Fuzzy Self-Learning Controllers for Elasticity Management in Dynamic Cloud Architectures

Pooyan Jamshidi*, Amir Sharifloo‡, Claus Pahl§, Hamid Arabnejad †, Andreas Metzger‡, Giovani Estrada ¶
*Imperial College London, UK
†IC4, Dublin City University, Ireland
‡University of Duisburg-Essen, Germany
§University of Bozen-Bolzano, Italy
¶Intel, Ireland

***Abstract*—Cloud controllers support the operation and quality management of dynamic cloud architectures by automatically scaling the compute resources to meet performance guarantees and minimize resource costs. Existing cloud controllers often resort to scaling strategies that are codified as a set of architecture adaptation rules. However, for a cloud provider, deployed application architectures are black-boxes, making it difficult at design time to define optimal or pre-emptive adaptation rules. Thus, the burden of taking adaptation decisions often is delegated to the cloud application. We propose the dynamic learning of adaptation rules for deployed application architectures in the cloud. We introduce FQL4KE, a self-learning fuzzy controller that learns and modifies fuzzy rules at runtime. The benefit is that we do not have to rely solely on precise design-time knowledge, which may be difficult to acquire. FQL4KE empowers users to configure cloud controllers by simply adjusting weights representing priorities for architecture quality instead of defining complex rules. FQL4KE has been experimentally validated using the cloud application framework ElasticBench in Azure and OpenStack. The experimental results demonstrate that FQL4KE outperforms both a fuzzy controller without learning and the native Azure auto-scaling.**

***Keywords*: Cloud Architectures; Fuzzy Control; Self-adaptive Systems; Self-learning; Q-Learning; Machine Learning.**

I. Introduction

The dynamic quality management of deployed architectures in the cloud, specifically the acquisition and release of resources is a challenge due to the uncertainty introduced by workload, cost and other quality requirements. In order to address this challenge, *auto-scaling* [21], [22] has been proposed. Current solutions typically rely on threshold-based rules, offered by several commercial cloud providers/platforms such as Amazon EC2, Microsoft Azure and OpenStack. Best practice is to define a comprehensible set of scaling rules, assuming a linear and constant dependency between resource assignments and performance improvements, while in Internet scale applications, the complexity of application architecture, the interferences among components and the frequency by which hardware and software failure happen typically invalidate the assumptions [11], calling for new approaches [23], [8].

Alternative approaches have been investigated to dynamically manage the quality of application architectures deployed in the cloud, e.g., based on classical control theory and on knowledge-based controllers and thus suffer from similar limitations [10]. Traditional capacity planning approaches based on queuing theory [26] do not fully address the dynamics of cloud application architectures due to over-simplifications and/or their static nature since the models are complex to be evolved at runtime, often resort to parameter tunings. Recent self-organizing controllers have shown to be a better fit for the complexity of cloud controllers [11]. However, a practical challenge for rule-based commercial approaches is the reliance on users for defining adaptation and controllers. First, from the cloud provider's perspective, details of the application architecture are often not visible, therefore, defining optimal scaling rules are difficult. Thus, the burden of determining these falls on the application developers, who do not have enough knowledge about workloads, infrastructure or performance modeling. Our aim is to design a controller that does not depend on the user-defined rules.

A. Research Challenges

In [18], we exploited fuzzy logic to facilitate user intuitive knowledge elicitation. The key strength of fuzzy logic is the ability to translate human knowledge into a set of intuitive rules. During the design process of a fuzzy controller, a set of `IF-THEN` rules must be defined. Although users are comfortable with defining auto-scaling rules using fuzzy linguistic variables [18], the rules have to be defined at design-time leading to the following issues: (i) Knowledge for defining such rules may not be available; (ii) Knowledge may be available but in part (partial rules); (iii) Knowledge is not always optimal (user can specify the rules but they are not effective, e.g., redundant rules); (iv) Knowledge may be precise for some rules but may be less precise for some other rules. (v) Knowledge may need to change at runtime (rules may be precise at design-time but may drift at runtime). As a result, user defined rules may lead to sub-optimal scaling decisions and loss of money for cloud application providers.

B. Research Contributions

Here, we develop an online learning mechanism, FQL4KE, to adjust and improve auto-scaling policies at runtime. We combine fuzzy control and Fuzzy Q-Learning (FQL) [15] to connect human expertise to a continuous evolution machinery. The combination of fuzzy control and the Fuzzy Q-Learning proposes a powerful self-adaptive mechanism where the fuzzy

control facilitates the reasoning at a higher level of abstraction and the Q-learning allows to adjust the controller.

The main *contributions* of this work are as follows: (i) a self-learning fuzzy controller, FQL4KE, for dynamic resource allocations. (ii) a tool, ElasticBench, as a realization and a means for experimental evaluations. The main implication of this contribution is that we do not need to rely on the knowledge provided by the users anymore, FQL4KE can start adjusting application resources with no a priori knowledge.

The paper is organized as follows. Section II motivates and introduces core concepts. Section III describes the mechanisms solution, followed by a realization in Section IV. Section V discusses experimental results, followed by implications and limitations in VI. Finally, Section VII discusses the related work and Section VIII concludes the paper.

II. Motivation and Background

A. Motivation

Dynamic resource provisioning (auto-scaling) is an online decision making problem. *Cloud controllers* that realize auto-scaling observe the resource consumption of applications and manipulate the provisioning plans to manage architecture quality. Computing resources are allocated to applications by monitoring workload, w, user requests over time and current performance, rt, as average response time of the application. The cloud controller decides to allocate or remove resources in order to keep the performance rt at a desired level $rt_{desired}$ while minimizing costs.

There are characteristics that often challenge existing auto-scaling techniques: (i) the environment is non-episodic, i.e., current choices will affect future actions; (ii) cloud infrastructures are complex and difficult to model; (iii) workloads are irregular and dynamic. These characteristics of the environment in which cloud controller operates require to solve sequential decision problems, where previous actions in specific states affect future ones. The common solution for this problem is to create a *plan*, *policy* or *strategy* to act upon. We use the term *policy* as the knowledge inside cloud controllers that we aim to learn at runtime. As a result, policies determine the decisions that controllers produce for different situations (i.e., the state in which the cloud application operates).

B. Reinforcement Learning for Elasticity Decision Making

In the reinforcement learning context, an agent takes action a_i when the system is in state s_t and leaves the system to evolve to the next state s_{t+1} and observes the reinforcement signal r_{t+1}. Decision making in elastic systems can be represented as an interaction between cloud controllers and environment. The cloud controller monitors the current state of the system through its sensors. Based on some knowledge, it chooses an action and evaluates feedback reward in the form of *utility functions* [27]. *Situation* allows the system to know when it must monitor the state, and also when it must take the action (i.e, triggers the scaling action). An elastic system may stay in the same state, but should take different actions in different situations and workload intensity.

