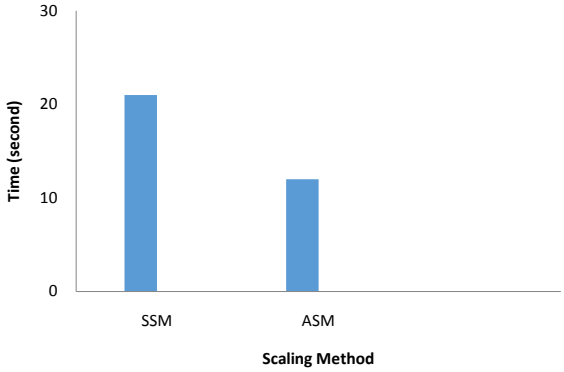

Figure 7. Comparing the SLA Violation Times

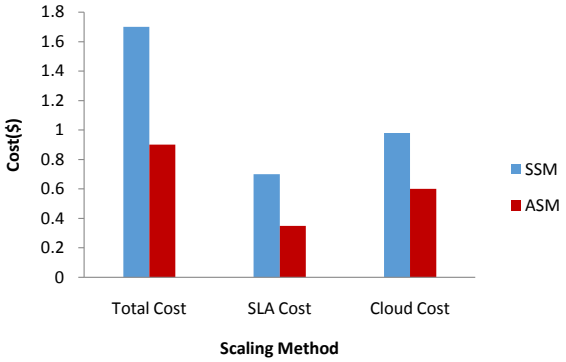

Figure 6. Cloud Cost and SLA Penalties

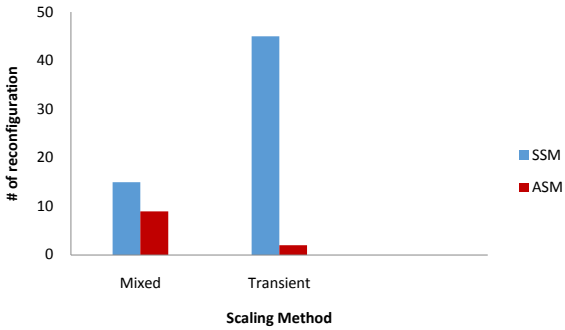

Figure 8. Number of Reconfigurations

in Figures 3 and 4, after a short period of time the LA converges and detects the workload type accurately.

Experiment 2: Total Cost
The total cost includes the SLA violation penalty and the cloud service usage for ASM and SSM approaches that is depicted in Figure 5. In this experiment it was observed that the total cost is initially the same in both methods. However after a short time, the total cost corresponding to the ASM method is drastically decreased in comparison to the SSM.

This is due to the fact that the ASM method predicts the workload type and applies the configuration with lower cost (it switches to the new more costly configuration only when the new load is stable; otherwise it keeps the current configuration).The results showed that the ASM had 52% reduction in total cost in comparison to the SSM.

The cost of SLA violation and cloud service usage is shown separately in Figure 6. Due to the performing useless reconfigurations in transient workload changes, the SSM approach resulted in higher cost of SLA violation. Each reconfiguration has indispensable SLA violation due to different service stop/start.

Experiment 3: SLA Violation It was observed that for transient workload pattern the SLA violation period was decreased by applying the ASM method as shown in Figure 7. Since ASM avoids useless reconfigurations during the transient load changes, less SLA violations are resulted in this method.

Experiment 4: The Number of Reconfigurations
In Figure 8, the number of reconfigurations in ASM method is compared with that of SSM method when using mixed and transient workloads. ASM approach was able to reduce nearly 50% of the reconfigurations in comparison to the SSM method. Obviously this is due to the fact that ASM method avoids a reconfiguration when the load is not stable. The effect of this behavior is very obvious when the workload pattern is transient rather than mixed.

5 Conclusion

In this paper the problem of recognizing the transient workloads in the cloud service is discussed. A self-management technique is proposed to recognize the transient workloads and also perform the auto-scaling process by interfacing the OpenStack platform. The Learning Automaton (LA) method was used in the proposed technique to learn the appropriate action during the load change. The scaling cost is also formu-

lated in order to achieve a cost effective configuration for the applications in cloud. The proposed technique is implemented using the Heat APIs in the well-known OpenStack platform. The experimental results show that the proposed method has higher accuracy, lower configuration costs, and reduced duration of the SLA violations in comparison to the SSM method.

References

[1] Chunming Rong, Son T. Nguyen, and Martin Gilje Jaatun. Beyond lightning: A survey on security challenges in cloud computing. *Computers & Electrical Engineering*, 39(1):47 – 54, 2013. ISSN 0045-7906. doi: http://dx.doi.org/10.1016/j.compeleceng.2012.04.015. URL `http://www.sciencedirect.com/science/article/pii/S0045790612000870`. Special issue on Recent Advanced Technologies and Theories for Grid and Cloud Computing and Bio-engineering.

[2] Jin Chen, G. Soundararajan, and C. Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *2006 IEEE International Conference on Autonomic Computing*, pages 231–242, June 2006. doi: 10.1109/ICAC.2006.1662403.

[3] C. C. Lin, J. J. Wu, J. A. Lin, L. C. Song, and P. Liu. Automatic Resource Scaling Based on Application Service Requirements. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 941–942, June 2012. doi: 10.1109/CLOUD.2012.32.

