National College of Ireland

Enhancing Static Auto-scaling Approach to Mitigate Resource Over-Provisioning in Cloud Computing

MSc Research Project
Cloud Computing

Manasi Patil
Student ID: x17156742

School of Computing
National College of Ireland

Supervisor: Adriana Chis

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Manasi Patil
Student ID:	x17156742
Programme:	Cloud Computing
Year:	2018
Module:	MSc Research Project
Supervisor:	Adriana Chis
Submission Due Date:	20/12/2018
Project Title:	Enhancing Static Auto-scaling Approach to Mitigate Resource Over-Provisioning in Cloud Computing
Word Count:	7775
Page Count:	29

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	24th January 2019

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing Static Auto-scaling Approach to Mitigate Resource Over-Provisioning in Cloud Computing

Manasi Patil
x17156742

Abstract

The elasticity property in cloud computing is favorable for both cloud providers and consumers because of its automatic adaptation to the dynamic workload that the application might experience. There are various approaches for scaling and auto-scalers choose the one to maximize the efficiency based on their application; keeping a balance between SLA violations and the cost of the resources. In almost all the approaches, the efficiency of the auto-scaler is directly proportional to the performance overhead it incurs. Hence, even the most popular approach, which can foresee and prepare itself to adapt to the dynamic workload beforehand, affects the performance of the auto-scaler due to its complex algorithms. This research, thus, focuses on improving a static auto-scaler by mitigating its drawback of resource over-provisioning. Additionally, the proposed solution incurs a negligible performance overhead. The paper introduces an algorithm which leverages the static auto-scaling to provide a solution to avoid over-provisioning. Consequently, the overall cost of the resources used by the application decreases. The results of the empirical evaluation show that the cost of the resources can be decreased by 20-25% depending on the scale of the application.

Keywords: Auto-Scaling, Resource over-provisioning, CloudSim Plus simulation, Cloud Computing, Elasticity, Static Auto-Scaling, MAPE model, Horizontal Scaling

1 Introduction

Cloud Computing has led to many innovations and ease of running varied-scale businesses in todays world. Its extensive range of applications comes from its thorough capabilities, applicable from personal-interest projects to large-scale businesses (Ghobaei-Arani et al.; 2018). One of the features of cloud computing is elasticity. Mell et al. (2009) state the following about elasticity:"Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand". When applications need extra resources than what is already allocated to them, auto-scaling plays a major role in monitoring and analyzing the need for the amount of resources at a particular time period. There are many models such as MAPE which the auto-scaler follows in order to enable the elasticity of the application (Lorido-Botran et al.; 2014). The scaling can either be horizontal (i.e. the instances are created or removed as a whole) or vertical (i.e. specific changes are made to the existing instances) (Chen et al.; 2018). Horizontal scaling is comparatively more widely than vertical scaling used type of scaling in cloud applications (Chen et al.; 2018). Hence, it can be said that

the auto-scaler decides the number of resources as a whole to be allocated to match the current demand as close as possible (Ghobaei-Arani et al.; 2018; Herbst et al.; 2013). Since there is a low probability of resources matching the demand exactly every time, it results in over- or under-provisioning (Ghobaei-Arani et al.; 2018).

Over-provisioning occurs when the number of resources is more than enough to meet the demand of the application (Ghobaei-Arani et al.; 2018). On one side, since, the commercial cloud providers use pay-per-use scheme for leasing the resources, the over-provisioning condition causes extra cost to the application owner (Ghobaei-Arani et al.; 2018; Buyya and Son; 2018). Besides, extra resources also utilize excess energy and electricity (Buyya and Son; 2018; Aslanpour et al.; 2017).

Under-provisioning, on the other hand, can violate SLA rules and result in loss of revenue (Ghobaei-Arani et al.; 2018). Because of the overload of resources, some functionalities may fail which can affect the reputation of application owner as well as the cloud provider.

To avoid both these scenarios (i.e. over- and under-provisioning), there are many methods provided by cloud providers in auto-scaling, so that the demand meets the resources as much as possible. One of the most classic and widely used methods is static or rule-based auto-scaling approach (Qu et al.; 2018; Chen; 2016). The approach follows the basic rule of provisioning a specific number of resources when the demand reaches a specified threshold (Chen; 2016). These values (threshold and number of resources) are to be fixed by the application owner before the application is hosted. Although it avoids complexity, it loses the flexibility of the application to adapt with respect to the demand (Qu et al.; 2018). Moreover, due to the lack of insight of how the workload of a application varies over time, there is always a probability that the application provider will specify a high value for the number of resources to be provisioned when there is a condition of over-provisioning, to avoid SLA violations (Chen; 2016). This method definitely will lower the probability of degrading their reputation and loss of customers but at the same time, the cost will increase due to provisioning the extra resources (Al-Dhuraibi et al.; 2018). Though there are many more methods which try to provide an efficient approach, there needs to be a way to avoid resource over-provisioning when using a static approach (as this is preferred over other approaches due to is negligible overhead) (Lorido-Botran et al.; 2014; Chen; 2016).

The main purpose of this paper is to solve the issue of resource over-provisioning in static auto-scaling. The proposed approach keeps the performance overhead under control, since it is one of the main reasons for the application owner to choose the static approach in the first place. The research introduces an algorithm built upon static auto-scaling which calculates the number of resources to be de-allocated safely without violating the SLA rules.

The remaining of the paper is structured as follows. Section 2 discusses different approaches of auto-scaling and provides the justification of enhancing the static mode, through the past research performed with similar interests. Section 3 states the basic methodology followed to solve the issue of over-provisioning in order to avoid the excess cost to the providers. Its implementation in cloudsim is explained in detail in section 4. Section 5 evaluates the performance, cost of resources and compares the results between the traditional and proposed approach. Finally, the concluding remarks and future scope is provided in section 6.

2 Related Work

This section provides the detailed information about how an auto-scaler works and critically analyses the revelant approaches followed by auto-scalers in the past.

2.1 Background on MAPE model

Auto-scaling is a continuous process which is iterated multiple times throughout the application lifetime. It follows an autonomic control loop which IBM proposed and referred to as MAPE model (Jacob et al.; 2004). The MAPE model is followed by many researchers as their base to propose various ideas to solve issues regarding auto-scaling. MAPE model has the following steps: Monitoring, Analysis, Planning, and Execution (Chen; 2016; Qu et al.; 2018). The steps in MAPE are discussed below.

Monitoring the resources allocated to the application is the first step of the model. The parameters for monitoring or key performance indicators (KPI) of the workload can vary from application to application. It can be the number of users for any hosted website (Kesavulu et al.; 2018), CPU utilization of virtual machines (Anand et al.; 2012), or resource usage in any application (Dhingra et al.; 2012). The efficiency of auto-scaling depends on these performance indicators as well as the time interval in between monitoring (Qu et al.; 2018). The information gathered in this phase will be forwarded to the analysis phase which will provide the model with the capability of deciding the stability of the application (Qu et al.; 2018).

The **Analysis** phase will analyze the information gathered by the monitoring phase for making further decisions. The monitored data is evaluated to decide to either scale up or down the resources (Qu et al.; 2018). In this phase, auto-scaler not only makes a decision to scale but also decides the schedule of the scaling. The decision for scheduling is based on the configuration of the auto-scaler to either predict the future workload or to wait for the workload to actually alter significantly (Lorido-Botran et al.; 2014). According to this configuration, there are two modes namely reactive and proactive.

The **Reactive** configuration follows the basic auto-scaling method. Whenever the workload varies significantly, the auto-scaler will allocate or deallocate the resources according to the threshold values (Galante and De Bona; 2015). It does not predict the future and only works with the current workload to decide the allocation of the specific number of resources (Galante and De Bona; 2015). Hence, this technique is suitable only for applications having a consistency in their workload graph (Qu et al.; 2018; Liao et al.; 2015). If in case, there is a sudden burst in the workload, the reactive-configuration auto-scaler will require some amount of time, termed as provisioning time, to allocate the resources (Lorido-Botrán et al.; 2012). This delay in provisioning while preparing and migrating the resources might violate SLAs of the cloud providers. Besides this drawback, reactive configured auto-scaler is used by many applications like Amazon, RightScale, Scalr and other commercial and academic applications since it does not have performance overhead due to an absence of computing complexity for prediction algoritms (Galante and De Bona; 2015).

Another type of configuration is termed as **Proactive**. To overcome the drawback of provisioning delay in reactive configured auto-scaling, the predictive configured auto-scaler tries to be prepared with the number of resources to be allocated beforehand. It anticipates the future workload and configures the system accordingly (Galante and De Bona; 2015). For predicting the future workload, it uses information based on the

monitored data, both current and historical (Galante and De Bona; 2015). Some of the work using predictive technique are performed by Galante and Bona (2012), Dawoud et al. (2011), Meinel et al. (2011), Roy et al. (2011), Gong et al. (2010). Unlike reactive-configured systems, predictive-configured systems are suitable for applications with varying workloads, since it mostly detects it beforehand and lessens the provisioning time. Along with the advantages, proactive systems have a disadvantage of over or under-provisioning of resources whenever the system results in incorrect prediction (Lorido-Botran et al.; 2014). Additionally, it also incurs performance overhead due to its complexity in forecast-computing (Lorido-Botran et al.; 2014).