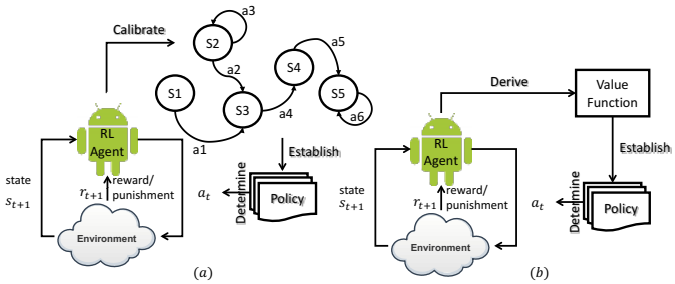

Fig. 1: Model-based vs. model free RL.

To derive an action, the agent uses a $policy$ that aims to increase the future rewards. A $model$ of the environment assists decision making (Figure 1(a)); however, it is not always feasible to have such a model available. Model-free reinforcement learning (hereafter *RL*) techniques have been developed to address this issue, which are relevant for cloud computing problems due to the lack of environmental models. We use the Q-Learning algorithm as a model-free approach that computes the optimal policy with regard to both immediate and delayed rewards. A cloud controller learns a value function (Figure 1(b)) that gives the *consequent* of applying different policies.

III. Fuzzy Q-Learning for Knowledge Evolution

This section presents our solution FQL4KE. By combining fuzzy logic and Q-Learning, FQL4KE deals with uncertainty caused by the incomplete *knowledge*. Expert knowledge, if available, is encoded in terms of rules. The fuzzy rules are continually tuned through learning from the data collected at runtime. In case there is no (or limited) knowledge available at design-time, FQL4KE is still able to operate.

A. FQL4KE Building Blocks

Figure 2 illustrates the main building blocks of FQL4KE. While the application runs on a cloud platform, FQL4KE guides resource provisioning. More precisely, FQL4KE follows the autonomic MAPE-K loop [19], where different characteristics of the application (e.g. workload and response time) are continuously monitored, the satisfaction of system goals are checked and accordingly the resource allocation is adapted in case of deviation from goals. The goals (i.e., SLA, cost, response time) are reflected in the reward function as we will define this in Section III-D.

The monitoring component collects low-level performance metrics and feed both cloud controller as well as the knowledge learning component. The actuator issues adaptation commands from the controller at each control interval to the underlying cloud platform. The cloud controller is a fuzzy controller that takes the observed data, and generates scaling actions. The learning component continuously updates the knowledge base of the controller by learning appropriate rules. These two components are described in Sections III-B and III-C respectively. Finally, the integration of these two components is discussed in Section III-D.

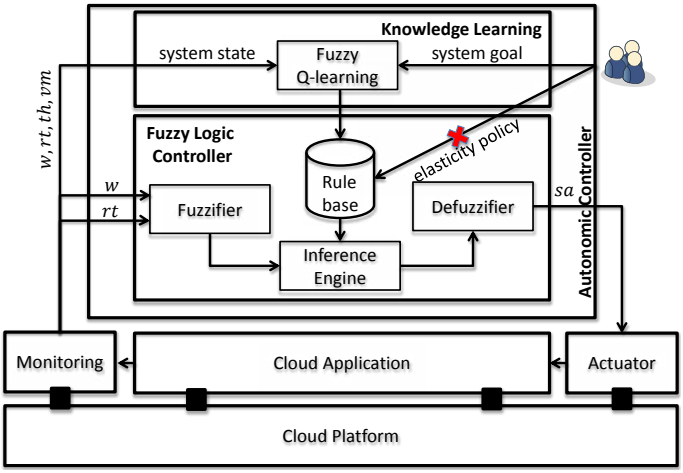

Fig. 2: FQL4KE (logical) architecture.

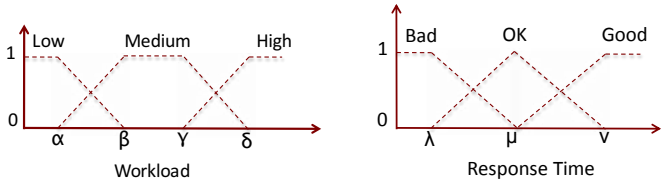

Fig. 3: Fuzzy membership functions for auto-scaling variables.

B. Fuzzy Logic Controller

Fuzzy inference is the process of mapping a set of control inputs to a set of control outputs through fuzzy rules. The inputs to the controller are the workload (w) and response time (rt) and the output is the scaling action (sa) in terms of increment (or decrement) in the number of virtual machines (VMs). The design of a fuzzy controller, in general, involves the following tasks: 1) defining the fuzzy sets and membership functions of the input signals. 2) defining the rule base which determines the behavior of the controller in terms of control actions using the linguistic variables defined in the previous task. The very first step in the design process is to partition the state space of each input variable into various fuzzy sets through membership functions. Each fuzzy set associated with a linguistic term such as "low" or "high". The membership function, denoted by $\mu_y(x)$, quantifies the degree of membership of an input signal x to the fuzzy set y (cf. Figure 3). In this work, the membership functions, depicted in Figure 3., are considered to be both triangular and trapezoidal based on our previous results in [18]. As shown, 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 to reduce the set of rules in the knowledge base.

The next step consists of defining the inference machinery for the controller. Here we need to define elasticity policies in terms of rules: "`IF` (w is $high$) `AND` (rt is bad) `THEN` ($sa = +2$)", where the output function is a constant value that can be an integer in $\{-2, -1, 0, +1, +2\}$, which is associated to the change in the number of deployed nodes. Note that this set can be any finite set but here for simplicity we constraint it to only 5 possible actions, but depending on the problem at hand can be any finite discrete set of actions. In this work, no a priori knowledge for defining such rules is assumed. In particular, FQL4KE attempts to find the consequent Y for the rules, see Section III-C.

Once the fuzzy controller is designed, the execution of the controller is comprised of three steps (cf. middle part of Figure 2): (i) fuzzification of the inputs, (ii) fuzzy reasoning, and (iii) defuzzification of the output. Fuzzifier projects the crisp data onto fuzzy information using membership functions. Fuzzy engine reasons on information based on a set of fuzzy rules and derives fuzzy actions. Defuzzifier reverts the results back to crisp mode and activates an adaptation action. The output is calculated as a weighted average:

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

where N is the number of rules, $\mu_i(x)$ is the firing degree of the rule i for the input signal x and a_i is the consequent function for the same rule. Then the output is rounded to the nearest integer, due to the discrete nature of scaling actions. Finally, this value, if endorsed by policy enforcer module (see Section IV), will be enacted by issuing appropriate commands to the underlying cloud platform fabric.

