2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling

Hamid Arabnejad*, Claus Pahl†, Pooyan Jamshidi‡ and Giovani Estrada§
*IC4, Dublin City University, Dublin, Ireland
†Free University of Bozen-Bolzano, Bolzano, Italy
‡Imperial College London, London, UK
§Intel, Leixlip, Ireland

***Abstract*—A goal of cloud service management is to design self-adaptable auto-scaler to react to workload fluctuations and changing the resources assigned. The key problem is how and when to add/remove resources in order to meet agreed service-level agreements. Reducing application cost and guaranteeing service-level agreements (SLAs) are two critical factors of dynamic controller design. In this paper, we compare two dynamic learning strategies based on a fuzzy logic system, which learns and modifies fuzzy scaling rules at runtime. A self-adaptive fuzzy logic controller is combined with two reinforcement learning (RL) approaches: (i) Fuzzy SARSA learning (`FSL`) and (ii) Fuzzy Q-learning (`FQL`). As an *off-policy* approach, Q-learning learns independent of the policy currently followed, whereas SARSA as an *on-policy* always incorporates the actual agent's behavior and leads to faster learning. Both approaches are implemented and compared in their advantages and disadvantages, here in the OpenStack cloud platform. We demonstrate that both auto-scaling approaches can handle various load traffic situations, sudden and periodic, and delivering resources on demand while reducing operating costs and preventing SLA violations. The experimental results demonstrate that `FSL` and `FQL` have acceptable performance in terms of adjusted number of virtual machine targeted to optimize SLA compliance and response time.**

***Keywords*-Cloud Computing; Orchestration; Controller; Fuzzy Logic;Q-Learning; SARSA; OpenStack**

I. Introduction

Automated elasticity and dynamism, as two important concepts of cloud computing, are beneficial for application owners. *Auto-scaling* system is a process that automatically scales the number of resources and maintains an acceptable Quality of Service (QoS) [19]. However, from the perspective of the user, determining when and how to resize the application makes defining a proper auto-scaling process difficult. Threshold-based auto-scaling approaches are proposed for scaling application by monitoring metrics, but setting the corresponding threshold conditions still rests with the user. Recently, automatic decision-making approaches, such as reinforcement learning (RL) [24], have become more popular. The key advantage of these methods is that *prior* knowledge of the application performance model is not required, but they rather learn it as the application runs.

Our motivation here is to compare two different auto-scaling services that will automatically and dynamically resize user application to meet QoS requirements cost-effectively. We consider extensions of two classic RL algorithms, namely SARSA and Q-Learning, for the usage with a fuzzy auto-scaling controller for dynamic resource allocations. RL is defined as interaction process between a learning agent (the auto-scaling controller) and its environment (the target could application). The main difference between SARSA and Q-learning is that SARSA compares the current state vs. the actual next state, whereas Q-Learning compares the current state vs. the best possible next states.

Generally, RL approaches suffer from the size of the table needed to store state-action values. As a solution, a fuzzy inference system offers a possible solution to reducing the state space. A fuzzy set is a mapping of real state to a set of fuzzy labels. Therefore, many states can be represented by only a few fuzzy states. Thus, we base our investigation on a fuzzy controller [17]. The combination of fuzzy logic control and RL approaches results in a self-adaptive mechanism where the fuzzy logic control facilitates the reasoning at a higher level of abstraction, and the RL approaches allow to adjust the auto-scaling controller. This paper extend previous results [14], [16] as follows. First, we specifically focus on architecture, implementation and experimentation aspects in OpenStack. Then, we utilise SARSA approach as an *on-policy* learning algorithm against Q-learning which is an *off-policy* approach. The advantage of using SARSA, due to following the action which is actually being taken in the next step, is the policy that it follows will be more optimal and learning will be faster. Furthermore, a comparison between the two strategies will be provided. The comparison analysis is an important goal to know the performance and scalability of each RL approaches under different workload patterns.

The contributions of this paper are:

- a review of cloud auto-scaling approaches;
- integrate RL and fuzzy approaches as an automatic decision-making in a real cloud controller;
- implementation of Fuzzy RL approaches in OpenStack (industry-standard IaaS platform);
- extensive experimentation and evaluation of wide range of workload patterns;
- comparison between two RL approaches, SARSA and Q-learning in terms of quality of results

We show that the auto-scaling approaches can handle various load traffic situations, delivering resources on demand while

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.15

reducing infrastructure and management costs alongside the comparison between both proposed approaches. The experimental results show promising performance in terms of resource adjustment to optimize Service Level Agreement (SLA) compliance and response time while reducing provider costs.

The paper is organized as follows. Section II describes auto-scaling process briefly, and discusses on related research in this area, Section III describes the OpenStack architecture and orchestration, Section IV describes our proposed `FSL` approach in details followed by implementation in Section V. A detailed experiment-based evaluation follows in Section VI.

II. Background and Related Work

The aim of auto-scaling approaches is to acquire and release resources dynamically while maintaining an acceptable QoS [19]. The auto-scaling process is usually represented and implemented by a MAPE-K (Monitor, Analyze, Plan and Execute phases over a Knowledge base) control loop [12].

An auto-scaler is designed with particular goal, relying on scaling abilities offered by the cloud providers or focusing on the structure of the target application. We can classify auto-scaling approaches based on usage theory and techniques:

A. Threshold-based rules

Threshold-based rules are the most popular approach offered by many platforms such as Amazon EC2[1], Microsoft Azure[2] or OpenStack[3]. Conditions and rules in threshold-based approaches can be defined based on one or more performance metrics, such as CPU load, average response time or request rate. Dutreilh et al. [6] investigate horizontal auto-scaling using threshold-based and reinforcement learning techniques. In [9], the authors describe a lightweight approach that operates fine-grained scaling at resource level in addition to the VM-level scaling in order to improve resource utilization while reducing cloud provider costs. Hasan et al. [10] extend the typical two threshold bound values and add two levels of threshold parameters in making scaling decisions. Chieu et al. [4] propose a simple strategy for dynamic scalability of PaaS and SaaS web applications based on the number of active sessions and scaling the VMs numbers if all instances have active sessions exceed particular thresholds. The main advantage of threshold-based auto-scaling approaches is their simplicity which make them easy to use in cloud providers and also easy to set-up by clients. However, the performance depends on the quality of the thresholds.

