An Auto-Scaling Cloud Controller Using Fuzzy Q-Learning - Implementation in OpenStack

Hamid Arabnejad, Pooyan Jamshidi, Giovani Estrada, Nabil El Ioini, Claus Pahl

An Auto-Scaling Cloud Controller using Fuzzy Q-Learning - Implementation in OpenStack

Hamid Arabnejad[1], Pooyan Jamshidi[2], Giovani Estrada[3], Nabil El Ioini[4], and Claus Pahl[4]

[1] IC4, Dublin City University, Dublin, Ireland
[2] Imperial College London, London, UK
[3] Intel, Leixlip, Ireland
[4] Free University of Bozen-Bolzano, Bolzano, Italy

Abstract. Auto-scaling, i.e., acquiring and releasing resources automatically, is a central feature of cloud platforms. The key problem is how and when to add/remove resources in order to meet agreed service-level agreements. 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. We propose a dynamic learning strategy based on a fuzzy logic algorithm, which learns and modifies fuzzy scaling rules at runtime without requiring prior knowledge. The proposed algorithm is implemented and evaluated as an extension to the OpenStack cloud platform, integrating it with the Heat and Ceilometer components for orchestration and monitoring, respectively, using Heat Orchestration Templates. We specifically focus on implementation and experimentation aspects here. Our auto-scaling approach can handle various load traffic situations, delivering resources on demand while reducing infrastructure and management costs. The experimentals show promising performance in terms of resource adjustment to optimize SLA compliance (response time) while reducing cloud provider's costs.

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

1 Introduction

Cloud computing allows easy deployment of elastic applications. Our focus is on Infrastructure as a Service (IaaS), which allows customers to increase or decrease their computational and storage resources on the fly. The consumer does not manage or control the underlying cloud infrastructure, but has control over operating systems, storage, and deployed applications [12]. IaaS provides virtualization, which enables running multiple operating system (OS) instances, called virtual machines (VMs), on the same physical server.

Important concepts of cloud computing are elasticity and dynamism. Managing physical and virtual resources is a key challenge in the IaaS model. However,

it allows applications to acquire and release resources dynamically, but deciding the correct number of resources to be released/acquired is the challenging concern. *Auto-scaling* is a process that automatically scales the number of resources and maintains an acceptable Quality-of-Service (QoS). The scaling process can be either vertical or horizontal. Vertical scaling involves modifying the amount of resources assigned to each VM (CPU and memory, mostly). Horizontal scaling involves acquiring or releasing of VMs. In most common operating systems, altering CPU core, memory or disk of the VM which it runs, is not possible without rebooting; for this reason, most cloud provider only offer horizontal scaling.

To address auto-scaling in IaaS infrastructures, we utilise `FQL4KE`, a technique for dynamic resource allocation, presented in [9]. The advantage of `FQL4KE` 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. We focus here on the implementation and evaluation of `FQL4KE` as an extension to the OpenStack cloud platform, integrating it with the Heat and Ceilometer components for orchestration and monitoring, respectively, using Heat Orchestration Templates for orchestration specification. Previously in [9], we performed the experiments on PaaS cloud platform whereas in this research, we specifically focus on architecture, implementation and experimentation aspects in OpenStack as an industry-standard IaaS cloud platform (we have documented the applicability to other platforms such as Azure elsewhere [8, 9]. New here is the in-depth implementation and experimental evaluation coverage. We also cover a wider range of workload patterns. We demonstrate that this auto-scaling approach can handle various load traffic situations, delivering resources on demand while reducing infrastructure and management costs. The experimental results show promising performance in terms of resource adjustment to optimize SLA compliance and response time while reducing cloud provider's costs.

The paper is organized as follows. Section 2 describes auto-scaling process briefly and discusses on related research in this area, Section 3 describes the OpenStack architecture and orchestration, Section 4 describes our approach in detail followed by implementation in Section 5 and evaluation in Section 6.

2 Background and Related Work

The aim of auto-scaling approaches is to acquire and release resources dynamically while maintaining an acceptable QoS [10]. The auto-scaling process is usually represented and implemented by a IBM's MAPE-K (Monitor, Analyze, Plan and Execute phases over a Knowledge base) control loop [7].