[4] Tzu-Chi Huang and Sherali Zeadally. Flexible architecture for cluster evolution in cloud computing. *Computers & Electrical Engineering*, 42:90 – 106, 2015. ISSN 0045-7906. doi: http://dx.doi.org/10.1016/j.compeleceng.2014.08.006. URL `http://www.sciencedirect.com/science/article/pii/S0045790614002195`.

[5] S. Dutta, S. Gera, A. Verma, and B. Viswanathan. SmartScale: Automatic Application Scaling in Enterprise Clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 221–228, June 2012. doi: 10.1109/CLOUD.2012.12.

[6] Jorge M Londoño-Peláez and Carlos A Florez-Samur. An Autonomic Auto-scaling Controller for Cloud Based Applications. *International Journal of Advanced Computer Science & Applications*, 1(4):1–6, 2013.

[7] J. Jiang, J. Lu, G. Zhang, and G. Long. Optimal cloud resource auto-scaling for web applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 58–65, May 2013. doi: 10.1109/CCGrid.2013.73.

[8] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, July 2011. doi: 10.1109/CLOUD.2011.42.

[9] Jingqi Yang, Chuanchang Liu, Yanlei Shang, Bo Cheng, Zexiang Mao, Chunhong Liu, Lisha Niu, and Junliang Chen. A cost-aware auto-scaling approach using the workload prediction in service clouds. *Information Systems Frontiers*, 16(1):7–18, 2014. ISSN 1572-9419. doi: 10.1007/s10796-013-9459-0. URL `http://dx.doi.org/10.1007/s10796-013-9459-0`.

[10] Emiliano Casalicchio and Luca Silvestri. *Autonomic Management of Cloud-Based Systems: The Service Provider Perspective*, pages 39–47. Springer London, London, 2013. ISBN 978-1-4471-4594-3. doi: 10.1007/978-1-4471-4594-3_5. URL `http://dx.doi.org/10.1007/978-1-4471-4594-3_5`.

[11] Jonathan Kupferman, Jeff Silverman, Patricio Jara, and Jeff Browne. Scaling into the cloud. Technical report, University of California, Santa Barbara, 2009.

[12] Guillermo Botella and Alberto A Del Barrio. Comparison of Auto-scaling Techniques for Cloud Environments. *Actas de las XXIV Jornadas de Paralelismo*, 2013.

[13] Omid Bushehrian and Zahra Reisi. iScale: An Intelligent Auto-Scaling Engine for Migrated Applications to the Cloud. *ARPN Journal of Systems and Software*, 4(5):116–122, 2014.

[14] Farahnaz Rezaeian Zadeh, Akhilesh Kumar Pandey, Mohamad Davarpanah Jazi, and Abhishek Kumar. Capacity Planning Scientific Workflow on Cloud through Ant Colony Approach. In *Eighth International Conference on Computer communication networks (ICCN 2014)*, pages 19–25. Elsevier, 2014.

[15] Hamid Beigy and M.R. Meybodi. Learning automata based dynamic guard channel algorithms. *Computers & Electrical Engineering*, 37(4):601 – 613, 2011. ISSN 0045-7906. doi: http://dx.doi.org/10.1016/j.compeleceng.2011.04.003. URL `http://www.sciencedirect.com/science/article/pii/S0045790611000413`.

[16] Leander Beernaert, Miguel Matos, Ricardo Vilaça, and Rui Oliveira. Automatic Elasticity in OpenStack. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, SDMCMM '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1615-6. doi: 10.1145/

2405186.2405188. URL http://doi.acm.org/10.1145/2405186.2405188.

[17] Auto scaling with Heat and Ceilometer. URL http://techs.enovance.com/5991/autoscaling-withheat-and-ceilometer.

[18] Ceilometer + Heat = Alarming. URL http://slideshare.net/NicolasBarcet/ceilometer-heatequalsalarming-icehousesummit.

[19] Hyungro Lee, Gregor von Laszewski, Fugang Wang, and Geoffrey C Fox. Towards understanding cloud usage through resource allocation analysis on xsede. Technical report, Technical Report March 25 2014, Community Grids Lab Publications.

[20] Pratibha R Gupta, Sheetal Taneja, and Aparna Datt. Using Heat and Ceilometer for providing Autoscaling in OpenStack. *International Journal of Information, Communication and Computing Technology Jagan Institute of Management Studies, New Delhi*, pages 84–89, July 2014.

[21] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, David A Patterson, et al. Rain: A workload generation toolkit for cloud computing applications. Technical report, University of California, Tech. Rep. UCB/EECS-2010-14, 2010.

Zahra Reisi received the B.Sc degree in software engineering from the Hakim Sabzevari University in 2010, IRAN, the MSc degree in software engineering from the Shiraz University of Technology, IRAN. Her research interests include cloud computing and data mining.

Omid Bushehrian received the BSc in software engineering from AmirKabir University of Technology (Tehran Polytechnics), IRAN, the MSc and Ph.D degrees in software engineering from the University of Science and Technology, IRAN. He is currently a faculty member at Shiraz university of Technology, IRAN. His main research interest is cloud computing and software quality.

Farahnaz Rezaeian Zadeh received her B.Sc degree in software engineering from the Hakim Sabzevari University in 2010. She received her M.Sc degree in Information Technology (IT) Engineering from the Foolad Institute of Technology, Foolad Shahr, Isfahan, Iran in 2014. She has already authored more than 10 international conference papers.