Planning is the subsequent step in the MAPE model. After getting an idea of the workload and determining the necessity of resources for the application for scaling, the planning phase is in charge of keeping a balance between SLA compliance and the cost of allocated resources (Lorido-Botran et al.; 2014). Basically, this phase will estimate the exact number of resources needed to manage the workload and to keep up with the SLA (Qu et al.; 2018). Since the need for resources will vary according to the workload, the auto-scaler needs to provision resources as close to the demand as possible, which is not an easy task (Lorido-Botran et al.; 2014). There are several configurations which help the auto-scaler to do the same. Lorido-Botran et al. (2014) classified the configurations in five categories, two of which are described in the subsequent subsection 2.2.

Execution is the last step which is followed by the model. As the name suggests, this phase is responsible for executing the plan which was decided in the previous phases (Lorido-Botran et al.; 2014). Executing, here, refers to provisioning or de-provisioning of resources.

2.2 Static mode

Cloud clients find static mode appealing because of its simplicity and intuitive nature (Lorido-Botran et al.; 2014). This mode abides by the traditional rule of provisioning a fixed number of resources whenever the workload varies (Al-Dhuraibi et al.; 2018). The threshold and the number of provisioning resources are provided by the application provider beforehand and are constant irrespective of the current demand (Han et al.; 2012). Thus the number of provisioned resources rarely meets the demand (Al-Dhuraibi et al.; 2018). Hence this mode can be suitable only for applications with a uniform workload graph where the application provider can estimate the resource demands roughly (Lorido-Botran et al.; 2014; Han et al.; 2012). Some examples of static mode auto-scaler are shown in Chen (2016), Wuhib et al. (2012), Maurer et al. (2012), Han et al. (2012), Chazalet et al. (2011), Copil et al. (2013) and Chen et al. (2018).

There are a number of modifications implemented in static mode to overcome the drawbacks and improve the efficiency of the auto-scaler. Every example seems to execute a different approach, nonetheless, follow the similar fundamentals of static mode configuration.

The basic rules of static mode are the thresholds which need to be provided by the application provider for both upper and the lower bound on the workload metric (Liao et al.; 2015). Along with it, the auto-scaler need to provide the number of resources to be provisioned as a constant. As mentioned in Chen et al. (2018), Cloudline (Galante and Bona; 2013) has the options to programmably configure the auto-scaling thresholds, even at runtime as required. This can be done only if the application provider has an expert knowledge about the application and the workload it might experience. In addition to

that, the auto-scaler cannot adapt itself to the dynamic workload. On the other hand, the advantage of this mode is that it is lightweight and incurs no complexity in the design.

Realizing the drawback of a fixed number of resources irrespective of the current workload, Netto et al. (2014) implemented an idea where the auto-scaler will increase the step size (number of resources fixed to scale up or down) holistically at runtime based on the increased number of resources and the current utilization. This type of approach was proved efficient for an application with bursty workload but incapable of improving auto-scaling for applications with all the other types of workloads (Netto et al.; 2014).

Cunha et al. (2014) worked on the same principle as Netto et al. (2014) trying to improve auto-scaling efficiency for all the types of workload. Along with the step size, the aggressiveness (step size increasing intervals as fixed by the application provider (Qu et al.; 2018)) of the parameter was also meant to be dynamically tunable according to the QoS of the application. Although with the dynamicity, the overhead in terms of performance and complexity originated.

After trying to improve the efficiency based on a fixed number of resources, some research revolved around the fixed threshold in static mode. Jeffrey and Chase (2010) and Lim et al. (2009) tried to change the threshold according to scaling. They realized that the application would need large threshold values for a small number of instances running on it. But it wont be suitable for a large cluster of instances. According to their research, threshold will increase after scaling in and decrease after scaling out (Jeffrey and Chase; 2010; Lim et al.; 2009). Hence the dynamically varying threshold will accommodate the varying allocated resources.

An ideal auto-scaler does not always point towards efficiency but also the cost of renting the extra resources (Aslanpour et al.; 2017). Lim et al. (2009) has hence tried to create an auto-scaler which is cost optimal along with being able to fulfill the workload demand. Considering the processing power, configuration cost, incoming workload and the capacity of clouds, Lim et al. (2009) compared the results with Amazon cloud in a specific configuration.

Calcavecchia et al. (2012) and *RightScale* (2018) worked on a totally different approach than the rest. They focused in a decentralized system of P2P network of instances (Calcavecchia et al.; 2012). The network was responsible for sharing their statuses and voting to decide whether to allocate or deallocate resources in the network (Calcavecchia et al.; 2012). It works on the basis of predefined rules and the following the majority in voting (*RightScale*; 2018).

2.3 Dynamic mode

To overcome the drawbacks of over and under-provisioning in static mode, the dynamic mode learns about the workload and its changes to predict the future workload for better resource provisioning (Qu et al.; 2018). Public cloud providers like Profitbricks [1] and CloudSigma [2] abide by dynamically configured auto-scaler (Galante and De Bona; 2015). Unlike static mode, applications using dynamic mode are allowed to have spikes in their workload. Despite the bursty workloads, the dynamic mode manages to scale efficiently. For this, it basically uses machine learning to create a resource provisioning model (Qu et al.; 2018). The machine learning starts without any prior configurations and then modifies itself according to the current workload. Since it follows learning from its own

[1] `https://www.ionos.com/`

[2] `https://www.cloudsigma.com/`

mistakes, many-a-times it violates SLA rules (Ghobaei-Arani et al.; 2018). In addition to it, dynamic mode also incurs additional performance overhead due to its machine learning computation (Chen; 2016).

2.4 Static versus Dynamic Approaches of Auto-Scaling

As described in Chen et al. (2018), although dynamic mode is preferred by many applications over static, there are several reasons why static is still chosen for auto-scalers in applications. Around 20% of applications still choose to use static over other approaches (Chen et al.; 2018; Gandhi et al.; 2018).

Cloud providers find static mode easy to provide in auto-scalers. Additionally, the clients also find it very easy to set-up the thresholds and number of resources in the auto-scaler (Lorido-Botran et al.; 2014). The applications which require a basic auto-scaler without much computation complexity might choose static over dynamic configuration in auto-scaler.

Applications with unvarying workload can suit themselves with static and not worry about the SLA violations and the extra cost of the resources (Aslanpour et al.; 2017; Galante and De Bona; 2015).

The ideal auto-scaler must perform scaling without affecting the normal performance of the application (Aceto et al.; 2013). As mentioned above, the dynamic model has a performance overhead. Hence small applications would prefer static mode because of its negligible overhead (Chen; 2016).

The starting phase of machine learning makes errors before stabilizing itself and hence violates many SLA rules by under provisioning of resources (Qu et al.; 2018; Chen; 2016). It requires many actuations to make on the physical system (Chen; 2016). Additionally, it could incur more cost of resources by over-provisioning. Since, high profile applications can not afford SLA violations, they might choose static over dynamic.

The complex trade-offs are handled only implicitly. Hence making amateur decisions is highly possible (Chen; 2016).

The application providers of academic or scientific applications (Galante and De Bona; 2015) with high intuitive and expert knowledge about the application can decide the threshold and the number of resources required very accurately. Hence, choosing static approach could avoid the above-mentioned drawbacks of dynamic mode and configure the auto-scaler statically.

This section described the working of auto-scaler with MAPE model and critically analysed the approaches followed in the past. Subsequently, it provided reasons for some applications to choose static over dynamic mode. The type, area and the outcomes of the research reveal that there is still a scope for improving the auto-scaling efficiency in static mode.

3 Methodology

This section describes the proposed solution to mitigate the over-provisioning issue. The proposed solution calculates the number of resources that can be deleted safely to maximize the overall CPU utilization. This value is computed using a formula which takes into consideration the upper and lower thresholds defined by the application provider and the current CPU utilization of the resources. In this way, the proposed approach tries to lessen the over-provisioned resources to the application. As mentioned in the section 2,

few applications providers prefer to use auto-scalers with static mode over dynamic mode for auto-scaling their applications either to avoid performance overhead, to not deal with SLA violations in the starting phase of machine learning, or need a basic approach since the application has unvarying workload. Bearing in mind the need of these applications, the proposed approach incorporates some existing configurations.

The auto-scaling is defined as horizontal scaling, which implies adding and removing resources as a whole. The scaling is performed only after the status of stability of the application changes. This corresponds to reactive-conFigured auto-scaling. Lastly, the auto-scaler follows static mode which means the thresholds are already fixed by the application provider programmatically.