Threshold-based rules. Threshold-based rules are supported by many cloud solutions such as EC2, Azure, or OpenStack. 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. Each rule includes an *upper* and a *lower* threshold that defines bound values for applying auto-scaling. Dutreilh et al. [3] investigate horizontal auto-scaling using threshold-based and reinforcement learning techniques. In [5], the authors describe an approach that

operates fine-grained scaling at resource level in addition to VM-level scaling in order to improve resource utilization while reducing cloud providers' costs. Hasan et al. [6] extend the typical two threshold bound values and add two levels of threshold parameters and use the three domains (CPU loads, response time and network link bandwidth) in making scaling decisions. Chieu et al. [2] propose a strategy for dynamic scalability of PaaS and SaaS applications based on the number of active sessions and scaling the number of VMs if all instances with active sessions exceed thresholds. The advantage is simplicity. However, the performance depends on the quality of the thresholds.

Reinforcement learning (RL). RL [17] is learning process for an agent to maximize its rewards. Here, the agent is an auto-scaler, the action is scaling up/down, the context 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 response to current state to maximize the rewards. The most used approach is Q-learning. It learns estimates of Q-values $Q(s, a)$, which map all system states s to the best action a. We initialise all $Q(s, a)$ and, during learning, choose an action a for state s based on an ϵ-greedy policy and apply it to 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). Tesauro et al. [18] propose a hybrid learning system for dynamic server allocation maximize the profits. Their approach combines a queuing network model (for online management) and reinforced learning using the SARSA approach (for offline training), making resource allocation decisions based on application workload and response time. Rao et al. [15] introduce VCONF, a reinforcement learning approach in the context of neural networks, for dynamic VM autoconfiguration according to the application requirements, i.e., it automatically changes VM configurations in order to achieve good performance for hosted applications.

In RL, there is no need of prior knowledge. It has the 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.

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 (the 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. The system is the target platform and system output are parameters to evaluate system performance (response time or CPU load).

Ali-Eldin et al. [1] use queueing theory to model a service. Two adaptive hybrid reactive/proactive controllers estimate future load in order to support elasticity. Padala et al. [13] 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.

3 OpenStack Orchestration

OpenStack is an open-source platform, mostly deployed as an IaaS and used for building public and private clouds. The platform consists of interrelated components that control hardware pools of processing, storage, and networking resources through a data center. Users either manage it through a web-based dashboard, through command-line tools, or through a RESTful API. Fig. 1 overviews the OpenStack core services. The important components here are:

- **Neutron:** is a system for managing networks and IP addresses and handles creation and management of a virtual networking infrastructure and gives users self-service ability over network configurations. It provides frameworks to mange and deploy advanced services such as load balancing.
- **Nova:** is the primary computing engine behind OpenStack. It provides deploying and managing virtual machines and other instances to handle computing tasks as the main part of an IaaS system.
- **Glance:** provides discovery, registration, and delivery services for disk and server images. It allows to use stored images as templates for new servers.
- **Heat:** is a service for managing the infrastructure needed for cloud applications to run. It provides templates to create and manage cloud resources such as storage, networking, instances, or applications. Templates are used to create stacks, which are collections of resources.
- **Ceilometer:** provides telemetry services to collect metering data. The collected data can be used for billing, system monitoring, or alerts.
- **Keystone:** provides user/service/endpoint authentication and authorization. Calling the API function requires authentication by Keystone.

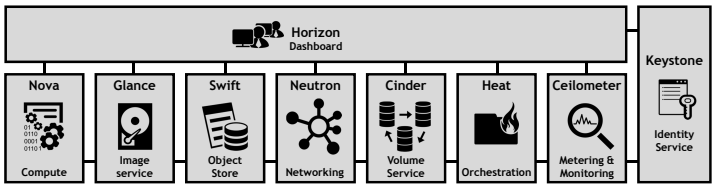


Fig. 1: An OpenStack block diagram

OpenStack orchestration allows 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, instances to deliver a service defined by policies. Heat, as OpenStack's orchestration component, implements an engine to launch multiple composite cloud 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 (such as compute instance) alongside using monitoring alerts (such as CPU utilization) provided by Ceilometer, Heat can dynamically adjust resource allocation, i.e., launching resources to meet application demand and removing them when no longer required. Fig. 2 shows the heat and ceilometer components. Heat executes HOT (Heat Orchestration Template) templates.

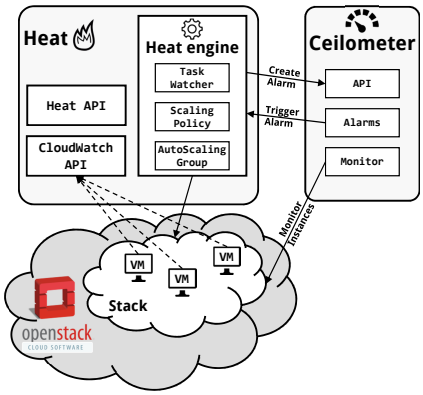


Fig. 2: Heat/Ceilometer architecture.

