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

Bingfeng Liu

Supervised By Dr Adel Nadjaran Toosi

Submitted in partial fulfilment of the requirements of the degree of

Master of Computer Science

Student Number: 639187
Total Credits: 75 points
Conventional research project
Currently enrolled in COMP90070

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

June 2018

Abstract

CLOUD computing provided the elasticity for the cloud 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 dynamic load. The threshold-based auto-scaling approach is among the most popular reactive auto-scaling strategies due to its high learnability and usability. But the static threshold would be undesired once the workload becomes dynamic. In this thesis, we propose a novel fuzzy logic based approach that automatically and adaptively adjusts thresholds and cluster size. The proposed auto-scaler aims at reducing resource consumption without violation of Service Level Agreement (SLA). The performance evaluation is conducted with the Wikipedia traces in the Amazon Web Services cloud computing platform, and the experimental results demonstrate that our reactive auto-scaler can efficiently save cloud resources usage and minimize the SLA violations.

Declaration

I certify that

- this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text,
- where necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the Department,
- the thesis is 17964 words in length (excluding text in images, table, bibliographies and appendices).

[illegible]

02/06/2018

Bingfeng Liu, June 2018

Acknowledgements

On the journey of completing my master of computer science course, I went through many challenges that I have never faced before and I had no research experience. It was the first time for me trying so hard to understand every related paper I found; it was the first time for me to think really creatively but also practically at the same time to develop an auto-scaler that can be done within three semesters; and it was also my first time to stay up really late for many times during this research journey and see tomorrow's sunrise.

I still remember the email that I sent to Dr Adel Nadjaran Toosi asking him if I could be his master research student. It was also the master introductory subject, distributed system, taught by him that opened up my sight that the distributed system is the way of the future computing. None of us can achieve anything great by ourselves, all the mankind's greatest work is mostly done by collaborating with each other and so does computers. I am so lucky that Dr Adel Nadjaran Toosi was willing to guide me throughout my research journey.

At the beginning of my research work, I had no idea what to do in cloud computing. But that was not too bad, at least I knew which area I wanted to focus on. After reading some resources suggested by Dr Adel Nadjaran Toosi I finally decided to go with auto-scaler. I am again very lucky that I think I chose an area I like, as it is fascinating to know that one day we can have computational resources provisioned to our application automatically, and it never went down because of the lack of computational power.

Dr Adel Nadjaran Toosi always suggested me some insightful ideas on developing the auto-scaler proposed in this thesis, without him I would have been struggling in completing this thesis. I want to say a big thank you here to my supervisor, Dr Adel

Nadjaran Toosi. Thank you for your great patience in finding papers for me, answering my questions, proofreading my thesis and spending time every week to meet me.

The research work is sometimes tedious and frustrating. There is always an endless number of papers you must read, and the new technologies you need to practice and learn. However, I am so lucky that I got very supportive family members who support me continuously and honestly believe in my capability on completing this thesis.

I was lucky to have my girlfriend always be with me every day and night during my master journey. She was the one who took care of my daily life. I could not have enough energy if you did not cook for me, I would have given up writing the thesis and performing experiments very early if you didn't comfort me whenever I felt frustrated and could not proceed in my research work. You were the one who genuinely listened to my frustration and complaints. Thank you so much for supporting me.

I want to thank the University of Melbourne for helping me to be who I am today. For all the five and half years, thank you for passing me the knowledge, thank you for challenging me so hard, thank you for giving me a chance to meet my teachers, friends and my true love.

Finally, Thank you all who presented in my life.

Bingfeng Liu
Melbourne, Victoria, Australia
13/05/2018

To my family, friends and supervisor.

This page intentionally left blank.

Contents

List of Figures

This page intentionally left blank.

List of Tables

Chapter 1

Introduction

1.1 What is cloud?

CLOUD is a vast pool of computational resources that people can pay money to rent the remote computers to meet their needs. For example, hosting website, processing data or playing the game can all be done with the use of the cloud computational resources (Figure 1.1). The cloud provider is the company or organization that owns a massive amount of physical computational hardware. By using the virtualization techniques, the cloud provider is able to divide the hardware into multiple isolated virtual computers for the cloud users to use. There are generally three different clouds namely public cloud, private cloud and hybrid cloud. The public cloud can be paid and accessed by the general public to meet cloud users' requirements [4]. The private cloud is used internally by a specific group of people within organizations. The hybrid cloud is a notion that the public cloud can borrow the idle resources from private cloud in needs for the public cloud user to use.

Cloud service is categorized into three different service model, namely IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and SaaS (Software as a Service), the flexibility of controlling the actual cloud hardware resource are decreasing. The IaaS user has the most flexibility and direct access to the cloud computational resource [4]. We can rent virtual machine (remote computer) to run whatever software we want. However, IaaS requires higher skill on management the computational resources. The thesis will mainly focus on IaaS which is creating an auto-scaler to provisioning virtual machines from IaaS provider such as AWS cloud platform.

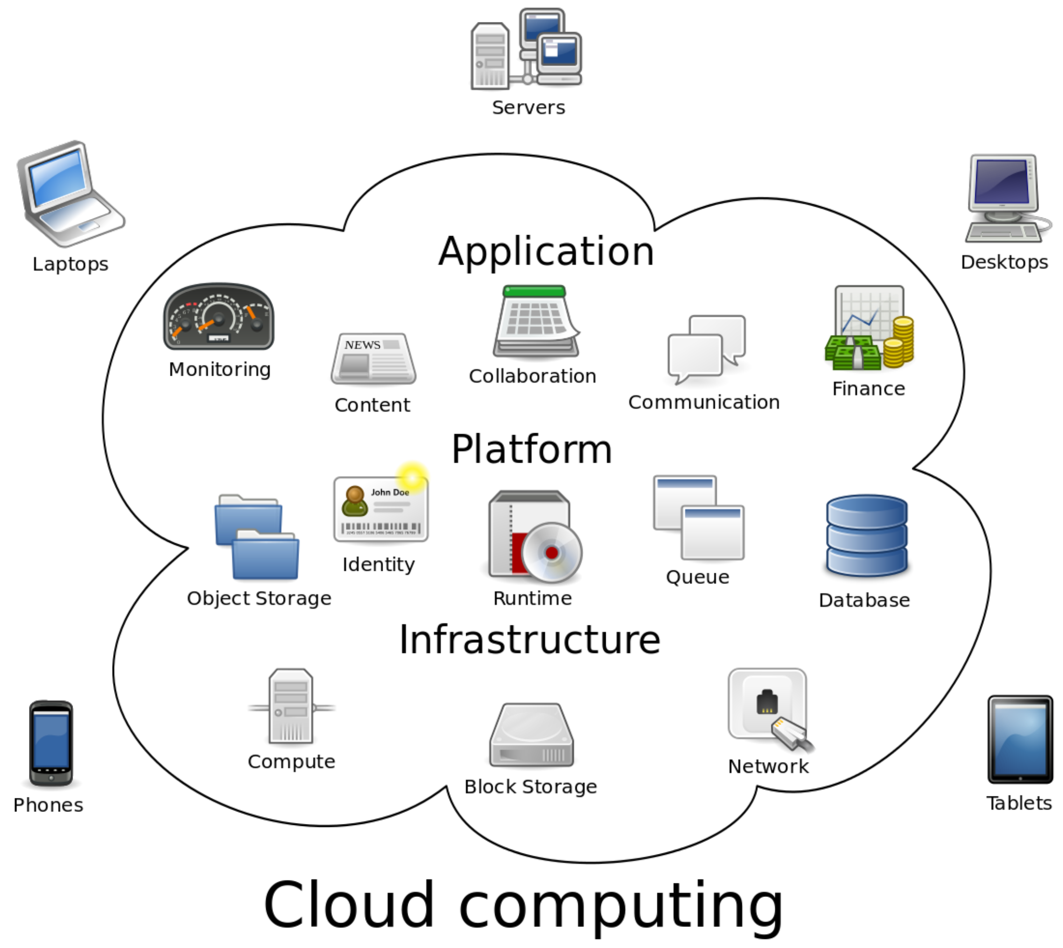


Figure 1.1: Cloud Computing [1]

For PaaS we have less flexibility in controlling the remote computational resource, the PaaS cloud provider only provided the platform for us to run specific application or code that we created. PaaS provider is responsible for maintaining and provisioning the application on our behalf. For example, one can use Haruko to host the web application, but he/she needs to follow their rules on how to deploy the web application.

SaaS user generally has no control over the remote computational resource. The user pays the software provider for accessing some services remotely, which means you do not have to install actual software on your personal computer locally (i.e. you can access them from the browser via internet). The most common SaaS example is Google Doc which helps you to keep your document in the cloud so that you will not lose your data due to the failure of your personal computer. For accessing the SaaS, we need the internet connection which is generally not a problem nowadays.

The core idea of the cloud is that you can delegate the computational task of running software to some remote computers that are reliable and robust. The cloud user only needs a computer that can access the internet to control the powerful cloud resources.

1.2 Why Cloud is attractive?

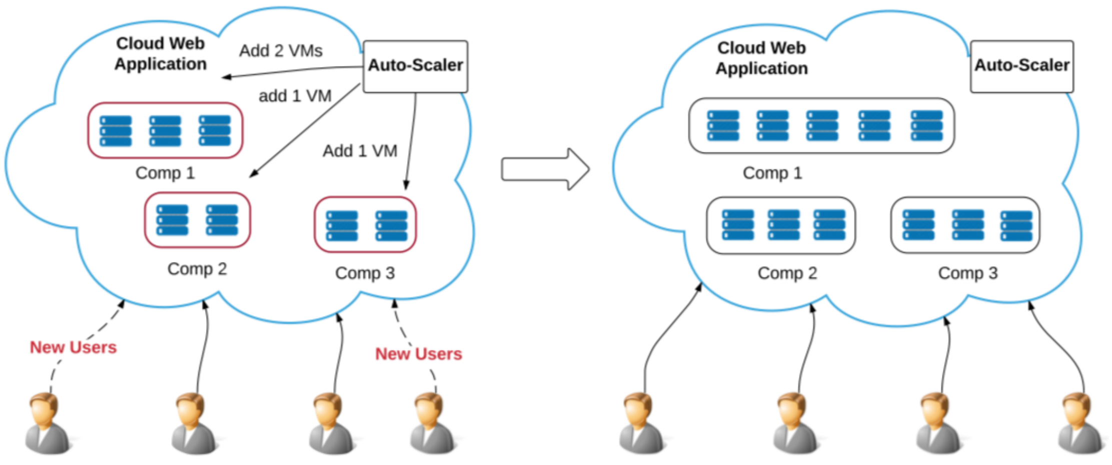
The cloud saves users from buying actual physical hardware, there would not be any trouble maintaining the physical hardware and the waiting time of installing the hardware is also eliminated. The cloud only charges us based on our usage of the cloud resource just like the traditional utility (e.g. electricity) [5, 6]. Cloud users only need to pay the use of the computational resource based on the time and the type of the remote computer resource used.

The cloud resource can be ubiquitously accessed and the device you own can be very weak on computational power, for example your mobile phone. The cloud resource can be easily accessed and controlled anywhere when you are connected to the internet. You can push the computational tasks to the cloud and have the results sent back to your device that might not be computationally powerful enough to process the tasks.

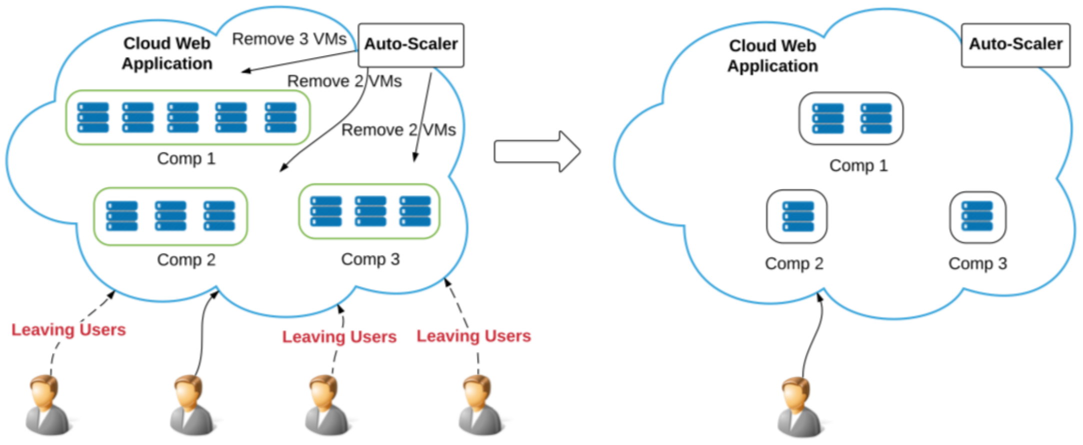
The pay-as-you-go is another leading feature that attracts people to host their application in the cloud nowadays. The elasticity of the cloud allows the users to *scale out* and *scale in* (i.e. add and remove virtual machines) their web application on demand. The cloud platform provides 'infinite' (more than you need) computational resources to prevent the computational task you run from the shortage of the computational resources. The client could simply acquire resources to use from 100 servers to 10000 servers when needed and release them afterwards. Using 1000 servers for one hour simultaneously cost the same as running one server for 1000 hours [5].

The on-demand service of the cloud is very suitable for small and medium organizations who neither want to lose clients nor pay an excess fee to handle short duration of the peak usage of the computational resources. However, it is still hard for the cloud users to know how much resources they need, since the number of web-application requests changes with time [7]. For example, web applications might receive fewer requests during the night while the maximum load might occur in the afternoon.

1.3 What is auto-scaling?



(a) Auto-scaling (Scaling out)



(b) Auto-scaling (Scaling in)

Figure 1.2: Typical Auto-scaling scenarios [2]

Manually watching the load of the web-application is both tedious and resource-wasting, which led to the birth of auto-scaling. The auto-scaler is used to automatically removes or adds the right amount of resources to optimize the performance and cost of the web-application (Figure 1.2). It is the cloud's elasticity that provides the chance for the notion of auto-scaling to become desired and the cloud user can pay for computational resource based on usage. The goal of the auto-scaler is to maximize the service performance of our application and minimize the cost of using the cloud resources.

There are two main categories of auto-scaling, namely vertical auto-scaling and horizontal auto-scaling. Vertical scaling adds more physical computational resources to existing hardware, for example, adding more CPUs to the current server. The vertical scaling method does not require the application to be distributed, hence the application is easier to be implemented. However, due to the physical constraints, the server could not have infinite physical resources added on, and the single computationally powerful computer is very vulnerable when it becomes the single point of failure. Therefore, even though the vertical auto-scaling is much faster in provisioning computational resource, the horizontal auto-scaling is preferred. Most of the huge cloud provider only provids limited vertical scaling and majored in providing virtualized computational resources. Horizontal scaling links many virtual machines together to host scalable and distributed applications. However, the communication among the virtual machines is generally through the internet, and for the application to take benefits from the horizontal scaling, they need to be developed with concerns of the communicating. The implementation of such distributed system is complicated and generally they are facing many challenges of the distributed system such as data synchronization, race conditions and failure of socket connections. Even though, the development of the distributed application is not easy, the attraction of infinite scaling out of the application is still very attractive.