C. Fuzzy Q-Learning

Until this stage, we have shown how to design a fuzzy controller for auto-scaling a cloud-based application where the elasticity policies are provided by users at design-time, like RobusT2Scale [18]. In this section, we introduce a mechanism to learn the policies at runtime, enabling knowledge evolution (i.e., KE in FQL4KE). As the controller has to take an action in each control loop, it should try to select those actions taken in the past which produced good rewards. Here by reward we mean "long-term cumulative" reward:

$$R_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2}$$

where γ is the discount rate determining the relative importance of future rewards. There exists a trade-off (cf. step 2 in Algorithm 1) between the actions that have already tried (known as exploitation) and new actions that may lead to better rewards in the future (known as exploration).

In each control loop, the controller needs to take an action based on $Q(s, a)$, which is the expected cumulative reward that can be received by taking action a in state s. This value directly depends on the policy followed by the controller, thus determining the behavior of the controller. This policy $\pi(s, a)$ is the probability of taking action a from state s. As a result, the value of taking action a in state s following the policy π

Algorithm 1 : Fuzzy Q-Learning

Require: γ, η, ϵ

1: Initialize q-values:
$q[i,j] = 0,\ 1 < i < N\ ,\ 1 < j < J$
2: Select an action for each fired rule:
$a_i = argmax_k q[i,k]$ with probability $1-\epsilon$ ▷ **Eq. 5**
$a_i = random\{a_k, k = 1, 2, \cdots, J\}$ with probability ϵ
3: Calculate the control action by the fuzzy controller:
$a = \sum_{i=1}^{N} \mu_i(x) \times a_i,$ ▷ **Eq. 1**
where $\alpha_i(s)$ is the firing level of the rule i
4: Approximate the Q function from the current q-values and the firing level of the rules:
$Q(s(t), a) = \sum_{i=1}^{N} \alpha_i(s) \times q[i, a_i],$
where $Q(s(t), a)$ is the value of the Q function for the state current state $s(t)$ in iteration t and the action a
5: Take action a and let system goes to the next state $s(t+1)$.
6: Observe the reinforcement signal, $r(t+1)$ and compute the value for the new state:
$V(s(t+1)) = \sum_{i=1}^{N} \alpha_i(s(t+1)).max_k(q[i, q_k])$.
7: Calculate the error signal:
$\Delta Q = r(t+1) + \gamma \times V_t(s(t+1)) - Q(s(t), a),$ ▷ **Eq. 4**
where γ is a discount factor
8: Update q-values:
$q[i, a_i] = q[i, a_i] + \eta \cdot \Delta Q \cdot \alpha_i(s(t)),$ ▷ **Eq. 4**
where η is a learning rate
9: Repeat the process for the new state until it converges

is formally defined as:

$$Q^{\pi}(s,a) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\}, \tag{3}$$

where $E_{\pi}\{.\}$ is the expectation function under policy π. When an appropriate policy is found, the RL problem at hand is solved. Q-learning is a technique that does not require any specific policy in order to evaluate $Q(s,a)$, therefore:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \eta[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)], \tag{4}$$

where γ is the learning rate. In this case, the policy adaptation can be achieved by selecting a random action with probability ϵ and an action that maximizes the Q function in the current state with probability $1-\epsilon$, note that the value of ϵ is determined by the exploitation/exploration strategy (cf. V-A):

$$a(s) = arg\max_k Q(s,k) \tag{5}$$

The fuzzy Q-learning algorithm is summarized in Algorithm 1. In the case of our running example, the state space is finite (i.e., 9 states as the full combination of 3×3 membership functions for fuzzy variables w and rt) and our controller has to choose a scaling action among 5 possible actions $\{-2, -1, 0, +1, +2\}$. However, the design methodology that we describe is general. Note that the convergence is detected when the change in the consequent functions is negligible in each learning loop.

D. FQL4KE for Dynamic Resource Allocation

In this work, for simplicity, only one instance of the fuzzy controller is integrated. Note that in the case of multiple controllers is also possible but due to the intricacies of updating a central Q table, we consider this as a natural future extension of this work for the problem areas that requires coordination between several controllers, see [9].

Reward function. The controller receives the current values of w and rt that correspond to the state of the system, $s(t)$ (cf. Step 4 in Algorithm 1). The control signal sa represents the action a that the controller take at each loop. We define the reward signal $r(t)$ based on three criteria: (i) numbers of the desired response time violations, (ii) the amount of resource acquired, and (iii) throughput, as follows:

$$r(t) = U(t) - U(t-1), \tag{6}$$

where $U(t)$ is the utility value of the system at time t. Hence, if a controlling action leads to an increased utility, it means that the action is appropriate. Otherwise, if the reward is close to zero, it implies that the action is not effective. A negative reward (punishment) warns that the situation becomes worse after taking the action. The utility function is defined as:

$$U(t) = w_1 \cdot \frac{th(t)}{th_{max}} + w_2 \cdot (1 - \frac{vm(t)}{vm_{max}}) + w_3 \cdot (1 - H(t)) \tag{7}$$

$$H(t) = \begin{cases} \frac{(rt(t) - rt_{des})}{rt_{des}} & rt_{des} \leq rt(t) \leq 2 \cdot rt_{des} \\ 1 & rt(t) \geq 2 \cdot rt_{des} \\ 0 & rt(t) \leq rt_{des} \end{cases}$$

where $th(t)$, $vm(t)$ and $rt(t)$ are throughput, number of worker roles and response time of the system, respectively. w_1,w_2 and w_3 are their corresponding weights determining their relative importance in the utility function. In order to aggregate the individual criteria, we normalized them depending on whether they should be maximized or minimized.

Knowledge base update. FQL4KE starts with controlling the allocation of resources with no a priori knowledge. After enough *explorations*, the consequents of the rules can be determined by selecting actions that correspond to the *highest* q-value in each row of the Q-table. Although FQL4KE does not rely on design-time knowledge, if even partial knowledge is available (i.e., operator of the system is confident with providing some of the elasticity policies) or there exists data regarding performance of the application, FQL4KE can exploit such knowledge by initializing q-values (cf. step 1 in Algorithm 1) This implies a quicker learning convergence.