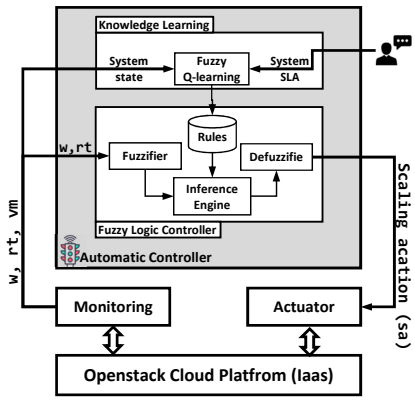


Fig. 3: Logical FQL4KE architecture.

4 Auto-Scaling Algorithm

In [8, 9] we proposed an elasticity controller, which is the basis of this investigation for OpenStack. This is an online learning mechanism by combining fuzzy control and fuzzy Q-learning (FQL) [4], called FQL4KE. The fuzzy Q-learning and control is a self-adaptive mechanism where the fuzzy control facilitates reasoning at a higher level of abstraction and the Q-learning allows to adjust the controller. The fuzzy rules are continually tuned through learning from the data collected. Here, we describe the implemention and evaluation in OpenStack.

4.1 FQL4KE Building Blocks

A fuzzy model is a qualitative model constructed from a set of fuzzy-rules to represent the relationship between system input and output [16]. However, there are some issues for defining rules at design-time such as: (i) whole knowledge or some parts of it may be available; (ii) knowledge is not an optimized model (existence of redundancy or ineffective rules); (iii) inaccuracy of some rules and (iv) instability of design-time rules requiring them to be changed at runtime. As a result, incomplete, inappropriate or not-optimized set of rules may lead to sub-optimal scaling decisions and loss of revenue for cloud application providers.

Fig. 3 illustrates the building blocks of FQL4KE. During the application lifecycle, FQL4KE guides resource provisioning following the autonomic MAPE-K loop by monitoring continuously different characteristics of the application (workload and response time), checking the satisfaction of system goals and adopting resource allocation to satisfy goals.

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

4.2 Fuzzy Logic Controller

Fuzzy inference maps a set of control inputs to a set of control outputs through fuzzy rules. The first step is to partition the state space of each input variable into fuzzy sets through membership functions. Each fuzzy set is associated with a linguistic term such as "low" or "high". The membership function $\mu(x)$ quantifies the degree of membership of input signal x to the fuzzy set y. The membership functions, see Fig. 4, are triangular and trapezoidal [8]. Three fuzzy sets are defined for each input (workload and response time) to achieve a reasonable granularity in the input space while keeping the number of states small.

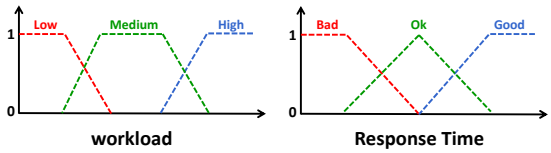


Fig. 4: Fuzzy membership functions for auto-scaling variables

For the inference mechanism, we define elasticity policies as 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. Here, no a priori knowledge for defining the rules is assumed. `FQL4KE` finds the consequent Y for the rules.

Once the fuzzy controller is designed, controller execution comprises of three steps (cf. middle part of Fig. 3): (i) fuzzification of inputs, (ii) fuzzy reasoning and (iii) defuzzification of output. The fuzzifier projects the crisp data onto fuzzy information using membership functions. The fuzzy engine reasons based on a set of fuzzy rules and derives fuzzy actions. The defuzzifier reverts results back to crisp mode and activates an adaptation action. This result is enacted by issuing appropriate commands to the underlying platform fabric.

4.3 Fuzzy Q-Learning

The mechanism learns the policies at runtime, enabling knowledge evolution (i.e., KE in `FQL4KE`). As the controller has to take action in each control loop, it selects past actions taken which produced good (long-term cumulative) rewards:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1}$$

The discount rate γ determines the relative importance of future rewards. There is a trade-off (step 2 in Alg. 1) between actions that have been tried (exploitation) and new actions that may lead to better rewards in the future (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 is defined as:

$$Q^{\pi}(s, a) = E_{\pi}\Big\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big\} \tag{2}$$

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

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \Big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big] \tag{3}$$

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

The policy adaptation is achieved by selecting a random action with probability ϵ and an action that maximizes Q in the current state with probability $1 - \epsilon$. The value of ϵ is determined by the exploitation/exploration strategy:

$$a(s) = arg\max_a Q(s, k) \tag{4}$$

The fuzzy Q-learning is summarized in Algorithm 1[5]. For our use case, 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 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.