3.1 Proposed solution

Basically, the proposed solution builds on the top of static auto-scaling approach. It takes into consideration the thresholds and values defined by the application owner and deletes some resources based on those values. The basic components needed for implementing the proposed solution in the auto-scaler are shown in Figure 1 . For enabling the elastic feature in the cloud, the auto-scaler is the main component of the system. It ensures proper provisioning of resources whenever the demand rises or drops. The resources represent the instances needed by the application deployed on the cloud. The load balancer balances the workload among the allocated resources to avoid overloading of any one of the allocated resources. Since the proposed solution is an enhancement in the static auto-scaler, Figure 1 shows the solution as a part of the static auto-scaler.

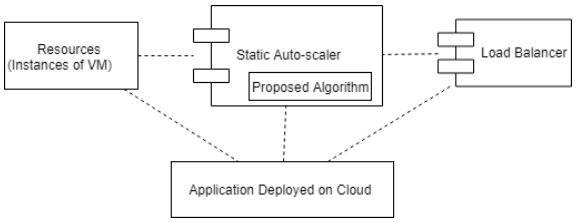


Figure 1: Component Diagram

```
while(application is running)
    if(CPU utilization > Thr_Upr)
        add Prov_Num resource
    else
        if(CPU utilization < Thr_Lwr)
            delete resource
        else
            if(overall CPU utilization < Thr_Lwr)
                calculate DeProv_Num
                delete DeProv_Num resources
        end
    end
end
```

Figure 2: Pseudo Code

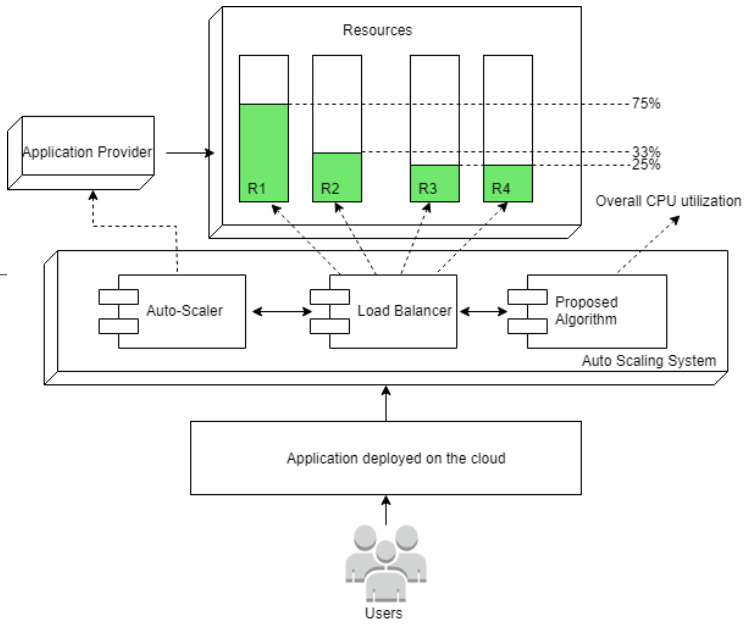


Figure 3: Deployment Diagram

These components are dependent on each other for performing their tasks. This dependency and detailed responsibilities of each element can be shown in the deployment diagram (3). The users are the consumers of the application which is hosted on the cloud. The application needs resources to provide service to users. The need for resources can

vary according to the demand. These resources are provided by the cloud providers with a specific configuration as needed by the application. The application provider is responsible for defining the thresholds to provide enough resources in order to avoid any kind of SLA violations. Since the auto-scaler used here is a static auto-scaler, the thresholds are provided by the application owner beforehand. The threshold values can be decided based on performance metrics like CPU utilization, request rate or response time of the application (Lorido-Botrán et al.; 2012). In this paper, the performance metric is chosen to the CPU utilization. The upper threshold (*Thr_Upr*) defines the upper bound of the CPU utilization reached by the resource to allocate new resources. Whereas the lower threshold (*Thr_Lwr*) defines the lower bound of the CPU utilization before deleting the resource. Also, the number of resources to be provisioned (*Prov_num*) for the application to reach stability is also fixed and pre-defined by the application provider.

As mentioned in the previous section, since the application owner tends to provide a higher number of resources to avoid SLA violation, there is an issue of over-provisioning of resources. To avoid this, the proposed solution monitors the resources, calculates *DeProv_num* and de-provisions resources to still meet the demand and avoid SLA violation. For example, as shown in the Figure 3, the resources R1 to R4 have an overall CPU usage of less than 40%. It means that even if one of these resources were shut down, the application will not suffer in terms of performance. The proposed algorithm provides the auto-scaler with a safe number of resources to be destroyed to mitigate over-provisioning depending on the threshold values and the current CPU utilization.

The pseudo code for the proposed algorithm is shown in Figure 2. First, similar to a basic static auto-scaler, it checks if the CPU utilization crosses some fixed threshold. If it does, the auto-scaler adds or removes a certain amount of resources. After auto-scaling, with the help of the load balancer, the workload is balanced among all the available resources. The load balancer makes sure that the resources are in a condition which will not violate any thresholds. At this stage, the proposed algorithm is implemented. When neither of the thresholds (Thr_Upr and Thr_Lwr) are exceeded, the proposed algorithm calculate the number of resources using the formula. The formula takes into consideration the thresholds provided by the application owner, number of currently allocated resources and their total CPU utilization.

$$DeProv\_Num = \frac{(Thr\_Upr \times n) - ((100 \times n) - sum)}{Thr\_Upr}$$

where,

$DeProv\_Num$ = Number of resources which can be deleted

Thr_Upr = higher bound set for checking VM overloading (in percentage)

n = Number of allocated resources

sum = Sum of CPU utilization of all VMs (in percentage)

For a better understanding of the flow of the auto-scaler and the implementation of the proposed solution, the Figure 4 presents the activity diagram of the proposed algorithm. There are three parts of the system as shown in the Figure 4. First, the application itself, which is hosted on the cloud. Second is the auto-scaler to monitor, provision or de-provision resources according to the current demand. The third is the load balancer which is responsible to balance the workload among the allocated resources. The application monitors the workload at specific time intervals to match one of the three conditions.

Rule1: If the CPU usage is higher than the threshold (*Thr_Upr*), add a predefined

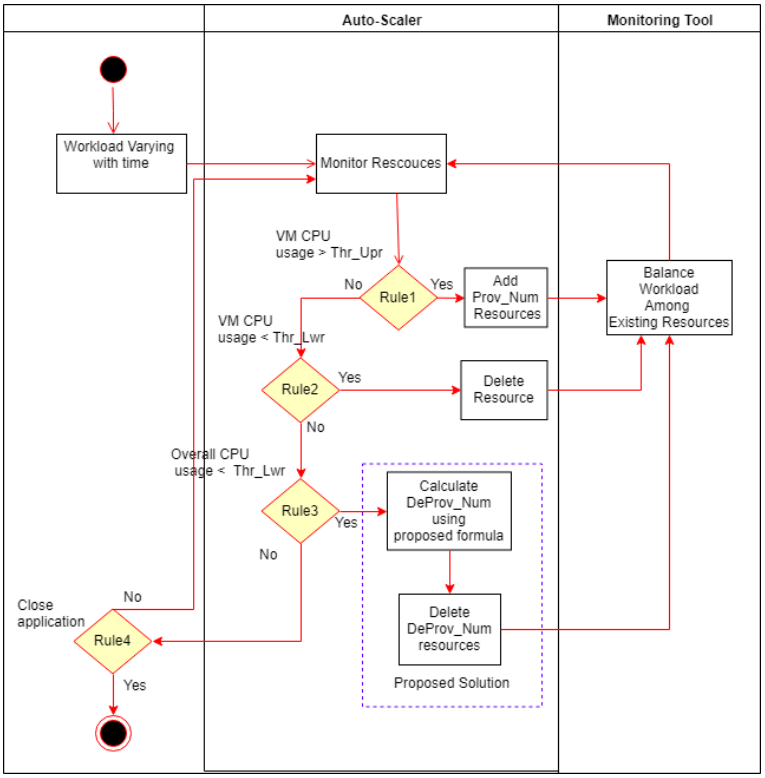


Figure 4: Activity Diagram

number of resources (*Prov_Num*). Rule2: If the CPU usage is lower than the threshold (*Thr_Lwr*), remove a resource. as If the CPU utilisation is within the interval defined by the two thresholds (i.e.*[Thr_Lwr, Thr_Upr]*) check the overall CPU utilization of the all the resources. Rule3: If the overall CPU utilization is less than the lower threshold (*Thr_Lwr*), calculate the number of resources to be destroyed using the proposed formula. Once the number of resources to be destroyed is calculated, the auto-scaler destroys the resources having the lowest CPU usage. This loop runs throughout the course of the application. The proposed solution analyzes the overall unutilized CPU percentage of all allocated resources and destroy the equivalent number of resources. This way, the issue of over-provisioning of resources can be mitigated. The implementation of this algorithm in a simulator is explained in the next section (section 4).