1.4 Motivation and objectives

The IaaS is growing rapidly over each year. The market revenue grows to $34.7 billion in 2017 which is about 36.6% increase from 2016 and it is predicted to reach 72.4 billion by 2020 [8]. The entire cloud industry is forecasted to reach $411.4 billion by 2020 from $219.6 billion in 2016 [8]. This huge trend of cloud services' growth of revenue shows that the needs of the cloud service are getting even stronger. The illusion of the 'infinite' computational resources in horizontal scaling is very attractive to most of the cloud users. Nobody wants to bottleneck their application due to the limited computational resources. The cloud users could start from small and gradually add more power to their application easily with more resources when they earn more profits from their cloud applications.

The variations of the loads require the users to use auto-scaler to optimize their cost and application performance to meet SLA. For example, the user request might increase in the afternoon and drop in the evening, so it is both slow and tedious for humans to manually watch the load changes and make the decision on removing or adding cloud resources. However, there is still no such commonly agreed horizontal auto-scaling techniques that could help the users to use the cloud resources efficiently. Therefore it is worthwhile to research the auto-scaling domain in this thesis.

The cloud provider like *AWS* [9] has already developed a simple auto-scaling methods for the cloud user to use. AWS has its auto-scaling policies by creating the *auto-scaling group* [10]. The AWS creates an auto-scaling group and uses *Cloud Watch* to monitor metrics. The user can set alarms on those metrics to trigger scaling policies once they are over certain thresholds. Since the AWS auto-scaler uses static threshold and static adjustment of adding or removing a certain number of VMs, it is highly likely to have over-provisioning and under-provisioning problems. Due to the simplicity and the adequate performance, most companies and organizations still prefer to use reactive approaches [2, 11]. The problem of reactive approach is that the threshold and the adjustment of the cluster size does not change according to dynamic workloads and cluster state. Therefore, the naïve static threshold-based approaches often have oscillation problem which means the auto-scaler keeps changing the number of VMs back and forth frequently that provide excessive or insufficient resources [11]. The reactive auto-scaler still has potential to be improved, and there is still no ideal auto-scaler provided by the cloud providers.

This thesis reviews the current state of art techniques that are used in horizontal scaling. Then based on the study of the horizontal auto-scaling literature, we aim to develop a reactive auto-scaler which has an adequate performance and is easy to use in the web application that intends to fully utilize the on-demand features of the cloud resources.

1.5 The research methodologies

The first step is to review the research papers which are related to cloud computing and horizontal auto-scalers. We will classify papers into different horizontal auto-scaling methods. After the classification, we will investigate the pros and cons of each method and propose an auto-scaling method to fill the research gap in this area. The prototype of the auto-scaler were built and many experiments were conducted with the real web request trace file in order to discover the limitations and the possible enhancement of the current approach. During the journey of completing this thsis, we aim to continously polishing our proposed auto-scaling method by exploring more domain knowledge, building many auto-scaler prototypes and performing as many experiences as possible.

1.6 Research questions

In this thesis, we aim to develop an efficient auto-scaling method for web application in cloud environments. To address the challenges of building such an auto-scaling method, we will investigate the following questions:

What is the best metrics that could be used to perform web application auto-scaling?

Most of the papers favour to use the CPU utilization to represent the current state of the entire cluster system, but the reason for choosing such performance metric is never or little explained. For the web application, the most important thing is to never violate the SLAs due to the shortage of the resources, so the metrics which could reflect the actual service time of the web application could be the major indicator for deciding whether to scale out or in a cluster. The potential service time metrics are response time, request complexity and computation capability of the instance. This thesis focuses on making reactive auto-scaler to be able to dynamically adjust the threshold and cluster size with fuzzy logic. The input of the fuzzy logic system, for making the scaling decisions, should also take the metrics that reflect the system's performance as a whole into consideration. Such as the number of instances used, the time it takes to start an instance, the type and computation capability of the instance available.

Why should we focus on reactive method?

The reactive method is easy to use and understand. Also, the load of the web application is very dynamic; no one can predict what is going to be the next popular website in the next second. The proactive methods require large history of loads to model the future loads, but the highly dynamic loads of a growing web application could not be fit into any models, and there are too many exceptions on incoming loads. Therefore, developing a reactive auto-scaler might be a general approach to solve most of the web application's needs of effective provisioning of the cloud resource with the observations of the current state of the cluster. The fuzzy system does not require a long time to learn; it only needs to take in the set of useful metrics and make the possible provisioning decision with fuzzy linguistic rules for auto-scaling.

Why we want to have an adaptive reactive auto-scaler?

Most of the reactive auto-scaler use static thresholds to trigger auto-scalings. However, the incoming loads and the cluster state are always changing. The static provisioning plan, like always adding or removing a fixed number of virtual machines, can easily cause over-provisioning and under-provisioning problems, since the static number can hardly provide the optimal number of virtual machines needed. The static thresholds might trigger the auto-scaling process too late or too early if we do not consider the ability of the current cluster size to tolerate the load changes. The oscillation problem is common, caused by using static upper threshold (used to add VM) and a static lower threshold (used to remove VM). Auto-scaler will add more VMs when static upper-threshold is breached. However, after provisioning more VMs, the auto-scaler might observe the static lower threshold is broken since the cluster can handle more loads now. The auto-scaler will keep performing scaling out and scaling in repeatedly. Therefore, it is not a wise decision to only use a static auto-scaling rule. Using dynamic upper threshold and dynamic numbers of virtual machines to be added or removed could make the reactive auto-scaler adaptive, hence improve its performance by making better auto-scaling decisions.

What is the missing part of the past researches on auto-scaling?

The past papers only focused on finding the best time to scale out or in the application, and the planning phase is rarely mentioned. The change of the number of VMs is also crucial at affecting the effectiveness of each scaling decision since adding and removing VMs in parallel save time than performing it in sequence. The delay of launching a new instance in the cloud also force us to add or remove the optimal number of the resource with minimum scaling process for mitigating the penalties of violating the SLA.

1.7 Overview of the proposed solution

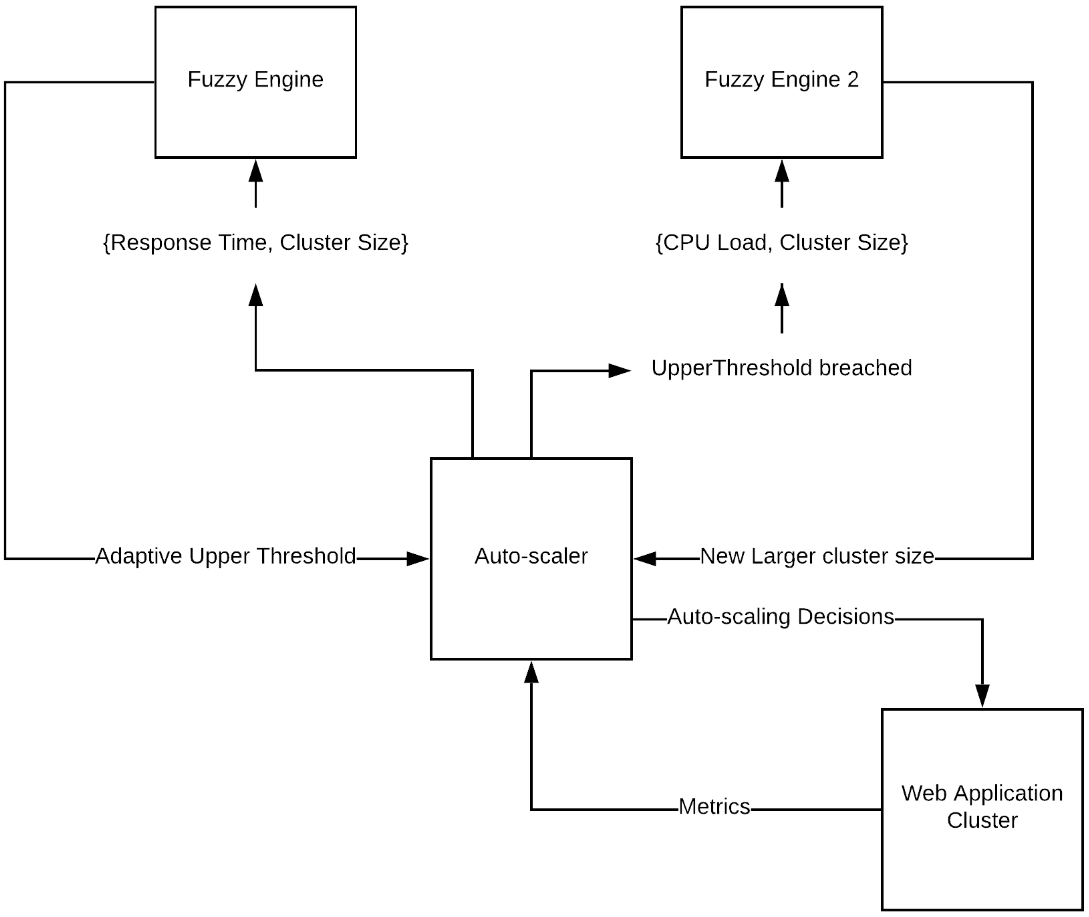


Figure 1.3: Fuzzy logic based auto-scaler overview

In this thesis, we propose a reactive auto-scaling approach that aims to avoid static thresholds and a static number of provisioning virtual machines (Figure 1.3). Our approach dynamically adjusts the upper threshold based on the current cluster size and response time. The number of provisioning virtual machines is also dynamically estimated with CPU Load and cluster size. To this end, the fuzzy logic looks very intuitive to use and unlike machine learning techniques, it does not need historical data and long training time to build the model. One can set up linguistic fuzzy rules with metrics that are critical to satisfy the Service Level Agreement (SLA). For example, one of the rules could be "If *CPU utilization* IS LOW THEN Shut Down One VM". Two fuzzy engines are presented in this thesis. The first one uses request response time and cluster size to output the appropriate upper threshold value for auto-scaling. If the current response time exceeds the dynamic upper threshold, then the second fuzzy engine kicks in, with cluster size and CPU Load as inputs. It determines the cluster size to prevent the web-application from violating the response time required by the SLA, due to the increase of incoming loads. The evaluation of the proposed auto-scaler is performed via a Wikijector agent [12] replaying the requests from the Wikipedia trace file to stress the cluster and monitoring how auto-scaler would adjust the cluster size to handle this stress. The performance of the proposed approach is compared to the AWS's default auto-scaler, and the AWS emulated auto-scaler. The results demonstrate that the proposed auto-scaler significantly reduces cost and SLA violations.

1.8 Why we prefer the fuzzy logic based reactive method

The proactive method is preferred under stable load and mature application that is runs for a long time and has enough historical traffic data. The model is constructed by learning patterns in the historical traffic data with the model building techniques like machine learning to make future traffic load predictions. The model is updated and rebuild throughout the entire lifetime of the application, so the end result is that the model will become very complicated and take more and more cluster resource to build, which means the auto-scaler will gradually become a burden of the cluster. The proactive method is also not very friendly to the newly launched web applications that lack of historical traffic data and our aim is to develop an auto-scaler that can be used by general web application regardless of having sufficient historical traffic data or not. The prediction model relies heavily on learning patterns and characterize from a lot of the historical data. Therefore, the newly launched applications that are trying to get popular might get killed by not satisfying the users expected response time due to the poor initial performance of the prediction model. Even though the machine learning technique could be used as a reactive method by learning correlations of each metrics in the historical data, due to the relying on the historical data, it might not be an appropriate technique to build a general purpose auto-scaler. Fuzzy logic has a lower learning curve when building or tuning it due to its linguistic rules. Our aim is to build an easy to use auto-scaler that does not rely on any historical data, therefore the fuzzy logic based auto-scaler is our chosen technique.

1.9 Overview of findings

We focus on the efficiency of using cloud resource to increase the SLA satisfaction. It is not possible to only increase SLA without increasing the cloud resource usage because the violation of SLA is always due to the under provisioning of the virtual machines. The experimental results shows the fuzzy logic being an effective method to make the reactive auto-scaler adaptive. Adaptiveness is crucial for auto-scalers to plan efficient scaling actions for different cluster states. The static auto-scaling method is not the best choice since it always uses the same scaling plan without considering the current incoming load and the current cluster state. The increase rate of the incoming load will be varied, so adding a static number of virtual machines will easily cause the over provisioning problem. The dynamic provisioning of new virtual machines and the adaptive upper-threshold show that their benefits on saving the cloud resource cost and maintaining the SLA satisfaction.

1.10 The structure of the thesis

The rest of the thesis is organized as follows: Chapter 2 provides some background on the auto-scaling of web applications, the problems of the auto-scaler and the current state of art techniques used in auto-scaler. Chapter 3 discusses how we come up the fuzzy logic auto-scaler, the overview of the auto-scaler and some detailed implementation of the auto-scaler. Chapter 4 shows the system architecture overview, the auto-scaler algorithm

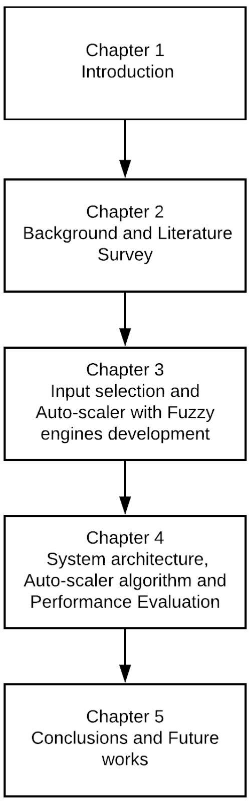

Chapter 1
Introduction
Chapter 2
Background and Literature Survey
Chapter 3
Input selection and Auto-scaler with Fuzzy engines development
Chapter 4
System architecture, Auto-scaler algorithm and Performance Evaluation
Chapter 5
Conclusions and Future works

Figure 1.4: Thesis organization

and the experiment result of the proposed auto-scaler against AWS auto-scaler. Chapter 5 concludes the paper and provides the future work based on the limitation of the proposed auto-scaler. Figure 1.4 shows the entire structure of this thesis.

Chapter 2
Background and Literature Survey

The auto-scaling domain has been well explored. There are many research works which develop creative ways to perform horizontal auto-scaling. This chapter will explain the domain knowledge of auto-scaling in details. The majority of this chapter will mainly focus on horizontal-scaling techniques.

2.1 What are the challenges?

Cloud resource provisioning still has many problems waiting to be solve. For example, due to the heterogeneity of the cloud providers, the cloud application need to cope with the cloud-provider-specific APIs. Therefore there is a risk of locking into one cloud provider. Especially for data critical cloud service, the security of the data stored in Clouds is still questionable [5]. By using cloud resources, it is likely that multiple users shared the same physical hardware through virtualization techniques, so the performance of the cloud resource is not stable if other users exhaust cloud resources.

However, in this thesis, we intend to solve the auto-scaling problem in the cloud. Most of the auto-scalers will go through the so-called MAPE (Monitoring, Analysis, Planning and Execution) loop. We will first discuss challenges of the auto-scaling in MAPE loop along with a concise introduction to each step of MAPE loop in the sections below.

2.1.1 Monitoring

Monitoring is responsible for collecting information about the low-level metrics (e.g., CPU utilization) and high level metrics (e.g., rate of request arrival). These different sets of monitoring parameters are stored in a knowledge base for use by the analysis phase to understand the incoming load and the state of cluster [2]. By using a monitoring tool like Ganglia, it is easy to grab all the required metrics for the application. However, it is hard to determine which metrics are the most relevant to your application. If the application is data-driven, the IO related metrics could be the best to reflect on the load of the application. On the contrary, the CPU Utilization might not be so meaningful in this case. Some Linux system metrics will report the metric as the percentage which is aggregated from the start of the system boot up to the current check so that if your application is runing for a long time and suddenly you have a load increase, the metric value you get might be extremely unreliable. Therefore, we should also know how the system metric is aggregated and then we can use them correctly to provide us the accurate state of the current virtual machine.