IV. Implementation

We implemented prototypes of FQL4KE on Microsoft Azure and OpenStack. As illustrated in Figure 4, the Azure prototype comprises of 3 components integrated according to Figure 6:

i A learning component FQL implemented in Matlab. [1]

[1] code is available at https://github.com/pooyanjamshidi/Fuzzy-Q-Learning

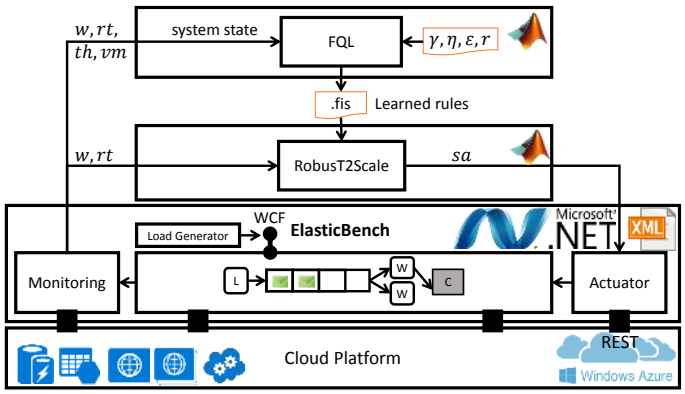

Fig. 4: FQL4KE implementation architecture.

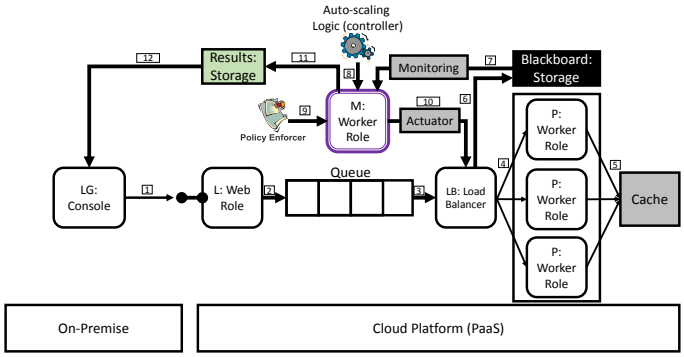

Fig. 5: ElasticBench: the experimental platform.

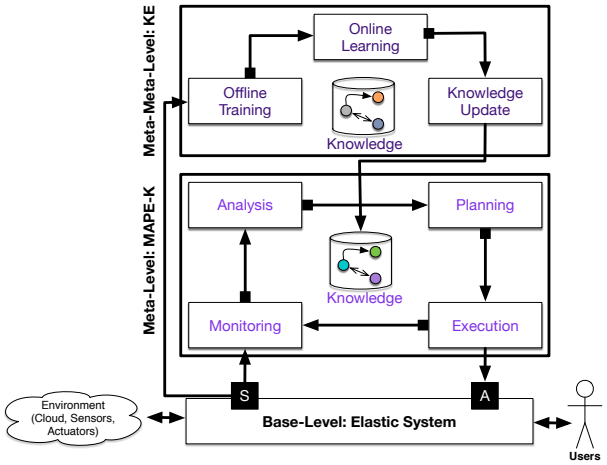

Fig. 6: Augmenting MAPE-K with online learning.

ii A cloud controller reasoning engine (RobusT2Scale [18]) implemented in Matlab. [2]

iii A cloud-based application framework (ElasticBench) implemented with Microsoft .NET technologies (.NET framework 4 and Azure SDK 2.5). [3]

iv The integration between these three components by software connectors (cf. Figure 5) developed in .NET.

A. *ElasticBench: implementation on Azure with .NET*

ElasticBench (cf. Figure 5) includes a workload generator that simulates different workload patterns to test and train the controller before execution. It also provides functionalities to perform a variety of auto-scaling experiments, therefore can be treated as a benchmark for auto-scaling research. In order to build a generic workload generator, we developed a service to generate Fibonacci numbers. A delay is embedded in the process of calculating Fibonacci numbers to simulate a process that takes a reasonably long period. Note that calculating Fibonacci numbers is an $O(N)$ task, making it an appropriate candidate for demonstrating different application types.

Two types of Azure services are used to implement ElasticBench: *web role* and *worker role*. Web and worker roles correspond to VMs at infrastructure level. The requests issued from the load generator are received by the web role, which puts a message on a task assignment queue. The worker instances continuously checks this queue and a background process (to calculate Fibonacci numbers) will be started. The worker roles communicate with a cache to acquire the data for processing (e.g., previously calculated Fibonacci numbers).

We implemented two types of worker role: P process the messages (i.e., calculate Fibonacci numbers), whereas the other type M implements the MAPE-K feedback control loop. The main functionalities in M worker role is as follows: (1) It reads performance metrics from the blackboard storage; (2) It calculates metrics for feeding the fuzzy controller; (3) It also implements a policy enforcer to check whether the number of nodes to be enacted is within the predefined range and whether the worker role is in a stable mode. (4) It is possible to plug-in other cloud controllers (i.e., controllers implementing other techniques) with few lines of code; (5) It also implements mechanisms comprising the resiliency of this worker role.

The design decision we made for implementing the MAPE-K functionalities inside a worker role in the cloud was strategic. In one hand, in order to avoid network latencies for decision enaction, we required an internal and isolated network between the decision maker module (i.e., M) and the scaling roles (i.e., P). On the other hand, we needed to provide a close design to the real world setting as it is the case for commercial solutions in public clouds that the auto-scaling controller sits near the scaling layer as opposed to be deployed on premise.

B. *Implementation on OpenStack with Python*

We also implemented FQL4KE on OpenStack. OpenStack is an open source cloud platform which controls large pools of compute, storage and networking resources, all managed through a dashboard called Horizon or via the OpenStack API. The main components of the OpenStack we use are:

- Nova: the engine to manage the resource life-cycle
- Swift: a storage system responsible for objects and files.
- Cinder: a block storage component, like Amazon EBS.

[2]code is available at https://github.com/pooyanjamshidi/RobusT2Scale
[3]code is available at https://github.com/pooyanjamshidi/ElasticBench

- Neutron: manages the networking.
- Keystone: the primary tools for user authentication.
- Glance: provides image services.
- Horizon: provides a web-based portal for users.

The core auto-scaling policy in OpenStack is based on threshold-based rules by measures such CPU utilization. In order to implement FQL4KE as a VM manager, we configured the following resources:

- AutoScalingGroup: a resource type that is used to encapsulate the resource that we wish to scale. Also, the minimum and maximum number of instances should be defined in this resource.
- ScalingPolicy: a resource type that is used to affect a scaling process on the current VM group.
- ControllerServer: the main core that has responsibility for the auto-scaling strategy in the platform. In this VM, we locate our FQL4KE algorithm in order to control and manage the VM instances. Due to unavailability of OpenStack API inside of this VM, in order to collect data form OpenStack environment, we need to call each API in two steps: first to get a token from the Keystone component as user authentication and then use this token for sending request to the Horizon component.

