A Fuzzy-based Auto-scaler for Web Applications in Cloud Computing Environments

Bingfeng Liu[1], Rajkumar Buyya[1], and Adel Nadjaran Toosi[2]

[1] School of Computing and Information Systems
The University of Melbourne, Australia
{bingfengl@student, rbuyya@}unimelb.edu.au
[2] Faculty of Information Technology, Monash University, Australia
adel.n.toosi@monash.edu

Abstract. Cloud computing provided the elasticity for its users allowing them to add or remove virtual machines depending on the load of their web applications. However, there is still no ideal auto-scaler which is both easy to use and sufficiently accurate to make web applications resilient under the dynamic load. The threshold-based auto-scaling approaches are among the most popular reactive auto-scaling strategies due to their high learnability and usability. However, the static threshold would become undesirable once the workload becomes highly dynamic and unpredictable. In this paper, we propose a novel fuzzy logic based approach that automatically and adaptively adjusts thresholds and cluster size for a web application. The proposed auto-scaler aims at reducing resource consumption without violation of Service Level Agreement (SLA). The performance evaluation is conducted with the real-life Wikipedia traces in the Amazon Web Services cloud platform. Experimental results demonstrate that our reactive auto-scaler efficiently reduces cloud resources usage and minimizes the SLA violations.

1 Introduction

The pay-as-you-go, elasticity, and on-demand nature of cloud resources are among the main features exhorting web service providers to host their application in clouds nowadays. Cloud users can rapidly launch Virtual Machines (VMs) for a particular time frame and only pay for their usage. In other words, they do not need to spend a significant amount of time and money to buy the actual hardware and pay maintenance fees [2, 5]. Since the rate of web application requests varies with time, it is often hard to determine the right amount of needed cloud resources [13]. Manually watching the load of the web-application is both tedious and resource-wasting led to the birth of auto-scaling. An auto-scaler automatically removes or adds the right amount of resources to optimize the performance and cost of the web-application.

The auto-scaling methods fall into two main categories: *reactive* and *proactive* approaches [16, 22]. The reactive approaches check if certain metrics (e.g., CPU utilization) exceed a threshold and provision VMs to meet the demand. The reactive methods only use recently observed metrics to understand the current state of the cluster and plan scaling decisions. The proactive approaches involve modeling and predicting the load of the web application based on the patterns of the historical data using machine learning techniques [16, 22]. However, proactive approaches are difficult to tune and often do not perform well for highly dynamic unforeseen workloads. The proactive methods need to continuously update their model to capture changes in the load patterns to predict future where the growth of the model complexity might become the burden of cluster resources.

Due to the simplicity and the adequate performance, most companies and organizations still prefer to use reactive approaches [16, 22]. In line with this view, we also focus on more commonly accepted reactive approaches in this paper. However, the problem of reactive approach is that the threshold does not change according to dynamic workloads and cluster state. Therefore, the naïve static threshold-based approaches often have oscillation problem where the auto-scaler frequently keeps changing the number of VMs back and forth to adjust the cluster size [22]. In this paper, we propose a reactive auto-scaling approach that aims at building dynamic thresholds to overcome these issues. Our approach dynamically adjusts thresholds based on the current cluster size and load.

To this end, the fuzzy logic looks very intuitive to use and unlike machine learning, it does not need historical data and long training time to build the model. One can set up fuzzy linguistic rules with metrics that are critical to satisfying the Service Level Agreement (SLA). For example, a rule can be as simple as "IF *cpu load* IS *low* THEN *cluster size* IS *small*". Two fuzzy engines are proposed in this paper. The first one uses request response time and cluster size to output the appropriate upper threshold value for the auto-scaler. If the current response time exceeds the dynamic upper threshold, then the second fuzzy engine with cluster size and CPU load as inputs comes into play to determine the cluster size avoiding the violation of the response time constraint required by the SLA.

The evaluation of the proposed auto-scaler is performed via a Wikijector agent [3] replaying the real-life requests from Wikipedia traces to stress the cluster and monitoring how auto-scaler adjusts the cluster size to handle this stress. The performance of the proposed approach is compared to both the Amazon Web Services (AWS)'s auto-scaler and its emulated version. The results demonstrate that the proposed auto-scaler significantly reduces cost and SLA violations.

2 Background

Cloud providers like AWS offer built-in auto-scaling services to their users for dynamic auto-scaling of resources used by applications [17]. AWS offers its auto-scaling service via *auto-scaling group* and uses *Cloud Watch* to monitor user-defined metrics [18]. The user can set alarms on metrics to trigger scaling policies once they are over certain thresholds. The AWS auto-scaler similar to many other approaches uses static thresholds and static steps for adding or removing VMs.