B. Control theory

Control theory deals with influencing the behaviour of dynamical systems by monitoring output and comparing it with reference values. By using the feedback of the input system (difference between actual and desired output level), the controller tries to align actual output to the reference. For auto-scaling, the reference parameter, i.e., an object to be controlled, is the targeted SLA value [15]. The system is the target platform and system output are parameters to evaluate system performance (response time or CPU load). Zhu and Agrawal [26] present a framework using Proportional-Integral (PI) control, combined with a reinforcement learning component in order to minimize application cost. Ali-Eldin et al. [2], [1] propose two adaptive hybrid reactive/proactive controllers in order to support service elasticity by using the queueing theory to estimate the future load. Padala et al. [21] propose a feedback resource control system that automatically adapts to dynamic workload changes to satisfy service level objectives. They use an online model estimator to dynamically maintain the relationship between applications and resources, and a two-layer multi-input multi-output (MIMO) controller that allocates resources to applications dynamically. Kalyvianaki et al. [18] integrate a Kalman filter into feedback controllers that continuously detects CPU utilization and dynamically adjusts resource allocation in order to meet QoS objectives.

C. Time series analysis

The aim of time series analysis is to carefully collect and study the past observations of historical collect data to generate future value for the series. Some forecasting models such as Autoregressive (AR), Moving Average (MA) and Autoregressive Moving Average (ARMA) focus on the direct prediction of future values, whereas other approach such as pattern matching and Signal processing techniques first try to identify patterns and then predict future values. Huang et al. [11] proposed a prediction model (for CPU and memory utilization) based on double exponential smoothing to improve the forecasting accuracy for resource provision. Mi et al. [20] used Browns quadratic exponential smoothing to predict the future application workloads alongside of a genetic algorithm to find a near optimal reconfiguration of virtual machines. By using ARMA, Roy et al. [23] presented a look-ahead resource allocation algorithm to minimizing the resource provisioning costs while guaranteeing the application QoS in the context of auto-scaling elastic clouds. By combining a sliding window approach over previous historical data and artificial neural networks (ANN), Islam et al. [13] proposed adaptive approach to reduce the risk of SLA violations by initializing VMs and perform their boot process before resource demands. Gong et al. [8] used the Fast Fourier Transform to identify repeating patterns. The Major drawback relies on this category is the uncertainty of prediction accuracy that highly on target application, input workload pattern, the selected metric, the history window and prediction interval, as well as on the specific technique being used [19].

D. Reinforcement learning (RL)

RL [24] is learning process of an agent to act in order to maximize its rewards. The standard RL architecture is given in Figure 1. The agent is defined as an auto-scaler, the action is scaling up/down, the object is the target application and the reward is the performance improvement after applying the action. The goal of RL is how to choose an action in

[1]http://aws.amazon.com/ec2
[2]http://azure.microsoft.com
[3]https://www.openstack.org

response to a current state to maximize the reward. There are several ways to implement the learning process. Generally, RL approaches learn estimates of the Initialized Q-values $Q(s, a)$, which maps all system states s to their best action a. We initialise all $Q(s, a)$ and during learning, choose an action a for state s based on ϵ-greedy policy and apply it in the target platform. Then, we observe the new state s' and reward r and update the Q-value of the last state-action pair $Q(s, a)$ with respect to the observed outcome state (s') and reward (r).

Two well-known RL approaches are SARSA and Q-learning [24]. Dutreilh et al. [5] use an appropriate initialization of the Q-values to obtain a good policy from the start as well as convergence speedups to quicken the learning process for short convergence times. Tesauro et al. [25] propose a hybrid learning system by combining queuing network model and SARSA learning approach to make resource allocation decisions based on application workload and response time.

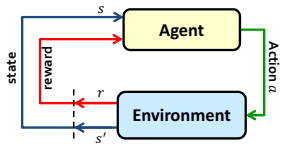


Fig. 1. The standard architecture of the RL algorithm

The important feature of RL approaches is learning without prior knowledge of the target scenario and ability to online learn and update environmental knowledge by actual observations. However, there are some drawbacks in this approach such as taking long time to converge to optimal or near optimal solution for solving large real world problems and requiring good initialization of the Q-function.

III. OpenStack Orchestration

OpenStack is an IaaS open-source platform, used for building public and private clouds. It consists of interrelated components that control hardware pools of processing, storage, and networking resources throughout a data center. Users either manage it through a web-based dashboard, through command-line tools, or through a RESTful API. Figure 2 shows a high-level overview of OpenStack core services.

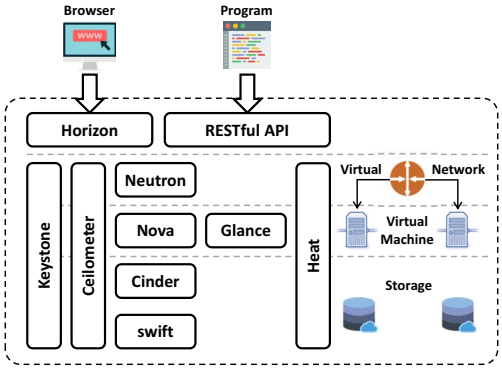


Fig. 2. An OpenStack block diagram

In OpenStack, 1) Neutron is a system for managing networks and IP addresses; 2) Nova is the computing engine for deploying and managing virtual machines; 3) Glance supports discovery, registration and delivery for disk and server images; 4) Ceilometer provides telemetry services to collect metering data; 5) Keystone provides user/service/endpoint authentication and authorization and 6) Heat is a service for orchestrating the infrastructure needed for cloud applications to run.

OpenStack Orchestration is about managing the infrastructure required by a cloud application for its entire lifecycle. Orchestration automates processes which provision and integrate cloud resources such as storage, networking and instances to deliver a service defined by policies. Heat, as OpenStack's main orchestration component, implements an engine to launch multiple composite applications described in text-based templates. Heat templates are used to create stacks, which are collections of resources such as compute instance, floating IPs, volumes, security groups or users, and the relationship between these resources. Heat along with Ceilometer can create an auto-scaling service. By defining a scaling group (e.g., compute instance) alongside using monitoring alerts (such as CPU utilization) provided by Ceilometer, Heat can dynamically adjust the resource allocation, i.e., launching resources to meet application demand and removing them when no longer required, see Figure 3. Heat executes Heat Orchestration Templates (HOT), written in YAML.