To implement FQL4KE in OpenStack, we created a ControllerServer. Additionally, all resources to be scaled are encapsulated in AutoScalingGroup. The implementation is divided between two different resource types: VMs instances and ControllerServer. In our experiment, each VM is defined as Web server and FQL4KE is located inside ControllerServer VM. Each request is sent to the LoadBalancer and redirected to the target server. At each control interval, the response time of the system LoadBalancer is measured, then FQL4KE decides to scale up/down based on end-to-end response time.

V. Experimental Results

We demonstrate the *efficiency* and *effectiveness* of FQL4KE via an experimental evaluation. More specifically, the key purpose of the experiments is to answer the following questions:

RQ1. Is FQL4KE able to learn how to efficiently acquire resources for dynamic systems in cloud architectures?

RQ2. Is FQL4KE flexible enough to allow the operator to set different strategies? and how the approach is effective in terms of key elasticity criteria (cf. criteria column in Table I)?

A. Experimental Setting

The main differentiating aspect is the delay in receiving rewards after each scaling action has been taken. The agent (i.e., cloud controller) deployed in a delayed-feedback environment (i.e., cloud) comes to know the reward after a non-negative integer indicating the number of time-steps between an agent taking an scaling action and actually receiving its feedback. In each monitoring cycle, which happens every 10 seconds, the controller knows about its state but in order to receive the reinforcement signal, it has to wait for example for 8-9 minutes for "scaling out" actions and 2-3 minutes for "scaling in" actions to be enacted. Such kinds of delayed feedback environments introduce some challenges for learning convergence. We tackled this by investigating different learning strategies. As depicted in Figure 7, we considered 5 different exploitation/exploration strategies (i.e., $S1 - S5$). For instance, in $S1$, the learning process starts by a high exploration rate, i.e, $\epsilon = 1$ (cf. Step 2 in Algorithm 1). We set this in order to explore all possible actions enough times in early cycles. Once the optimal fuzzy rules are learned, the controller with updated elasticity policies will replace the current one. In other words, FQL starts with exploration phase and after a first learning convergence happened, it enters the balanced exploration-exploitation phase. However, in order to compare the performance of FQL4KE under different strategies, we consider other learning strategies as well. For instance, in $S2$, after initial learning by high exploration, we set $\epsilon = 0$ in order to fully exploit the learned knowledge.

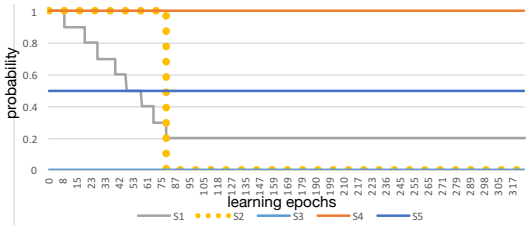

Fig. 7: The learning (exploitation/exploration) strategies.

The learning rate in the experiments are set to a constant value $\eta = 0.1$ and the discount factor is set to $\gamma = 0.8$. The minimum and maximum number of nodes is set to 1 and 7 respectively. The control interval is set to $10sec$. The worker role that our FQL4KE is deployed is *small* VM with 1 core and $1792MB$ memory while the P worker roles are *extra small* VMs with 1 CPU core and $768MB$ memory. Initially, we set Q table to zero, assuming no a priori knowledge. We set the weights in the reward function all equal, i.e., $w_1 = w_2 = w_3 = 1$ (cf. Eq. 7). The experiment time has been set to $24hours$ to monitor the performance of the system in adequate learning steps (on average due to the delay in reward observation, each step takes between $2 - 9mins$). We collected data points in each control loop (more than 8600 data points). The learning overhead is in the order of $100ms$ and the monitoring and actuation delay is about $1000ms$ (excluding enaction time).

B. FQL4KE Efficiency (RQ1)

The temporary evolution of the q-values associated to each state-action for S1 is shown (for partial set of pairs) in Figure 9. Note that the change in the q-values occurs when the corresponding rule is activated, i.e., when the system is in state $S(t)$ and takes action a_i. As the figure shows, some q-values changed to a negative value during exploration phase and this means that these actions are punished and are not appropriate to be taken in the future. The optimal consequent for each rule in the rule base is determined by the largest q-value at the end of the learning phase. For instance, action a_5 is the best consequent for rule number 9 in learning strategy S1 (cf. 5th

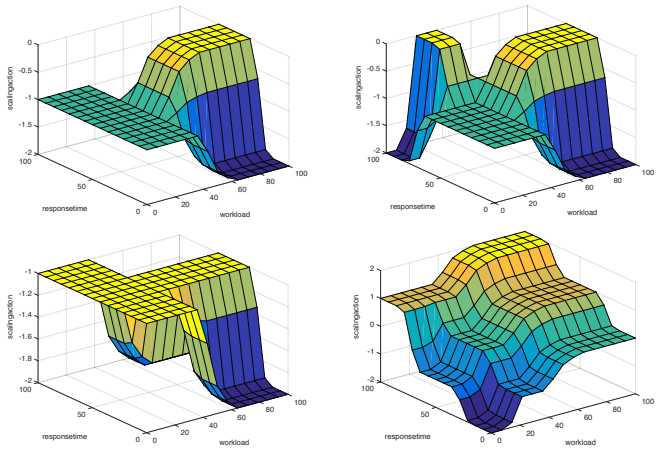

Fig. 8: Temporal evolution of control surface.

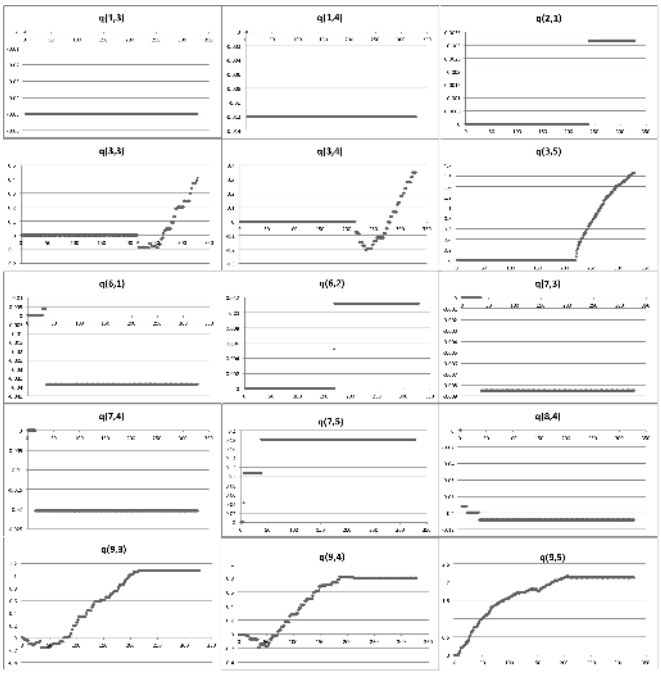

Fig. 9: Temporal evolution of q-values.

row in Figure 9). With changing q-values, the control surface of the fuzzy controller is also changing. Figure 8 shows the temporal evolution of the control surface of the fuzzy controller. The surface evolves until the learning converges.