Most of the auto-scalers (including our proposed method) adopts *Monitoring*, *Analyzing*, *Planning* and *Execution* (MAPE) loop to adjust the size of the cluster to fulfill the SLA requirement [12, 16]. *Monitoring* is not only necessary for the decision making of the auto-scaler but also important for evaluating and improving its performance by logging all the relevant metrics data for offline analysis. In this work, *Ganglia* [14] is used for the monitoring purpose. The *Analyzing* phase is for making the gathered metrics meaningful and to pass them as the inputs to the auto-scaler to trigger the scaling policies at the right time. The *Planning* phase is a challenging phase in MAPE loop – there is no commonly agreed way to know how many VMs are needed to avoid over-provisioning or under-provisioning. The final step which is called *Execution* is responsible to perform the actual scaling-in or scaling-out processes according to the decision of the new cluster size made in the Plannig phase via the cloud provider's APIs.

3 Related work

There are many reactive auto-scaling approaches proposed in the literature. Lim et al. [11] focused on using the averaged CPU utilization of VMs to determine the number of VMs needed to be removed or added. They use a pair of threshold boundaries to make a range but not just a single limit, and then varied the lower bound of the thresholds to make sure the auto-scaler does not switch the number of VMs back immediately after changing the values of the thresholds. Hasan et al. [8] proposed a novel static threshold technique to prevent the oscillation problem for a short period of fluctuating load. The authors first choose static upper and lower bound and then select extra sub-upper and sub-lower bound in between. If the current metrics exceed any of the thresholds for a specific pre-set duration, the system starts a scaling-out or scaling-in the cluster.

The oscillation problem is a common issue in auto-scaling, and many solutions have been proposed to address it in the literature. The oscillation problem occurs when request load changes slightly and triggers the scaling-out process to add a new VM into the cluster. However, the newly added VM causes a load drop, and the new cluster size becomes too powerful for the current load which in turn triggers a scale-in process in a short period. Unlike [8] and [11], our proposed auto-scaler adaptively updates the upper-threshold based on the cluster size. The main idea is that a larger cluster has more tolerance to small changes in the load and upper threshold is set to a higher value.

Frey et al. [7] developed a fuzzy logic auto-scaler to improve Quality of Service (QoS). The result of the fuzzy auto-scaler shows the technique is effective in reducing the response time. Their method uses inputs such as 'high prediction' (request load) and the 'slope of the response time'. The high prediction input helps the cluster perform scaling earlier and the slope of response time helps to add multiple VMs to reduce the response time. The high prediction input in [7] is formed based on the knowledge of the application's history load pattern whereas the proposed auto-scaler does not assume a priori knowledge of the application.

Lama et al. [10] proposed an adaptive fuzzy system using machine learning without offline learning. The neural network dynamically generates fuzzy rules, membership functions and input parameters with online learning algorithms. Contrary to our approach that focuses on adapting thresholds, they focus on constructing the fuzzy control structure and adapting control parameters using online learning approaches. In their approach, fuzzy engine has to adapt itself to changes in the load that takes time and could seriously affect the small web applications that are growing popularity [16, 22]. Jamshidi et al. [9] also developed an adaptive Fuzzy logic based auto-scaler with Fuzzy Q-learning method to modify the fuzzy rules at run time so that defining fuzzy rules are not longer a manual process. Arabnejad et al. [1] enhanced the output (actions) of the fuzzy controllers with reinforcement learning. The reinforcement learning uses the current state of the system, and randomly tries out different actions to set the actions of the fuzzy controllers. It takes a random approach to explore all the possible actions until the model cannot improve the overall performance. Müller et al.[15] built a governance platform to satisfy multiple users' QoS requirements where users are routed to different provisioning levels depends on their SLAs.

Kubernetes is a cluster manager for containerized applications (e.g. applications run in docker). Kubernetes currently (at the time of writing this paper) supports horizontal auto-scaling for pods, a group of containers deployed together on the same host. The auto-scaling algorithm is used to keep CPU utilization at a target level. The algorithm determines the number of pods by dividing the sum of CPU utilization in the pods by the target CPU utilization. In cloud environment, users provision cloud resource in form of of VMs. Thus, our work focuses on auto-scaling of VMs. However, our method can be used along with container cluster management for the management of web application micro services. Baresi et al. [4] enhanced their framework to be able to deploy multi-tier applications with Docker containers. The framework performs resource management with control theory to satisfy SLA (response time) and use cloud resources (CPU cores) effectively.

In web applications, the correlation of CPU utilization and response time might not be strong. The trigger should be driven by the response time since it is possible that the SLA is well satisfied while the CPU utilization triggers unnecessary scaling. The service-time-related metrics of the application are better than the metrics which only show the state of the VMs [16]. This is also why [11], [8], [7] and [10] tend to use response time to benchmark their auto-scaler.

Contrary to our approach, most of the auto-scaling studies only focus on the *Analysis* part of the MAPE loop; they do not mention much about the planning phase and how many VMs should be added or removed if the threshold is triggered (including [11] [8] [7] [10]). The time taken to start new VM is usually around several minutes [12], so naïvely launching a single VM once at a time might not be feasible to solve the under-provisioning issue [16].