4 Implementation

This section provides details about the implementation of the proposed solution. The proposed solution is implemented in CloudSim Plus [3], a cloud simulation framework used for simulating cloud applications. With a variety of commercial cloud providers available for implementation and testing the proposed solution such as Amazon, Google, OpenStack, etc., cloud simulator was preferable due to the following reasons. Since evaluating the approach needs a variation of workload at different times in the application, simulation can provide a better control at managing the same. Moreover, testing the experiments with instances having varied configurations would be time-consuming. Cloud Simulator provides an added advantage of flexibility in sampling instances of different configura-

[3] `www.cloudsimplus.org`

tions. It allowed iterating the experiments a countless number of times without worrying about the time and expense. Besides, most of the commercial cloud infrastructure would have been expensive for renting instances.

4.1 CloudSim Plus: Simulation Framework

CloudSim Plus is an extended version of CloudSim 3 simulator framework (Manoel C. et al.; 2018). CloudSim Plus is a simulation framework which provides basic entities like a data center, VMs, SANs, and hosts, Additionally, it postulates the behavior of such entities exposed to diverse workloads (Silva Filho et al.; 2017). Moreover, it lets the user specify characteristics like RAM, storage, and processing elements (PEs) for the entities. Apart from the basic functionalities of simulating the cloud environment, CloudSim Plus has some some exclusive features like VM scaling, vertical and horizontal scaling and event listeners. These features are used as the foundation for building the algorithm proposed in this research. CloudSim Plus framework is written in Java, hence the algorithm also uses Java as the programming language. The use of CloudSim plus simulator framework for implementing and evaluating the algorithm is explained next.

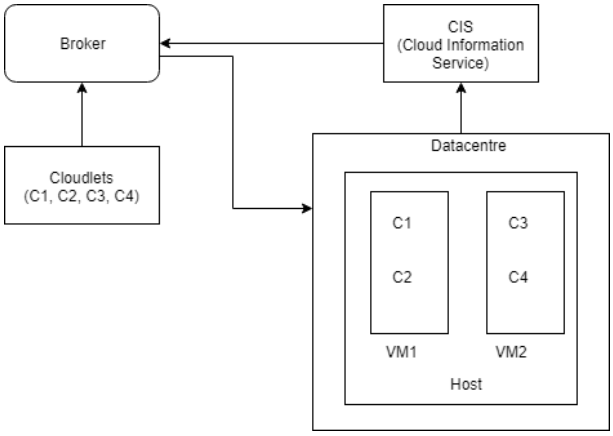


Figure 5: CloudSim Simulator

For simulating the cloud environment, CloudSim Plus has a class named *CloudSim* which provides the user with entities shown in fig 5. The Cloud Information Service (CIS) contains the registry of all the other entities once they are created. To build a data center, there are several entities that need to be created and registered to CIS. The data center contains hosts which have hardware characteristics like RAM, bandwidth, storage, number of processing elements (PEs) and their capacity . The datacenter passes the list of these hosts as parameters for its creation. These hosts are virtualized in the cloud infrastructure providing a number of VMs with specific configurations. The VM is configured with RAM, bandwidth, storage, number of PEs, PE capacity and scheduling algorithms. The number of hosts and VMs are adjusted for simulating different scenarios. In this research, every VM is provided with a feature of horizontal scaling. This feature defines a function which checks the CPU utilization every few seconds for overloading and a function which provides a specific number of VM if the condition stands true. The simulation framework provides with a datacenter broker to mediate between SaaS and the cloud provider (Calheiros et al.; 2011). The workload is simulated via tasks performed on the cloud called as cloudlets. Cloudlets are assigned to every VM tagged along with a space-shared scheduler. The cloudlets capacity is measured in Million

Instructions (MI). The number of cloudlets is varied according to the need of workload for the experiment. To mimic the authenticity of a real-world scenario, the cloudlets are dynamically created throughout the simulation at random times. These cloudlets are submitted to the broker and the broker forwards it to the VM in the datacenter for its completion. All the entities are destroyed once all the cloudlets finish their execution. The function *setVmDestructionDelayFunction()* makes sure that the VMs are destroyed only after few seconds of them being idle, namely 5 seconds. The time period 5 seconds is chosen to nake sure the VM has finished the execution of all cloudlets and is not just waiting for a cloudlet to get allocated to it.

4.2 Entities' Characteristics

The cloudsim entities like host, VMs and cloudlets has characteristics which are defined as per the application requirements.

- **Host**
  - The number of hosts to be created in the datacenter.
  - A list of Processing Elements (PEs) for each host.
  - Capacity of each PE in Million Instructions per second (MIPS)
  - RAM to be provided in megabytes
  - Storage to be provided in megabytes
  - Bandwidth to be provided in megabits
  - Enable State History for obtaining the CPU utilization for every host.
- **VM**
  - The number of VMs to be created.
  - A list of Processing Elements (PEs) for each VM.
  - Capacity of each PE in Million Instructions per second (MIPS)
  - RAM to be provided in megabytes
  - Storage to be provided in megabytes
  - Bandwidth to be provided in megabits
  - Enable Utilization History for obtaining the CPU utilization for every VM.
  - Enable Horizontal Scaling (ref figure 6)- For providing every VM with a feature of scalability, the horizontal scaling class supplies with a function *isVMOverloaded()* which checks the CPU utilization of the VM every 5 seconds. This time can be adjusted as per requirement in the variable named *SCHEDULING_INTERVAL*. Furthermore, the class also lets the application provision VMs if the VM has crossed the higher bound (*Thr_Upr*).

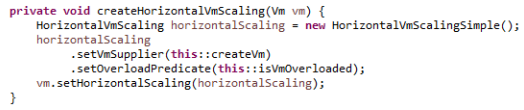


Figure 6: Horizontal Scaling

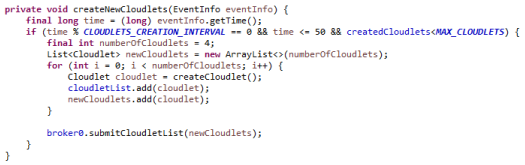


Figure 7: New Cloudlet Creation

- **Cloudlets**
  - The number of Cloudlets to be created.
  - Length of the Cloudlets in Million Instructions (MI).
  - Utilization Model for defining usage of RAM, CPU and Bandwidth.
  - Input, Output file sizes
  - Enable Event listener (ref figure 7)- Notify the system when any cloudlet finishes its execution and run the function named *onCloudletFinishListener()*.
  - Create Cloudlets at run time. The cloudlets are scheduled to get created and allocated to the VMs every few seconds until they reach the maximum cloudlets value (*Max_Cloudlets*) via a function *addOnClockTickListener* of the CLoudSim class. The scheduling interval can be adjusted by changing the variable value (*CLOUDLETS_CREATION_INTERVAL*).

4.3 Auto-Scaling

The auto-scaling works on a basic principle of monitoring, analyisng, planning and executing in a loop. For monitoring, the horizontal scaling class has provided with a function is *isVMOverloaded()*. Basically, the CPU utilization of the VM is checked against the two thresholds (*Thr_Upr*) and (*Thr_Lwr*) and accordingly resources are either provisioned or de-provisioned. If the CPU utilization value crosses *Thr_Upr*, a flag variable *overload* is set to 1. This flag is checked while creating a new VM for provisiong. The flag set to 1, indicates that the resources are overloaded, and hence it will create *Prov_Num* number of VMs.

If the CPU utilization value is lower than *Thr_Lwr*, a function for deleting the VM (*destroyVM()*) is called. This function will migrate the cloudlets from the waiting list of the underutilized VM to another VM chosen from the *VmList* (List of allocated VMs). Since, the CPU utilization of the VM is low but not zero, it denotes that there are cloudlets running on the VM. Hence the VM is added to the *vmDestroyList*. This list is checked everytime any cloudlet finished its execution via the event listener function *onCloudletFinishListener()*. So whenever a cloudlet finishes its execution, this function will check if the VM to whoch the cloudlet was assigned, is in the *vmDestroyList* list. If it stands true, then it deletes that particular VM.

4.3.1 Static Approach

The basic auto-scaling with static configuration is leveraged to include the proposed algorithm. This helps in evaluating the performance difference between the proposed algorithm and the basic one. The static algorithm needs several thresholds to be defined like the CPU percentage as a high bound (*Thr_Upr*) for allocating instances, low bound

(*Thr_Lwr*) for deleting instances, number of instances (*Prov_Num*) to be allocated on up-scaling. Before starting the simulation, all the entities must be created by defining their hardware configurations and registered with CIS. The VMs and the cloudlets are submitted to the broker mentioning their scheduling policy. Once the simulation starts, the cloudlets are assigned to the VMs and the VMs start executing the cloudlets in a space-shared manner. The simulation follows the auto-scaling mechanism as stated in subsection. It will monitor the CPU utilization of VMs, check the value against the thresholds set and allocate or de-allocate resources accordingly.