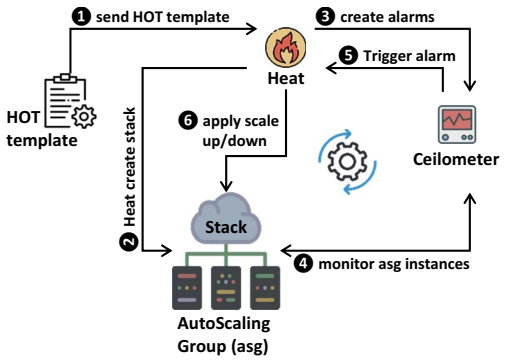


Fig. 3. Heat + Ceilometer architecture

By sending a HOT template file to the Heat engine, a new autoscaling group (`asg`) is created by launching a group of VM instances. The maximum and minimum number of instances should be defined in HOT file. Then, Ceilometer alarms that monitor all of the instances in `asg` are defined. Basically, at each Ceilometer interval time, the system checks the alarm metric and if it passed the defined threshold values, the scaling up/down policy will be performed based on defined action in the HOT file. During the life cycle of the application, all checking, testing and actions are performed automatically.

IV. On-Policy and Off-Policy RL Auto-Scaling

In [14], an elasticity controller based on a fuzzy logic system is proposed. The motivation factor for using fuzzy control systems is the fact that they make it easier to incorporate

human knowledge in the decision-making process in the form of fuzzy rules, but also reduce the state space.

We extend the fuzzy controller in the form of a SARSA-based Fuzzy Reinforcement Learning algorithm as an *on-policy* learning approach, called FSL, and describe this in more detail. Then, we related this to a Q-Learning-based *off-policy* learning approach, called FQL, by describing the differences.

A. Reinforcement Learning (RL)

Reinforcement learning [24] is learning by trial and error to map situations to actions, which aims to maximize a numerical reward signal. The learning process consists of two components: a) an agent (i.e., the auto-scaler) that executes actions and observes the results and b) the environment (i.e., the application) which is the target of the actions. In this schema, the auto-scaler as an agent interacts with an environment through applying *scaling actions* and receiving a response, i.e., the *reward*, from the environment. Each *action* is taken depending on the current *state* and other environmental parameters such as the input workload or performance, which moves the agent to a different *state*. According to the *reward* from system about the action quality, the auto-scaler will learn the best scaling action to take through a trial-and-error.

B. Fuzzy Reinforcement Learning (FRL)

We extend fuzzy auto-scaling with two well-known RL strategies, namely Q-learning and SARSA. We start with a brief introduction of the fuzzy logic system and then describe proposed FSL and FQL approaches.

The purpose of the fuzzy logic system is to model a human knowledge. Fuzzy logic allows us to convert expert knowledge in the form of rules, apply it in the given situation and conclude a suitable and optimal action according to the expert knowledge. Fuzzy rules are collections of IF-THEN rules that represent human knowledge on how to take decisions and control a target system. Figure 4 illustrates the main building blocks of a Fuzzy Reinforcement Learning (FRL) approach. During the lifecycle of an application, FRL guides resource provisioning. More precisely, FRL follows the autonomic MAPE-K loop by monitoring continuously different characteristics of the application (e.g., workload and response time), verifying the satisfaction of system goals and adapting the resource allocation in order to maintain goal satisfaction. The goals (i.e., SLA, cost, response time) are reflected in the reward function that we define later in this section.

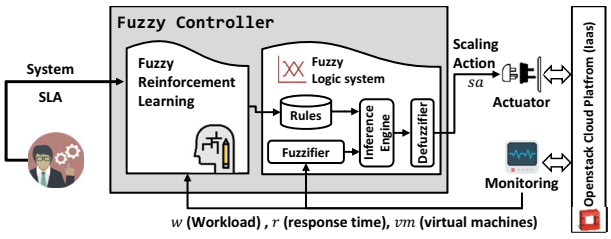


Fig. 4. FRL (logical) architecture

The monitoring component collects required metrics such as the workload (w), response time (rt) and the number of virtual machines (vm) and feeds both to the controller and the knowledge learning component. The controller is a fuzzy logic controller that takes the observed data, calculates the scaling action based on monitored input data and a set of rules, and as output returns the scaling action (sa) in terms of an increment/decrement in the number of virtual machines. The actuator issues adaptation commands from the controller at each control interval to the underlying cloud platform.

Generally, the design of a fuzzy controller involves all parts related to membership functions, fuzzy logic operators and IF-THEN rules. The first step is to partition the state space of each input variable into fuzzy sets through membership functions. The membership function, denoted by $\mu(x)$, quantifies the degree of membership of an input signal x to the fuzzy set y. Similar to [14], the membership functions, depicted in Figure 5, are triangular and trapezoidal. Three fuzzy sets have been defined for each input (i.e., workload and response time) to achieve a reasonable granularity in the input space while keeping the number of states small.

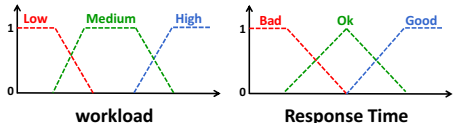


Fig. 5. Fuzzy membership functions for auto-scaling variables

For the inference mechanism, the elasticity policies are defined as rules: "IF (w is $high$) AND (rt is bad) THEN ($sa+=2$)", where w and rt are monitoring metrics stated in the SLA and sa is the change constant value in the number of deployed nodes, i.e., the VMs numbers.

Once the fuzzy controller is designed, the execution of the controller is comprised of three steps (cf. middle part of Figure 4): (i) fuzzification of the inputs, (ii) fuzzy reasoning, and (iii) defuzzification of the output. The fuzzifier projects the crisp data onto fuzzy information using membership functions. The fuzzy engine reasons based on information from a set of fuzzy rules and derives fuzzy actions. The defuzzifier reverts the results back to crisp mode and activates an adaptation action. This result is enacted by issuing appropriate commands to the underlying platform fabric.