4 Fuzzy Auto-scaler

In this section, we propose our fuzzy logic-based approach for auto-scaling of web applications. First, we identify inputs and metrics that are relevant to the auto-scaling. Then, we discuss the importance of dynamically adapting the upper threshold. Finally, we propose our algorithm to set the right size for the cluster.

Table 1: Request Details for Figure 1

Request	**Url**	**Response Body Size (Byte)**	**Request Type**
1	index.php?action=raw&title=Cliff_Clavin	6867	Text
2	index.php/Main_Page	13397	Text
3	index.php/Bufotenin	30930	Text
4	index.php/List_of_dog_breeds	65521	Text
5	skins/common/ajax.js?99	5320	Javascrpit

4.1 Input Selection

The most important aim of autoscaling is to maintain the response time below a certain level to provide a satisfactory experience to users. Therefore, the use of the response time as an input to the auto-scaler is highly desired. The response time alone does not provide us information regarding how many more VMs are needed to avoid SLA violations. The naïve approach is to launch a small static number of VMs to see if the response time is lowered to the desired range. However, the time of booting up and configuring the newly launched VMs is often too long that would lead to SLA violation. The long booting time of VMs forces the auto-scaler to make the best decision with a minimal number of trials.

CPU utilization can be used to determine the cluster size. However, the problem with CPU utilization is that it does not reflect the actual overall load of the VMs accurately. For example, consider a web-application like Mediawiki, which is I/O intensive rather than CPU intensive. The use of CPU is low during the I/O processes hence a delusion of the low CPU utilization happens, while the load of the VM would be high. In addition, the maximum CPU utilization of a single core VM only goes up to 100%, so it is hardly useful to determine how many more VMs are needed.

Service rate, the rate at which requests are processed, also sounds relevant to determine the cluster size. The tools like 'Jmeter'[3] or 'Ab'[4] can be used to flood the VMs to profile the maximum number of requests that can be processed by a VM. This number can be used to set up the optimal number of required VMs. However, the service rate is highly dependent on the complexity of the request's response body. Table 1 shows a group of 5 Mediawiki requests with various response body sizes and request types. Figure 1 shows that the larger response body size (Table 1), the smaller service rate (Request 1 to 4). However, the size of the response body is not the only factor which could affect the service rate. The request 5 has similar response body size to request 1 but the maximum number of processed requests per second is much higher. The reason is that Request 5 is a JavaScript file, so it needs lower computations and I/O operations to produce the response and hence has lower complexity than request 1. It is hard to define the complexity of the request since the content of the page varies greatly. Therefore, the use of service rate to determine the cluster size is not effective if the request received by the VM has a huge difference to the requests used for profiling.

[3] http://jmeter.apache.org/
[4] https://httpd.apache.org/docs/2.4/programs/ab.html

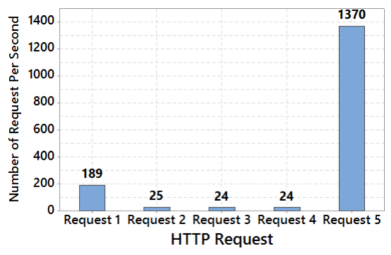


Fig. 1: Jmeter load test with different requests

4.1.1 Metrics Correlations with Response time

We conduct a load test to see how the changes in the metrics are correlated. Wikijector is used to send requests from Wikipedia trace files to the load balancer instance. We apply a simple scaling out rule that only adds one new VM each time the response time is over the static upper threshold. According to the test, Table 2 shows Pearson and Spearman Rho correlation value of each metric to request response time. Results demonstrate that the CPU load has the highest Pearson and Spearman Rho correlation value to the response time. Thus it could be a potentially useful metric for determining the cluster size. While CPU utilization reflects the CPU usage, the CPU load metric can give us the actual demand of the computer's resources (not only CPU). Unlike CPU utilization, CPU load can go beyond 100%. The analogy between CPU utilization and load is the high way traffic. CPU utilization is how often the freeway has cars running on it, whereas CPU load is how many cars are both running on and waiting to enter the freeway [21].

Table 2: Metrics correlations with Response Time (P-value represents the significance of the correlation)

Metric	Pearson value	p-value	Spearman value	p-value
CPU load last minute	0.42	0.00	0.92	0.00
CPU System	0.11	0.12	0.80	0.00
CPU User	0.08	0.30	0.80	0.00
CPU IO Wait	0.26	0.00	-0.08	0.25
Memory cached	-0.24	0.00	-0.25	0.00
Number of Request per second	-0.28	0.00	0.27	0.00

4.2 Dynamic upper threshold

The main idea behind the fuzzy rules for the dynamic upper threshold is that when the cluster size is large (e.g. 100 web-servers running), then a moderate change of the load will only fluctuate the response time in a small amount so the fuzzy engine should output a higher upper threshold of response time. Whereas, when there is only one running instance, the cluster is very likely to violate the SLA (large fluctuation) with even a small amount of request load increase. In this case, the threshold should be more sensitive (lower). Therefore, the inputs of the fuzzy engine are chosen as the current normalized *response time*, a ratio to the maximum allowed response time in SLA, and the normalized *cluster size*, a ratio to the maximum cluster size.