4.3.2 Proposed Approach

For implementing the proposed algorithm, the main difference lies in the analysis part of the auto-scaling. In the static auto-scaling, the monitored CPU utilization value is checked against the upper bound (*Thr_Upr*) and lower bound (*Thr_Lwr*) in the *isVMOverloaded()* function. However, in the proposed solution, this function will also check if the overall CPU utilization of all the VMs included, is lower than *Thr_Lwr*. If this condition stands true, the algorithm (mentioned in section 3) calculates the number of resources to be destroyed, is processed. In order to delete the VMs with the lowest CPU utilization, the VMs are sorted in ascending order according to their current CPU utilization. For every VM to be deleted, the function named *deleteVm()* is called.

5 Evaluation

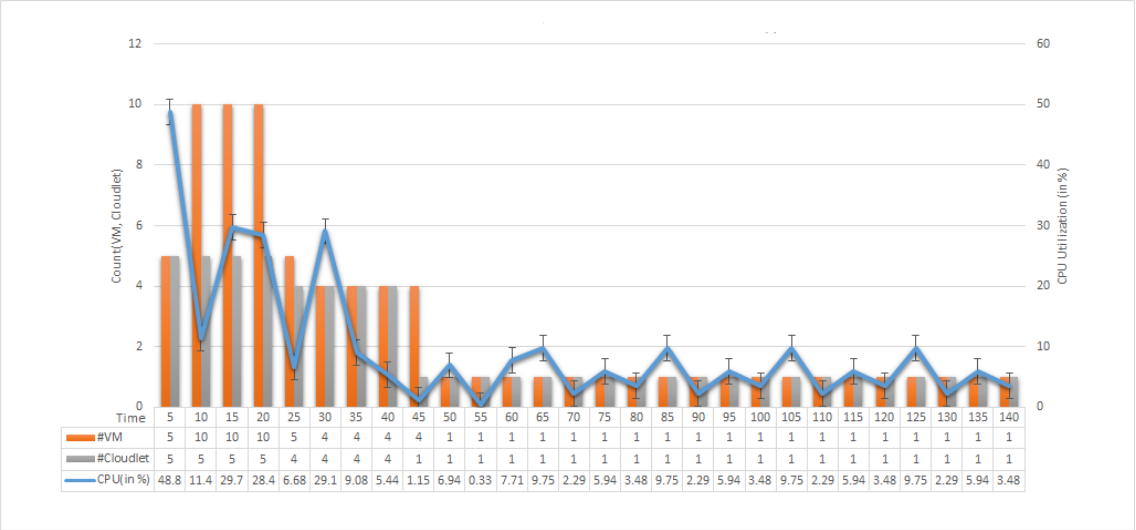
This section presents the empirical evaluation of the proposed algorithm for its performance, accuracy and reliability. There are a total of four case studies examining various parameters in scenarios simulating either a small scale application or a large scale application. The performance of the proposed solution is evaluated by examining the basic functionality of an auto-scaler. Depending on the workload, the auto-scaler is examined for its provisioning and de-provisioning of resources (i.e. allocates resources when the workload is high and de-allocates when the workload is low). Furthermore, since the proposed solution is built upon a static auto-scaler, this section compares both the static and the proposed approach under similar workload for the cost of resources they might incur.

The performance of the auto-scaler is examined by monitoring the allocation time of the VMs and the overall CPU utilization of the resources according to the variation of the workload. The VMs are expected to be allocated whenever there is a rise in workload, Moreover the graph of CPU utilization over time is also expected to be in accordance with the graph of workload over time.

As stated in section 1, the motivation behind improving the static auto-scaler is to obtain a cost-effective auto-scaling approach. Since the cloud providers provide a pay-per-use scheme for the resources being allocated, the application provider has to pay unnecessarily for the over-provisioned resources. Hence to ensure the proposed approach optimizes cost of resources over the static approach, the cost of the allocated resources per unit time throughout the course of the application is used for comparison.

5.1 Performance Analysis of the basic (static) approach

This experiment is conducted to verify the basic functionalities of an auto-scaler. Any basic auto-scaler is expected to provision resources when the workload rises and de-provision them when the workload stabilizes. Furthermore, the CPU utilization is also expected to rise as the workload rises. Hence for this experiment, the CPU utilization and the allocated VMs is compared with the number of cloudlets running (depicting the workload) at particular time intervals. The configuration of the simulation is set according to a small scale application.



Time	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120	125	130	135	140
#VM	5	10	10	10	5	4	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
#Cloudlet	5	5	5	5	4	4	4	4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
CPU(in %)	48.8	11.4	29.7	28.4	6.68	29.1	9.08	5.44	1.15	6.94	0.33	7.71	9.75	2.29	5.94	3.48	9.75	2.29	5.94	3.48	9.75	2.29	5.94	3.48	9.75	2.29	5.94	3.48

Figure 8: Performance Analysis of Static Approach

The figure 8 represents the performance analysis of the proposed solution. A dual axis graph provides a better comparison between two parameters having different units of measurements. The x-axis depicts the time interval of the course of the application. The y-axis on the left-hand side represents the number of cloudlets executing and the number of VM allocated. The secondary axis on the right-hand side represents the CPU utilization of the host. In the first time interval (0-5), the number of cloudlets and VMs are 5 each. The system allocates one cloudlet to each VM. During the same time interval (0-5) in the figure, the CPU utilization has a high value of around 50%. Due to this high CPU utilization, the auto-scaler allocates 5 additional VMs to the application. As soon as the VMs are allocated, the overall CPU utilization drops to 12%. This shows that the workload is balanced between all the allocated VMs. The auto-scaler de-provisions the VM at the time 25 when the cloudlets starts finishing their execution. At time interval 40-45, the number of cloudlets executing drops from 4 to 1. Hence the VM are de-provisioned in the next time interval (45-50). Moreover, the accuarcy of the proposed approach can also be analysed from the figure 8.The number of allocated VMs seem to follow the number of cloudlets at each time interval, which signifies that the auto-scaler provisions and de-provisions the VMs according to the workload. Theerfore, considering the performance and the accuracy, we can say that the auto-scaler using the proposed approach operates as expected.

5.2 Comparison of Proposed and Static approach on a small scale application

This experiment is carried out to compare the proposed solution and the static auto-scaling approach in terms of the number of resources being used in any small-scale application. In other terms, the experiment is simulating any small-scale application with the threshold values in accordance with the scale of the application. This experiment is simulating the situation where the application provider has the knowledge of the application and the workload it might experience. Hence, the thresholds provided by them will be well-suited for the type of the application and the workload. The evaluation metric used here is the cost per unit time since the experiment is trying to compare the cost which the application provider might be charged for the number of resources used by both types of auto-scalers.

The hardware configurations of CloudSim Plus simulator for this experiment is the same for both static auto-scaling approach and the proposed approach are shown in Table 2.

Table 1: Configuration for small scale application

Entity	**Number of Entities**	**RAM (MB)**	**Storage (MB)**	**Bandwidth (MegaBits)**	**No of PEs**	**PE capacity**
Host	4	6000	1,000,000	10,000	50	1000
VM	6	512	10,000	1000	2	1000

The threshold values and configuration to simulate a small scale application is stated below:

Prov_Num=2; *Thr_Upr*=0.8; *Thr_Lwr*=0.3;

Minimum number of cloudlets= 10; Maximum Number of Cloudlets=100;

Cloudlet length= 10000 in MI (Million Instructions)

The threshold values are provided by the application owner. The *Thr_Upr* and *Thr_Lwr* are the upper and lower bound against which the CPU utilization is checked for up-scaling or down scaling. *Prov_Num* defines the number of resources to be allocated when the CPU utilization reaches *Thr_Upr*. The number of cloudlets are provided considering the scale of the application. The minimum number of cloudlets that are created at the start of the simulation. When the cloudlets are created at runtime, the cloudlets are not supposed to exceed the maximum number. All these cloudlets have their lengths (capacity) provided in million instructions (MI).

With the given setup, the simulation for static auto-scaling with both the basic approach and the proposed approach is run on CloudSim Plus simulator. The start and finish time for each of the VMs are monitored and aggregated in order to obtain the total execution time of all the VMs. The total cost of using the VMs is calculated by multiplying this time by the cost per unit time. This experiment is repeated 110 times and the data is collected for both the approaches. A line graph is plotted with cost per unit time on x-axis and number of cloudlets on the y-axis. The cost of a resource is considered as 1 unit and the number of cloudlets represent the intensity of workload the application might be experiencing.