2.1.2 Analysis

The analysis phase is a process of understanding the correlation between measured metrics and the incoming loads to decide the time to perform scaling process. Aggregation and modelling are usually used in analysis phase to understand the incoming load of the application. The analysis of the incoming load and the system metrics of the virtual machine are used as a foundation in the planning phase to determine the actual number of computational resources needed [13]. Aggregating metrics, like calculating the mean of the CPU Utilization and setting an arbitrary value like 50% to indicate the trigger of the scaling process, is the easiest analytic method in auto-scaling. The metric aggregation will work better with the profiling of the capacity of the virtual machine.

The load of the application is measured according to the capacity of handling the incoming request of the current cluster based on the SLA satisfaction. If the incoming load is likely to break the SLA of the application, the scaling process should be triggered. The metric value observed when the virtual machine violats the SLA, and can be used to determine the time of the scaling and the possible number of virtual machines needed to mitigate the SLA violation. For example, we observed a virtual machine breach the SLA when its CPU Utilization is over 50% so that we can use 50% CPU Utilization as an indicator to trigger scaling out. We could also use 50% as an indicator of how many virtual machines we need to launch. For example, if we observed the current CPU Utilization as 100% we might decide to launch 2 virtual machines (100% / 50% = 2). However, it is hard to profile the application, since the load comprised of the different type of user requests can result in different changes in the metrics values when the SLA is violated.

The analytic model like queuing model could be used to understand the stress of the current cluster when handling the incoming load. The downside of using analytical model is that the model is a static one and assumes that the incoming load to follow some specific patterns like the same distribution of the arrival request. Therefore, analytically model is suitable for a large stable web application with a stable request pattern, but for an application that is just getting popular then the analytical model approach may suffer since its load patterns are often changed. The machine learning techniques could be used to update the model continuously. The model applied to correlate the response time and the system metrics or to predict the future incoming load might take a massive amount of the resource to build and could burden the existed application. The historical data required for training the model could not cope with the unexpected rise in the demand for the application as well, since it is still a model built based on the previously known knowledge. For the new web application, it hardly has sufficient historical load to build a mature machine learning model so its poor performance at the start could be deadly for a young web application with growing popularity.

2.1.3 Planning

After knowing the application is overloaded or under-loaded. It is time to calculate how many virtual machines you need to remove or add based on the estimation of the incoming traffics. In planning section, while overprovisioning cloud resources can provide better SLA but costs much more. To optimize the cost, only offering enough resources is not sufficient if the incoming traffics are rising continuously since it takes long time to start up new virtual machines. Therefore, the optimal number of 'trial and error' to provision cloud resources are preferred to be minimized. The planning of the cloud resources is a balance on a budget of the cloud user and the importance of preserving SLA. It is highly depending on the cloud users' needs and capability of using cloud resources. The planning phase will become more complicated if we host the web application in multi-clouds. Each cloud platform will surely offer different price of using virtual machines and the problem of choosing the most reliable and cheap cloud resource will become critical to save cloud user's money and maintain the SLA [6,14].

2.1.4 Execution

Execution is the easiest step in the MAPE loop which simply executes the plan of provisioning cloud resources. However, due to the heterogeneity of the cloud provider's APIs, it is hard to write one code that works on all platforms. Using multiple cloud provider is preferred since different provider tends to have geographical segregation which could make your application offer servers close to the users' location, hence faster application accessing speed could be achieved. Being able to use multiple cloud provider via the same APIs will avoid you from locking in one cloud provider and prevent you from 'single point of failure'.

2.1.5 The problem of auto-scaling during provisioning

The auto-scaler also needs to face the delay of acquisition of the new cloud resources which is usually in minutes (booting and configuration of the VMs). To make profits, it is infeasible for cloud providers to always turn on a significant amount of resources and waiting for people to use it. Therefore we can not set a trigger of the previously mentioned example of CPU Utilization over 50% and then launch one new virtual machine each time until SLA is satisfied, since the new virtual machines need several minutes to start and configure. If your application is QoS (Quality of Service) critical the slow join of the virtual machine will cause the loss of application users. Therefore, developing an auto-scaler could be relatively easy if there is no delay like vertical scaling which could join more computational resource instantly [2].

2.2 Reactive and proactive horizontal scaling

The horizontal auto-scaling methods fall into two main categories: *reactive* and *proactive* approaches [2, 11]. In reactive approaches, if certain metrics (e.g., CPU utilization or request rate) exceed a threshold, the auto-scaler provisions a certain amount of VMs. The proactive approaches are more complicated and involve modelling and predicting the future workloads of the web application based on the historical data [2,11].

The distinction between proactive and reactive method is that the proactive method will try to predict the future incoming load based on the pattern and characteristic studied from the web application's historical data whereas the reactive method is aimed to find the best scaling solutions based on the current state of the cluster (i.e. what metrics values are measured now).

Popular proactive approaches like machine learning techniques are used to find models to fit the historical data and forecast future accordingly. However, proactive approaches are difficult to tune and often do not perform well for highly dynamic (unseen) workloads of web applications. The reactive method does not burden the cluster computational resource since it does not need to build complicated models to fit the historical data and predict the expected future load. However, the reactive method might not perform as good as the proactive method if there is a seasonal drastic increase or decrease of the incoming traffics, because the reactive method does not try to learn from the historical data.

2.3 What is fuzzy logic?

Fuzzy logic is a technique used to express the degree of truth (i.e. partial truth) which is different from the Boolean logic that everything is either completely true or false [15]. The partial truth of fuzzy logic usually corresponds to a range of continuous number between **0.0** and **1.0**. The grey area between absolute true and false is modelled by the fuzzy engine to simulate the human-like logic. We will often find ourselves in between specific states, for example, you might find the glass of water to be warm but the notion of warm is actually very ambiguous it could be close to either hot or cold since everybody has different tolerance on temperature. Only when the glass of water is boiled or frozen, we are confident to say the water is hot or cold (i.e. easy to tell the extreme case). Fuzzy logic is often used in control system like brake system to control the strength applied on the brake. Similarly, we could use fuzzy logic to tell us the degree of the stress of our web application and control threshold and amount of cloud resources provisioned based on the continuous fuzzy output. Rather than classifying the cluster to be overloaded, we can have granular values of clusters being overloaded or underloaded (e.g. the cluster **is** 0.95 overloaded and it is **0.05** percent under-loaded) so that we can offer granular control on when to trigger the scaling and how many virtual machines we want to add to our cluster. The fuzzy logic system is mainly comprised of the fuzzy set, membership functions and fuzzy rules. The fuzzy engine received crisp input value (numeric values) and mapped to members in the fuzzy set through membership function. The members in the fuzzy set are categorical terms, like hot, warm, cold in the glass of water example above [15]. The mapping process of turning crisp value to members of the fuzzy set is called fuzzification. The crisp input value can belong to multiple members in the fuzzy set, for example the temperature of 50 degrees can be 20% hot and 80% warm, these membership value of representing the belongingness of the crisp input value to fuzzy set members could later be used in linguistic rules to determine a crisp output value to satisfy those membership values.

The process of defuzzification will turn the output fuzzy set members into a continuous output crisp value by calculating the centroid of the area under fuzzy membership functions of the output fuzzy set member values. The fuzzy rules will map the input fuzzy member to the member of output fuzzy set. 'AND' and 'OR' Boolean operators are normally represented by 'min' and 'max' functions respectively.

The fuzzy rules are constructed with linguistic terms, so it is very intuitive to understand hence easy for us to apply the knowledge on controlling the cloud resources during auto-scaling. The fuzzy logic system does not need a learning time, and it does not model the workload in a linear or continuous way, therefore it is less vulnerable on the incoming traffic which is never observed before. Fuzzy logic's ability on interpreting the vagueness could help us with dynamically control the cluster by knowing the fine degree of severity of the cluster, in order to avoid the static threshold and the static adjustment of the cluster size in the naive reactive auto-scaler.

2.4 Related work

There are many horizontal auto-scaling techniques that have been explored in the literature. The following will show some mainstream horizontal auto-scaling methods and the papers which adopts the method. Table 2.1 shows the overview of the auto-scaling methods mentioned in the papers included in this section.

2.4.1 Threshold-based

The majority of the reactive auto-scalers are threshold based. The threshold is usually an indicator metric that could be used to reflect the SLA. If the safe value of the indicator metrics is over, say 50% of the CPU utilization, the auto-scaler will start the scaling out process to keep the indicator metric value down.

Lim et al. [16] focused on using the averaged CPU utilization of the VMs to determine the number of VMs needed to be removed or added. They use a pair of threshold boundaries to make it 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 scaling out.

Hasan et al. [17] proposed a novel static threshold technique which could prevent the oscillation problem for a short duration of the frequent changing loads. The authors first choose static upper and lower bounds and then select extra sub-upper and sub-lower bounds in between. If the current metrics exceed any of the thresholds for a certain preset duration, the system starts to scale-out or scale-in the cluster.

The oscillation is a common problem of auto-scalers, and there are many solutions proposed in the literature. Unlike [17] and [16], our proposed auto-scaler in this thesis uses a different approach by increasing the upper-threshold when the cluster size is large, so it has a stronger tolerance on the small fluctuation of the request load. Normally 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 the stress of incoming 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.

2.4.2 Fuzzy logic based

Fuzzy logic could be used to easily express the expert's auto-scaling knowledge with its linguistic rules. The input and crisp output values are generally between some range like 0 and 1 which allows more granular control on when to trigger the scaling process.

Frey et al. [18] developed a fuzzy logic auto-scaler to improve QoS of the cloud computing. The result of the fuzzy auto-scaler shows the technique is effective in reducing the response time. They used different fuzzy 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. However, the 'high prediction' input is based on the knowledge of the application's history load pattern whereas the proposed auto-scaler does not assume the prior knowledge of the application.

2.4.3 Fuzzy logic with neural network learning

Lama et al. [19] 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. The neural network generated fuzzy logic linguistic rules are harder to reason. Learning algorithms takes time for the fuzzy engine to correctly construct its rules at the starting phase, since the learning algorithm needs time to build a mature model. The learning model building phase cloud be deadly for the small web applications that are starting to get popular [2,11].

2.4.4 Fuzzy logic with Reinforcement learning

Instead of hard-coding static scaling number of virtual machines, Arabnejad et al. [20] enhanced the output (actions) of the fuzzy controllers with reinforcement learning. In [20], the state of the system is reduced by fuzzy sets and their corresponding membership functions. The state of the system is represented by workload, response time, number of virtual machines. The reinforcement learning consumes the current state of the system, and randomly trying out different actions to be the actions of the fuzzy controllers. By getting feedback of the actions represented by the cost of the resource acquired and the SLA violations, the reinforcement learning takes a random approach to explore all the possible actions until the model cannot improve the actions taking for each state. As we can see the reinforcement learning takes trial and error actions to construct the rules used to produce the fuzzy engine's output, the initial performance of the auto-scaler will be random (i.e. poor), so the auto-scaler user need to be prepared for violating SLA frequently at the start of the program.

2.4.5 Profiler based

Profiler technique involves determining the capability of the certain type of virtual machines with load stress test. During the load test, the requests are firing to the virtual machines until the SLA is broken, then the max number of request of that type of virtual machine can handle without breaking SLA will be recorded. The capacity of each virtual machine will be used like an indicator to determine the number of virtual machines that are required to be launched when the auto-scaler observes whether the incoming load would caused the SLA violation with the current cluster size. However, in chapter 3, we will discuss why it is hard to decide on how many requests a virtual machine can handle since different request types use different amounts of computational resource.

Fernandez et al. [21] developed an auto-scaler comprised of a predictor, a profiler and a dynamic load balancer. The profiler is the major component of the auto-scaler, which is used to measure the capacity of the hardware (virtual machines) used to host the application. The predictor is used to estimate the future incoming traffic. The load balancer will dynamically distribute the request to each type of virtual machines according to their capacity of handling request. Knowing the capability of each type of virtual machine is very handy to scale out or in a different number of virtual machines based on the predicted incoming loads. However, the authors do not mention exactly what kind of trace file they used to profile the virtual machine and how they conducted the profiling process.

2.4.6 Analytic model based

The analytic model is a mathematical model used to represent the behaviour of the system. The correlation between the dependent variables and the independent variables is used to make the optimal scaling plans. The analytic model discussed below is static since those model are invented based on previous analysis of the system patterns and it might need the machine learning technique to keep it updated to cope with the characteristic of the current cluster.

However, the use of the analytic model in the auto-scaling area is also a popular method. In [22], ARIMA is adopted to solve the problem. ARIMA (Autoregressive integrated moving average) is a statistical model used to analyse the time series data. The authors used this model for predicting the future load of the web application. Thus the author needs historical traffic data to construct the ARIMA model. Solely estimate the future load is not enough to auto-provision the resources for hosting the web application (i.e. do not know how many virtual machines need to be added). Queuing network of n*M/M/1/k is used to estimate the optimal number of VMs needed with the predicted future load. For the queuing theory n*M/M/1/k, the first M means the Poisson distribution of the request arrival rate, the second M means the exponential distribution of service time, 1 means one queue per server, n means there are n servers and each server has one queue, k is the queue size inferred from max response time allowed for each virtual machine [23]. The goal is to find the optimal number of virtual machines (n) needed in the cluster. The predicted future load is inputted as 'Request arrival rate' in the queuing model. If the current number of VMs is smaller than the optimal number of VM then more VMs added otherwise shut down the exceeded number of VMs (not immediately shutting down but wait for all requested served). The analytic model used in this research paper assume the pattern of the incoming request and the queue capacity value k differs from each type of virtual machine, which means the author needs some kind of profiling step to work out the k value. However, the detailed steps of measuring k value is not well explained.

2.4.7 Predictive Model Control

Predictive Model Control is a technique which tries to find the optimal moves that could satisfy our target while meeting constraints on selecting the optimal solutions. In our case, the target can be the SLA satisfaction and the constraints are the cost of the cloud resource that should be used effectively. MPC has a look ahead mechanism that predicts what will happen in the future timeslot (prediction horizon) [2, 24]. This technique requires solving the optimisation problem, and it requires good mathematical skills when using it.

Ghanbari et al. [25] adopts the Model Predictive Control (MPC) with a Stochastic model to predict the optimal future actions with the constraints to balance the cost of greedy actions and the reactive actions. The greedy action will only care about the best cost in the current stage which could be potentially under-provisioning. The reactive action will take the cost of the violation of the SLA in the future in to consideration. The auto-scaler will try the best to minimize the cost with the consideration of using greedy or reactive actions.

2.4.8 Random Forrest approach

Random Forrest is a machine learning technique which is used to solve classification problem. By gathering training data (historical load) and the corresponding metrics value related to each data point collected. The random forest technique could establish a learning tree to find the patterns from the metrics of the cluster and connect it to the corresponding incoming traffics.