Based on this background, we can now combine the fuzzy logic controller with the two RL approaches.

C. Fuzzy SARSA Learning (FSL)

By using RL approaches instead of relying on static threshold values to increase/decrease the amount of VMs, the performance of target application can be captured after applying each sa decision. In this paper, we use SARSA and Q-learning as RL approaches that we combine with the fuzzy controller. In this schema, a state s is modeled by a triple (w,rt,vm) for which an *RL* approach looks for best action a to execute. The combination of the fuzzy logic controller with SARSA [24] learning, called FSL, is explained in the following.

1) Initialize the q-values: unlike the threshold policy, the *RL* approach captures history information of a target application into a value table. Each member of the q-value table is assigned to a certain rule that describes some state-action pairs and is updated during the learning process. It can tell us the performance of taking the action by taking into account the reward value. In this study, we set all q-values to 0 as simplest mode.
2) Select an action: to learn from the system environment, we need to explore the knowledge that has already been gained. The approach is also known as the *exploration/exploitation* strategy. ϵ-greedy is known as a standard exploration policy [24]. Most of the time (with probability $1-\epsilon$), the action with the best reward will be selected or a random action will be chosen (with low probability ϵ) in order to explore non-visited actions. The purpose of this strategy is to encourage exploration. After a while, by decreasing ϵ, no further exploration is made.
3) Calculate the control action inferred by fuzzy logic controller: The fuzzy output is a weighted average of the consequences of the rule, which can be written as:

$$a = \sum_{i=1}^{N} \mu_i(x) \times a_i \tag{1}$$

where N is the number of rules, $\mu_i(x)$ is the firing degree of the rule i (or the degree of truth) for the input signal x and a_i is the consequent function for the same rule.
4) Approximate the Q-function from the current q-values and the firing level of the rules: In classical RL, only one state-action pair (rule) can be executed at once, which is not true for the condition of fuzziness. In a fuzzy inference system, more rules can be taken and an action is composed of these rules [7]. Hence, the Q value of an action a for the current state s is calculated by:

$$Q(s,a) = \sum_{i=1}^{N} \Big(\mu_i(s) \times q[i,a_i] \Big) \tag{2}$$

The action-value function $Q(s,a)$ tells us how desirable it is to reach state s by taking action a by allowing to take the action a many times and observe the return value.
5) Calculate reward value: The controller receives the current values of vm and rt that correspond to the current state of the system s. The reward value r is calculated based on two criteria: (i) the amount of resources acquired, which directly determine the cost, and (ii) SLO violations.
6) Calculate the value of new state s': By taking action a and leave the system from the current state s to the new state s', the value of new state denoted $V(s')$ by is calculated by:

$$V(s') = \sum_{i=1}^{N} \mu_i(s').\max_k(q[i,a_k]) \tag{3}$$

where $\max(q[i,a_k])$ is the maximum of the q-values applicable in the state s'.
7) Calculate error signal: As an *on-policy* approach, SARSA estimates the value of action a in state s using experience actually gathered as it follows its policy, i.e., it always incorporates the actual agent's behavior. We mark $\Delta Q(s,a)$ as the error signal given by:

$$\Delta Q_{\texttt{FSL}}(s,a) = r + \gamma \times Q(s',a') - Q(s,a) \tag{4}$$

where γ is a discount rate which determines the relative importance of future rewards. A low value for γ means that we value rewards that are close to time t, and a higher discount gives more value to the ones that are further in the future than those closer in time.
8) Update q-values: at each step, q-values are updated by :

$$q[i,a_i] = q[i,a_i] + \eta.\Delta Q.\mu_i\big(s(t)\big) \tag{5}$$

where η is the learning rate and takes a value between 0 and 1. Lower values for η mean that preferring old values slightly with every update and a higher η gives more impact on recent rewards.

The FSL solution is sketched in Algorithm 1.

Algorithm 1 Fuzzy SARSA learning(FSL)

Require: discount rate (γ) and learning rate (η)
initialize q-values
observe the current state s
choose partial action a_i from state s (ϵ-greedy strategy)
compute action a from a_i (Eq. 1) and its corresponding quality $Q(s,a)$ (Eq. 2)
repeat
    apply the action a, observe the new state s'
    receive the reinforcement signal (reward) r
    choose partial action a'_i from state s'
    compute action a' from a'_i (Eq. 1) and its corresponding quality $Q(s',a')$ (Eq. 2)
    compute the error signal $\Delta Q_{\texttt{FSL}}(s,a)$ (Eq. 4)
    update q-values (Eq. 5)
    $s \leftarrow s'$,$a \leftarrow a'$
until convergence is achieved

D. Fuzzy Q-Learning (FQL)

As we explained before, the major difference between Q-learning and the SARSA approach is their strategy to update q-values, i.e., in Q-learning q-values are updated using the largest possible reward (or reinforcement signal) from the next state. In simpler words, Q-learning is an *off-policy* algorithm and updates Q-table values independent of the policy the agent currently follows. In contrast, SARSA as an *on-policy* approach always incorporates the actual agent's behavior. Thus, the error signal for FQL is given by :

$$\Delta Q_{\texttt{FQL}}(s,a) = r + \gamma \times V(s') - Q(s,a) \tag{6}$$

The FQL is presented in Algorithm 2

As an example, we assume the state space to be finite (e.g., 9 states as the full combination of 3×3 membership functions for fuzzy variables w (workload) and rt (response

Algorithm 2 Fuzzy Q-Learning (FQL)

Require: discount rate (γ) and learning rate (η)

initialize q-values
observe the current state s
repeat
    choose partial action a_i from state s (ϵ-greedy strategy)
    compute action a from a_i (Eq. 1) and its corresponding quality $Q(s, a)$ (Eq. 2)
    apply the action a, observe the new state s'
    receive the reinforcement signal (reward) r
    compute the error signal $\Delta Q_{\text{FQL}}(s, a)$ (Eq. 6)
    Update q-values (Eq. 5)
    $s \leftarrow s'$
until convergence is achieved

time). Our controller might have to choose a scaling action among 5 possible actions $\{-2, -1, 0, +1, +2\}$. However, the design methodology that we demonstrated in this section is general and can be applied for any possible state and action spaces. Note, that the convergence is detected when the change in the consequent functions is negligible in each learning loop.