4.3 Dynamic cluster size

In an ideal situation, a fully utilized single CPU core VM has CPU load at 100%, and all CPU load beyond 100% means the VM is overloaded. Note that, 100% CPU load only means that the VM is fully loaded, but it can still provide a decent response time. In a single core computer like *t2.micro*, the 100% of the CPU load indicates that the computational resource is fully utilized and if the value is 200%, it indicates that we need one more instance to handle the load.

The scaling-in approach used in the proposed auto-scaler is conservative but dynamic. We scan through all launched VMs and stop all the VMs which have CPU load below a certain threshold (e.g., 50%) instead of pre-defining the number of VMs to scale-in. Since, a single type of instance is used in our approach, thus the total CPU load of the cluster not overloading VMs is calculated by:

$$TotalCPULoadNotOverloading = MaxNo.ofInstance \times 100\% \tag{1}$$

The cluster stress value is the ratio of the average CPU overload value (CPU load beyond 100%) to the summation of the CPU load of a cluster without overloads (equal and below 100% CPU load) and is calculated as:

$$ClusterStress = \frac{AverageCPUoverload}{TotalCPULoadNotOverloading} \tag{2}$$

For example, if we have 2 VMs and they have CPU load of 150% and 200% respectively, the average CPU overload value will be (150+200)/2 - 100% = 75%. If we only allow a maximum of 5 VMs in the cluster, then we will have total CPU load without overloading VMs of 500% (5 * 100%). Therefore, the cluster stress is equal to 75/500 = 0.15. The cluster stress represents how huge the average CPU overload is with respect to the total CPU load that the cluster can accommodate without overloading VMs.

For scaling out, we use the cluster stress value and the normalized current cluster size as inputs to the fuzzy engine. We use these inputs to calculate the needed number of VMs. Since the user's budget is often limited, a maximum number of VMs is set for the cluster. The output of the fuzzy engine, *NewClusterPower*, is the ratio of the cluster size to the maximum value. Based on this ratio, the new number of VMs is calculated as:

$$\lceil NewClusterPower \times MaxNumberOfVirtualMachines \rceil \tag{3}$$

The correlation between CPU load metric and response time is not 100%, so the average CPU overload value cannot be fully trusted to perform scaling out. Therefore, the output of the cluster size fuzzy engine for scaling out should consider the current cluster size. The fuzzy engine acts conservatively by adding less number of VMs determined from the average CPU overload value when the cluster size is small and also when the cluster size is near the maximum.

4.4 The proposed Auto-scaling Algorithm

Algorithm 1 shows the overview of how adaptive response time threshold and dynamic cluster size work together to perform auto-scaling. Two fuzzy engines

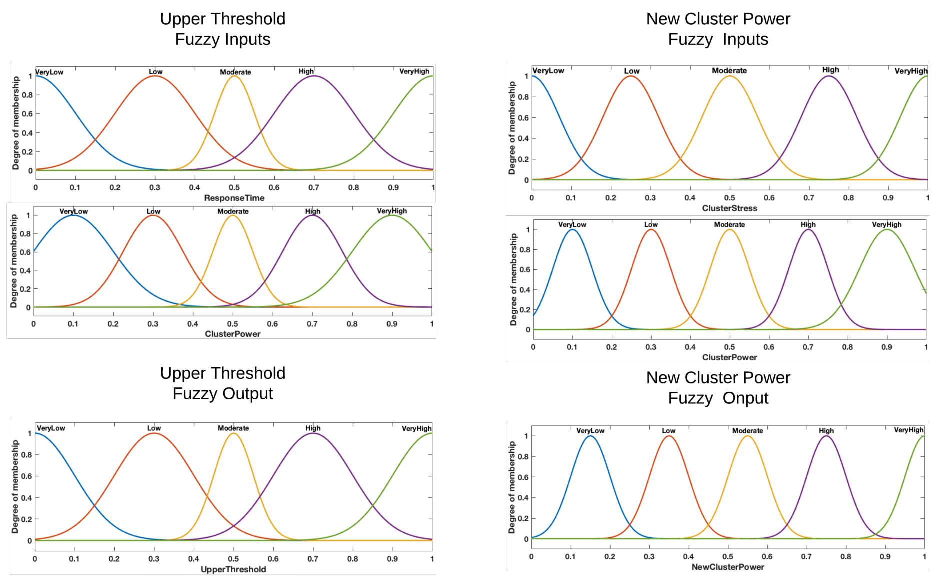


Fig. 2: Fuzzy engines for setting upper-threshold and adding new virtual machines

are developed: *upperThresholdFuzzy* and *clusterSizeFuzzy*. Both of them use the Mamdani fuzzy inference system and the defuzzification method of the centroid of the area to produce the crisp output value. All the membership functions for mapping input values (all of them are from 0 to 1) to fuzzy sets are Gaussian membership functions as shown in Figure 2. Sample fuzzy rules used in our fuzzy engines are specified in Tables 4 and 3.