Instead of using static model, the machine learning approach tries to adjust the model dynamically to fit the current and future state of the application's incoming traffic (learn from observation, statistics and video streaming service). The author [26] collects all the numeric system metrics from Linux SAR tools to form the feature sets for later training. Different regression models have experimented in the paper, and the final results show the random forest model has the lowest error rate of predicting the service-level metrics (Video frame rate, Audio buffer rate and Network read rate). Like most of the machine learning techniques, the model training in the paper relies heavily on the historical data. The author uses syntactic traces to simulate the web application run and collect the system metrics as their training and testing data set. Therefore, it could possibly be that the random forest model is biased to the artificial traffics generated by the authors.

2.4.9 Neural Network

The neural network consists of three primary layers, namely input layers, hidden layers and output layers. Neural network uses the propagation technique to correlate the input value (e.g. system metrics value) and the output value (predicted traffic load). As a machine learning technique, the neural network also needs to use historical data as a training set to build the model and neural network's computational intensity entirely depends on the number of hidden layers.

Islam et al. [27] use the neural network approach to predict the future CPU Utilization for an E-commerce website hosted in AWS (Amazon Web Services) platform. The purpose of predicting the future CPU Utilization is to mitigate the severe consequence of receiving sudden colossal request (Slashdot effect). The training data is obtained by using author developed TPC-W request generator to simulate the ramp-up phase of the web request pattern named 'flash crowd web traffic'. The training data gathering experiment also use AWS's simple auto-scaling API to provide resource provisioning. Predicting CPU Utilization might not reflect the actual workload of the cluster. The author also demonstrates the effects on prediction accuracy with different time input window size.

2.4.10 Commercial Auto-scaling tool with container approach

Kubernetes is a cluster manager which makes the deployment of the cluster application easier. Kubernetes mainly focused on deploying containerised applications (e.g. applications run in docker). Kubernetes launches the applications in 'Pod' which is like a light-weight virtual machine that could further divide the virtual machine's computational resources into smaller pieces. Kubernetes currently (at the time of this thesis written) supports horizontal pod auto-scaling. The auto-scaling algorithm used is to keep CPU Utilization within a target value. Kubernetes works on adding more metrics to its auto-scaler. However, the auto-scaler works best with CPU Utilization now.

The algorithms mainly depends on the following formula to determine the number of pods to be in the system.

$$TargetNumOfPods = \lceil \frac{SumCurrentPodsCPUUtilization}{TargetCPUUtilization} \rceil \tag{2.1}$$

If 'TargetNumOfPods' is smaller than or larger than 10% of the current number of pods, the scaling out or in of the process will be triggered to prevent the auto-scaler from being overly sensitive [28].

The auto-scaler of Kubernetes is still under development, but the advantage of using cluster manager like Kubernetes to deploy a scalable application is substantial. Therefore, having a decent performance auto-scaler integrated into cluster manager like Kubernetes could be a killer feature (so does the cloud resource provider). Since Kubernetes is open-sourced, it is likely that the auto-scaler will be enhanced nicely, for example [29] attempted to try to develop a proactive auto-scaler with model predictive control in Kubernetes and try to merge it to the master branch of Kubernetes in GitHub.

2.5 Slash-dot effect

Slashdot (or flash crowd) usually happens when the popular website links to a smaller website. The sudden massive increase in the incoming load of the website will cause the web application to be unavailable due to the lack of computational resources to process the user request [30]. Slash-dot effect often happens when the new website is gaining popularity on the internet. For handling load burst, you need some way of predicting the future load since the starting time of new instance are long. However, the current prediction methods are based on studying the historical data of the web application, so if the load bursting has no patterns in the past data, we could not handle it easily. The problem of sudden unexpected load burst is still a dilemma waiting for us to solve. The reactive auto-scaler might have decent performance when encountering slash-dot effect since it is simply reacting to the currently observed metrics of the cluster and unlike inputting unexpected load to the predictive model, it does not make a random move in this case.

2.6 Auto-scaling indicator metrics

The proposed auto-scaler in this thesis is much simpler to create and tune, and since it is a reactive auto-scaler, it does not have poor performance at the start. By not relying on the historical data, the auto-scaler proposed in this thesis could be more resilient under different load pattern and suitable for newly arisen application that does not have enough historical traffics to use the model based auto-scaler.

Most of the auto-scaling studies only focused on the *Analysis* part of the MAPE loop; they did not mention much about the planning phase that what metric can indicate how many VMs should be added or removed if the thresholds are triggered. The time taken to start new VM is usually around several minutes [13], so naïvely launching single VM one at a time might not be feasible to solve the under-provisioning issue [2].

The service-time-related metrics of the application are better than the metrics which only show the state of the VMs [2]. The web applications are all about meeting SLAs and providing appropriate Quality of Service (QoS). This is also why most of the research works tend to use response time to benchmark their auto-scaler. We believe that the response time should be used as a scaling out trigger. For example, the VMs with high CPU Utilization might still provide 50ms response time whereas the SLAs only asked for 100ms response time per request, so in this case, there is no reason (except future load predicted) to add more VMs and spend more money just because the CPU utilization is high. Therefore the best trigger metric that could be used to invoke the scaling policy is the request response time in web applications' case.

Table 2.1: Table of Auto-scaling method papers

Work	Indicator Metrics	Method	Reactive or Proactive	Application
[16]	CPU Utilization	Threshold-based with dynamic lower threshold	Reactive	Elastic Storage
[17]	CPU Utilization	IACRS ALGORITHM (Threshold based)	Reactive	General Cloud application
[19]	Relative Delay Deviation and Delay Target violation	Self-Adaptive Neural fuzzy controller	Reactive	General Cloud Application
[18]	Response Time, Application History workload	Fuzzy logic engines	Proactive	General Cloud Application
[21]	CPU Utilization, request rate and response time	Profiler based	Proactive	General Cloud Application
[22]	Web application load	Analytical Model (ARIMA)	Proactive	Web application
[26]	All numeric system metrics from Linux SAR command	Random Forrest	Proactive	Video on-demand services
[25]	Estimated Queue length, Already reserved resource, Response time	Model Predictive Model	Proactive	General Cloud Application
[27]	CPU Utilization	Neural network	Proactive	E-commerce website
[28]	CPU Utilization	Feedback control model	Reactive	General Cloud Application

Chapter 3

Fuzzy Auto-scaler

In this chapter, we propose our fuzzy logic-based auto-scaling approach for web applications. First, we identify metrics that are relevant to the auto-scaling process and use them in our proposed fuzzy engines. Then, we discuss the ideas behind the fuzzy rules of setting the upper threshold and estimating the number of virtual machines needed for scaling out. Finally, we present our proposed algorithm to give you a overview on how the proposed auto-scaler works.

3.1 Introduction

THE proposed fuzzy logic based auto-scaler is aimed to be used for web applications and its goal is to minimise the cost of the servers used to host the web application while keeping a high SLA satisfaction of the web application. The web application nowadays is generally three-tiers architecture. The first tier is a load balancer used as the entry point for the user to connect with the second tier's web servers. The third tier is the database instance which stores the data needed to generate the response to user's request. Our auto-scaler is focused on making the best auto-scaling decisions for the second tier web servers since the second tier is easy to get bottlenecks due to the insufficient supply of cloud resources. In this chapter, we will explain all the crucial steps involved to develop the proposed fuzzy logic based auto-scaler. The process of how we choose the metrics, the logic behind the fuzzy engines and the algorithms will all be shown in the rest of the chapter.

3.2 Input Selection

Many research studies are focusing on how the speed of opening a website can affect the user's desirability of using the website [31,32]. The most important aim of the web-application auto-scaler is to maintain the low response time to ensure the website provide the satisfactory user experience. Therefore, the use of the response time as an input to the auto-scaler is desired.

Most proposed auto-scalers only use one of the *CPU Utilization* or *Request Rate* as their key performance indicator to trigger the auto-scaling policies since the authors assume the change of those metrics are strongly related to response time, but relying on a single metric is hard to make an optimal scaling decision.

The response time used alone as the indicator does not provide us with enough information about how many more VMs are needed to avoid violating the SLA. The naïve approach is always to launch a static number of VMs to see if the response time is lowered to the desired range. But the time of booting up and configuring the newly launched VMs are often quite high, which could cause the problem of over-provisioning or under-provisioning. Due to the slow join of the new web servers, we cannot simply apply the 'trial-and-error' approach that adds one VM at a time to see if it reaches the desired response time with optimal cluster size. The long launching time forces the auto-scalers to make the best decision with a minimal number of trials to shorten the time of violating the SLA.

CPU Utilization is often used to determine the new cluster size. However, the problem with CPU Utilization is that it does not reflect the actual overall load of the VM, but how busy your CPU is in the last time interval measured. Even though there is one process occupying the CPU cycles, the CPU Utilization could be 100%. Therefore it does not show the demand for other processes which are waiting for CPU cycles. Also, different web applications tend to demand different system resource. For example, consider a web-application like Mediawiki, which is I/O intensive rather than CPU intensive. The use of CPU could be delayed during the I/O processes hence a delusion of the low CPU Utilization will happen, but the load of the VM is actually high. 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. We need the metric which could tell us the demand of the system resource. Hence we can plan an optimal number of virtual machine to add or remove for each scaling process to quickly satisfy the SLA.

Request processing rate per second sounds relevant to the new cluster size. The tools like 'Jmeter'[1] or 'Ab'[2] could be used to flood the VMs to profile the maximum number of requests to be processed per VM then use it to set up an 'optimal' number of VMs. Normally the profiling process needs to be repeated for a different type of virtual machine since their capacity on processing request per second is different. However, the maximum request rate is also highly dependent on the complexity of the request's response body.

Table 3.1 shows a group of 5 Mediawiki requests with various response body size and request type. Figure 3.1 shows that larger request response body size (Table 3.1) causes the max request number per second to decrease (Request 1 to 4). However, the size of the response body is not the only factor which could affect the max request per second. The request 5 has similar response body size to request 1 but the max request per second is much higher. The reason is that the Request 5 is a JavaScript file, so it needs fewer computations and I/O operations to construct and return the response body hence the complexity of generating this response body is much lower than the Request 1 which requires querying data from the database and constructing the page content in the MediaWiki web server. It is hard to tell the complexity of the request since the content of the page has high variations that we do not know the complexity of database queries and computational works need for the page user requested. The use of maximum request rate to determine the cluster size is not effective if the request received by the VM has a big difference to the requests used to measure the maximum request rate. For example, if the profiling trace file used mainly contains the request of the pages which requires only little computation then the maximum request rate observed will be relatively high. In reality, we might receive much more complicated user request, therefore the number we believed is the capacity of the virtual machine from profiling might be inaccurate. The auto-scaler will never add more virtual machines since the actual number of request we

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

can process is lower than the one observed from profiling. Hence the consequence is the auto-scaler never performs scale out, and we would experience under provisioning for a long time.

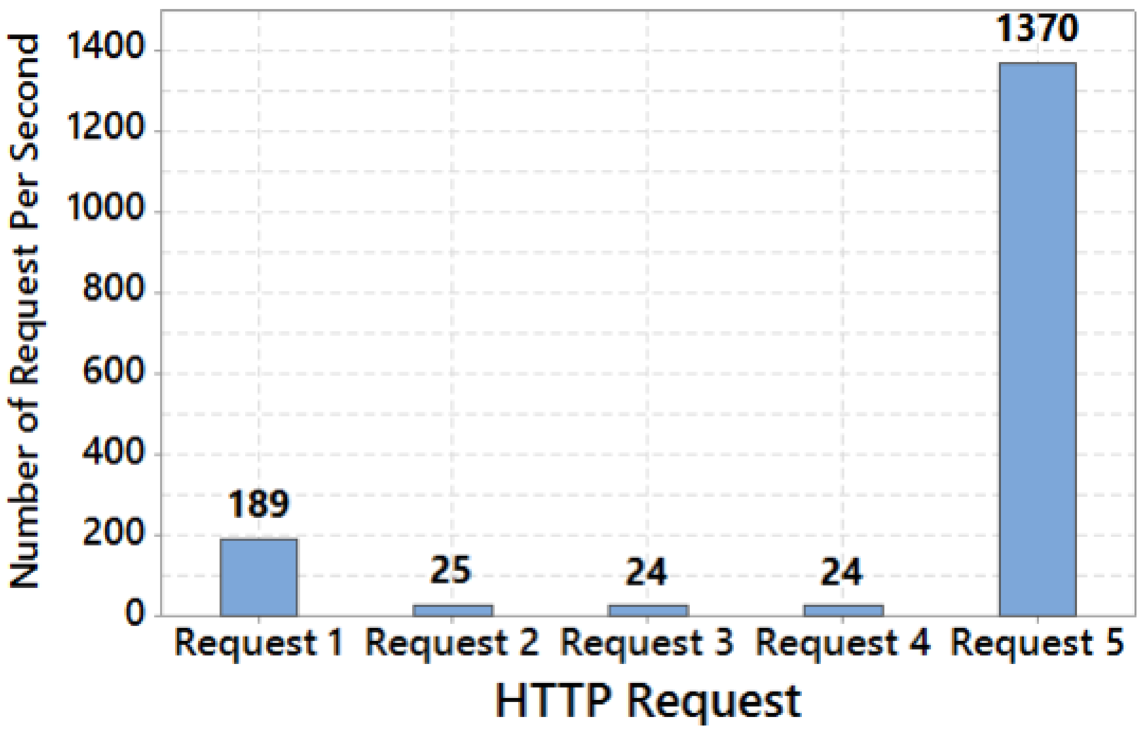


Figure 3.1: Jmeter load test with different requests

Table 3.1: Table of Request Details for Figure 3.1

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	Text (JavaScript)

3.3 Metrics Correlations with Response time

Since the web applications use multiple resources of the VMs (CPU, Memory, etc.), it would be better if there is one metric that reflects the system resources as a whole and is also sensitive to response time variation [33]. Table 3.2 shows each metric's Pearson

Table 3.2: Table of Metrics correlations with Response Time (P-value represents the significance of the correlation)

Metric	Pearson	Pearson (P-Value)	Spearman	Spearman (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

(linear relationship) and Spearman Rho (monotonic relationship) correlation value with respect to request response time. Table 3.2 is produced by applying simple scaling out rules that only add one new VM each time when the response time is over the static upper threshold of response time. Wikijector is used to perform load test by sending the requests from Wikipedia trace files to the load balancer instance. Unlike flooding a single VM with an overwhelming amount of requests, the auto-scaler with the simple scale-out rule is applied to record the fluctuation of all the metrics while the cluster size is changing. Overloading the single virtual machine causes all the resources to raise up continuously until hitting the resource limit, so the correlation of all the metrics will likely to be strongly positive or negative. The aim of the load test used here is to try to add new virtual machines gradually and see how the changes in the metrics are correlated. Table 3.2 shows that the CPU Load has the highest Pearson and Spearman Rho correlation value to the response time, so it could be a potentially useful metric for determining the cluster size.

3.4 What is CPU Load?

The CPU Utilization means the percentage of time that the CPU runs our program for. Since CPU Utilization is a discrete state, it is not the metric that could reflect on how much power of the CPU we have used, but the proportion of the time our program occupies CPU cycles. That is also why the CPU Utilization can only be raised to 100%, whereas the computational load metric (CPU Load) could go beyond 100%.