C. FQL4KE Flexibility and Effectiveness (RQ2)

In this section, we study how the learning component of FQL4KE improves the functionality of dynamic resource allocation over static rule-based or native mechanisms. Table I summarizes the criteria that we considered for comparing different auto-scaling strategies with respect to different workload patterns. Note that S5 corresponds to the fuzzy controller with initial knowledge extracted from users at design-time (RobusT2Scale) with no learning component and the

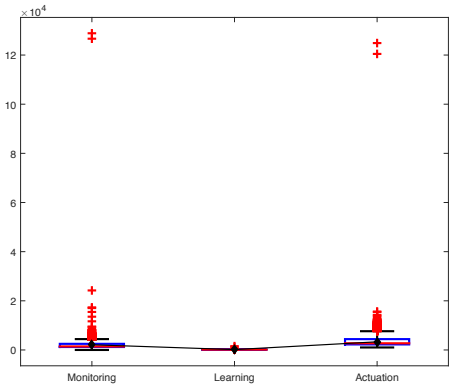

Fig. 10: Runtime delay (ms) for MAPE loop activities.

last strategy corresponds to Azure native auto-scaling. We synthetically generated 6 different workload patterns (see Figure 11) in order to provide enough environmental conditions for this comparison. The x axis shows the experimental time and the y axis shows the number (in $[0, 100]$) for which the Fibonacci series needs to be calculated, demonstrating the workload intensity similar to the number of concurrent users for a web-based application. A key parameter in learning-based approaches is the convergence delay to reach the optimal policy. The response time of the system under different workloads is also considered as another comparison criterion. The average number of VMs acquired throughout the experiment interval as well as the number of changes in the underlying resources (i.e., sum of issued scaling actions) is also considered as a comparison criterion. Figure 12 shows the changes in the number of VM instances in a trial run. The main findings described in Table I can be summarized as follows:

- Sequential decreasing of exploration factor (cf. S1) is effective in accelerating learning convergence. However, it is also effective for highly dynamic workloads such as "quickly varying" as in Figure 11 because it keeps a minimum of $\epsilon = 0.2$ when initial knowledge has been learned and it keeps learning more suitable rules when new situations arise.
- Initial high exploration (cf. S2) is effective for quick convergence, but in non-predictable workloads such as "quickly varying", the decisions become sub-optimal. This is evident by comparing the average number of VMs and the number of learning iterations until convergence for "large variation" and "quickly varying" patterns.
- Although high constant exploration (cf. S3) is effective in unpredictable environments (see response time and compare it with other strategies), it is not optimal in terms of convergence, number of changes and acquired resources. Note that the higher number of changes in the resources means that for quite considerable period in time, instability in the deployment environment of the application has been experienced.
- Maximum exploration rate (cf. S4) is not a good learning strategy by no means as it only produces random actions

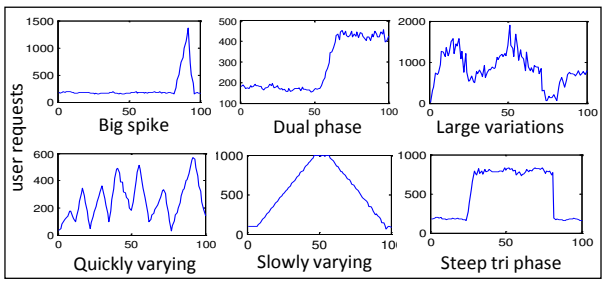

Fig. 11: Synthetic workload patterns.

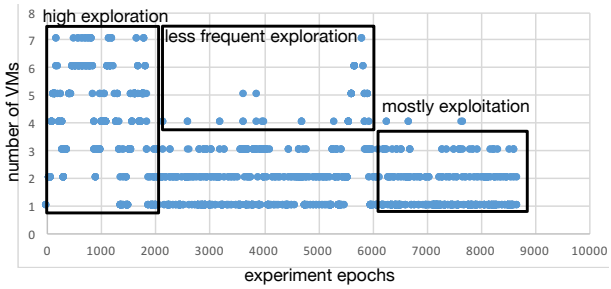

Fig. 12: Auto-scaling behavior during exploration/exploitation.

and it never converges to an optimal policy.

- The strategy S5 is equal to RobusT2Scale, representing a policy-based adaptation without any policy learning. By comparing response time, number of changes and average number of resources (almost in all aspects and for all patterns it is relatively lower), we can observe that FQL4KE is more effective in terms of learning optimal policies and updating them at runtime.
- Both the cloud controller without learning mechanism and with learning are more effective than the native cloud platform reactive auto-scalers. Note that for the controller without learning, we consider a reasonably logical set of rules to govern the elasticity decision making. But if we consider a non sensible set of rules, the native auto-scaling of Azure performs better than RobusT2Scale.

TABLE I: FQL4KE performance under different strategies.

Strategy	Criteria	Big spike	Dual phase	Large variations
S1	$rt_{95\%}, \overline{vm}$	1212ms, 2.2	548ms, 3.6	991ms, 4.3
	node change	390	360	420
	convergence	32	34	40
S2	$rt_{95\%}, \overline{vm}$	1298ms, 2.3	609ms, 3.8	1191ms, 4.4
	node change	412	376	429
	convergence	38	36	87
S3	$rt_{95\%}, \overline{vm}$	1262ms, 2.4	701ms, 3.8	1203ms, 4.3
	node change	420	387	432
	convergence	30	29	68
S4	$rt_{95\%}, \overline{vm}$	1193ms, 3.2	723ms, 4.1	1594ms, 4.8
	node change	487	421	453
	convergence	328	328	328
S5	$rt_{95\%}, \overline{vm}$	1339ms, 3.2	729ms, 3.8	1233ms, 5.1
	node change	410	377	420
	convergence	N/A	N/A	N/A
Azure	$rt_{95\%}, \overline{vm}$	1409ms, 3.3	712ms, 4.0	1341ms, 5.5
	node change	330	299	367
	convergence	N/A	N/A	N/A
		Quickly varying	**Slowly varying**	**Steep tri phase**
S1	$rt_{95\%}, \overline{vm}$	1319ms, 4.4	512ms, 3.6	561ms, 3.4
	node change	432	355	375
	convergence	65	24	27
S2	$rt_{95\%}, \overline{vm}$	1350ms, 4.8	533ms, 3.6	603ms, 3.4
	node change	486	370	393
	convergence	98	45	28
S3	$rt_{95\%}, \overline{vm}$	1287ms, 4.9	507ms, 3.7	569ms, 3.4
	node change	512	372	412
	convergence	86	40	23
S4	$rt_{95\%}, \overline{vm}$	2098ms, 5.9	572ms, 5.0	722ms, 4.8
	node change	542	411	444
	convergence	328	328	328
S5	$rt_{95\%}, \overline{vm}$	1341ms, 5.3	567ms, 3.7	512ms, 3.9
	node change	479	366	390
	convergence	N/A	N/A	N/A
Azure	$rt_{95\%}, \overline{vm}$	1431ms, 5.4	1101ms, 3.7	1412ms, 4.0
	node change	398	287	231
	convergence	N/A	N/A	N/A