Table 3: Partial add Virtual Machines fuzzy rules

ClusterPower	ClusterStress	NewClusterPower
VeryLow	VeryLow	VeryLow
VeryLow	Low	Low
Low	VeryHigh	Moderate
Moderate	Moderate	High
High	High	VeryHigh
VeryHigh	VeryHigh	VeryHigh

Table 4: Partial Upper threshold fuzzy rules

ClusterPower	ResponseTime	UpperThreshold
VeryLow	VeryLow	VeryHigh
VeryLow	Low	VeryLow
Low	Moderate	Low
Moderate	VeryHigh	VeryLow
High	High	High
VeryHigh	VeryHigh	High

The algorithm first gathers the current average response time ($RTime$), CPU Loads ($curLoad$) and the cluster size ($CSize$) from the load balancer instance (Line 3). After gathering all metrics needed, the current response time and the current cluster size are used by the *upperThresholdFuzzy* engine to set the upper-threshold (Line 4). If the current response time breaches the adaptive upper threshold twice consecutively (Lines 5 and 6), the *clusterSizeFuzzy* engine is called (Line 7) to determine the new cluster size. For each scaling out or in, the $warmUpTime$ is set to a certain value (e.g., 30 seconds) which means we do not allow any scaling policies to be executed in this period. The reason for setting $warmUpTime$ is that the newly launched VM normally has an unstable metrics due to the operating system's bootstrapping and all the configurations for the applications. For scaling in, we loop through each running VM and check if their CPU load is under *idleLoad* value, e.g. 50% (Line 17) and if the same VM CPU load is under *idleLoad* twice consecutively (Line 19), it will be stopped and the *warmUpTime* is reset. The scaling in/out loop is executed periodically, e.g. every 5 seconds (Line 27), that is because checking the metrics too frequently adds pressure to VMs, and often many metrics do not have high-resolution updates.

Algorithm 1 Fuzzy Auto-scaler

```
Require: idleLoad, checkTime, consecutiveOverThreshold, warmUpTime
 1: procedure StartScaling
 2:     while true do
 3:         RTime, CSize, curLoad ← AllLaunchedInstances.getStats()
 4:         $upperThreshold$ ← upperThresholdFuzzy(RTime, CSize)
 5:         if $RTime >$ upperThreshold then
 6:             if $consecutiveOverThreshold == true$ and $warmUpTime <= 0$ then
 7:                 newClusterSize ← clusterSizeFuzzy(curLoad,CSize)
 8:                 scalingUp(newClusterSize)
 9:                 Set $warmUpTime$
10:                 $consecutiveOverThreshold \leftarrow false$
11:             else
12:                 $consecutiveOverThreshold \leftarrow true$
13:         else
14:             $consecutiveOverThreshold \leftarrow false$
15:         if warmUpTime <= 0 then
16:             for Instance in AllLaunchedInstances do
17:                 if Instance.load < idleLoad then
18:                     $Instance.consecutiveBelowThreshold+ = 1$
19:                     if $consecutiveBelowThreshold == 2$ then
20:                         $Instance.stop()$
21:                         Set $warmUpTime$
22:             if $warmUpTime! = 0$ then
23:                 setConsecutiveBelowThresholdToZero(AllLaunchedInstances)
24:         else
25:             setConsecutiveBelowThresholdToZero(AllLaunchedInstances)
26:         sleep(checkTime)
27:         $warmUpTime- = checkTime$
```

5 System Prototype

The instances used in this system are all *t2.micro* since it offers the free-tier policy (no costs of using VMs) and also keeps the system homogeneous, which lowers the system overheads for evaluating the fuzzy auto-scaler (Figure 3).

The HAproxy instance (load balancer instance) receives all the HTTP requests from the users first; then it evenly distributes them to each Mediawiki-server instance. The HAproxy instance also hosts the master node of the Ganglia monitoring system which is used to gather all the instances' current performance metrics (e.g. CPU load). Since all the requests go through the HAproxy instance first, the HTTP request related metrics like request response time could be retrieved from the HAproxy. The fuzzy auto-scaler communicates to the HAproxy instance to gather metrics of all the AWS t2.micro instances and processes them locally to decide on whether to perform scaling process and to determine the suitable size of the current Mediawiki-server cluster needed to satisfy the SLA. The orchestration of the Mediawiki web-application cluster is controlled by the