Unlike CPU Utilization, CPU Load could go beyond 100%. The CPU Load metric could give us the actual 'demand' of the computer's resources (not only CPU power), whereas CPU Utilization could only reflect how busy the CPU is. The analogy of the difference between CPU Load and CPU Utilization is the high way traffic. CPU Utilization is how often the freeway is found to have cars running on it, CPU Load is how many cars are demanding the freeway both running on the freeway or waiting to enter the freeway [34]. In Figure 3.2, we can see if the freeway is fully used the CPU Load is 100%, if only half of the freeway is used the CPU Load is 50% and if the freeway is fully used and there are some cars waiting to enter the freeway the CPU Load will go beyond 100% and it is 170% in the figure. In a single core computer like *t2.micro*, ideally the 100% of the CPU Load indicates that the computational resource is fully utilized and if the value is 200%, it means we need one more instance to handle the load. Since the CPU Load could reflect the overall usage of the resources, the CPU Load could be a crucial performance indicator metric in our fuzzy logic based auto-scaler.

The CPU Load is initially designed to show the demand of the CPU resource that the task is using the CPU cycles and the task which is waiting to run but not blocked by IO. However the Linux kernel later evolve the CPU Load metric to become different from other non-Linux based operating system. The Linux based operating system will have CPU Load metrics included the TASK_UNINTERRUPTIBLE which are the task waiting for IO and locks [35]. Therefore, the CPU Load metric in Linux based operation should be correctly named as the 'System average load' which represent the demand of the computational resources as whole. Therefore, the CPU Load metric could be a perfect indicator metric for estimating the number of virtual machines needed to satisfy the SLA.

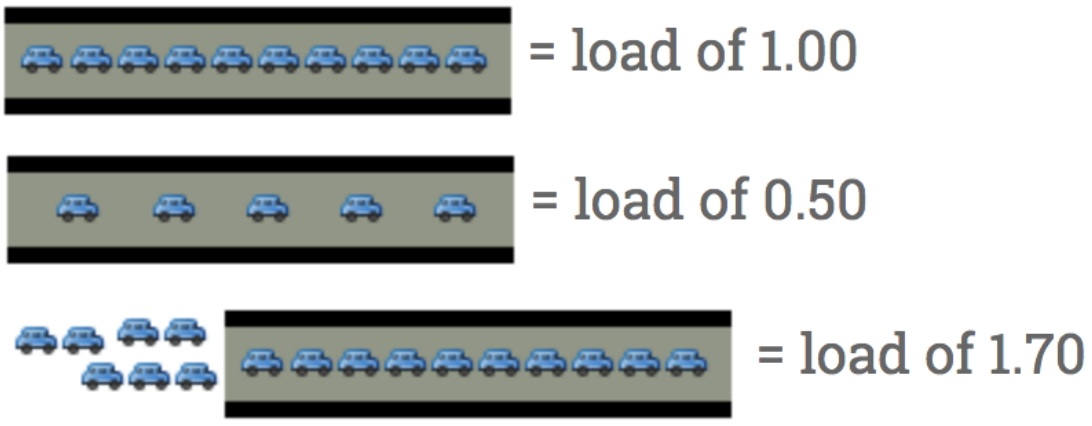


Figure 3.2: CPU Load Freeway analogy illustration [3]

3.5 Dynamic upper threshold

The response time can be collected from the load balancer instance, and it could also be collected directly from the client. But due to the network partitioning or network jamming, the response time perceived by the actual user could be high. All the web application aims to finish processing the request as soon as possible. Therefore, collecting response time directly from the load balancer instance seems more reasonable to do, since it indicates how fast the corresponding request's response body is generated in MediaWiki server instances.

We decided to go conservative on scaling-in as it is risky to terminate instances earlier than expected which may cause under-provisioning and eventually results in losing users. Therefore we shut down a virtual machine which is only used below half of its computational resources in the proposed auto-scaler. The inputs of the fuzzy engine is the current *response time*, as a ratio of the maximum response time allowed in SLA, and the *cluster size* which is also a ratio of the maximum cluster size (e.g., 0.5 of cluster size means half of the maximum number of VMs and 0.5 of the response time means half of the maximum response time allowed in SLA).

The upper threshold used to trigger the scaling out should be adjusted with the consideration of the current cluster size. For example, if you have only 1000 dollars (i.e. small cluster size) and if you accidentally lost 500 dollars you will be pretty upset since you lost half of your total asset, you might need to plan to save your spending. However, if you have 1 billion dollars (i.e. large cluster size) and you accidentally lost 500 dollars of your total wealth you could pretty much still live in the same lifestyle as before. This analogy is to show that, the cluster size is very critical when setting the dynamic upper threshold to prevent unnecessary scaling out and save the cost of cloud resource when there are many fluctuations of the response time.

The idea behind the fuzzy rules is that when the cluster size is large (e.g. 100 web-servers running), then the changes of the request load will only fluctuate the response time in a small amount so the fuzzy engine should output higher upper threshold of response time, hence higher tolerance on response time increase but still makes sure the response time is within the SLA. Whereas, when there is only one running instance, the cluster is very likely to violate the SLA (large fluctuation) with a small amount of request load increase. In this case, the threshold should be more sensitive (lower) in order to trigger early scaling out in order to save the SLA satisfaction faster.

3.6 Dynamic cluster size

Ideally, a single-CPU VM with fully used system resources has CPU Load at 100%, and all CPU Load beyond 100% means the VM is overloaded. Note that, 100% CPU Load or over only means that the VM is fully loaded, but it could still provide decent request response time since the response time mentioned in SLA could be handled very well even the system resource is slightly overloaded. The scaling-in approach used in the proposed auto-scaler is conservative but dynamic. We can scan through all launched VMs and stop all the VMs which have CPU Load below 50% instead of pre-defining a number of VMs to scale-in. Since the cost we can afford is often limited, the maximum number of the launched VMs must be set for scaling-out since even the cloud provider has relatively infinite cloud resources we can use, but the budget we have are typically limited. The minimum number of VMs is needed to provide baseline performance for our web applications (i.e. it is not a good idea to remove all of your virtual machines).

The total CPU Load of our cluster without overloading VMs could be calculated by:

$$TotalCPULoadNotOverloading = MaxNo.ofInstance \times 100\% \tag{3.1}$$

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

$$ClusterStress = \frac{AverageCPULoadOver100\%}{TotalCPULoadNotOverloading} \tag{3.2}$$

For example, if we have 2 VMs and they have CPU Load of 150% and 200% respectively, the average CPU Load beyond 100% 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% (i.e. 5 * 100%). Therefore, the cluster stress is equal to 75/500 = 0.15.

The ratio of the overloaded CPU Load value of the total cluster CPU Load is the representation of the current cluster's stress that is how huge the overloaded average CPU Load is, with respect to the total CPU Load that we can fully use without overloading the VM.

Finally, we could use the cluster stress value and current cluster size as the inputs to the fuzzy engine to adjust the cluster size for scaling out. The output of the fuzzy engine is the ratio of the cluster size to the maximum cluster size. Based on this ratio, the new number of VMs could be calculated by:

$$\lceil ClusterSizeRatio \times MaxNumberOfVirtualMachines \rceil \tag{3.3}$$

The correlation between CPU Load metric and response time is not 100%, so the overloaded CPU Load (beyond 100%) value could not be fully trusted to perform scaling out especially when the cluster size is very small comparing to the huge incoming loads. Therefore, the output of the cluster size fuzzy engine for scaling out should take the current cluster size into consideration. The fuzzy engine will act conservatively by adding less number of VMs determined from the overloaded CPU Load when the cluster size

Table 3.3: Partial add VM fuzzy rules

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

is small because when the cluster size is small the metric we observed could heavily be overloaded and they are less reliable to be used for estimating the cloud resources we need. It is the same that when the cluster size almost reaches the maximum size, we should be conservative because we should not readily give out all the cloud resource we can afford.

3.7 The proposed Fuzzy engines

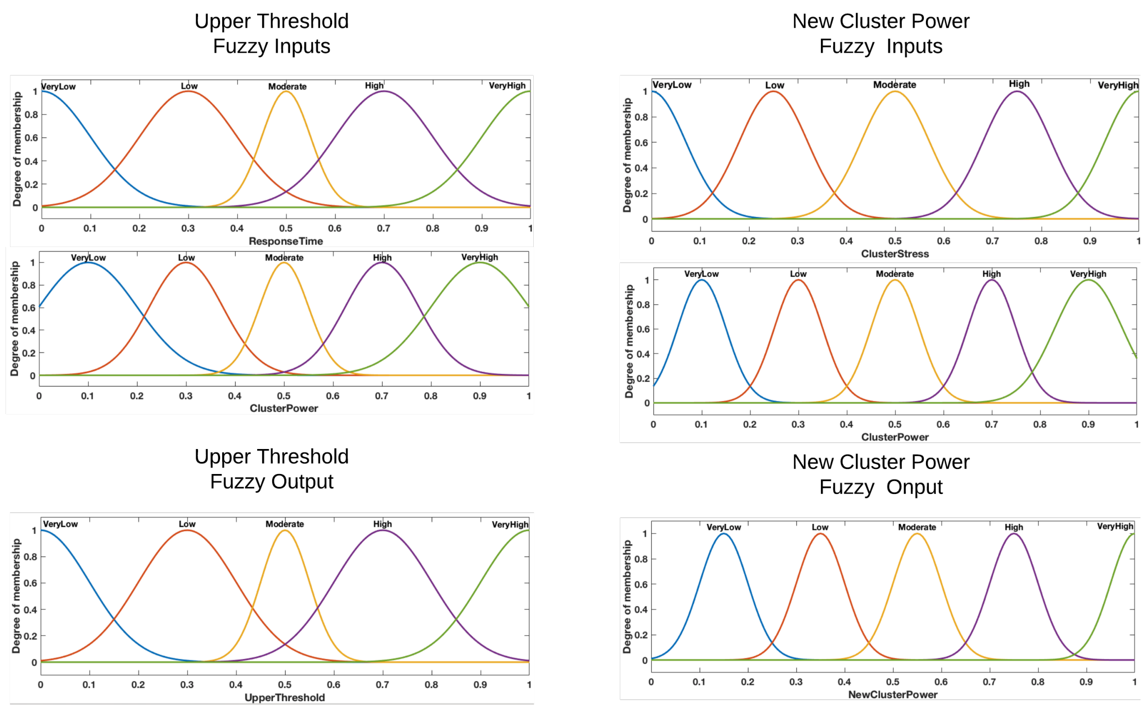


Figure 3.3: Fuzzy engines for setting upper-threshold and adding new virtual machines

Table 3.4: 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

Algorithm 1 shows the overview of how adaptive response time threshold and dynamic cluster size work together to perform auto-scaling. The fuzzy engines are applied in *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 crisp values (all of them are from 0 to 1) to the fuzzy sets are Gaussian membership functions. The fuzzy membership functions' detail is shown in Figure 3.3 and the partial fuzzy rules are specified in Table 3.4 and Table 3.3. For example, if we have a maximum of 10 VMs and 200ms is our desired response time (i.e. SLA), and if we input 0.1 as current cluster power (10 * 0.1 = 1 running VM) and input 0.1 as current response Time (0.1 is equivalent to 200ms * 0.1 = 20ms) to our adpative uppper-threshold fuzzy engine, then based on the fuzzy membership functions of fuzzy inputs they will have the highest membership value in *VeryLow ResponseTime* and *VeryLow ClusterPower*, then the fuzzy inference system will check the fuzzy rules in Table 3.4. The final output upper-threshold's crisp value will falls in the *high* and *VeryHigh* Gaussian function of the output membership functions. The cluster has a low response time, so it is reasonable for the fuzzy engine to set a high threshold to prevent scaling out. Same fuzzy procedures also applied to *clusterSizeFuzzy* fuzzy engine, but instead of using Response Time, we use the stress of the cluster calculated based on the maximum number of cluster and current average overloaded CPU Load metric value.

3.8 Overview of the implementation

The entire auto-scaler is implemented in Python3, only the fuzzy engine part is a Java program that could be called via command line with Python code to provide inputs through stdin to fuzzy engines and get back outputs via stdout. The general AWS cloud resource provisioning is through Python Boto3 library and the Boto3 is working closely with Ansible to tell Ansible the right time to start configuring the new instance. The newly started virtual machine will go through 'Pending' then to 'Running' state. The Boto3 library provide wrapper function of AWS command line to let us know which state the instance is currently in. Only in the running state, the Ansible can start running, otherwise, the virtual machine is not booted up and Ansible is not able to start an ssh session to configure the virtual machine [36]. Boto3 is also used to call AWS's auto-scaling API to perform experiment in Chapter 4. Ansible is used to do small configuration task like updating the list of MediaWiki server instance IP address in the Haproxy instance. The heavy lift of provisioning like installing MediaWiki application's dependency like PHP and the data needed in the MySQL database instance is done via the Amazon machine image (AMI). AMI works as a template that we can install all applications needed in the virtual machine then create an AMI of that virtual machine to replicate the state of the virtual machines over many different VMs (i.e. all of them have same configurations, applications and files installed). AMI saves a huge amount of configuration time needed for each virtual machine; otherwise, we will take 10 or 20 minutes to finish configuring a virtual machine. If the time taken to configure and start a virtual machine is considerably long, then the benefits of on-demand service is minimal since we are not able to provision our cluster fast enough to mitigate SLA violation. AMI also reduces the risk of installing all the dependencies from the internet that might have huge time variation due to all the uncertainties exist in our Internet (e.g. internet jamming).

Chapter 4
Performance Evaluation

In this chapter, we will benchmark our fuzzy logic based auto-scaler against the AWS auto-scaling methods. The efficiency of using cloud resources to increase SLA satisfaction is the key comparison factor in the experiments. The effectiveness of our proposed auto-scaler will be shown with experimental results obtained by simulating user requests with Wikipedia trace file via Wikijector. The overview of the system architecture and the details of proposed auto-scaler algorithm will also be included in this chapter.

4.1 Introduction

THE experiment environment is the MediaWiki system host in AWS as shown in Figure 4.1. The virtual machines used are all from Amazon AWS cloud platform. To minimize the cost of running experiments, we only use the free tier standard t2.micro instances. The homogeneous instances used in the experiment keep the overheads low to let us only focus on the core idea of the proposed algorithms. The experiments involve three different auto-scalers which are the fuzzy logic based auto-scaler, AWS emulated auto-scaler and AWS native auto-scaler. AWS emulated auto-scaler is a mimic of the AWS native auto-scaler using our framework. Unlike the AWS native auto-scaler, the AWS emulated auto-scaler does not call the AWS's auto-scaling API to perform the auto-scaling but does it locally to reduce the overheads. The SLA satisfaction and cloud resources cost are compared among those three auto-scalers in this chapter with real world Wikipedia traffic traces. The experiment aims to benchmark the performance of the proposed auto-scaler regarding the cloud resources used and the satisfaction of the SLA. The experiments also show the relation of CPU Load metric and the trace file (i.e. incoming load) to further evidence that the CPU Load is a reliable metric to be used in estimating the cloud resources needed to handle the load variations. The AWS native auto-scaler is used as a baseline when analysing the experimental results since it is expected to perform the worst.