VI. DISCUSSION

A. Computational Complexity and Memory Consumption

The runtime overhead of the feedback control loop activities (cf. Figure 4) is depicted in Figure 10. Step 2 to Step 8 in Algorithm 1 are computationally intensive and based on our experiments are in the order of few minutes for 9 states and for 10,000 learning epochs. However, runtime overhead of the learning is not an issue in our setting because of the actuation delays that are in the order of magnitude of several minutes, 8-9 minutes for scaling out an extra small VM on Azure platform and 2-3 minutes for removing an existing VM. Note the few outliers as the result of failures of the PaaS level actions (cf. actuation) and network communication delays (cf. monitoring) on Azure platform during the course of our experiments.

In addition, the memory consumption of our approach is given by the dimensions of the look up table that saves and updates the q-values. In other words, the space complexity of our approach is always $O(N \times A)$, where N is the number of states and A is the number of actions. In the setting that we described in this paper, the table is composed by 9 states $\times 5$ actions $= 45$ q-values, thus memory consumption is negligible.

B. FQL4KE for Policy-based Adaptations

Although in this paper we integrated FQL4KE with RobusT2Scale, this approach is general and can be integrated with any knowledge-based controllers. By knowledge-based controller, we mean any controller that have explicit notion of knowledge that can be specified in terms of rules and used for reasoning and producing the control signal. Basically FQL4KE can be integrated with such controllers to learn rules and populate the knowledge base at runtime. Such policy-based controllers are not only applied for resource scaling but have also been previously applied to the rule-based adaptations of software architecture at runtime [14].

C. Limitations

Besides the provided features of FQL4KE, it comes with some limitations. Firstly, performance of scaling actions produced by FQL4KE during initial learning epochs at runtime may be poor. The key reason is that in environments such as cloud, in which every interaction with the environment (i.e., adding/removing cloud resources) takes several minutes, the learning typically converges slowly. This imposes some

difficulties. First, at early stages when the learning process has not been converged there might be some over-provisioning or under-provisioning due to such decisions. However, some other strategies (e.g., temporary over-provisioning) can be adopted in parallel in order to let the approach learns policies and when the optimal policies have been learned, it becomes the sole decision maker for resource allocation. Secondly, the learning process may be sensitive to the selection of the reinforcement signal (cf. Equation 7). It is also dependent on the fact that the system states must have been visited sufficiently [24].

D. Threats to Validity

There are a number of sources of threats to validity of the results presented in Section V. First, the results presented in Table I may be slightly different depending on the utility function defined in Eq. 7. We defined a reasonable function to measure the reward, while this can be defined differently leading to a different effectiveness of learning strategies. We expect the results would be consistent with the effectiveness (cf. Table I) of our solution as long as the function is appropriate, i.e., only consider both reward or punishment even with different metrics that we used, but not only one aspect.

The other threat to the validity of the result is the application framework that we built for our experiment, i.e., ElasticBench. Although we embed different characteristics of a cloud-based application by using Fibonacci based calculation and using cloud based technologies such as caching, but the results presented in Table I may be slightly different for other types of application. However, since we can simulate different functionalities with this framework, we expect that results on a different application is consistent with the ones presented in Section V. This requires further investigations with real-world software applications. Also note that we have implemented the solution in both Azure and OpenStack to demonstrate cross-platform applicability, though more experimentations are necessary to fully validate FQL4KE in particular in OpenStack.

Although FQL4KE does not impose any constraints on the possible number of scaling actions, for simplicity, we only considered five possible scaling actions (i.e., $-2, -1, 0, +2, +2$). This limited set of actions has some implications on the performance (cf. Section V-B) and effectiveness of learning (cf. Section V-C).

Finally, limited number of workload patterns (6 patterns is used in this work for evaluation, cf. Figure 11) is another threats to the validity. As it is also used in other research [12], this set of patterns, although not comprehensive, but provides a reasonably enough environmental conditions for evaluation.

VII. Related Work

In autonomic computing, policy-based adaptation techniques have been used to build self-adaptive software. We here focus on *policy-based adaptation*, related to *software adaptation* and *dynamic resource allocation*.

Policy-based adaptation. In self-adaptive software literature, policy-based adaptation has gained momentum due to its efficiency and flexibility for planning [16]. A policy-based approach can potentially decouple adaptation logic with how to react when necessary. Rainbow [13] exploits architecture-based adaptation, in which system chooses new architectural reconfigurations, at runtime, based on rules defined at design-time. In a similar line, Sykes et al. [25] propose an online planning approach to architecture-based self-managed systems. Their work describes a plan as a set of condition-action rules, which has been generated by observing a change in the operational environment. Georgas and Taylor [14] present an architecture-centric approach in which adaptation polices are specified as reaction rules. Not all of the policy-based approaches exploit `if-then` rules, other resemblances of policy have been also utilized. For instance, model-based approaches in terms of variability models has been adopted in [6]. While policy-based approaches have been shown useful in some settings (e.g., enforcing certain characteristics in the system), they cannot deal with unseen situations or uncertainties. System hence produces suboptimal decisions. The solution proposed here, FQL4KE, is in the same line of research, but applied fuzzy Q-learning, for the first time, to the problem of dynamic resource allocation through online policy evolution.

Dynamic adaptation planning. In [3], dynamic decision networks are proposed to deal with the uncertainty in decision-making of self-adaptive systems. The initial models are provided by experts; however, the models are updated at runtime as more evidences are observed through monitoring. Esfahani et al. [7] discuss the application of black-box learning models to understand the impact of different features in a self-adaptive system. Given a system goal, a function is learned to formulate the impact of different features. Amoui et al. [1] present an approach based on reinforcement learning to select adaptation actions at runtime. Through an adaptive web-based case study, it is shown that the approach provides similar results comparing to a voting-based approach that uses expert knowledge. Kim et al. [5] discuss the application of Q-learning to plan architecture-based adaptations, a similar policy-based architecture adaptation is also proposed in [14], applied in robotics domain. FQL4KE addresses decision making in autonomic systems, particularly focusing on resource allocation in cloud-based applications.