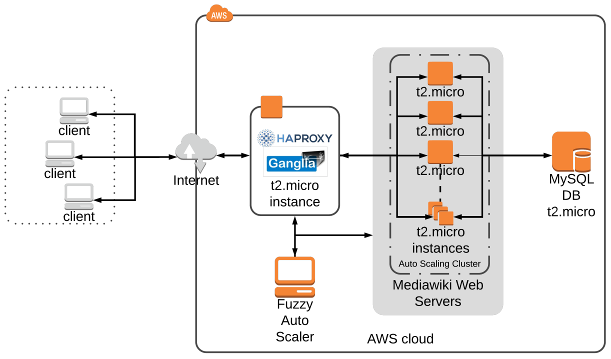


Fig. 3: System architecture diagram

fuzzy auto-scaler instance which uses Python Boto3 (AWS SDK for Python [19]) library to manipulate the AWS cloud resources. The Fuzzy auto-scaler instance starts or stops VMs based on the metrics retrieved from the HAproxy instance. The automation process of configuring different instances are done with Ansible [6] tool which is also executed in a Fuzzy auto-scaler instance.

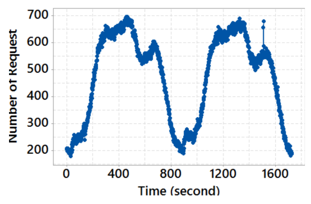
6 Performance Evaluation

6.1 Experimental Setup

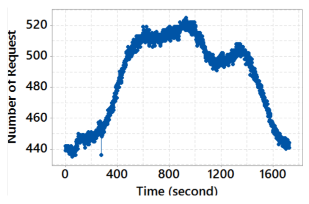
Each experiment is carried out for 30 minutes and is repeated for three times. We report the best result obtained by each auto-scaler. The AWS t2.micro instances used for the experiments have only 30 CPU credits with 1 point per minute for the full power CPU usage. Once the CPU credits are exhausted the VM will be restricted to a baseline (10% CPU utilization) [20]. The trace files used in the experiments to evaluate the auto-scaler are all produced from the Wikipedia workload trace file. The 'shape' of the original trace is preserved while the time dimension is scaled down to 30 minutes.

The AWS emulated auto-scaler is a copied version of the AWS auto-scaling policy which uses Ganglia instead of AWS CloudWatch for monitoring. The reason for adding it is that the AWS native auto-scaler does not offer fine grain control and monitoring data for validating scaling policies.

In experiments, we have a maximum limit of 10 t2.micro instances, and the desired response time is set to 200ms for the SLA. For the AWS native and emulated auto-scalers, the upper threshold is 100ms (static), and the fuzzy auto-scaler uses the adaptive upper-threshold from the fuzzy engine. Since AWS native auto-scaler cannot read individual VM's metric to stop a VM, we set a lower threshold of 70% CPU load (average of all launched VMs) for both AWS native and emulated auto-scalers, while the fuzzy auto-scaler has a lower threshold of 50% for each VM. We also deliberately use a cluster size of two VMs at the beginning of the experiment to show the differences of each auto-scalers. Thus, there will be many SLA violations at the begining.



(a) Two full day trace of Wikipedia



(b) The one day trace of Wikipedia

Fig. 4: The trace files used in the experiments, both of them are 30 minutes long

6.2 Experimental Results

The trace file used in experiments (Figure 4a) is extracted from the two day Wikipedia trace and is scaled down into 30 minutes. Figure 5 shows the CDF (Cumulative Distribution Function) of the average response time below 200ms for all auto-scalers. The fuzzy auto-scaler preforms about **8.94%** better than the AWS emulated auto-scaler (adding one VM each scaling out) and **16.28%** better than the AWS native auto-scaler. The cost of each auto-scaler can be calculated from the area under the step-curve in 'Number of Instance vs Time' scatter graph. The cost of the AWS native one is the lowest since it often under-provisions resources. The cost of the fuzzy auto-scaler is similar to the AWS emulated auto-scaler, but the fuzzy auto-scaler provides better results in terms of keeping the response time under 200ms.

The number of instances versus time scatter graphs show that the fuzzy auto-scaler performs a lower number of scaling out actions to reach the desired cluster size, whereas AWS native auto-scaler and emulated auto-scaler with adding one VM require more time to do the same. The amazon emulated auto-scaler with adding two VMs options is the fastest in this regards. However, it causes the unnecessary cost to provide the same performance. Therefore, adding multiple VMs during the scaling-out is very important to process the web request load. However, the lack of prediction ability of the fuzzy auto-scaler leads in several trials to gradually reach the desired cluster size. The initial cluster size is small, so it can quickly get flooded with requests and metrics raise up to the maximum. Hence using metric values is not reliable to determine the actual request load. Thus, the method of measuring the current request load needs further investigations.