4.2 The overview of the system prototype architecture

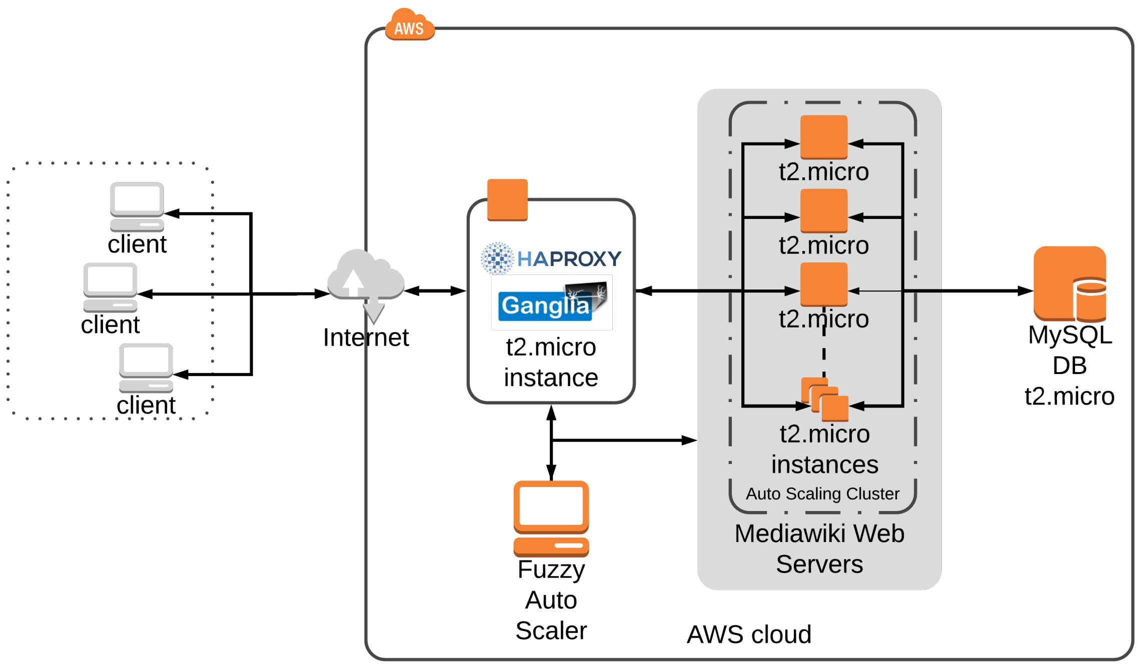


Figure 4.1: System architecture diagram

The entire system is a multi-tier architecture. The first tier is a load balancer, the second tier is MediaWiki web server, and the third tier is our database instance. In this thesis, we only focused on performing auto-scaling in the MediaWiki server tier.

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 4.1). The communications inside AWS cloud happen through private IP addresses to keep the cost of running the experiment low.

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 is also 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 in this system is not in AWS cloud due to the logging and performance evaluation purpose of the auto-scaler. However, it can be easily installed in another t2.micro instance in the AWS cloud. The fuzzy auto-scaler mainly communicates to the HAproxy instance to gather metrics of all the AWS t2.micro instances used, 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 fuzzy auto-scaler instance which uses Python Boto3 (AWS SDK for Python [37]) library to manipulate the AWS cloud resources. The automation process of configuring different instances are done with Ansible [38] tool which is also executed in a Fuzzy auto-scaler instance for configuring the launched instances. The Fuzzy auto-scaler instance is in control of the resources of the AWS cloud cluster; it can start or stop VMs based on the metrics retrieved from the HAproxy instance. The client group shown in (Figure 4.1) is replaced with a t2.micro instance that has Wikijector installed to fire the real Wikipedia trace file's request to the MediaWiki server cluster we used to test the performance of the auto-scaler.

4.3 Metrics

There are many Linux commands which can be used to monitor the performance metrics of each computer. The 'Uptime' command is used to obtain the CPU Load in this system. The response time is gathered from the HAproxy's statistic page's CSV file link via 'curl' command. The 'telnet' command is used to retrieve Ganglia's metrics data of all the online virtual machines via port '8649'. For the metrics correlation analysis, we need far more metrics provided from the default metrics of Ganglia. The Ganglia's 'gmetric' command is used to send the extra performance metrics like 'Memory Cached' to Ganglia master node which resides in the Haproxy virtual machine. We could easily obtain the virtual machines metrics locally via Linux command like 'SAR' and pass the metric value to 'gmetric' with a shell script to constantly send the new metrics value to Ganglia then Ganglia will store those extra metrics from each online virtual machines for later analysis. The cluster size value can be retrieved from various methods since all currently launched and configured virtual machines will connect to Haproxy and Ganglia, so they can provide the cluster size information. Also, Boto3 can directly call the AWS's APIs to provided more detailed information about the status of each virtual machine. Each virtual machine has a life cycle in AWS platform [36], for example knowing a virtual machine is in 'pending' state is extremely useful to know when the right time is to trigger Ansible to configure the virtual machine, otherwise running Ansible when the newly launched virtual machine is still booting up will cause a lot of troubles.

4.4 The proposed algorithm

The algorithm 1 first gathers current average response time and current cluster size from the load balancer instance (*AllLaunchedInstance.getStats*, Line 9), then it will input current response time and current cluster size to set the dynamic upper-threshold (*upperThresholdFuzzy*, line 10). If the current response time breaches the adaptive upper threshold twice consecutively (line 11 and 12), the new cluster size fuzzy engine is called (clusterSizeFuzzy, line 13) to determine new cluster size to handle the request load. For each scaling out or in, the *warmUpTime* is set to 30 seconds which means in this 30 seconds we do not allow any scaling policies to be executed. The reason for setting warm-up time is that the newly launched VM normally has an unstable metrics due to the operating system's bootstrapping and all the configurations of the applications. Therefore we need some time to wait for the metric to become stable. For scaling in, we loop through each running VM and check if their CPU load is under 50% (considered idle, line 23) and if the same VM is under 50% of CPU load twice consecutively (line 25), it will be stopped and the warm-up time is updated to 30 seconds. The scaling loop will execute every 5 seconds (line 32), that is because checking the metrics too frequently adds pressure to VMs, and some metrics do not have high-resolution updates as well. *setConsecutiveBe-*

Algorithm 1 Fuzzy Auto-scaler

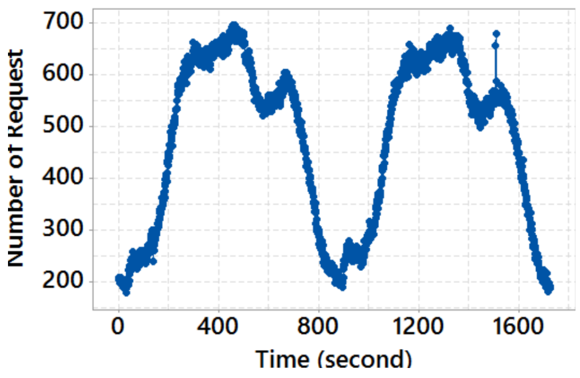
```
procedure StartScaling
    idleLoad $\leftarrow$ 50
    $warmUpTime \leftarrow$ -1
    $checkTime \leftarrow$ 5s
    $consecutiveOverThreshold \leftarrow$ false
    $RTime \leftarrow 0$                                  ▷ Current Average Reponse Time
    $CSize \leftarrow 0$                                  ▷ Current Cluster size
    while true do
        RTime, CSize $\leftarrow$ AllLaunchedInstances.getStats()
        $upperThreshold \leftarrow$ upperThresholdFuzzy(RTime, CSize)
        if $RTime >$ upperThreshold then
            if $consecutiveOverThreshold == true$ and $warmUpTime <= 0$ then
                newClusterSize $\leftarrow$ clusterSizeFuzzy(curLoad,CSize)
                scalingUp(newClusterSize)
                $warmUpTime \leftarrow$ 30s
                $consecutiveOverThreshold \leftarrow false$
            else
                $consecutiveOverThreshold \leftarrow true$
        else
            $consecutiveOverThreshold \leftarrow false$
        if warmUpTime $<=$ 0 then
            for Instance in AllLaunchedInstances do
                if Instance.load $<$ idleLoad then
                    $Instance.consecutiveBelowThreshold += 1$
                    if $consecutiveBelowThreshold == 2$ then
                        $Instance.stop()$
                        $warmUpTime \leftarrow$ 30s
            if $warmUpTime != 0$ then
                setConsecutiveBelowThresholdToZero(AllLaunchedInstances)
        else
            setConsecutiveBelowThresholdToZero(AllLaunchedInstances)
        sleep(checkTime)
        $warmUpTime -= checkTime$
```

lowThresholdToZero is used to clear the array used to record the idle instances. We only want to execute the scaling in when there are instances that are consecutively breaching the *idleLoad* (lower threshold). Both *upperThresholdFuzzy* and *clusterSizeFuzzy* fuzzy logic engines are implemented in Java FuzzyLite library [39] which uses the '.fis' file created by MatLab Fuzzy Logic Designer [40].

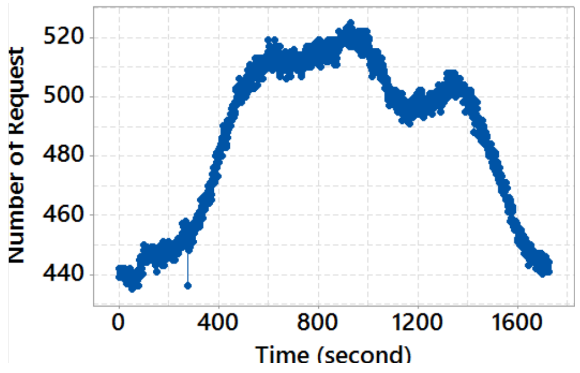
4.5 Experimental Setup

The duration of each experiment is kept at 30 minutes. This is due to the fact that all standard t2.micro type instances only have 30 CPU credits allocated when launched and the use of full power of the CPU cause 1 point each minute. Once the CPU credits are exhausted the VM resources will be restricted to 10% of CPU utilization [41]. In the experiment, we have a maximum of 10 t2.micro instances, and the desired response time is set to 200ms for the SLA. For the AWS native and emulated auto-scaler their upper threshold is 100ms (static). The fuzzy auto-scaler uses the adaptive upper-threshold from the fuzzy engine. Since AWS native auto-scaler cannot easily use individual VM's metric to stop the specified VM, both AWS native and AWS emulated auto-scaler have a higher lower threshold at 70% CPU Load (average of all launched VMs). The fuzzy auto-scaler has a lower threshold at 50% (for each VM). We also deliberately use a minimal cluster size at the beginning to show how auto-scalers perform under such circumstances, so there will be many SLA violations at the start.

4.5.1 What trace file is used?



(a) The trace file with two full day trace of Wikipedia



(b) The one day trace file with slow increase of number of request

Figure 4.2: The trace files used in the experiments, both of them are 30 minutes long

The Wikipedia trace file is sent by Wikijector to benchmark our auto-scaler's performance. The trace files (Figure 4.2a) used in the experiments to evaluate the auto-scaler are all produced from the real-life Wikipedia trace files. The trace files used preserved the 'shape' of the original trace file and only the frequency of the number of requests in each second is changed. The trace file is extracted from the two day Wikipedia trace and scaled into 30 minutes. The trace file has two peaks, which are the highest number of request received during each day, since it originally consists of two-day traces. The auto-scaler developed in this thesis is a reactive one which does not rely on the historical data, so the peak shape (increase then decrease) pattern in the trace file is suitable to evaluate the ability of the reactive method to decide the scaling process for saving the cost and satisfying the SLA.

4.6 Remarks for performing experiments in AWS

Even this section is not directly related to the auto-scaler performance; we still want to address it here for people who want to use AWS for their auto-scaling experiments. The free tier provided by AWS is excellent for a researcher who cannot afford huge amount of cost on running experiments. However, we can only use the t2.micro instance in this case. The t2.micro instance is the shared instance, which means we need to share the hardware with other users. The experimental results might have slight variations in each run of the experiment, but the result should not vary too much since the isolation features in AWS's vitalization technique is mature. AWS has the limit of the maximum number of instance we can use at the start to prevent inexperienced cloud user accidentally launching too many virtual machines. Therefore, before start running experiments, we should request a larger number of virtual machines we can use. The CPU credits will get reset only when the instance is restarted. Therefore, we should prepare to have an algorithm to detect the CPU credits to stop and start the virtual machine if we are planning to run any experiment longer than 30 minutes, but this might complicate your auto-scaler.

4.7 The overall performance in terms of the cost saving and the SLA satisfaction

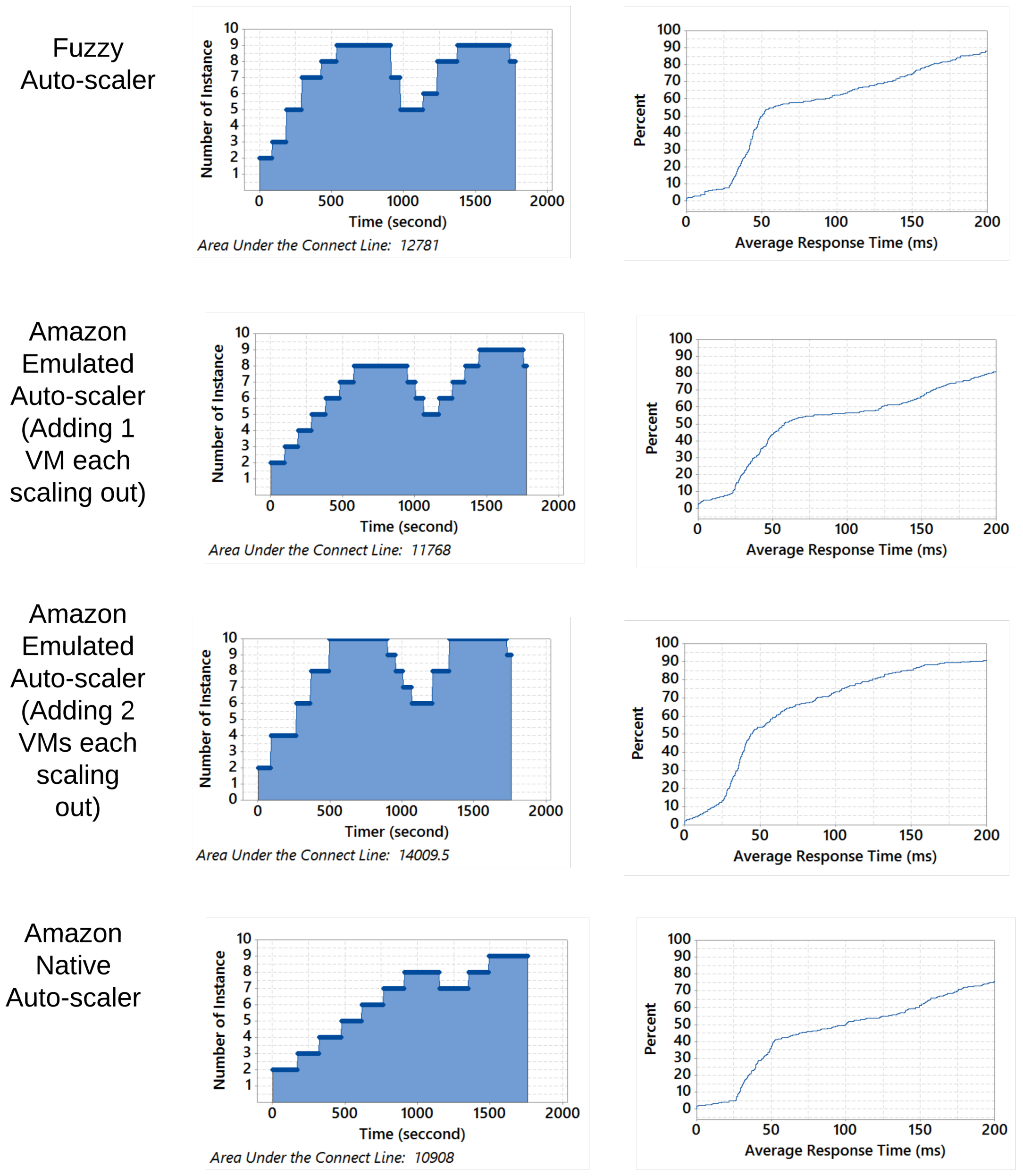


Figure 4.3: The experiments results of the auto-scalers

Table 4.1: Experiment result summary (trace file - Figure 4.2a)