Figure 9 shows the behavior of a small scale application in terms of cost per unit time of the application as the workload varies for both static and proposed approach. The x-axis represents the number of cloudlets and y-axis represents the median of cost

per unit time. As shown in the figure 9, according to the workload, the total cost per unit time of the resources differs. This denotes that for higher workload, the application tends to use a higher number of resources regardless of the approach used. However, the proposed approach represented by the blue line seems to incur overall less cost of resources per unit time. This signifies that the proposed solution uses comparatively less number of resources for the same configurations. The proposed approach incurs 24.17% less cost per unit time than the static approach.

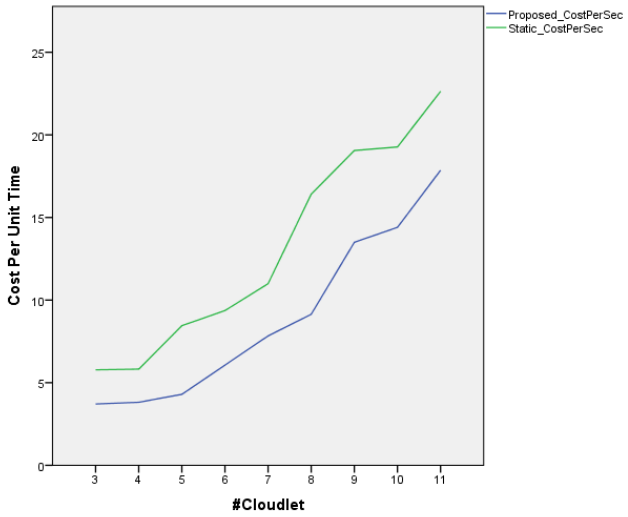


Figure 9: Cost per unit time for small scale applications

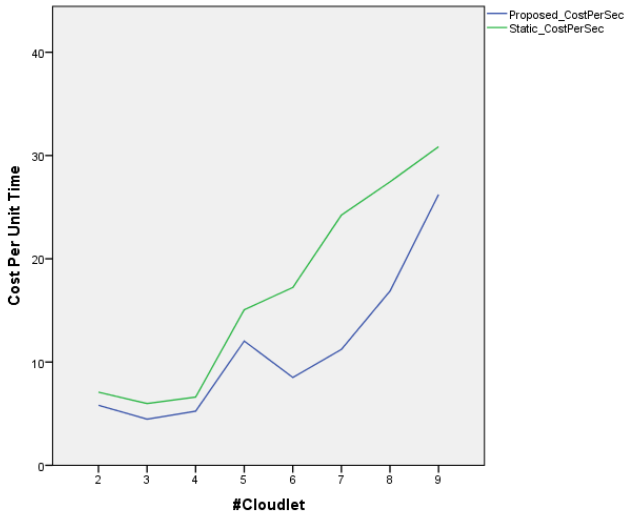


Figure 10: Cost per unit time for small scale applications with high provisioning value

5.3 Comparison of Proposed and Static approach on a small scale application with a high provisioning value (*Prov_Num*)

An application provider has a tendency to provide a provisioning value (Prov_Num) higher than the range of the application scale. Since small-scale applications tend to have smaller bursts of workload, a smaller provisioning value is appropriate for a those kinds of applications . However, some application providers unnecessarily provide a higher *Prov_Num* for their small-scale application to ensure that the application will not violate any SLA rules. As mentioned in section 2, this is one of the main reasons causing resource over-provisioning. This experiment simulates a similar scenario of a small scale application with a high provisioning value (*Prov_Num*). The experiment is repeated 100 time under the configuration similar to the previous experiment for a small scale application (refer table 2). Since this experiment is simulating a scenario where the application provider provides a higher provisioning value, instead of Prov_Num as 2, this experiment has the provisioning value (Prov_Num) set to 5.

Evaluating the same metric as the previous experiment, the figure 10 represents the comparison of the proposed and static algorithm in terms of the cost of resources per unit time. The x-axis represents the number of cloudlets. The y-axis represents the median of cost per unit time. The figure 10 shows that as the number of workload increases, the cost of resources er unit time also increases. It signifies that even if the application provider provides a high number of provisioning resources, the proposed approach tends

to use only necessary number of resources. Hence, we can say that, regardless of the higher provisioning number, the proposed approach abides to the basic cost optimization in the auto-scaler than static approach. The proposed approach is approximately 21.9% cheaper per unit time than the static approach.

5.4 Comparison of Proposed and Static approach on a large scale application

This experiment simulates a large scale application and the effects of both proposed and static auto-scaling approaches on it. The hardware configurations of CloudSim Plus simulator for this experiment simulating a large scale applcation is the same for both static auto-scaling approach and the proposed approach and is shown in Table 2.

Table 2: Configuration for large scale application

Entity	**Number of Entities**	**RAM (MB)**	**Storage (MB)**	**Bandwidth (MegaBits)**	**No of PEs**	**PE capacity**
Host	20	600000	100000000	1000000	100	1000
VM	10	6000	100000	10000	10	1000

The threshold values and configuration to simulate a small scale application is stated below:

Prov_Num=10; *Thr_Upr*=0.8; *Thr_Lwr*=0.3;

Minimum number of cloudlets= 100; Maximum Number of Cloudlets=5000;

Cloudlet length= 10000 in MI (Million Instructions)

The threshold values provided by the application owner for large scale application which denote the upper and lower bound against which the CPU utilization is checked for up-scaling or down-scaling. *Prov_Num* defining the number of resources to be allocated when the CPU utilization reaches *Thr_Upr* is higher considering the high values of hardware configurations of the large scale application. Similarly the minimum and maximum number of cloudlets provided have a higher value. With this configuration, the experiment is repeated 100 times to collect enough data for analysis.

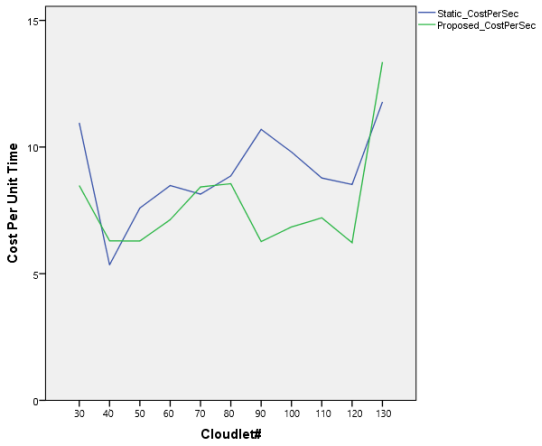


Figure 11: Cost per unit time for large scale application

With the configuration mentioned above, the experiment is repeated 100 times to obtain a result as shown in figure 11. The x- axis represents the number of cloudlets. The y-axis represents the cost per unit time. The lines representing both the approaches does not seem to have a significant difference regardless of the workload. This implies that the number of resources allocated in proposed approach are not significantly lower than the number of resources allocated in static approach. This experiment signifies that the proposed approach does not perform significantly better in a large scale application. Observing closely, the proposed algorithm costs only 1% lower than the static approach would.

5.5 Discussion

This section provides a critical analysis of the outcomes of the experiments carried out. The improvements and limitations of the proposed approach over static and other approaches are mentioned and discussed.

The proposed solution is evaluated and verified that it operates according to the basic functionalities accurately. Moreover the results showed that the proposed approach has significant enhancements over the static approach. The first experiment (5.1) examined if the proposed approach operates as expected from any auto-scaling approach. In figure 8, the number of VMs are provisioned as the workload (number of cloudlets) increased and de-allocated whenever the workload dropped. Moreover, the CPU utilization graph which varies according to both the number of VMs and cloudlets signifies that the workload is distributed among the provisioned VMs. Since the proposed solution seems to operate as expected, the subsequent experiments (5.2, 5.3, 5.4) evaluates the performance (in terms of cost per unit time) of the approach against the static approach. The results of the second experiment (refer figure 9) claimed that the number of resources used in any small scale application is lower if the proposed approach is followed as compared to the static approach. Hence we can say that the proposed approach optimizes cost by around 24.17%. Moreover, despite of the application provider providing a higher provisioning value (*Prov_num*) to avoid SLA violations, as simulated in the third experiment 5.3, the proposed approach is able to satisfy the workload demand using lower number of resources as compared to static approach. This is shown in figure 11 which presents that the cost per unit time is low for the proposed approach. Hence we can say that the proposed approach can take care of a higher provisioned value (*Prov_Num*) by allowing the small-scale application use only the number of resources necessary. The fourth experiment 5.4 provides a simulation of a large scale application, and the outcomes reveal that proposed approach does not out-perform the static approach in terms of resources used. Although,it does not cost more than static approach, since the cost optimixation according to the experiment 5.4is 1%. It means that the proposed solution uses comparatively (though not significantly) less number of resources for most of the values of number of cloudlets in the case of large-scale application.