V. Implementation

We implemented prototypes of the FQL and FSL algorithms in OpenStack. Orchestration and automation within OpenStack is handled by the Heat component. The auto-scaling decisions made by Heat on when to scale application and whether scale up/down should be applied, are determined based on collected metering parameters from the platform. Collecting measurement parameters within OpenStack is handled by Ceilometer (see Figure 3). The main part of Heat is the stack, which contains resources such as compute instances, floating IPs, volumes, security groups or users, and the relationship between these resources. Auto-scaling in Heat is done using three main resources: (i) *auto-scaling group* is used to encapsulate the resource that we wish to scale, and some properties related to the scale process; (ii) *scaling policy* is used to define the effect a scale process will have on the scaled resource; and (iii) an *alarm* is used to define under which conditions the scaling policy should be triggered.

In our implementation, the environment contains one or more VM instances that are controlled by a load balancer and defined as members in autoscaling group resources. Each instance (VM) includes a simple web server to run inside of it after launching. Each web server listens to an input port (here port 80), returns a simple HTML page as the response. User data is the mechanism by which users can define their own pre-configuration as a shell script (the code of web server) that the instance runs on boot.

In Fig. 6, the template used for the web server is shown. For the VM web-server instance type, we used a minimal Linux distribution: the cirros[4] image was specifically designed for use as a test image on clouds such as OpenStack [3].

[4]CirrOS images, https://download.cirros-cloud.net/

The next step is defining the *scaling policy*, which is used to define the effect a scaling process will have on the scaled resource, such as "add -1 capacity" or "add +10% capacity" or "set 5 capacity". Figure 7 shows the template used for the scaling policy.

```
user_data_format: RAW
user_data: |
  #!/bin/sh
  ...
  while true
  do
    {
      echo "HTTP/1.1 200 OK"
      echo "Content-Length:$(wc -c /tmp/index.html | cut
 ↪  -d' ' -f1)"
      echo
      cat /tmp/index.html
    } | sudo nc -l -p 80
  done
  ...
```

Fig. 6. The simple web server

```
resources:
  ...
  web_server_scaleup_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      auto_scaling_group_id: {get_resource: asg}
      adjustment_type: change_in_capacity
      scaling_adjustment: 1
  ...
```

Fig. 7. The template for scaling policy

The scaling policy resource is defined as a type of OS::Heat::ScalingPolicy and its properties are as follows: 1) auto_scaling_group_id is the specific scaling group ID to apply the corresponding scale policy, 2) adjustment_type is the type of adjustment (absolute or percentage) and can be set to allowed values such as *change in capacity*, *exact capacity*, *percent change in capacity*, and 3) scaling_adjustment is the size of the adjustment in absolute value or percentage.

We used our auto-scaling manager instead of the native auto-scaling tool in OpenStack, which is designed by setting alarms based on threshold evaluations for a collection of metrics from Ceilometer. For this threshold approach, we can define actions to take if the state of the watched resource satisfies specified conditions. However, we replaced this default component by the FRL approaches, to control and manage scaling options. In order to control and manage scaling option by the two FRL approaches (FQL and FSL), we added an additional VM resource, namely ctrlsrv, which acts as an auto-scaling server and enacts the scale up/down decision proposed by either of the two FRL approaches. For ctrlsrv, due to the impossibility of installing any additional package in the cirros image, we considered a VM machine running a Linux Ubuntu precise server. Figure 8 illustrates the implemented system in OpenStack. The created load balancer distributes client's HTTP request across a set of web-servers, i.e., auto-scaling group members, collected in load balancer pool. The algorithm used to distribute load between the members of the pool is ROUND_ROBIN.

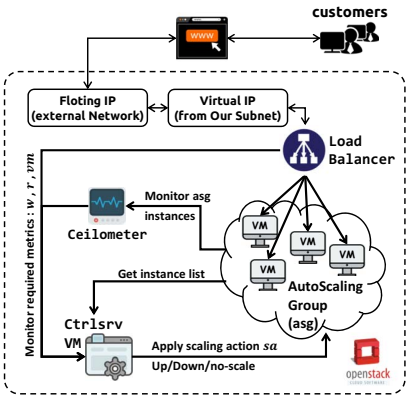


Fig. 8. Overview of the implemented system

Figure 8 shows the complete process of how the proposed fuzzy auto-scaling approach works. First, `ctrlsrv` gathers information from the load balancer, ceilometer and the current state of members (web-servers) in an autoscaling group, then decides which horizontal scaling, i.e., up or down, should be applied to the target platform. For instance, the scale-up even will launch a new web-server instance, which may take a few minutes as the instance needs to be started, and adds it to the load-balancer pool. The two proposed auto-scaling algorithms, `FQL` and `FSL`, are coded and run inside of the `ctrlsrv` machine. We implemented and added a complete fuzzy logic library. This is functionally similar to the respective matlab features and implements our `FRL` approaches.

For some parameters in the proposed algorithm, such as the current number of VM instances or workload, we need to call the OpenStack API. For example, the command `nova list` shows a list of running instances. The API is a RESTful interface, which allows us to send URL requests to the service manager to execute commands. Due to the unavailability of direct access to the OpenStack API inside of the `ctrlsrv` machine, we used the popular command line utility `cURL` to interact with a couple of OpenStack APIs. `cURL` lets us transmit and receive HTTP requests and responses from the command line or a shell script, which enabled us to work with the OpenStack API directly. In order to use an OpenStack service, we needed authentication. For some OpenStack APIs, it is necessary to send additional data, like the authentication key, in a header request. In Figure 9, the process of using `cURL` to call OpenStack APIs is shown. The first step is to send a request authentication token by passing credentials (username and password) from OpenStack Identity service. After receiving `Auth-Token` from the Keystone component, the user can combine the authentication token and Computing Service API Endpoint to send an HTTP request and receive the output. We used this process inside the `ctrlsrv` machine to execute OpenStack APIs and collect required outputs.

By combining these settings, we are able to run both `FRL` approaches, i.e., `FQL` and `FSL`, as the manager and controller of auto-scaling process in OpenStack.