Auto-scaler	Response Under 200ms (%)	Cost (Area Under Graph)
Fuzzy	87.97	12781.00
AWS emulated (adding 1 VM)	80.75	11768.00
AWS emulated (adding 2 VMs)	90.43	14009.50
AWS native	75.65	10908.00

Table 4.2: Experimental results summary with respect to AWS native auto-scaler (trace file - Figure 4.2a)

Auto-scaler	Response Under 200ms Change (%)	Cost Change (%)
Fuzzy	+12.32	+17.17
AWS emulated (adding 1 VM)	+5.10	+7.88
AWS emulated (adding 2 VMs)	+14.78	+28.43
AWS native	0.00	0.00

AWS native, as a baseline, performs the worst by only satisfying SLA for 75.65% (Table 4.1). In Figure 4.3, the CDF (Cumulative Distribution Function) graph of the average response time under 200ms shows that the fuzzy auto-scaler is about 7.22% better than the AWS emulated auto-scaler (adding one VM for each scaling out) and 12.32% better than the AWS native auto-scaler (Table 4.2). The cost of each auto-scaler could be calculated from the area under the step-curve in 'Number of Instance vs Time' scatter graph (Figure 4.3). The cost of the AWS native one is the lowest since it is often under-provisions resources. This implies that the higher SLA satisfaction requires more resource provisioning hence higher cost required (i.e trade-offs between SLA satisfaction and resource cost). 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 shows that the fuzzy auto-scaler performs less number of scaling out to reach the desired cluster size, whereas both AWS native auto-scaler and AWS emulated auto-scaler (adding one VM) requires more time to reach the desired cluster size. The fewer number of scaling means quicker responses to changes of the load and less time wasted on the orchestration of new instances. Therefore, adding multiple virtual machines 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 those metrics are not reliable to determine the actual request load and we need to use them very conservatively. Comparing both AWS emulated auto-scaler (adding one VM) and the fuzzy auto-scaler, they increase SLA satisfaction by 5.1% and 12.32% respectively. To examine the efficiency of using cloud resources between AWS emulated (adding 1 VM) and fuzzy logic, we can see that with 1% increase in SLA satisfaction, the increase of cost is 1.55% for emulated auto-scaler (adding 1 vm) (5.1% SLA satisfaction increase with 7.88% cost increase) and the fuzzy auto-scaler improves SLA satisfaction by 1% will have 1.39% increase in the cost (12.32% SLA increase and 17.17% cost increase). The fuzzy auto-scaler uses the resources more efficiently to provide better SLA satisfaction. In this case we also observe that providing better SLA satisfaction will cause the cost of cloud resources become higher, since the violation of SLA is mainly due to the lack of provisioned virtual machines, so in another word it is not possible to maintain SLA with small cloud resources if the incoming load is increasing.

When AWS emulated auto-scaler uses the strategy of always adding two virtual machines for each scaling out step, its SLA satisfaction outperforms our proposed auto-scaler. However, the increase of SLA satisfaction is traded by using 11.26% more cost. The auto-scaler aims to minimize the cost and maximize the SLA satisfaction. Therefore, simply adding more virtual machines to get better performance without considering the cost will lead to a huge bill for the web application provider.

4.8 Dynamic scaling-out evaluation

Figure 4.3 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 other auto-scalers only use the maximum of 9

VMs without violating the target 200ms reponsetime in the SLA. This happens due to the lack of ability to dynamically adjusting the number of VMs needed to be added to the cluster size. The number of virtual machines to add to the cluster also affects the number of scaling out required to reach a point that the SLA and the upper-threshold are both satisfied. The static number of provisioning can hardly balance the number of scaling out and the efficient use of the cloud resources. For example, the AWS emulated auto-scaler which only adds one new virtual machine for each scaling out requires 6 times of scaling out to reach a point that the upper-threshold is not breached. Increasing the static number to add 2 virtual machines each time in emulated auto-scaler reduces the scaling out number to 4, but adding 2 virtual machines each time could have the consequence of adding an unnecessary extra virtual machines.

The static number of adding in virtual machines causes the auto-scaler waste cluster resources. 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 comparing to the fuzzy auto-scaler to reach this result. With respect to the AWS native auto-scaler, AWS emulated auto-scaler (adding 2 VMs each scaling out) needs 1.92% increase in the resources to achieve 1% (i.e. 14.78% SLA satisfaction increase with 28% cost increase) increase of the response time under 200ms, while fuzzy auto-scaler only needs 1.39% (i.e. 12.32% SLA satisfaction increase with 17.17% cost increase) and the maximum of 9 VMs is used in fuzzy logic auto-scaler also implies that the 9 VMs is enough to handle the maximum loads without breaching 200ms response time (SLA) threshold in this trace. The use of the dynamic number for scaling out the cluster is essential to keep the cloud user's cost low and maintain decent SLA satisfactions. The dynamic number of provisioned virtual machines is computed based on the fuzzy engine that takes CPU Load and cluster size into account as inputs (mentioned in Chapter 3). From this experiment, we can also see that the CPU Load has potentials to determine the optimal number of virtual machines needed for handling current incoming traffics. Since the virtual machines we used are all t2.micro instance, the CPU Load does not need to take the different types of instance's computational power into consideration. Hence, if we want to use a different type of virtual machines, we might need to know the CPU Load proportion between each type of virtual machines. For example, 100% of CPU Load might be equivalent to 50% of CPU Load for a stronger type of virtual machine.

4.9 Adaptive upper-threshold evaluation

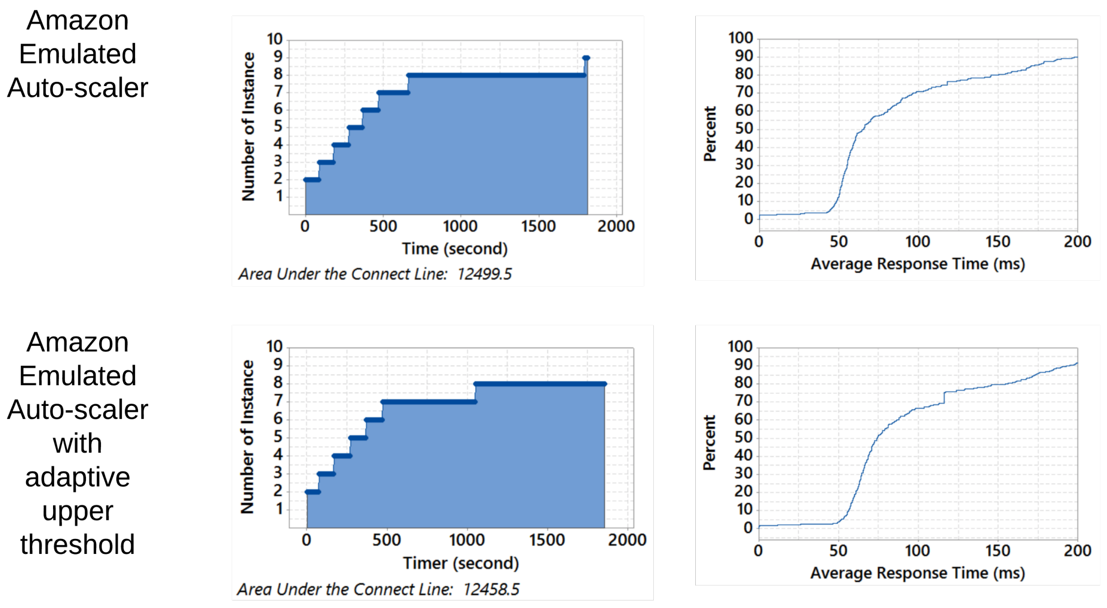


Figure 4.4: The adaptive upper threshold experiment result

Due to the limitation of the experiment time, we are not able to run the experiment over a full day, so the change in the request rate happens considerably quick in Figure 4.2a. Therefore, a 30-minute version of the one-day trace file is made (Figure 4.2b). The maximum and minimum request number has a smaller difference (slow increase and decrease in requests number) to simulate lower fluctuations of the request load.

In this experiment, only the two AWS emulated auto-scalers are used. The only difference between these two AWS emulated auto-scalers is that one uses the adaptive upper-threshold produced from our fuzzy engines and the other one uses the static upper threshold, which is the response time, is set to 100ms.

As shown in Figure 4.4, the AWS emulated auto-scaler with adaptive upper threshold keeps the cluster size at 7 instances, which is longer than the AWS emulated autoscaler with static upper-threshold. The percentage of response time under 200ms are very close for both methods in this case, while AWS emulated with adaptive threshold has 91.74% and the non-adaptive upper threshold one has 90.14%. Due to the restriction of the experiment time, the cost saving is not clearly observable, but we can expect that the dynamic adaptive upper threshold utilizes the cloud resources more efficiently in a long run, since small load fluctuation is typical in web applications. Normally the web application requires high availability so that people can access it anytime, so there will be many small fluctuations of response time during the long run of the web application hence it is highly likely that the dynamic upper-threshold could save us a significant amount of money. This experiment shows that the dynamic upper-threshold is desired since the larger cluster size has stronger ability to prevent the fluctuation of the incoming traffic from breaking the SLA. Hence we should increase the upper-threshold when the cluster size is larger to prevent unnecessary trigger of scaling out process to save the cloud resources cost but still not violate the SLA.

4.10 The effectiveness of using CPU Load as an provisioning indicator metrics

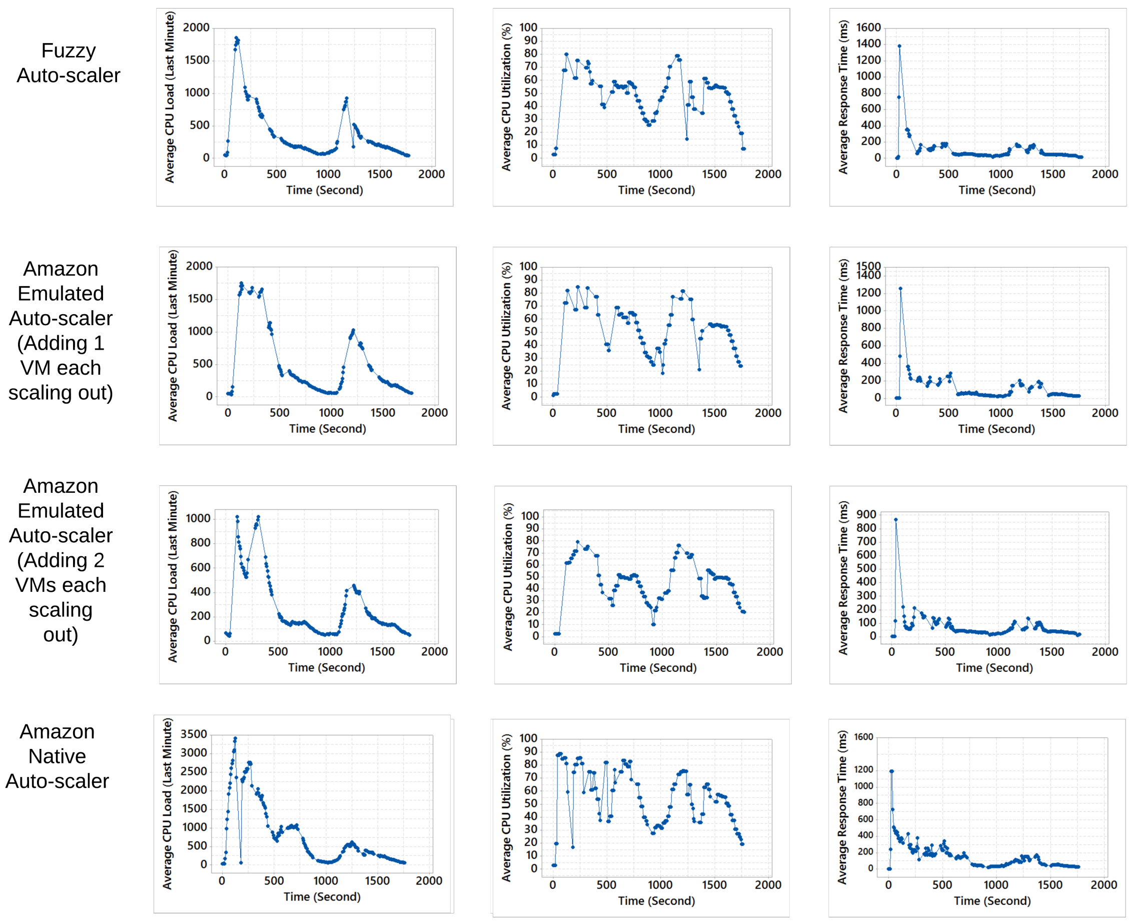


Figure 4.5: The experiments results of the auto-scalers metrics

The Figure 4.5 shows the CPU Utilization during the entire experiment time for all the auto-scalers. From the CPU Utilization figure, we can see that it does not perfectly match the load represented in trace file (Figure 4.2a). Dramatic increases in the CPU Utilization in some places does not follow the load changes in the trace file. In Figure 4.5 CPU Load shows two peaks matching to the two peaks in the Mediawiki trace file and the response time in Figure 4.5. The reason for the second peak is smaller is that the initial number of MediaWiki instances is considerably small. The small number of virtual machines results in the CPU Load shoot-up and is caused by significant disproportionate growth of incoming traffic and the computational power in the beginning. The other metric which is directly related to the incoming traffics and the SLA is the response time. In response time graphs in Figure 4.5, there are also two peaks reflecting the patterns in the MediaWiki trace file. From the observations of CPU Utilization, CPU Load and Response time in Figure 4.5, we can see that CPU Load is not a wrong choice to indicate the computational load of the incoming traffics, hence the use of CPU Load can effectively determine the number of new virtual machines needed. CPU Load can represent the overall system usage. It can potentially work with some other specific metrics to work out which resource in the virtual machine is causing the SLA to be broken. It could be the slow IO, or there is not enough memory causing many page faults or the CPU does not have enough cores to perform parallel processing. We can try out more combinations of CPU Load with other system metrics to have a detailed view of the virtual machine's stress on particular resources and then we can choose the best type of the virtual machine to be added in our cluster.

4.11 Experimental results summary

From the experiments, we can see that the used trace file has very regular patterns and is very suitable to model it in forms of time series. Therefore, when the time is up to a certain point, we can be sure that the load will be increasing hence we will take action to handle the future load before it occurrs. However, we aim to develop an auto-scaler that does not require the use of historical data for medium and small size web applications.