4.4 Dynamic Resource Allocation by FQL4KE

For the *reward function*, the controller receives the current values of w and rt that correspond to the system state, $s(t)$ (step 4 in Alg. 1). The control signal sa

[5] A Matlab implementation: https://github.com/pooyanjamshidi/Fuzzy-Q-Learning

Algorithm 1 Fuzzy Q-Learning

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

1: Initialize q-values:
2: Select an action for each fired rule:
$a_i = argmax_k q[i,k]$ with probability $1-\epsilon$ ▷ Eq4
$a_i = random\{a_k, k = 1, 2, \cdots, J\}$ with probability ϵ
3: Calculate the control action by the fuzzy controller
4: Approximate the Q function from current q-values and firing level of the rules:
$$Q\big(s(t),a\big) = \sum_{i=1}^{N}\Big(\alpha_i(s) \times q[i,a_i]\Big)$$
where $Q\big(s(t),a\big)$ 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:
$$V\big(s(t+1)\big) = \sum_{i=1}^{N}\alpha_i\big(s(t+1)\big).\max_a(q[i,q_k])$$
7: Calculate the error signal:
$\Delta Q = r(t+1) + \gamma \times V_t\big(s(t+1)\big) - Q\big(s(t),a\big)$ ▷ Eq3
8: Update q-values (where η is a learning rate):
$q[i,a_i] = q[i,a_i] + \eta.\Delta Q.\alpha_i\big(s(t)\big)$ ▷ Eq3
9: Repeat the process for the new state until it converges

represents the action a that the controller takes in each iteration. We define the reward signal $r(t)$ based on two criteria: (i) SLA violations, and (ii) the amount of resources acquired, which directly determines the cost as follows:

$$r(t) = U(t) - U(t-1) \tag{5}$$

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

$$U(t) = w_1.\Big(1 - \frac{vm(t)}{vm_{max}}\Big) + w_2.\Big(1 - H(t)\Big)$$

$$H(t) = \begin{cases} \frac{rt(t) - rt_{des}}{rt_{des}} & , rt_{des} \le rt(t) \le 2.rt_{des} \\ 1 & , rt(t) \ge 2.rt_{des} \\ 0 & , rt(t) \le rt_{des} \end{cases} \tag{6}$$

where $vm(r)$ and $rt(t)$ are workload and response time (actual & desired) of the system. w_1 and w_2 are their corresponding weights determining their relative importance in the utility function. In order to aggregate the individual criteria, we normalize them depending on whether they should be maximized or minimized.

For the *knowledge base update*, we start with controlling the allocation of resources with no a priori knowledge. After enough *explorations*, the consequents

of the rules can be determined by selecting those actions that correspond to the *highest* q-value in each row of the Q-table. Although we do not rely on design-time knowledge, if even partial knowledge is available or there exists data regarding performance of the application, our solution can exploit such knowledge by initializing q-values (cf. step 1 in Alg. 1) with more meaningful data. This implies a quicker learning convergence.