Table 5 shows results summary. Compared to AWS native auto-scaler, AWS emulated auto-scalers with adding one and two VM(s) each scaling out and fuzzy auto-scaler generate 7.88%, 28.43% and 17.17% higher cost respectively. However, with 1% increase in SLA satisfaction, the fuzzy auto-scaler only requires 1.05% (17.17/16.28) increase in cost while AWS eumulated auto-sacalers with adding one and two VM(s) each scaling out need 1.17% (7.88/6.74) and 1.46% (28.43/19.53) increase in cloud resource usage respectively. The fuzzy auto-scaler use resources more efficiently to provide better SLA satisfaction. The AWS na-

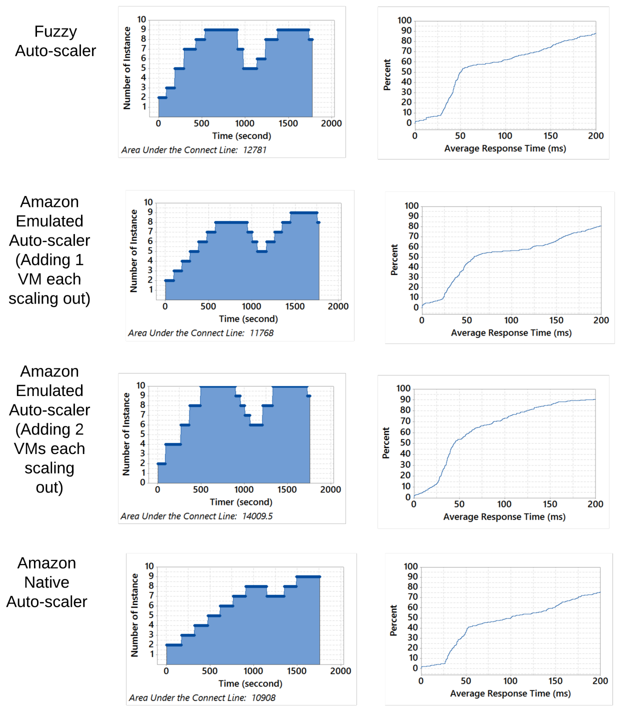


Fig. 5: Resource usage and average response time for different auto-scalers

tive auto-scaler and AWS emulated auto-scaler adding one VM each scaling out have overall lower cost than fuzzy auto-scaler. However, they cause more SLA violations by under-provisioning resources.

Table 5: Experimental results summary

Auto-scaler	**% of Requests with Response Time Under 200ms (Improvment to AWS Native)**	**The Area Under the Graph or Cost (Improvment to AWS Native)**
Fuzzy Auto Scaler	87.97 (16.28%)	12781.00 (17.17%)
AWS emulated (adding 1 VM)	80.75 (6.74%)	11768.00 (7.88%)
AWS emulated (adding 2 VMs)	90.43 (19.53%)	14009.50 (28.43%)
AWS native	75.65 (0%)	10908.00 (0%)

6.3 Dynamic scaling-out evaluation

Figure 5 shows the downside of using a static number to perform scaling out. The AWS emulated auto-scaler (adding 2 VMs each scaling out) reached the maximum number of 10 VMs whereas in the same experiment only the maximum of 9 VMs used by fuzzy auto-scaler without violating the upper threshold. This

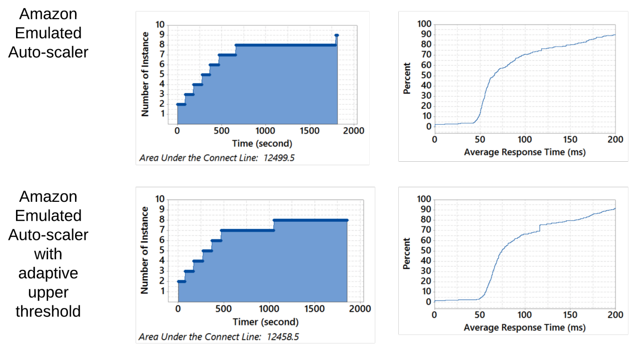


Fig. 6: The adaptive upper threshold experiment result

happens due to the lack of ability to dynamically adjusting the added number of VMs in the cluster size. Even though AWS emulated auto-scaler (adding 2 VMs each scaling out) has a better result of not violating the SLA (90.43%), but it also uses the resource less efficiently compared to the fuzzy auto-scaler in order to reach this result. With respect to the AWS native auto-scaler, AWS emulated auto-scaler (adding 2 VMs each scaling out) needs **1.46%** more resources to achieve 1% increase of the response time under 200ms while fuzzy auto-scaler only needs **1.05%**.

6.4 Adaptive upper-threshold evaluation

In order to evaluate the impact of an adaptive upper threshold, we use Amazon emulated Auto-scaler with and without the adaptive upper threshold fuzzy engine. The static threshold is set to 100ms for the one without the fuzzy engine. The change in the request rate of the trace shown in Figure 4a is considerably quick which decreases the chance of observation of adaptive upper-threshold benefits. Thus, a 30 minute version of the one day trace file is made (Figure 4b).