The experimental results show that our fuzzy logic based reactive auto-scaler improves the static threshold's approach. The dynamic threshold does help us to save the cost by increasing the resilience of our web application when handling the fluctuation of incoming traffics. The CPU Load also prove that it can better reflect the usage of the cloud resources and shows a strong correlation to response time, whereas the CPU Utilization metric does not have a strong pattern matching with the trace file we used. Our auto-scaling rules and CPU Load metrics together provide decent performance on estimating the number of new virtual machines needed for keeping the response time from violating the SLA. The accuracy of estimating the provisioning cloud resources is important to lower the number of scaling out steps, save the cloud resources cost and improve the SLA satisfaction. In future, we will compare reactive and proactive auto-scalers. Since each of these auto-scaler takes different approach that one studies historical data and one does not. The trace file used in this experiment will surely favour the proactive method since the trace file has regular trace patterns which are easy to model. The proactive methods are mostly model-based approachs which requires observing the historical data to predict the expected patterns so if we use a trace file that the proactive auto-scaler never get trained before it might put the proactive method in disadvantage. However, from the experimental results' point of view, we are confident with the usage of the fuzzy logic engines to dynamically change upper threshold and dynamically estimate the new instance needed for scaling out.

This page intentionally left blank.

Chapter 5
Conclusions

In this chapter, we will summarise the key ideas in this thesis and list out the possible future works based on the limitations of our proposed fuzzy logic based auto-scaler.

5.1 Summary

In this thesis, we had explored different possible methods to tackle the problems of performing auto-scaling for web application. We believe that the trend of Cloud computing will keep getting more popular, if the deployment of the distributed applications becomes easier and there exists a general purposed auto-scaler that could automatically handle the cloud resource provisioning to maximise the cloud user's profits and their applications' SLA satisfaction. Clouds with horizontal scaling provides us the infinite computational resources and the pay-as-you-go feature, which is very attractive for small or medium size businesses who do not have enough budget to pay upfront cost to set up their servers and do not want the resource provisioning issues stops them from growing big. Horizontal scaling is a possible solution for developing a large application which is not going to be bottlenecked by the lack of computational resources. This thesis conducted an experiment on how different type of user requests can cause the profiling method to fail for using a trace file that has very different load patterns to the actual load patterns.

We used fuzzy logic to develop our reactive auto-scaler. Fuzzy logic is very intuitive to understand, and the rules of auto-scaling can be well expressed with its linguistic rules. For a web application, the response time of each user request is crucial to provide appropriate QoS and to have your website users satisfied with the QoS of your web application. The high response time generally implies that the web servers are overloaded and do not have enough computational resources to serve the user request as fast as possible. Therefore to maintain the SLA satisfaction, the cost increase of using cloud resources is reasonable.

In our proposed auto-scaler, CPU load is used as the metric for determining the stress of the cluster since it reflects the overall average system resource usage in Linux based operating system. It is essential to use the correct metric to identify the usage of the computational resource of your cluster then you can provide the optimal amount of cloud resources to prevent you from spending significant budget on satisfying the SLA. The experimental results show that the CPU Load is a decent metric for estimating the number of virtual machines we need.

The fuzzy logic based auto-scaler is benchmarked with AWS auto-scalers and the result shows that the proposed fuzzy logic method does improve the SLA satisfaction while saving the cost of cloud resources at the same time. The fuzzy logic approach increases the percentage of the requests serviced within 200ms by 12.32% and each 1% of SLA satisfaction increases only 1.39% more cloud resources compared to the Amazon native auto-scaler. The experimental results also showed that the dynamic upper threshold produced by the fuzzy engine with current cluster size and response time as inputs helps with increasing the tolerance of the response time fluctuations to avoid unnecessary scaling out. The fuzzy engine is also used to dynamically determine the number of virtual machines needed and it shows the strength on quickly satisfying the SLA but not using unnecessary cloud resources. Overall, the thresholds and provisioning virtual machines should all be adaptive to prepare optimal auto-scaling plans for each state of the cluster. Since the cluster's incoming loads and observed metrics are always changing, using a static auto-scaling rule is not able to deal with all the situations.

5.2 Limitations and Future works

The auto-scaler proposed in this thesis is still far from being perfect, more investigation needed to be performed in this area. The following will list out some of the possible improvements on the proposed auto-scaler base on its shortcomings.

5.2.1 Multiple tier auto-scaling strategies

Even though the MediaWiki cluster is set up in a multiple tier fashion, our proposed auto-scaler only explores the possibility of scaling the MediaWiki web server instance but does not take the load balancer instance and the database instance into consideration of the auto-scaling. Even though the load balancer and database are normally hosted in a stronger instance and less likely to get bottlenecked comparing to web server instances, to have a robust web application cluster, we should not leave them out from the auto-scaling process. If the load balancer instance and database instances reach their limit, then the scaling out of the middle tier (MediaWiki server instance) becomes meaningless. The auto-scaling of load balancing tier and database tier requires more considerations, since they are shared between multiple MediaWiki server instances and have a high risk of failing the entire system if they are not available. The load balancer instance could possibly be more advantageous with vertical scaling to add in more powerful hardware while keeping the same IP. The database cluster can use the distributed database technology, which can automatically handle the data synchronisation and load distributions, but we still need to determine a way to know when to perform auto-scaling and how many new database virtual machines are needed.

5.2.2 Predicting seasonable incoming loads

The proactive method could not predict the unexpected sudden change of the incoming load. However, if the load of the application is very stable and regular, the proactive method is the best by performing early scaling to mitigate SLA violation. The reactive method does not take the load patterns into account so that it is not the best method when the load pattern is predictable. It is worth to explore combining our reactive auto-scaler with a proactive method to mitigate the shortcomings of both approaches. We could still use the reactive fuzzy logic based auto-scaler to perform auto-scaling and switch to proactive auto-scaler method whenever is appropriate. However, it is a hard problem to tell when to use the reactive method and when to use the proactive method, since witnessing unexpected load patterns in the proactive method might result in undesired scaling process hence causing our web application to have low SLA satisfaction.

5.2.3 Sticky session support

Nowadays most of the web application are session based. The session is used to keep the active client's current activities states so that the user can resume their actions afterwards. From the perspective of user experience, having session based web application is very attractive. However, the active session involves data used to remember client's activities; those data could cause a problem when the virtual machine is classified as an idle resource and required to shut down immediately to save the resource cost. The active session data in the idle resource forces us to wait for all the user connections to be closed before shutting down the virtual machine. It is very costly to run virtual machines to handle very few clients before they are disconnected. In this thesis, we only deal with static web pages that the user does not have to repeat a series of process to reach to their previous state in the website. Therefore, it is easy to take down the server, and the impact on user experience is minimal.

5.2.4 Auto-scaler for multiple cloud

The auto-scaler developed in this thesis only works for AWS's APIs, since most of the cloud platforms tend to have their own set of APIs so it is hard to make a generic auto-scaler which can be used in multiple clouds. If the auto-scaler can use APIs from all the platforms, we can wisely select the most competitive virtual machine with minimum cost and most robust computational resource to save more cost while meeting our SLA. As discussed in Chapter 2, if the web application can be hosted in multiple clouds we also have advantages like not locking into single cloud vendor and fast web application access with different geographically located virtual machines. Since there is no agreement on standardization of the APIs of different cloud providers, the work of making auto-scaler could be a tedious manual task to implement all the APIs of all the cloud providers.

5.2.5 Auto-scaler with differernt performance and cost mode

There are always trade-offs between SLA satisfaction and the cloud resources cost. The auto-scaler proposed here is very conservative, we could implement different mode to let the auto-scaler user decide on the tradeoff between cost and SLA satisfaction. Over-provisioning can prevent the SLA violations for most of the times, but it costs much more and does not take full benefits of using the elastic cloud resources. However, the different application might favour SLA satisfaction more than the cloud resources cost or vice-versa. Therefore, having a different mode for different scaling modes could be useful to provide some degrees of customisation on the auto-scaler.

5.2.6 Dynamic cooling time

Cooling time is a duration used to prevent any scaling executions after the previous scaling process. The provisioning of the new instance to join the cluster requires configurations (e.g start daemon processes), so the cooling time is used to wait for the new instance to be ready to serve the incoming request. Also, the metrics of the newly configured instance could be very high (overloaded) due to all the booting and configuration steps. Therefore immediately using the metrics from those newly configured instances might trigger inaccurate scaling out due to the high metrics value caused by configuration but not from incoming traffics. Currently, the cooling time of our auto-scaler is an approximate value, that means enough for the metrics value to become stable from automation. The auto-scaler might be more responsible (trigger scaling out/in faster) if we can dynamically detect the time when new instance's system metrics are recovered from automation.

5.2.7 Combining auto-scaler and load balancer

The accurate way of estimating the complexity of generating the response body for each user request could be very useful to plan on how many virtual machines we need. As mentioned in Chapter 3 (Input Selection), each user request might require different data resources to respond. The auto-scaler might be able to work closely with the load balancer to estimate the complexity of each request. One possible method to know the complexity of each user request is storing the temperate data about how much time is used to process each request. If we gradually learn all complexity of the request, the load balancer can figure out the strength of the incoming load and give our auto-scaler better information on how to plan the scaling. However, the method proposed here assume the content of the website is stable otherwise it will be very costly to learn all the request's processing time. In the web application's case, each web page might request different database queries and computational work to generate. It is a hard problem to develop an efficient way to know the complexity of each request when received, so we need to perform more researching works in this area. The HAproxy used in our auto-scaler system evenly distributes the user requests to each instance without considering the load of each individual instance. Therefore, the SLA could be violated if there are few instances that would get all the complicated user requests that demand high computational resource to generate the response body. Those virtual machines will accumulate many requests in the queue, and they will be overloaded eventually, since they cannot process the complicated requests fast enough while the load balancer still keeps sending requests to them. The hybrid of auto-scaler and load balancer can achieve efficient load balancing and auto-scaling process for satisfying SLA.

5.2.8 Improving the use of CPU Load

As mentioned in Chapter 3, CPU Load metric can represent the average system resource usage. CPU Load only gives us an overview of the load of the computer resource. However, there are many different system resources can be used by the application for example the web application could be CPU intense or memory intense. Combining CPU Load with metrics like memory usage could give us more information about the demand of specific system resource that is affecting the SLA satisfaction. By knowing what resource is reducing our SLA satisfaction, we can choose a virtual machine with suitable computational resources. For example, if the current incoming load is more memory demanding we can use the virtual machines with a larger memory size. This feature is very suitable for the cloud provider like AWS that offers many different types of virtual machines, we can easily choose the one with the best price for our need of the system resource like memory.

This page intentionally left blank.

Bibliography

[1] Wikipedia, the free encyclopedia, "Cloud computing," 2018, [Online; accessed February 28, 2018]. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/b/b5/Cloud_computing.svg/440px-Cloud_computing.svg.png

[2] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: a taxonomy and survey," *In: ACM Computing Survey (2018)*, 2016.

[3] Andre. (2009) Understanding linux cpu load - when should you be worried? [Online]. Available: http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages

[4] W. Voorsluys, J. Broberg, and R. Buyya, "Introduction to cloud computing," *Cloud computing: Principles and paradigms*, pp. 1–41, 2011.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[6] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, vol. 25, no. 6, pp. 599–616, 2009.

[7] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12*, 2012.

[8] I. Gartner. (2017) Gartner forecasts worldwide public cloud services revenue to reach $260 billion in 2017. [Online]. Available: https://www.gartner.com/newsroom/id/3815165

[9] A. W. Services. (2018) Amazon web services (aws) - cloud computing services. [Online]. Available: https://aws.amazon.com/

[10] ——. (2018) Aws auto scaling. [Online]. Available: https://aws.amazon.com/autoscaling/

[11] L. Yazdanov, "Towards auto-scaling in the cloud: online resource allocation techniques," Ph.D. dissertation, Dissertation, Dresden, Technische Universität Dresden, 2016.

[12] E.-J. van Baaren, "Wikibench: A distributed, wikipedia based web application benchmark," *Master's thesis, VU University Amsterdam*, 2009.

[13] 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*, vol. 12, no. 4, pp. 559–592, 2014.

[14] C. Qu, R. N. Calheiros, and R. Buyya, "A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances," *Journal of Network and Computer Applications*, vol. 65, pp. 167–180, 2016.

[15] L. A. Zadeh, "Fuzzy logic: A framework for the new millennium," in *Fuzzy Logic*. Springer, 2002, pp. 1–6.

[16] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *Proceedings of the 7th international conference on Autonomic computing*. ACM, 2010, pp. 1–10.

[17] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Proceedings of 2012 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2012, pp. 1327–1334.

[18] S. Frey, C. Lüthje, C. Reich, and N. Clarke, "Cloud qos scaling by fuzzy logic," in *Proceedings of 2014 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2014, pp. 343–348.

[19] P. Lama and X. 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.

[20] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "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.

[21] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 195–204.

[22] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications qos," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, 2015.

[23] K. Teknomo. (2014) Queuing theory tutorial. [Online]. Available: http://people.revoledu.com/kardi/tutorial/Queuing/

[24] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. Scokaert, "Constrained model predictive control: Stability and optimality," *Automatica*, vol. 36, no. 6, pp. 789–814, 2000.

[25] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal autoscaling in a iaas cloud," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 173–178.

[26] R. Yanggratoke, J. Ahmed, J. Ardelius, C. Flinta, A. Johnsson, D. Gillblad, and R. Stadler, "Predicting service metrics for cluster-based services using real-time ana-

lytics," in *Network and Service Management (CNSM), 2015 11th International Conference on*. IEEE, 2015, pp. 135–143.

[27] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[28] CNCF. (2018) Horizontal pod autoscaling. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/horizontal-pod-autoscaler.md

[29] M. Matt, "Predictive pod auto-scaling in the kubernetes container cluster manager," 2016. [Online]. Available: https://unbound.williams.edu/theses/islandora/object/studenttheses%3A119/datastream/OBJ/download

[30] I. Ari, B. Hong, E. L. Miller, S. A. Brandt, and D. D. Long, "Managing flash crowds on the internet," in *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*. IEEE, 2003, pp. 246–249.

[31] S. Work, "How loading time affects your bottom line," *KISSmetrics, April*, 2011.

[32] D. An, "Find out how you stack up to new industry benchmarks for mobile page speed," vol. 26, p. 2018, 2018. [Online]. Available: https://www.thinkwithgoogle.com/marketing-resources/datameasurement/mobile-page-speed-new-industry-benchmarks/.Accessed

[33] D. Ferrari and S. Zhou, "An empirical investigation of load indices for load balancing applications," California Univ Berkeley Dept Of Electrical Engineering And Computer Science, Tech. Rep., 1987.

[34] R. Walker, "Examining load average," *Linux Journal*, vol. 2006, no. 152, p. 5, 2006.

[35] B. Gregg. (2017) Linux load averages: Solving the mystery. [Online]. Available: http://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html

[36] A. W. Services. (2018) Instance lifecycle. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-lifecycle.html

[37] ——. (2018) Aws sdk for python (boto3). [Online]. Available: https://aws.amazon.com/sdk-for-python/

[38] M. DeHaan. (2018) Ansible. [Online]. Available: https://www.ansible.com/

[39] J. Rada-Vilela, "Fuzzylite: a fuzzy logic control library," 2014. [Online]. Available: http://www.fuzzylite.com

[40] M. U. Guide, "The mathworks," *Inc., Natick, MA*, vol. 5, p. 333, 1998.

[41] A. W. Services. (2018) Cpu credits and baseline performance. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/t2-credits-baseline-concepts.html