5 Implementation

We implemented `FQL4KE` in OpenStack. Orchestration and automation within OpenStack is handled by the Heat component. It provides a declarative structure for defining auto-scaling processes. A new OpenStack native standard has also been developed for providing templates for Orchestration called HOT (Heat Orchestration Template) which meant to replace the Heat CloudFormation-compatible format (CFN). Heat automatically provisions infrastructure (compute, network, storage) based on a YAML template file. 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 platform. Collecting measurements in OpenStack is handled by Ceilometer.

```
heat_template_version: 2013-05-23
parameters:
  max_instances:
    type: number
    default : 5
  min_instances:
    type: number
    default : 1
  desired_capacity:
    type: number
    default : 1
  OpenStack_AUTH_URL:
    type: string
    default : NO OpenStack_AUTH_URL
  web_server_image:
    type: string
    default: cirros-0.3.4-x86_64-disk
  web_server_flavor:
    type: string
    default: m1.nano
  ctrlsrv_image:
    type: string
    default: precise-server-cloudimg-amd64-disk1
  ctrlsrv_flavor:
    type: string
    default: m1.micro
  SSH_key:
    type: string
    default: mykeypair
  network:
    type: string
  subnet_id:
    type: string
  external_network_id:
    type: string
...
resources:
  # FQL Auto-scaling server
  controller:
    type: OS::Nova::Server
    properties:
      name: ctrlsrv
      flavor: {get_param: ctrlsrv_flavor}
      image: {get_param: ctrlsrv_image}
      key_name: {get_param: SSH_key}
      networks: [{network: {get_param: network} }]
      user_data_format: RAW
      user_data:
        str_replace:
          template: |
            # srouce code of FQL goes here and will
              ↪    executed

  # web server
  asg:
    type: OS::Heat::AutoScalingGroup
    properties:
      desired_capacity: {get_param: desired_capacity}
      min_size: {get_param: min_instances}
      max_size: {get_param: max_instances}
      resource:
        type: lb_server.yaml
        properties:
          flavor: {get_param: web_server_flavor}
          image: {get_param: web_server_image}
          key_name: {get_param: SSH_key}
          network: {get_param: network}
          pool_id: {get_resource: pool}
          metadata: {"metering.stack": {get_param:
            ↪    "OS::stack_id"}}
          user_data: |
            # srouce code of web-server goes here and
              ↪    will executed
  scaleup1_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: {get_resource: asg}
      cooldown: 10
      scaling_adjustment: 1
  scaledown1_policy:
    type: OS::Heat::ScalingPolicy
    properties:
      adjustment_type: change_in_capacity
      auto_scaling_group_id: {get_resource: asg}
      cooldown: 10
      scaling_adjustment: -1
  scaleup2_policy:
    ...
  scaledown2_policy:
    ...
  monitor:
    type: OS::Neutron::HealthMonitor
    properties:
      type: TCP
      delay: 1
      max_retries: 10
      timeout: 1
  pool:
    type: OS::Neutron::Pool
    properties:
      protocol: HTTP
      monitors: [{get_resource: monitor}]
      subnet_id: {get_param: subnet_id}
      lb_method: ROUND_ROBIN
      vip:
        protocol_port: 80
  lb:
    type: OS::Neutron::LoadBalancer
    properties:
      protocol_port: 80
      pool_id: {get_resource: pool}
  lb_floating:
    type: OS::Neutron::FloatingIP
    properties:
      floating_network_id: {get_param:
        ↪    external_network_id}
      port_id: {get_attr: [pool, vip, port_id]}
```

Fig. 5: Excerpts from OpenStack HOT Template, including resource boundaries and configurations, scaling policies and the integration of e.g. load balancer and monitoring tools, as part of the auto-scaling group (asg) definition.

A combination of Heat and Ceilometer is used (see Fig. 2). 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. An exceprt of our Heat implementation, defined in YAML, is shown in Fig. 5. This YAML structure defines the required resources for auto-scaling process. Auto-scaling in Heat is done using three main resources:

- **auto-scaling group :** defined using type `OS::Heat::AutoScalingGroup` and is a resource type that is used to encapsulate the resource that we wish to scale, and some properties related to the scale process.
- **scaling policy :** defined using type `OS::Heat::ScalingPolicy` and is a resource type to define the effect a scaled process has on the resource.
- **alarm :** defined using type `OS::Ceilometer::Alarm` and is a resource type to define under which conditions the ScalingPolicy is triggered.

In the following, we describe the architectural integration of our controller with the OpenStack platform components – with turns out to be not straighforward.

In our implementation, the environment contains one or more VM instances that are members of a load balancer and defined as members in an AutoScalingGroup resource. Each instance (VM) includes a simple web server run inside of it after launching. The implemented server listens to an input port (here port 80) and return simple HTML pages as the response. The simple web server is created and coded as a part of the `user date` property of each VM. 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. The response time from each web server includes the VM instance's hostname. For the VM instance type, we used a minimal Linux distribution, `cirros`[6], an image that was designed for use as a test image on clouds such as OpenStack.

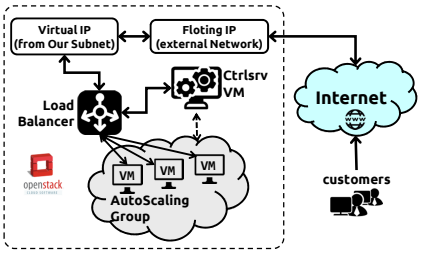


Fig. 6: System Overview.