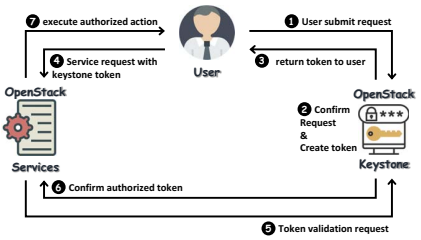


Fig. 9. `cURL` process of calling OpenStack API

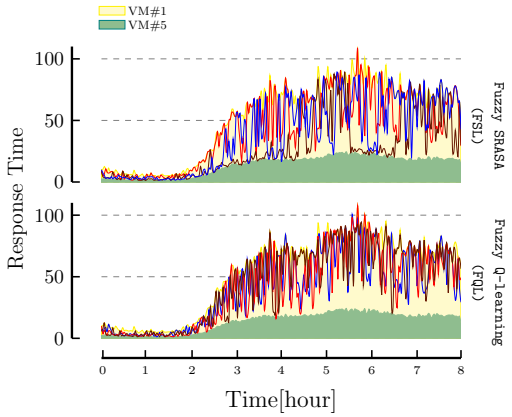


Fig. 14. The observed end-to-end response time for Wikipedia workload

VI. Experimental Comparison

The experimental evaluation aims to show the effectiveness of two proposed approaches `FQL` and `FSL`, but also to look at differences. Furthermore, the cost improvement by proposed approaches for cloud provider is demonstrated.

A. Experimental setup and benchmark

In our experiment, the two proposed approaches `FQL` and `FSL` were implemented as full working systems and were tested in the OpenStack platform. As the required parameters, the maximum and minimum number of VMs that were allowed to be available at same time were set to 5 and 1, respectively. Here, we considered low number of VMs to demonstrate

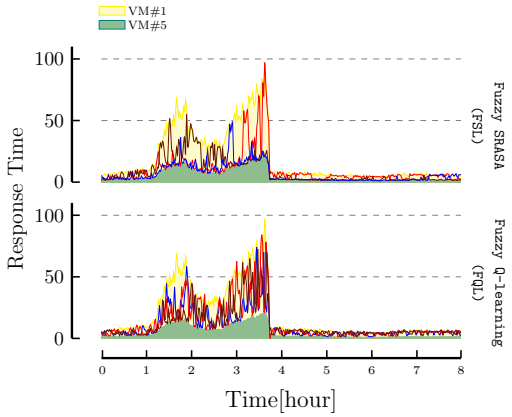
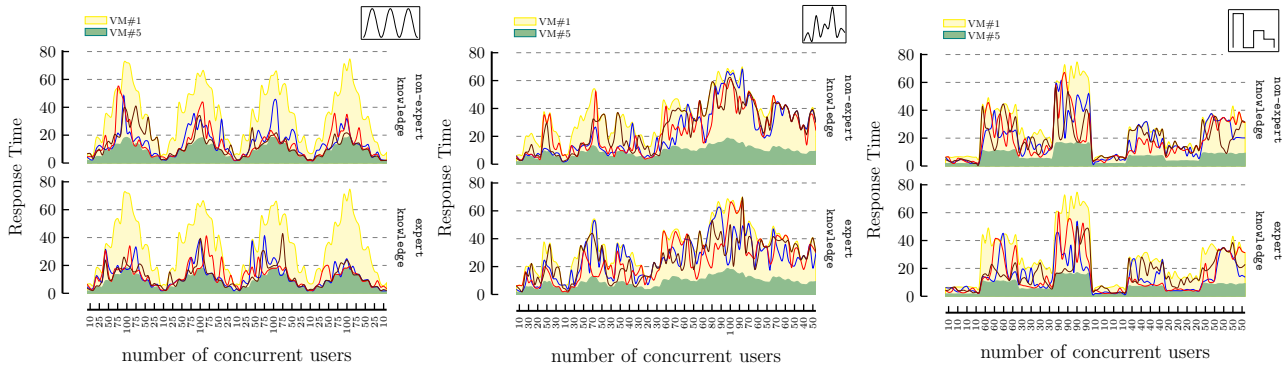
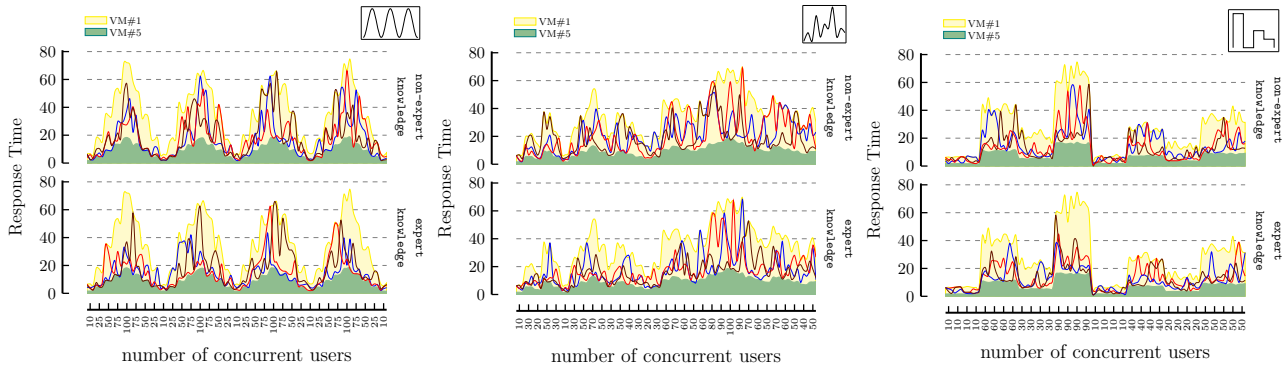


Fig. 15. The observed end-to-end response time for FIFA'98 workload



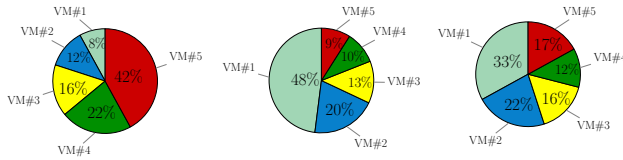
(a) Predictable Bursting pattern (b) variations pattern (c) ON&OFF pattern