Thus, the experiments prove that the proposed approach has shown enhancements over the static approach to some extent and that it still has scope to improve. Since the proposed solution was only compared against the static approach, it is still unsure if the approach is better than the other approaches which use the machine learning algorithms to predict the workload. Moreover, since the proposed approach is built upon static approach, it will not provide its best efficiency on bursty workloads.

6 Conclusion and Future Work

This paper proposes an algorithm in order to mitigate the issue of resource over-provisioning in the static approach of auto-scaling. Application providers tend to provide a higher resource provisioning value while setting up the application in order to steer clear of violating any SLA rules. This causes resource over-provisioning and eventually incurs unnecessary cost to the application provider. To solve this issue, the proposed algorithm calculates the number of instances which can be shut down without affecting the performance of the application. The algorithm is a basically a mathematical formula which considers the thresholds provided by the application provider for up-scaling and the current CPU utilization of the allocated resources and hence does not have any performance overhead. The proposed algorithm operates the basic functionalities as expected in the simulator accurately. Considering cost per unit time as the evaluation metric, the paper compares the performance of the proposed approach with the static one. The results denote that de-allocating the extra resources proved to be around 20-25% cost optimized in various scenarios of a small scale application. The proposed algorithm basically outperforms the static approach in most of the circumstances.

The future work of this research can be focused on mitigating the issues found during the evaluation of the proposed solution. In the case study of the large scale application, the proposed approach showed the cost optimization of only 1%. Besides, the proposed approach might have an additional cost issue due to the frequent creation and destruction of resources (Liao et al.; 2015). Additionally, the algorithm can be compared against the dynamic approach and enhanced accordingly. Since the algorithm exhibited significant improvement over static approach, it could be applied to an auto-scaler for an application deployed on the cloud.

References

Aceto, G., Botta, A., De Donato, W. and Pescapè, A. (2013). Cloud monitoring: A survey, *Computer Networks* **57**(9): 2093–2115.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. and Merle, P. (2018). Elasticity in cloud computing: state of the art and research challenges, *IEEE Transactions on Services Computing* **11**(2): 430–447.

Anand, A., Dhingra, M., Lakshmi, J. and Nandy, S. (2012). Resource usage monitoring for kvm based virtual machines, *Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on*, IEEE, pp. 66–70.

Aslanpour, M. S., Ghobaei-Arani, M. and Toosi, A. N. (2017). Auto-scaling web applications in clouds: A cost-aware approach, *Journal of Network and Computer Applications* **95**: 26–41.

Buyya, R. and Son, J. (2018). Software-defined multi-cloud computing: A vision, architectural elements, and future directions, *arXiv preprint arXiv:1805.10780* .

Calcavecchia, N. M., Caprarescu, B. A., Di Nitto, E., Dubois, D. J. and Petcu, D. (2012). Depas: a decentralized probabilistic algorithm for auto-scaling, *Computing* **94**(8-10): 701–730.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. and Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, *Software: Practice and experience* **41**(1): 23–50.

Chazalet, A., Tran, F. D., Deslaugiers, M., Lefebvre, R., Telecom, F., Exertier, F. et al. (2011). Adding self-scaling capability to the cloud to meet service level agreements an open-source middleware framework solution.

Chen, T. (2016). Self-aware and self-adaptive autoscaling for cloud based services, *arXiv preprint arXiv:1608.04030* .

Chen, T., Bahsoon, R. and Yao, X. (2018). A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems, *ACM Computing Surveys (CSUR)* **51**(3): 61.

Copil, G., Moldovan, D., Truong, H.-L. and Dustdar, S. (2013). Multi-level elasticity control of cloud services, *International Conference on Service-Oriented Computing*, Springer, pp. 429–436.

Cunha, R. L., Assunçao, M. D., Cardonha, C. and Netto, M. A. (2014). Exploiting user patience for scaling resource capacity in cloud services, *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, IEEE, pp. 448–455.

Dawoud, W., Takouna, I. and Meinel, C. (2011). Elastic vm for cloud resources provisioning optimization, *International Conference on Advances in Computing and Communications*, Springer, pp. 431–445.

Dhingra, M., Lakshmi, J. and Nandy, S. (2012). Resource usage monitoring in clouds, *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*, IEEE, pp. 184–191.

Galante, G. and Bona, L. C. E. (2013). Constructing elastic scientific applications using elasticity primitives, *International Conference on Computational Science and Its Applications*, Springer, pp. 281–294.

Galante, G. and Bona, L. C. E. d. (2012). A survey on cloud computing elasticity, *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, IEEE Computer Society, pp. 263–270.

Galante, G. and De Bona, L. C. E. (2015). A programming-level approach for elasticizing parallel scientific applications, *Journal of Systems and Software* **110**: 239–252.

Gandhi, A., Dube, P., Karve, A., Kochut, A. and Zhang, L. (2018). Model-driven optimal resource scaling in cloud, *Software & Systems Modeling* **17**(2): 509–526.

Ghobaei-Arani, M., Jabbehdari, S. and Pourmina, M. A. (2018). An autonomic resource provisioning approach for service-based cloud applications: A hybrid approach, *Future Generation Computer Systems* **78**: 191–210.

Gong, Z., Gu, X. and Wilkes, J. (2010). Press: Predictive elastic resource scaling for cloud systems., *CNSM* **10**: 9–16.

Han, R., Guo, L., Ghanem, M. M. and Guo, Y. (2012). Lightweight resource scaling for cloud applications, *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, IEEE Computer Society, pp. 644–651.

Herbst, N. R., Kounev, S. and Reussner, R. H. (2013). Elasticity in cloud computing: What it is, and what it is not., *ICAC*, Vol. 13, pp. 23–27.

Jacob, B., Lanyon-Hogg, R., Nadgir, D. K. and Yassin, A. F. (2004). A practical guide to the ibm autonomic computing toolkit, *IBM Redbooks* **4**: 10.

Jeffrey, H. C. L. S. B. and Chase, S. (2010). Automated control for elastic storage, *Proceedings of the seventh international conference on autonomic computing, Washington, DC*, pp. 7–11.

Kesavulu, M., Nguyen, D., Tien, D., Helfert, M. and Bezbradica, M. (2018). An overview of user-level usage monitoring in cloud environment.

Liao, W.-H., Kuai, S.-C. and Leau, Y.-R. (2015). Auto-scaling strategy for amazon web services in cloud computing, *Smart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference on*, IEEE, pp. 1059–1064.

Lim, H. C., Babu, S., Chase, J. S. and Parekh, S. S. (2009). Automated control in cloud computing: challenges and opportunities, *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACM, pp. 13–18.

Lorido-Botrán, T., Miguel-Alonso, J. and Lozano, J. A. (2012). 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.

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

Manoel C., S. F., Raysa L., O., Claudio C., M., Pedro R. M., I. and Freire, M. M. (2018). Cloudsim plus, *Departamento de Informtica* .
URL: *http://cloudsimplus.org/*

Maurer, M., Brandic, I. and Sakellariou, R. (2012). Self-adaptive and resource-efficient sla enactment for cloud computing infrastructures, *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, IEEE, pp. 368–375.

Meinel, C., Dawoud, W. and Takouna, I. (2011). Elastic vm for dynamic virtualized resources provisioning and optimization.

Mell, P., Grance, T. et al. (2009). The nist definition of cloud computing, *National institute of standards and technology* **53**(6): 50.

Netto, M. A., Cardonha, C., Cunha, R. L. and Assunçao, M. D. (2014). Evaluating auto-scaling strategies for cloud computing environments, *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, IEEE, pp. 187–196.

Qu, C., Calheiros, R. N. and Buyya, R. (2018). Auto-scaling web applications in clouds: A taxonomy and survey, *ACM Computing Surveys (CSUR)* **51**(4): 73.

RightScale (2018).
URL: *https://www.rightscale.com/*

Roy, N., Dubey, A. and Gokhale, A. (2011). Efficient autoscaling in the cloud using predictive models for workload forecasting, *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, IEEE, pp. 500–507.

Silva Filho, M. C., Oliveira, R. L., Monteiro, C. C., Inácio, P. R. and Freire, M. M. (2017). Cloudsim plus: a cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness, *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, IEEE, pp. 400–406.

Wuhib, F., Stadler, R. and Lindgren, H. (2012). Dynamic resource allocation with management objectives: Implementation for an openstack cloud, *Proceedings of the 8th international conference on Network and Service Management*, International Federation for Information Processing, pp. 309–315.

National College of Ireland

Configuration Manual

MSc Research Project
Cloud Computing

Manasi Patil
Student ID: x17156742

School of Computing
National College of Ireland

Supervisor: Adriana Chis

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Manasi Patil
Student ID:	x17156742
Programme:	Cloud Computing
Year:	2018
Module:	MSc Research Project
Supervisor:	Adriana Chis
Submission Due Date:	20/12/2018
Project Title:	Configuration Manual
Word Count:	285
Page Count:	5

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	20th December 2018

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Manasi Patil
x17156742

1 System requirements

- OS: Windows 10
- Processor: Intel(R) Core(TM) i3-4030 CPU@1.90Ghz
- RAM: 8 GB
- System Type: 64 bit

2 Install an IDE

Eclipse IDE with integrated plugin for JAVA development is a platform used to simulate the cloud environment.The steps [1] to be followed to install it are:

- Install Java Development ToolKit (JDK), since the simulation is using Java as the programming language. Use

  `http://www.ntu.edu.sg/home/ehchua/programming/howto/JDK_HowTo.html#jdk-install`

  for the steps to be followed for installation.