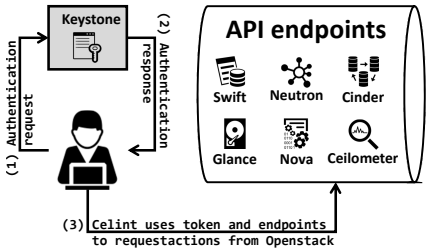


Fig. 7: `cURL` calling OpenStack API.

Generally, the native auto-scaling approach in OpenStack 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 by the `FQL4KE` algorithm, to control and manage scaling option. We added a resource type `OS::Nova::Server` to create an additional VM, named `ctrlsrv`, which acts as an auto-scaling server and enacts the scale up/down decision proposed the FQL learning algorithm. Fig. 6 illustrates the implemented system in OpenStack. The created load-balancer distributes client HTTP requests across

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

a set of web servers, i.e., auto-scaling group resources, collected in a load balancer pool. The algorithm used to distribute the load between members of the pool is `ROUND_ROBIN`. As shown, the `ctrlsrv` machine, by gathering information from the load-balancer and the current state of the AutoScalingGroup resource, decides which horizontal scaling, i.e., up or down, should applied in the target platform. The scale-up will launch a new server instance. which may take a few minutes as the instance needs to be started and added to the load-balancer pool. Once all preconfigured settings are installed, i.e., the defined user data file, it will go to active and receive or answer to requests sent by the load-balancer.

The proposed auto-scaling algorithm is coded in Python and runs inside the `ctrlsrv` machine. We added a complete fuzzy logic library. This is functionally similar to the respective matlab features and implements used our `FQL4KE` approach. However, for some parameters in the proposed algorithm, such as the current number of VM instances or workload, we used the OpenStack API command line. For example, command `nova list` shows a list of running instances. Due to the unavailability of direct access to the OpenStack API inside of `ctrlsrv` machine, we used the popular command line utility `cURL` to interact with a couple of OpenStack APIs. `cURL` allows transmitting and receiving HTTP requests and responses from the command line or a shell script, which enables working with the OpenStack API directly. For some OpenStack APIs, it is necessary to send additional data, such as authentication keys, as a header request. In Fig. 7, the process of using `cURL` to call OpenStack APIs is demonstrated.

The first step is to send a request authentication token by passing correct credentials (username and password) from the OpenStack identity service. After receiving `Auth-Token` from Keystone, the user can combine the authentication token and Computing Service API Endpoint and send as HTTP request and receive the output. We used this process inside of `ctrlsrv` machine to execute OpenStack APIs and collect required outputs. By combining the settings, we are able to inetgrate and run the `FQL4KE` technique as the manager and controller of auto-scaling processes in OpenStack.

6 Experimental Results and Discussion

The experimental evaluation is designed to show the effectiveness of proposed `FQL4KE` approach as part of the OpenStack platform. Furthermore, the cost improvement by proposed approach for cloud provider is demonstrated.

6.1 Experimental Setup and Benchmark

In our experiment, `FQL4KE` was implemented as a full working system and was tested on OpenStack. A web server was considered as target cloud application. Each server is configured and installed on one dedicated VM, which uses `Cirros` images (Linux distribution), and random response times between 0 and 1 sec. For the auto-scaling control server, due to the Impossibiliity of installing any additional package in the `Cirros` image, we considered a VM machine running a

Linux Ubuntu precise server. The maximum and minimum number of VMs that are allowed to be available at the same time is set to 5 and 1, respectively.

The term workload refers to a number of concurrent user request arrival in different time. Workload is defined as the sequential of users accessing the target application that need to be handled by auto-scaler. According to [10, 11], application workload types can be categorized in four representative patterns: (a) The *Predictable Bursting* pattern indicates the type of workload that is subject to periodic peaks and valleys such services with seasonality trends or high performance computing, (b) the *Variations* pattern reflects applications such as News&Media, event registration or rapid fire sales, (c) the *Fast Growth* pattern presents applications such as events, business growth and slashdot effect and (d) 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.

We used `Siege`[7] as our performance measuring tool. `Siege` is a HTTP load testing and benchmarking utility which simulates web browsers. It can generate concurrent user requests and measure performance metrics such as average response time. For each number N of concurrent users, we generate N request per second by `Siege` for 10 minutes. The learning rate is set to a constant $\eta = 0.1$ and the discount factor is set to $\gamma = 0.8$. Here, considered lower value for η causes to giving more impact on old rewards with every update. After sufficient epochs of learning, we update the controller's knowledge base (FIS rules) and decrease the exploration rate (ϵ) until a minimum value is reached, here 0.2. So, FQL starts with exploration phase and after a first learning convergence happens, it enters the balanced exploration-exploitation phase.