Fig. 10. The observed end-to-end response time of FSL



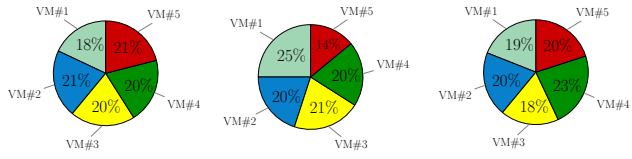
(a) Predictable Bursting pattern (b) variations pattern (c) ON&OFF pattern

Fig. 11. The observed end-to-end response time of FQL



(a) Predictable Bursting pattern (b) variations pattern (c) ON&OFF pattern

Fig. 12. Percentage number of VMs used by FSL



(a) Predictable Bursting pattern (b) variations pattern (c) ON&OFF pattern

Fig. 13. Percentage number of VMs used by FQL

the effectiveness of our proposed approaches under heavy load user request traffic. However, larger VM number can be applied for these parameters. The term workload refers to the number of concurrent user request arrivals in given time. Workload is defined as the sequence of users accessing the target application that needs to be handled by the auto-scaler.

Application workload patterns can be categorized in three representative patterns [19]: (a) the *Predictable Bursting* pattern indicates the type of workload that is subject to periodic peaks and valleys typical for services with seasonality trends or high performance computing, (b) the *Variations* pattern reflects applications such as News&Media, event registration or rapid fire sales, and (c) the *ON&OFF* pattern reflects applications such as analytics, bank/tax agencies and test environments. In all cases, we considered 10 and 100 as minimum and maximum number of concurrent users per second.

Additionally, we validated our approaches with real user request traces of the Wikipedia[5] and the FIFA WorldCup[6] websites, which are the number of requests/users accessing these two websites per unit time. We used `Siege`[7], a HTTP load testing and benchmarking utility, as our performance measuring tools. It can generate concurrent user requests, and measure the performance metric such as average response time. For each concurrent user number N , we generate N requests per second by `Siege` for 10 minutes.

[5]Wikipedia Access Traces : http://www.wikibench.eu/?page_id=60
[6]FIFA98 View Statistics : http://ita.ee.lbl.gov/html/contrib/WorldCup.html
[7]https://www.joedog.org/siege-home/

For fuzzy controller parameters, the learning rate is set to a constant value $\eta = 0.1$ and the discount factor is set to $\gamma = 0.8$. Here, we considered lower value for η, thus giving more impact on old rewards with every update. After sufficient epochs of learning, we decrease the exploration rate (ϵ) until a minimum value is reached, which here is 0.2. FRL approaches start with an exploration phase and after the first learning convergence occurs, they enter the balanced exploration-exploitation phase.

Additionally, we compared the two proposed approaches with a base-line strategy. The results of comparing with fixed numbers of VMs equal to a minimum and maximum permitted value are also shown as based-line (benchmark) approaches, named $VM\#1$ and $VM\#5$, reflecting under- and over-provisioning strategies.

Furthermore, in order to investigate the effects of initialized knowledge, we considered two types of fuzzy inference system (FIS) as the primary knowledge for fuzzy controller, *expert* and *not-expert* knowledge.

B. Comparison of effectiveness

Figures 10 and 11 show the fluctuation of the observed end-to-end response time for three type of workload patterns obtained by two approaches FSL and FQL, respectively. In order to investigate the behaviour of the auto-scaler, we considered two types of initialized knowledge (expert and non-expert) and each algorithm FQL and FSL was executed several times and represented by a different color in presented figures.

During the test, workloads were dynamically changed. Depending on incoming workload (the concurrent input requests submitted by individual users) and the number of available VMs, corresponding response timed varied between upper or lower bound. Both FQL and FSL algorithms with adaptive policies continuously monitored these fluctuation of the response time and identified workload changes. The scaling decisions were applied accordingly as recommended by the proposed algorithms. In our experiment, the up/down scaling process can be completed in a few seconds, due to simplicity and fast booting of Cirros image.

We compared FQL and FSL with $VM\#1$ and $VM\#5$ as the base-line approaches, which have a fixed number of VMs during the test. Figures 10 and 11 show that the proposed auto-scalers are able to dynamically set the number of required resources to the current workload, providing only resource allocations that are needed to meet the user's QoS expectations. As seen from Figures 10 and 11, both algorithms FQL and FSL adapt themself to input workload in order to meet SLA parameters, which here is the response time.

The difference in the algorithms can be seen from the quality of the solution, i.e., the scaling value. Both algorithms represent dynamic resource provisioning to satisfy upcoming resource demand. However:

1) For the *Predictable Burst* workload pattern (Figures 10(a) and 11(a)), FSL finds a significantly better solution compared to FQL. The reason can be explained by the speed of convergence for each RL approach. Q-learning does not learn the same policy as it follows which consequences that it learns slower. This means that although the learning improves the approximation of the optimal policy, it does not necessarily improve the policy which is actually being followed. On the other hand, *on-policy* learning used in by FSL learns faster and enters the balanced exploration-exploitation phase, i.e., completes learning phase quickly and reaches a minimum exploration rate (ϵ) that avoids more exploration in the action selection step.
2) As a result of the performance improvement achieved by SARSA, FSL has a tendency to get more VMs launched to obtain a good solution which can be realized by comparing the percentage number of VMs used by these two algorithms (Figure 12(a) and Figure 13(a)).
3) For the *Variations* workload pattern, FQL is superior to the solution found by FSL approach. Due to faster learning of the *on-policy* approach used in FSL alongside high fluctuation and non-periodic behaviour of this pattern, the non-explorative policy used after the learning phase is not optimized for the these workloads. For the *ON&OFF* (Figures 10(c) and 11(c)) workload patterns, the value of the solution is more and less similar.

The effectiveness of having expert (optimal) knowledge can be figured out by comparison between the two types of initial knowledge used for the experiment. In all presented cases, the good initial knowledge significantly improves the quality of results compared to non-expert (sub-optimal) knowledge.

In addition, to validate the applicability of approaches against real-life situations, we used two real workloads: the Wikipedia workload and the FIFA WorldCup Website access logs. While the Wikipedia workload shows a steady and predictable trend, the FIFA workload has a bursty and an unpredictable pattern. For the Wikipedia trace in figure 14, FSL shows slightly better performance compared to FQL. For the FIFA results shown in Figure 15, the situation is different. FSL as an *on-policy* approach behaves better in terms of the measured response time, while FQL is still in exploration/exploitation phase.