As shown in Figure 6, the adaptive upper threshold keeps the cluster at the size of 7 longer than non-adaptive threshold AWS emulated auto-scaler that leads to **0.33%** cost saving. The percentage of response time under 200ms is close in this case while AWS emulated with adaptive threshold has **91.74%**, and the non-adaptive upper threshold one has **90.14%** which shows **1.78%** improvement. Due to the restriction of the experiment time, the cost saving is not clearly observable, but we expect that the dynamic adaptive upper threshold utilizes cloud resources more efficiently in a long run since small load fluctuation is typical in a long-running web application.

It is worth mentioning that the load prediction can enhance the number of trials to reach the desired number of virtual machines quicker when encountering the seasonal loads. The experiment results also shows that the response time is decreased while adding more Mediawiki instances so that in our experiments the single MySql database instance in not a bottleneck for the Mediawiki cluster performance.

7 Conclusions and Future Work

In summary, a dynamic upper-threshold auto-scaler for the web application is proposed in this paper. We designed and presented two fuzzy engines to dynamically adjust upper threshold and cluster size respectively. The fuzzy logic approach increases the percentage of the service rate within the SLA target by 16.28% in comparison to AWS native auto-scaler. The experimental results suggest that threshold and the added number of VMs should be dynamically selected since the static values are not always right choices for all cluster sizes.

The creation of the optimal auto-scaler is far from trivial. In the future, we will explore techniques to determine the best candidate VMs for scaling-in. Scaling-in is riskier than scaling-out, as over-provisioning only costs money, but under-provisioning means losing customers. The auto-scaling of session-based web applications, where users have sticky sessions needs more attention since it requires session migration. It will be interesting to perform a more in-depth investigation of the correlation between proposed metrics which allows us to design more efficient auto-scalers. In the future, we will also try to implement different auto-scaling methods and benchmark them against our fuzzy auto-scaler with different workloads.

References

[1] Hamid Arabnejad et al. “A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling”. In: *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on.* IEEE. 2017, pp. 64–73.

[2] Michael Armbrust et al. “A view of cloud computing”. In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[3] Erik-Jan van Baaren. “Wikibench: A distributed, wikipedia based web application benchmark”. In: *Master’s thesis, VU University Amsterdam* (2009).

[4] Luciano Baresi et al. “A discrete-time feedback controller for containerized cloud applications”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM. 2016, pp. 217–228.

[5] Rajkumar Buyya et al. “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility”. In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.

[6] Michael DeHaan. *Ansible.* 2018. URL: `https://www.ansible.com/` (visited on 02/28/2018).

[7] Stefan Frey et al. “Cloud QoS scaling by fuzzy logic”. In: *Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E).* IEEE. 2014, pp. 343–348.

[8] Masum Z Hasan et al. “Integrated and autonomic cloud resource scaling”. In: *Proceedings of 2012 IEEE Network Operations and Management Symposium (NOMS).* IEEE. 2012, pp. 1327–1334.

[9] Pooyan Jamshidi, Claus Pahl, and Nabor C Mendonça. “Managing uncertainty in autonomic cloud elasticity controllers”. In: *IEEE Cloud Computing* 3 (2016), pp. 50–60.

[10] Palden Lama and Xiaobo Zhou. “Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee”. In: *Proceedings of 2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2010, pp. 151–160.

[11] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. “Automated control for elastic storage”. In: *Proceedings of the 7th international conference on Autonomic computing*. ACM. 2010, pp. 1–10.

[12] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. “A review of auto-scaling techniques for elastic applications in cloud environments”. In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592.

[13] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. “Auto-scaling techniques for elastic applications in cloud environments”. In: *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12* (2012).

[14] Matt Massie et al. *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale.* " O’Reilly Media, Inc.", 2012.

[15] Carlos Müller et al. “An elasticity-aware governance platform for cloud service delivery”. In: *Services Computing (SCC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 74–81.

[16] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. “Auto-scaling web applications in clouds: a taxonomy and survey”. In: *In: ACM Computing Survey (2018)* (2016).

[17] Amazon Web Services. *Amazon Web Services (AWS) - Cloud Computing Services*. 2018. URL: `https://aws.amazon.com/` (visited on 02/28/2018).

[18] Amazon Web Services. *AWS Auto Scaling*. 2018. URL: `https://aws.amazon.com/autoscaling/` (visited on 02/28/2018).

[19] Amazon Web Services. *AWS SDK for Python (Boto3)*. 2018. URL: `https://aws.amazon.com/sdk-for-python/` (visited on 02/28/2018).

[20] Amazon Web Services. *CPU Credits and Baseline Performance*. 2018. URL: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-credits-baseline-concepts.html` (visited on 02/28/2018).

[21] Ray Walker. “Examining load average”. In: *Linux Journal* 2006.152 (2006), p. 5.

[22] Lenar Yazdanov. “Towards auto-scaling in the cloud: online resource allocation techniques”. PhD thesis. Dissertation, Dresden, Technische Universität Dresden, 2016.