Additionally, we compare the `FQL4KE` approach 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.

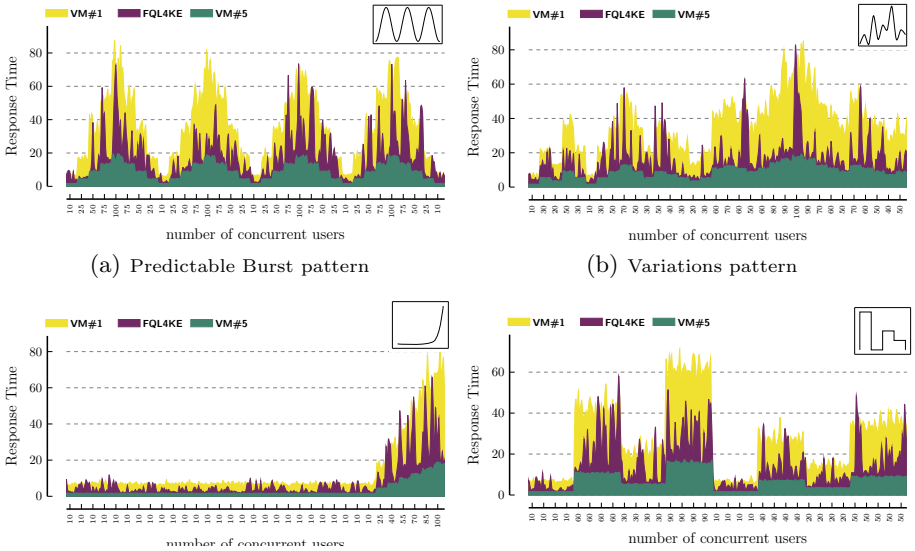
6.2 Performance

The metric to evaluate dynamic auto-scaling of resource allocation must represent response time in order to measure the QoS experienced by the users. To compare our approach with other approaches, we use the observed end-to-end response time for four workload patterns (Figs. 8(a) to 8(b)), here achieving the system goals. Additionally, to evaluate the infrastructure provider's operational cost, we used the percentage number of used VMs as represented metrics to show proposed approach effectiveness in auto-scaling applications to meet their QoS requirements while reducing infrastructure providers cost (Figs. 9(a) to 9(d)).

6.3 Effectiveness of the FQL4KE Algorithm

Figs. 8(a) to 8(b) show the fluctuation of the observed end-to-end response time for the workload patterns, e.g., Predictable Bursting (sine wave) or Variation. For

[7] Siege, https://www.joedog.org/siege-home/



(a) Predictable Burst pattern (b) Variations pattern

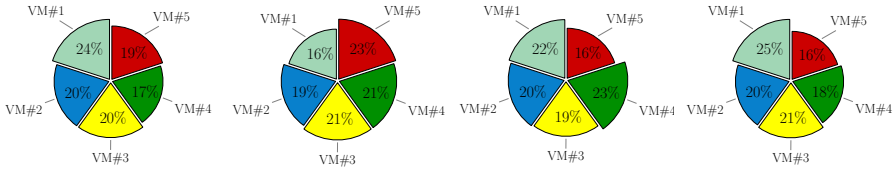
(c) Fast Growth pattern (d) ON&OFF pattern

Fig. 8: The Observed End-to-End Response Time for the four workload Patterns.

each change of the input workload, i.e., the concurrent input request submitted by individual users, the corresponding response time varies between upper or lower bound, depending on current load balance and number of available VMs. `FQL4KE` monitors the target application and scales up/down the current number of VMs by detecting these fluctuation of the response time. In our experiment, the scaling process, up/down, can be completed in a few milliseconds, due to simplicity and fast booting of `Cirros` image.

As shown in Fig. 8, FQL4KE demonstrates good performance compared with the base-line approaches, $VM\#1$ and $VM\#5$, which have a fixed number of VMs during the test. Firstly, performance of scaling actions produced by `FQL4KE` during the initial learning epochs at runtime may be poor. However, after some iterations and updating the knowledge base, the proposed approach adopts itself and is able to make more accurate decisions for the current status of system.