- Download the Eclipse IDE from

  `https://www.eclipse.org/downloads/packages/release/juno/sr2/eclipse-ide-java-ee-developers`

- Unzip the downloaded folder and open the execution(.exe) file to open Eclipse IDE.

[1] `http://www.ntu.edu.sg/home/ehchua/programming/howto/eclipsejava_howto.html`

3 Simulate Cloud Environment

For simulating the cloud environment, a simulation framework, CloudSim Plus is used. The steps [2] to implement the simulation in Eclipse IDE are:

- Download the project source from `http://cloudsimplus.org/`
- Unzip the downloaded folder in the eclipse IDE's workspace.
- Open Eclipse IDE and import the cloudsim plus folder as an existing MAVEN project. The project will be saved in a structure as shown in figure 1.

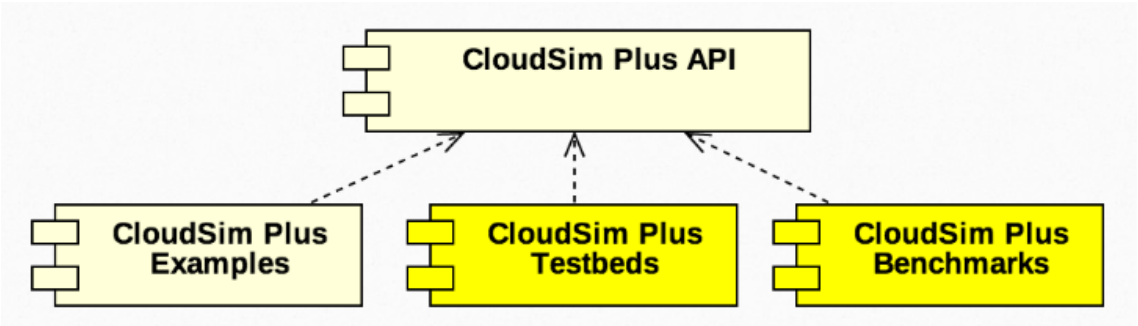


Figure 1: CloudSim Structure

- Allow the IDE to automatically install the packages required to run the cloudsim plus project.

4 Execution

- **Implement the approach**

  Select *src/main/java* under *cloudsim-plus-examples* and import the package *org.cloudsim.cloudbus.staticautoscaling*.

- **Set the configurations**

  The cloud environment has hardware configurations and characteristics for hosts, VMs and cloudlets which can be set to simulate particular instance type or a scenario. In addition to that, the thresholds values(*Thr_Upr, Thr_Lwr, Prov_Num*) can also be provided. Section 5 provides the configuration for simulating the scenarios under which the proposed approach is evaluated in the research.

- **Run the java code and obtain simulation results**

  The simulation results can be obtained similar to the figure 2. The CPU utilization of every VM or Hosts can also be obtained.

[2] `http://cloudsimplus.org/`

SIMULATION RESULTS

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	0	20	0	2	10000	2	0	17	18
5	SUCCESS	1	0	20	0	2	10000	2	17	38	22
10	SUCCESS	1	0	20	0	2	10000	2	38	65	27
6	SUCCESS	1	0	20	0	2	10000	2	65	85	20
7	SUCCESS	1	0	20	0	2	10000	2	85	105	20
8	SUCCESS	1	0	20	0	2	10000	2	105	125	20
9	SUCCESS	1	0	20	0	2	10000	2	125	145	20
1	SUCCESS	1	0	20	1	2	10000	2	0	17	18
11	SUCCESS	1	0	20	1	2	10000	2	17	39	23
2	SUCCESS	1	0	20	2	2	10000	2	0	17	18
12	SUCCESS	1	0	20	2	2	10000	2	17	39	23
3	SUCCESS	1	0	20	3	2	10000	2	0	17	18
13	SUCCESS	1	0	20	3	2	10000	2	17	39	23
4	SUCCESS	1	0	20	4	2	10000	2	0	17	18

Figure 2: Simulation Result

5 Implementing Case Studies

The evaluation performed on the proposed solution simulates different scenarios for small scale or large scale application. The configurations for small scale and large scale differ in thresholds, and hardware configuration and entity characteristics.

5.1 Performance Analysis of Proposed Solution

For this experiment, the configuration tries to simulates a small scale application. The configurations are shown in figure 3. For collecting the data for evaluation, the CPU utilization of the hosts for every 5 seconds are collected and stored in a file. The figure 4 presents a code snippet of the function *showCpuUtilizationForAllHosts()* where the CPU utilization and time is stored in a text file named *Host.txt*. Moreover for evaluating the performance the execution start and finish time for cloudlets and VMs are also required. The figure 5 shows the code snippet of the function *printSimulationResults()* which adds the cloudlet id and the start and finish time of all the cloudlets in a textfile named *cloudlet.txt*. Whereas the VM id with the start and finish time of all the VMs is stored in a *VM.txt*. All the created text files can be found under *org.cloudsim.cloudbus.staticautoscaling*. Using this data, graph can be plotted in order to analyze the performance and accuracy of the proposed algorithm.

```
private static final int SCHEDULING_INTERVAL = 5;

//The interval to request the creation of new Cloudlets.
private static final int CLOUDLETS_CREATION_INTERVAL = SCHEDULING_INTERVAL * 2;

private static final int HOSTS = 1; //No of Hosts
private static final int HOST_PES = 20; //No of PEs for every Host
private static final int VMS = 5;   //No of VMs
private static final int VM_PEs=2;  //No of PEs for each VM
private static final int CLOUDLETS =10;     //Minimum No of Cloudlets
private static final int MAX_CLOUDLETS =50; //Maximum No of Cloudlets
private static final int Prov_Num = 2;  //No of VMs to be provisioned on overloading
```

Figure 3: Configuration for simulation

```
for (Map.Entry<Double, Double> entry : host.getUtilizationHistorySum().entrySet()) {
    final double time = entry.getKey();
    final double cpuUsage = entry.getValue()*100;
    numberOfUsageHistoryEntries++;
    if(time%5==0)
    {
    //write to a text file
    try (
        PrintStream out = new PrintStream(new FileOutputStream("Host.txt", true))) {
             out.println(cpuUsage+","+time);
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
   }
 }
```

Figure 4: Collect CPU utilization

```
for(int i=0;i<cloudletList.size();i++)
{
    Cloudlet cl=cloudletList.get(i);
    double starttime=cl.getExecStartTime();
    double stoptime=cl.getFinishTime();
//write to a text file
try (
    PrintStream out = new PrintStream(new FileOutputStream("cloudlet.txt", true))) {
         out.println(cl.getId()+","+starttime+","+stoptime);

} catch (FileNotFoundException e) {
    e.printStackTrace();
}}
for(int i=0;i<vmList.size();i++)
{
    Vm vm=vmList.get(i);
    double starttime=vm.getStartTime();
    double stoptime=vm.getStopTime();
     //write to a text file
    try (
        PrintStream out = new PrintStream(new FileOutputStream("VM.txt", true))) {
        out.println(vm.getId()+","+starttime+","+stoptime);

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
```

Figure 5: Collect Cloudlet and VM data

5.2 Comparison of Proposed and Static Approach

The experiments for comparing the proposed approach with the static one have either simulated a small scale application or a large scale application. After setting the configurations for the application as provided in the research, these experiments evaluate the cost per unit time with the number of cloudlets. This data can be gathered as shown in the figure 6. The figure 6 is a code snippet of the function *printSimulationResults()* in the static approach, which adds the cost per unit time and the number of finished cloudlets to a text file named *static.txt*. This code must be added in both the approaches(proposed and static) and run 100 times to collect enough data for comparison. For repeating the experiments around a 100 times, the code *runExperiments.java* can be run. Both the text files can be analyzed and have the capability of obtaining a line graph for comparison.

```
for(int i=0;i<vmList.size();i++)
{
    Vm vm=vmList.get(i);
    double starttime=vm.getStartTime();
    double stoptime=vm.getStopTime();
    if(starttime!=-1 && stoptime!=-1)
    time=time+(stoptime-starttime);
}
time=time/simulation.clock();
//write to a text file
try (
    PrintStream out = new PrintStream(new FileOutputStream("static_file.txt", true))) {
         out.println(finishedCloudlets.size()+","+time);

} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
```

Figure 6: Collect #Cloudlet and Cost Per Unit Time