Dynamic resource allocation. Xu et al. [4] present an approach to learning appropriate auto-configuration in virtualized resources. It uses multiple agents, each of which apply reinforcement learning to optimize auto-configuration of its dedicated environment. Barrett et al. [2] investigate the impact of varying performance of cloud resources on application performance. They show that a resource allocation approach, considering this aspect, achieves benefits in terms of performance and cost. To reduce the learning time, a parallelized reinforcement learning algorithm is proposed through which multiple agents are employed to deal with the same tasks to speed up the procedure to explore the state space. Huber et al. [17] propose DML, a domain-specific language, that enables parametric performance modeling and adaptation process for self-adaptive resource management in heterogeneous environments. The internal knowledge is based

on linear regression model that is kept up to date based on runtime observations and the prediction based on the internal model triggers resource adaptations. Lama et al. [20] integrate NN with fuzzy logic to build adaptive controllers for autonomic server provisioning. Similar to our approach, NNs define a set of fuzzy rules, and the self-adaptive controller adapts the structure of the NN at runtime, therefore automatically updating rules. The above mentioned approaches enable horizontal elasticity, however some approaches like [9] enable vertical elasticity. Unlike the above approaches, FQL4KE offers a seamless knowledge evolution through fuzzy control and RL, putting the burden of defining adaptation rules off the users, while keeping the internal model relevant throughout the system operation.

VIII. Conclusions and Future Work

We have investigated dynamic quality management for deployed cloud-based application architectures. The scenario under investigation assumes no a priori knowledge is available regarding the policies that cloud controllers can exploit for quality management. Instead of specifying elasticity policies in auto-scaling solutions, for system operations it is only required to provide the importance weights in the reward function for designing such elasticity controller. In order to realize this, a fuzzy rule-based controller (lower feedback control loop) linked with a reinforcement learning algorithm (upper knowledge evolution loop) for learning optimal elasticity policies, has been proposed. The advantages are:

1) FQL4KE is *robust* to highly dynamic workload intensity due to its self-adaptive and self-learning capabilities.
2) FQL4KE is *model-independent*. The variations in the performance of the deployed applications and the unpredictability of dynamic workloads do not affect the effectiveness of the proposed approach.
3) FQL4KE is capable of automatically updating the control rules through a *fast online learning*. FQL4KE auto-scales and learns to improve its performance simultaneously.
4) Unlike supervised techniques that learn from the training data, FQL4KE *does not require off-line training* that saves significant amount of time and efforts.

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. Also (iii) we aim to extend the IBM MAPE-**K** adaptation loop to MAPE-**KE** that is able to update different knowledge sources (cf. Figure 6).

Acknowledgment

This work has received funding from IC4 (an Irish national technology centre funded by EI) and the EU's Programme FP7/2007-2013 under grant agreement 610802 (CloudWave).

References

[1] M. Amoui and et al. Adaptive action selection in autonomic software using reinforcement learning. In *ICAC'08*, 2008.
[2] E. Barrett, E. Howley, and et al. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12), 2013.
[3] N. Bencomo and et al. Dynamic decision networks for decision-making in self-adaptive systems: A case study. In *SEAMS'13*, 2013.
[4] X. Bu, J. Rao, and C.Z. Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Trans. Parallel Distrib. Syst.*, 24(4), 2013.
[5] K. Dongsun and P. Sooyong. Reinforcement learning-based dynamic adaptation planning method for architecture-based self-managed software. In *SEAMS'09*, May 2009.
[6] A. Elkhodary, N. Esfahani, and S. Malek. FUSION: a framework for engineering self-tuning self-adaptive software systems. In *FSE '10*, New York, New York, USA, 2010. ACM Press.
[7] N. Esfahani, A. Elkhodary, and S. Malek. A learning-based framework for engineering feature-oriented self-adaptive software systems. *IEEE Transactions on Software Engineering*, 39(11), Nov 2013.
[8] S. Farokhi, P. Jamshidi, I. Brandic, and E. Elmroth. Self-adaptation challenges for cloud-based applications: A control theoretic perspective. In *Feedback Computing*, 2015.
[9] S. Farokhi, P. Jamshidi, E.B. Lakew, I. Brandic, and E. Elmroth. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *Future Generation Computer Systems, under evaluation*, 2016.
[10] A. Filieri, M. Maggio, and et al. Software engineering meets control theory. In *SEAMS'15*, 2015.
[11] A. Gambi, M. Pezze, and G. Toffetti. Kriging-based self-adaptive cloud controllers. *IEEE Transactions on Services Computing*, 2014.
[12] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. Adaptive, model-driven autoscaling for cloud applications. In *ICAC'14*, pages 57–64. USENIX, 2014.
[13] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10), October 2004.
[14] J.C. Georgas and R.N. Taylor. Policy-based self-adaptive architectures: Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In *SEAMS '08*. ACM Press, 2008.
[15] P.Y. Glorennec and L. Jouffe. Fuzzy Q-learning. In *Proceedings of 6th International Fuzzy Systems Conference*, volume 2. IEEE, 1997.
[16] H.N. Ho and E. Lee. Model-based reinforcement learning approach for planning in self-adaptive software system. In *Conference on Ubiquitous Information Management and Communication*, 2015.
[17] N. Huber, J. Walter, M. Bahr, and S. Kounev. Model-based autonomic and performance-aware system adaptation in heterogeneous resource environments: A case study. In *ICCAC*, pages 181–191. IEEE, 2015.
[18] P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic resource provisioning for cloud-based software. In *SEAMS*, pages 95–104, 2014.
[19] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1), January 2003.
[20] P. Lama and X. Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for percentile-based delay guarantee. *ACM Transactions on Autonomous and Adaptive Systems*, 2013.
[21] T. Lorido-Botran, J. Miguel-Alonso, and J.A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 2014.
[22] M. Netto and et al. Evaluating auto-scaling strategies for cloud computing environments. *MASCOTS'14*, 2014.
[23] C. Pahl and P. Jamshidi. Software architecture for the clouda roadmap towards control-theoretic, model-based cloud architecture. In *Software Architecture*, pages 212–220. Springer, 2015.
[24] R.S. Sutton and A.G. Barto. *Introduction to reinforcement learning*. MIT Press, 1998.
[25] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *SEAMS '08*, New York, New York, USA, 2008. ACM Press.
[26] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. 33(1):291–302, 2005.
[27] W.E. Walsh, G Tesauro, J.O. Kephart, and R. Das. Utility functions in autonomic systems. *International Conference on Autonomic Computing, 2004. Proceedings.*, 2004.