6.4 Cost-Effective Scaling by FQL4KE



(a) Predictable Burst (b) Variations (c) Fast Growth (d) ON&OFF

Fig. 9: Percentage Numbers of VMs used by `FQL4KE` for 4 Pattern Types.

Figs. 9(a) to 9(d) show the percentage of used VMs during the trial for all workload patterns. Our approach depends on current workload and relative re-

sponse time of the system at the current time, increasing the number of available VMs in scaling up and decreasing the number of idle VMs in scaling down. The FQL4KE algorithm conducts distributed-case scaling and allocates suitable numbers of VMs during the trail according to the workload. For example, the maximum number of VMs used is only in 19% of time during our experiment. This implies our approach 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. Thereby, FQL4KE can perform auto-scaling of application as well as save provider costs by increasing resource utilisation.

7 Conclusions and Future Work

We investigated VM-level scaling of cloud applications. In most real cases, no priori knowledge is available regarding elasticity policies that cloud controllers could exploit. A fuzzy Q-learning approach, called FQL4KE, that uses a fuzzy rule-based system combined with a reinforcement learning algorithm for learning optimal elasticity policies, has been implemented in OpenStack, an open-source IaaS platform. FQL4KE is capable of automatically updating the controller and learns to improve its performance simultaneously. Unlike supervised techniques, it does not require off-line training. FQL4KE can efficiently scale up/down cloud resources to meet the given QoS requirements while reducing cloud provider costs by improving resource utilisation. FQL4KE has been implemented in OpenStack platform to demonstrate the practical effectiveness of proposed approach has been successfully tested and presented.

A key contribution here is the demonstration of the architectural integration requirements that need to be overcome to actually implement an advanced autoscaling technique in an industrial setting. We have described the architectural challenges and solutions, including the successful evaluation in a real-world system. It reflects our experience in moveing a conceptual soliution that has been tested through simulations into a real setting. This includes how to use the platform mechanism to orchstrate the core tool like load balancer or networking, replacing the built-in controller and integrating this with identity management and the monitoring tools, which is critical in this context.

We plan to carry out further long-term experiments beyond the selected workload patterns induced. Conceptually, extensions are worth considering that extend the solution for environments which are partially observable. An exploration of different learning approaches, online and offline, is also planned. Here, possible dynamic changes in the fuzzy rule set need to be taken into account. A further direction is to investigate container virtualisation [14] and its workload and performance management.

8 Acknowledgement

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

References

1. A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212. IEEE, 2012.
2. TC Chieu, A Mohindra, and AA Karve. Scalability and performance of web applications in a compute cloud. In *Intl Conf on e-Business Engineering*, 2011.
3. X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck. From data center resource allocation to control theory and back. In *Intl Conf on Cloud Computing (CLOUD)*, pages 410–417. IEEE, 2010.
4. P.Y. Glorennec. Fuzzy q-learning and dynamical fuzzy q-learning. In *Fuzzy Systems, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the Third IEEE Conference on*, pages 474–479. IEEE, 1994.
5. R. Han, L. Guo, M.M. Ghanem, and Y. Guo. Lightweight resource scaling for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 644–651. IEEE, 2012.
6. 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.
7. M.C. Huebscher and J.A. McCann. A survey of autonomic computingdegrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.
8. P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic resource provisioning for cloud-based software. In *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 95–104. ACM, 2014.
9. 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 *Intl ACM Sigsoft Conference on the Quality of Software Architectures QoSA'2016*. ACM, 2016.
10. 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*, 12(4):559–592, 2014.
11. M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2011.
12. P. Mell and T. Grance. The NIST definition of cloud computing. 2011.
13. 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 *Europ Conf on Computer systems*, pages 13–26, 2009.
14. C. Pahl. Containerisation and the PaaS Cloud. IEEE Cloud Computing, 2(3). pp. 24-31. 2015.
15. J. Rao, X. Bu, C.Z. Xu, L. Wang, and G. Yin. Vconf: a reinforcement learning approach to virtual machines auto-configuration. In *Intl conference on Autonomic computing*, pages 137–146, 2009.
16. M. Sugeno and T. Yasukawa. A fuzzy-logic-based approach to qualitative modeling. *IEEE Transactions on fuzzy systems*, 1(1):7–31, 1993.
17. R.S. Sutton and A.G. Barto. *Introduction to reinforcement learning*, volume 135. MIT Press Cambridge, 1998.
18. G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Intl Conf on Autonomic Computing*, pages 65–73, 2006.