C. Comparison of cost-effectiveness of scaling

Figures 12 and 13 show percentage numbers of used VMs for all workload patterns. The approaches work on the current workload and relative response time of the system at the current time, increasing the number of available VMs (scale-up) and decreasing the number of idle VMs (scale-down). Both FQL and FSL conduct distributed-case scaling and allocate suitable numbers of VMs according to the workload.

For different types of workload patterns, the average maximum number of VMs used during our experiment by FQL and FSL algorithms are 18.3% and 22.6%, respectively. This implies our approaches can meet the QoS requirements using a smaller amount of resources, which is an improvement on resource utilisation for applications in terms of hosting VMs. Thus, the FQL and FSL approaches can perform auto-scaling of application as well as save cloud provider cost by increasing resource utilisation.

VII. Conclusion

We investigated horizontal scaling of cloud applications. Many commercial solutions use simple approaches such as threshold-based ones. However, providing good thresholds for auto-scaling is challenging. Recently, machine learning approaches have been used to complement and even replace expert knowledge to design self-adaptable solutions to capable to react to unpredictable workload fluctuations.

We proposed a fuzzy rule-based system, based on which we compared two well-know RL approaches, resulting in Fuzzy Q-learning (`FQL`) and Fuzzy SARSA learning (`FSL`). Both approaches can efficiently scale up/down cloud resources to meet the given QoS requirements while reducing cloud provider costs by improving resource utilisation. However, differences also emerge. In the SARSA experiment, given the reward at each time step improves the quality of solutions for periodic workload pattern. Both algorithms have been implemented in OpenStack, an open-source IaaS platform, to demonstrate the practical effectiveness of proposed approach has been successfully tested and presented and the validity of the comparison results are established.

In conclusion, this paper identifies the promising auto-scaling concepts for cloud computing: (i) developing an autonomic and complete auto-scaler for a cloud platform system by combining of techniques such as a fuzzy logic system and reinforcement learning to provide optimal resource management approach tailored to different types of workload pattern, and (ii) defining the concept of a complex auto-scaler, that can replace traditional threshold-based ones, (iii) implement the proposed auto-scaler in an open-source cloud platform and presenting results for different type of workloads.

We have demonstrated the overall suitability of the different types of on-policy and off-policy RL approaches for auto-scaling, but also differences for specific workload patterns and converging times. We plan to extend our approach in a number of ways: (i) extending `FQL4KE` to perform in environments which are partially observable, (ii) exploiting clustering approaches to learn the membership functions of the antecedents (in this work we assume they do not change once they specified, for enabling the dynamic change we will consider incremental clustering approaches) in fuzzy rules and (iii) look at other resource types such as containers [22].

VIII. Acknowledgement

This work was partly supported by IC4 (Irish Centre for Cloud Computing and Commerce), funded by EI and the IDA.

References

[1] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Efficient provisioning of bursty scientific workloads on the cloud using adaptive elasticity control. In *Workshop on Scientific Cloud Computing Date*, 2012.

[2] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS)*, pages 204–212. IEEE, 2012.

[3] H. Arabnejad, P. Jamshidi, G. Estrada, N. El Ioini, and C. Pahl. An auto-scaling cloud controller using fuzzy q-learning - implementation in openstack. In *European Conf on Service-Oriented and Cloud Computing ESOCC 2016*, pages 152–167, 2016.

[4] T.C. Chieu, A. Mohindra, A.A. Karve, and A. Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. In *IEEE Intl Conf on e-Business Engineering*, pages 281–286, 2009.

[5] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *International Conference on Autonomic and Autonomous Systems*, pages 67–74, 2011.

[6] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck. From data center resource allocation to control theory and back. In *IEEE 3rd International Conference on Cloud Computing*, pages 410–417, 2010.

[7] D. Fang, X. Liu, I. Romdhani, P. Jamshidi, and C. Pahl. An agility-oriented and fuzziness-embedded semantic model for collaborative cloud service search, retrieval and recommendation. *Future Generation Computer Systems*, 56:11 – 26, 2016.

[8] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[9] R. Han, L. Guo, M.M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *12th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 644–651, 2012.

[10] M.Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S.L.D. Gudreddi. Integrated and autonomic cloud resource scaling. In *Network Operations and Management Symposium (NOMS)*, pages 1327–1334, 2012.

[11] J. Huang, C. Li, and J. Yu. Resource prediction based on double exponential smoothing in cloud computing. In *Intl Conf on Consumer Electronics, Communications and Networks*, pages 2056–2060, 2012.

[12] M.C. Huebscher and J.A. McCann. A survey of autonomic computing: Degrees, models, and applications. *ACM Comp Surveys*, 40(3):7, 2008.

[13] S. Islam, J. Keung, K. Lee, and A. Liu. Empirical prediction for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.

[14] P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and self-Managing Systems (SEAMS)*, pages 95–104, 2014.

[15] P. Jamshidi, C. Pahl, and N. C. Mendona. Managing uncertainty in autonomic cloud elasticity controllers. *IEEE Cloud Computing*, 3(3):50–60, 2016.

[16] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA)*, 2016.

[17] P. Jamshidi, A.M. Sharifloo, C. Pahl, A. Metzger, and G. Estrada. Self-learning cloud controllers: Fuzzy q-learning for knowledge evolution. In *International Conference on Cloud and Autonomic Computing*, pages 208–211, 2015.

[18] E. Kalyvianaki, TheT.mistoklis Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th international conference on Autonomic computing*, pages 117–126. ACM, 2009.

[19] T. Lorido-Botran, J. Miguel-Alonso, and J. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[20] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan. Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 514–521. IEEE, 2010.

[21] P. Padala, K.-Y. Hou, K.G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *ACM Europ Conf on Computer systems*, pages 13–26, 2009.

[22] C. Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

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

[24] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[25] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *IEEE Intl Conference on Autonomic Computing*, pages 65–73, 2006.

[26] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *ACM Intl Symp on High Performance Distributed Computing*, pages 304–307, 2010.